Thesis Dissertation

HIGH BIAS BRANCH PREDICTION ON HARDWARE LEVEL

George Demetriou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

HIGH BIAS BRANCH PREDICTION ON HARDWARE LEVEL

George Demetriou

Supervisor
Dr. Yiannakis Sazeidis

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2025

Acknowledgments

I would like to begin by expressing my sincere gratitude to my thesis supervisor, Dr. Yiannakis Sazeidis. His unwavering support, availability, and guidance throughout the duration of this work have been invaluable. I am particularly thankful for his insightful feedback and thoughtful advice, which consistently motivated me to persevere and improve. His ability to offer alternative perspectives greatly enriched my understanding and approach to the subject matter.

I also wish to extend my appreciation to all the professors who have contributed to my academic development over the past four years. Their dedication to teaching and their commitment to fostering intellectual growth provided me with a strong academic foundation, upon which this thesis has been built.

To all those acknowledged, as well as to anyone whose contributions may not be explicitly mentioned, I offer my heartfelt thanks.

Summary

This thesis investigates a static program-based approach to branch prediction, aiming to complement existing hardware-based predictors. Motivated by the limited research on highly biased branches, it explores the feasibility of accurately predicting the direction of such branches using static analysis techniques.

Two classification tasks are addressed. The first is predicting whether a branch is taken or not-taken. The second involves identifying high-bias branches. To solve both tasks, a diverse set of machine learning models was evaluated, including convolutional neural networks (CNNs), multilayer perceptrons (MLPs), Random Forests (RF), and XGBoost (XGB). Features were extracted from a corpus of SPEC CPU-style benchmarks, and models were trained under both per-benchmark and cross-benchmark regimes using only static information such as opcode patterns, operand types, jump span, and control-flow context.

The empirical results show that tree-based ensemble models (RF, XGB) significantly outperform neural models (CNN, MLP) in the taken/not-taken task, while CNN and MLP suffer from poor recall. In contrast, all models perform well on high-bias classification, with XGBoost achieving the highest accuracy, confirming the separability of this class using static features alone.

Based on these findings, the thesis proposes a hardware-level architectural enhancement: the Static Program-Based Branch Prediction Unit (SPBBPU). This unit targets high-bias branches specifically and integrates with existing branch predictors and compiler hints. Its predictions are combined via a multiplexer that selects the output based on predicted bias status, allowing for improved accuracy and robustness in biased scenarios.

The approach is shown to be feasible through data analysis, model performance, and per-benchmark breakdowns. High-bias branches are common across all programs; their bias can be accurately detected; and, once identified, their outcomes can be predicted reliably. These results suggest a practical path toward incorporating learned static branch behavior into future microarchitectures, enabling more accurate, low-overhead predictions at decode time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Branch prediction remains a fundamental challenge in modern computer architecture, with significant implications for instruction-level parallelism and overall processor performance. Despite decades of advancements in hardware-based dynamic prediction mechanisms [15, 23], certain categories of branches like those that are infrequently executed or highly biased continue to present difficulties. These branches often arise in rare execution paths such as error handling, boundary checks, or infrequent conditions, yet their misprediction can impose disproportionate penalties on performance.

To mitigate prediction failures, compilers can emit static branch hints [14], which serve as lightweight guidance for the processor. However, these hints are typically derived from heuristics or limited profiling and are often insufficient for accurate prediction. When uncertainty remains, responsibility is deferred to hardware predictors, which rely on runtime behavior to adapt. These predictors, while powerful, require sufficient historical execution to make accurate inferences, rendering them less effective for cold paths or rare events [22].

An alternative approach leverages program-based prediction: the use of static code features to predict branch behavior without executing the program. This technique offers distinct advantages in terms of input-independence, scalability, and integration with compilation pipelines. The central hypothesis of this work is that certain branch behaviors, particularly those that are highly biased, can be effectively inferred through static analysis and machine learning models trained on program features.

This thesis aims to evaluate the effectiveness of this static, program-based approach, particularly in predicting whether a branch is likely to be taken. In doing so, it addresses several fundamental research questions regarding the feasibility, limitations, and comparative advantages of static branch prediction models.

1.2 Contributions

This work makes the following primary contributions:

1. **Empirical Evaluation of Bias-Aware Static Prediction.** We assess whether highly biased branches (e.g., taken or not-taken more than 99% of the time) can be reliably predicted using only static features extracted from binary code. The goal is to determine the extent to which such branches exhibit predictable patterns that can be captured without runtime information.

2. **Quantitative Characterization of Branch Bias.** We perform a systematic analysis of branch bias prevalence across a range of SPEC CPU2017 benchmarks [25], measuring the distribution and density of biased branches to establish whether they represent a meaningful optimization target.

3. **Feature-Source Analysis and Interpretability.** Using feature selection techniques (e.g., sequential forward selection and feature importance from tree-based models), we identify which program features (e.g., opcode, operand type, control flow position) are most predictive of branch bias. We explore potential reasons for their predictive power and examine the semantic role of influential features.

4. **Model Evaluation under Unified Framework.** We implement and evaluate multiple machine learning models—including Convolutional Neural Networks (CNNs), Multi-Layer Perceptrons (MLPs), and Gradient Boosting Machines (GBMs)—under a consistent training and evaluation framework. This comparison allows for fair assessment of model performance in terms of accuracy, recall, precision, and execution-weighted F1-score.

5. **Justification for Bias-Targeted Prediction.** We argue that static prediction is particularly suitable for highly biased branches, as hardware predictors require repeated executions to learn such patterns—a luxury not afforded for rare or cold branches.Rarely executed branches are common in many benchmarks. Our analysis supports the claim that static models can provide value where dynamic predictors are weakest [5].

6. **Comparative Study Against Generalized Models.** We compare the performance of bias-specialized models against generalized machine learning models trained to predict all branches. Our results demonstrate that targeting only biased branches leads to higher accuracy and better robustness, highlighting the benefit of specialization in branch prediction.

7. **Feature Ranking and Model Interpretability.** We rank the most significant features contributing to branch predictability and provide insights into their relative importance. This serves both to validate the model and to inform future feature engineering for compiler-assisted prediction systems.

8. **Cross-Benchmark vs. Per-Benchmark Evaluation.** We investigate whether models trained on data from multiple benchmarks generalize well to unseen programs or whether benchmark-specific models yield superior results. This analysis informs the scalability and generality of the proposed approach.

Chapter 2

Background

2.1 Modern CPUs and Branch Prediction

Central Processing Units (CPUs) serve as the core component of computer systems, responsible for executing programs and managing computational tasks [13]. As computing demands have grown more complex, CPU architectures have significantly evolved to handle sophisticated software and intricate processing requirements efficiently [21]. Among the key features of modern CPU design is the use of pipeline architecture, which improves instruction throughput by overlapping execution stages [23].

A pipeline allows multiple instructions to be processed simultaneously at different stages, similar to an assembly line [24]. However, this parallelism introduces new challenges, particularly in the presence of conditional branches. Since the outcome of a branch is not known until late in the pipeline, fetching the correct next instruction in advance becomes speculative. To mitigate the potential performance penalties of mispredicted branches, modern CPUs employ branch prediction mechanisms.

The basic stages of a modern CPU pipeline include:

1. **Fetch**: The CPU retrieves an instruction from memory.
2. **Decode**: The instruction is analyzed and interpreted.
3. **Execute**: The CPU performs the operation described by the instruction.
4. **Memory Access**: Data is accessed from memory if required by the instruction.
5. **Write Back**: The results of execution are stored back into memory or registers.

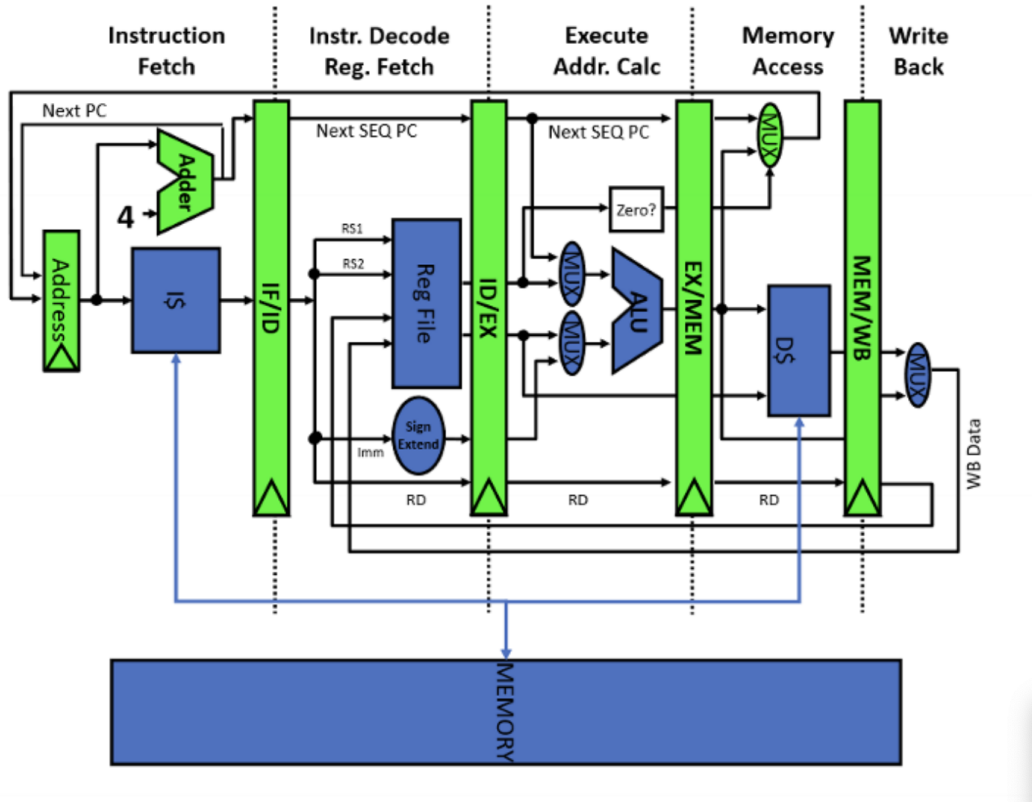


Figure 2.1: Five-stage RISC pipeline showing instruction flow from fetch to write-back. Branch instructions are resolved during Execute, making early prediction essential.

The Fetch stage is particularly critical when considering branch prediction. It initiates the instruction cycle and interacts with the memory hierarchy [21]. If a conditional branch is encountered and its outcome is unknown, the processor must speculate on the target address to continue fetching instructions without stalling the pipeline. Accurate prediction reduces control hazards, improves instruction throughput, and sustains high performance [18].

Misfetches caused by incorrect predictions lead to instruction squashing and pipeline flushing, which are costly in terms of performance. As a result, both static and dynamic prediction strategies are utilized to forecast branch directions. Static predictors use compile-time heuristics or code structure, while dynamic predictors rely on runtime history and hardware state.

Another key stage in the pipeline is Memory Access, which may be affected by prior control decisions [24]. For example, in speculative execution paths, incorrect branch predictions can cause unnecessary memory access operations, compounding performance inefficiencies. Understanding how branches interact with instruction fetch and memory access is essential for evaluating and optimizing branch prediction strategies.

Memory interactions can also be categorized into types such as Load, Read For Own-

ership (RFO), Prefetch, and Writeback. Each of these may be affected by branch prediction, especially in speculative execution contexts:

- **Load**: Retrieves data for computation, which may be on a speculative path.
- **Read For Ownership (RFO)**: Acquires exclusive access for modification, where mispredictions may introduce false sharing [2].
- **Prefetch**: Anticipates future data use—prediction accuracy can significantly influence its effectiveness [18].
- **Writeback**: Commits results back to memory, which must be managed carefully in speculative paths [24].

In summary, pipeline performance and memory hierarchy interactions are intricately tied to the behavior of branches. Understanding these components provides the necessary context for exploring and evaluating branch prediction techniques, which aim to maximize instruction-level parallelism and minimize execution stalls in modern processors [2, 23].

2.2 Program-Based Branch Prediction

Program-based branch prediction refers to the technique of estimating the outcome of a conditional branch specifically, whether it will be taken or not—taken based solely on static properties of the program's code. Unlike dynamic branch prediction, which relies on runtime execution history and hardware structures such as branch history tables or pattern predictors, program-based prediction operates without executing the program. It instead utilizes information extracted from the program's binary or intermediate representation [3, 7, 8].

The goal of program-based prediction is to infer branch behavior using features such as instruction opcodes, operand types, surrounding control-flow structure, and immediate values or offsets. These features can be derived during compilation or static binary analysis and serve as input to machine learning models trained to classify branches.

This approach is particularly useful for:

- **Profile-free environments:** where runtime data collection is impractical or costly.
- **Cold or rarely executed code:** which lacks sufficient execution history for accurate dynamic prediction [12].
- **Compiler optimizations:** where early prediction of branch direction can inform instruction scheduling, code layout, or speculative transformations [7, 20].

Program-based prediction is often implemented as a supervised learning task: a labeled dataset of branches (with known taken/not-taken outcomes) is used to train a predictive model. During inference, this model can then classify new, unseen branches based on their static features alone.

In this thesis, program-based prediction is studied in the context of highly biased branches (branches that are overwhelmingly taken or not taken). These branches are good candidates for static prediction, as their behavior tends to be governed by structural program logic rather than dynamic inputs.

2.2.1 Program-Based Prediction Features

These features are extracted statically from code, without requiring program execution:

Branch Opcode The specific instruction opcode (e.g., JNE, JZ, JL) of the conditional branch. It indicates the type of comparison or jump condition being evaluated.

Direction Whether the target of the branch is forward or backward in the instruction stream. Backward branches are often associated with loops, which are typically taken frequently.

Operand Opcode / Function / Type Metadata for the operand being compared in the branch:

Opcode: Instruction used to compute the operand (e.g., SUB, CMP)

Function: Abstract semantic category, e.g., arithmetic, logical, comparison

Type: The data type or representation, such as integer, float, or pointer

RA and RB Instruction Metadata (Opcode, Function, Type) Static tracking of the instruction(s) that produced the source operands:

RA/RB Opcode: The last instruction that wrote to the RA/RB operand

RA/RB Function: Semantic role of that instruction (e.g., load, compare, shift)

RA/RB Type: Operand type classification (e.g., integer, floating-point)

These features capture the computation context of the branch decision inputs [8].

Flag Setter Identifies whether a prior instruction explicitly sets condition flags used by the branch (e.g., via CMP, TEST). The presence and type of flag-setting instruction can influence predictability.

Jump Span The static distance (in bytes or instructions) between the branch and its target. Longer distances may imply cross-function jumps, while short distances are typical of loops or short conditionals.

Static Before / After The number of static instructions before and after the branch within its containing function. These can reflect control flow density and local complexity.

Opcode Sequence or Frequency Histogram A sequence or histogram of opcode types in a fixed window around the branch (e.g.,±5 instructions). This captures local

instruction patterns that may correlate with branch semantics [8].

Instruction Distance (Offset) An absolute offset (often encoded as an immediate in the instruction) that points to the branch target. Useful as a spatial feature, sometimes correlated with branch role (e.g., loops vs. conditionals) [12].

Control Flow Context Encodes the shape or topology of the local control flow graph, such as fan-in, fan-out, loop headers, or merge points. This can help differentiate structural branches from incidental ones [7].

Instruction Density Measures how many static instructions exist around the branch. Denser regions may correspond to hot paths or performance-critical code blocks.

2.3 Profile-Based Branch Prediction

Profile-based branch prediction leverages dynamic information collected during program execution to predict the direction of conditional branches. This approach relies on profiling runs (executions of the program with representative inputs) to gather statistical data about each branch's behavior, such as how frequently it is taken or not taken [5, 9].

In contrast to program-based prediction, which uses only static code features, profile-based prediction incorporates actual runtime behavior to improve accuracy. Branch direction frequencies obtained from profiling can then be embedded into the program binary as metadata or used during compilation to influence optimizations. For instance, compilers can use profile information to guide branch ordering, layout decisions, or to emit static hints (e.g., using the `likely`/`unlikely` annotations in GCC or LLVM) [13, 20].

Profile-based prediction is commonly used in:

- **Profile-Guided Optimization (PGO):** where branch behavior statistics help the compiler optimize code layout and reduce mispredictions [9].
- **Software-based prediction models:** that use profile data as training labels for machine learning [8].
- **Compiler-inserted static hints:** where frequently taken branches are annotated to help the processor make better static predictions [14].

However, profile-based prediction has limitations. It requires representative inputs to be effective, if the input distribution shifts at deployment time, the predictions may become inaccurate. Additionally, collecting profile data introduces overhead and complexity, making it less suitable for fast or lightweight deployment scenarios.

This thesis focuses on program-based prediction to avoid the reliance on profile data and explore whether static features alone can offer accurate prediction for specific branch classes, particularly those that are highly biased.

2.3.1 Profile-Based Prediction Features

These features are computed or observed at runtime during profiling runs:

Taken / Not Taken Count Counts of how many times the branch was taken or not. These are the primary indicators of branch bias used in profile-guided optimization [9].

Execution Frequency The total number of times the branch was executed. Branches with low frequency may not be well-learned by dynamic predictors.

Confidence Estimator A measure of how consistent the prediction outcome is. High-confidence predictions may be used to bypass further prediction logic [5].

Speculative Update Behavior Tracks how often speculative predictions are later corrected. Frequent rollbacks may indicate a mispredict-prone branch.

Stall Cycles on Mispredict Measures how many cycles the processor stalls when a misprediction occurs at the branch. This reflects the performance cost of inaccurate predictions [13].

Reorder Buffer Pressure The degree to which mispredictions congest the instruction pipeline's reorder buffer, preventing forward progress [13].

Cache Miss Association Observes whether cache misses tend to occur after certain branches. Some predictors associate branch behavior with memory access patterns [5].

Global History Register (GHR) A bit-vector encoding recent global branch outcomes. Used to capture long-range inter-branch dependencies [30].

Local History Table (LHT) Stores local histories of each individual branch, capturing consistent behavior across multiple executions of the same branch [30].

Pattern History Table (PHT) A table indexed by hashed history patterns; each entry holds a saturating counter that biases prediction. Common in two-level predictors [30].

Path History Sequence of recently taken basic blocks or function calls. Used to correlate long execution paths with branch behavior.

Branch Target Buffer (BTB) Caches recently taken indirect branches and their targets. Used to accelerate prediction for indirect or computed jumps [13].

Return Address Stack (RAS) Tracks return addresses for call-return pairs, enabling prediction of return instructions.

2.4 Hardware-Level Branch Prediction: Dynamic and Static Hint Bit Approaches

Branch prediction is a critical technique for improving instruction-level parallelism in modern pipelined processors. Since control-flow instructions such as conditional branches can alter the program counter (PC) in non-sequential ways, processors must predict the outcome of branches early in the pipeline to avoid costly stalls or wasted work from

mispredicted instructions. This section reviews hardware-level branch prediction mechanisms, with a focus on both traditional dynamic prediction and static program-based prediction using hint bits, as represented in Figure 2.3.

2.4.1 Traditional Dynamic Branch Prediction

In conventional designs, branch prediction is performed by a dedicated hardware unit known as the *Branch Prediction Unit* (BPU). The BPU typically consists of structures such as the Branch History Table (BHT), Pattern History Table (PHT), Branch Target Buffer (BTB), and Return Address Stack (RAS), which collectively track the outcomes and targets of previously encountered branches [13, 23, 30].

When a branch instruction is fetched from the instruction cache, the BPU consults these hardware tables to predict the next PC:

- **Local predictors** use the outcome history of individual branches.
- **Global predictors** use the outcomes of recent branches to form a global history.
- **Hybrid predictors** combine multiple prediction schemes for higher accuracy.

If the BPU predicts that a branch will be taken, the pipeline fetches instructions from the predicted target. If the prediction proves incorrect during execution, a pipeline flush is required, incurring a performance penalty.

2.4.2 Dynamic Branch Prediction without Hint Bits

Figure 2.2 illustrates a pipeline that relies solely on dynamic branch prediction, without support for statically embedded hint bits. In this architecture, all branch predictions are made by the hardware Branch Prediction Unit (BPU), which maintains internal structures such as local/global history tables, pattern history tables, or other predictive mechanisms [13, 23].

When the processor fetches instructions from the instruction memory, the BPU receives the current program counter (PC) and uses it to index its prediction tables. Based on accumulated runtime history and the outcomes of previous branches, the BPU predicts the next PC for the instruction fetch unit, allowing the pipeline to speculatively continue execution without stalling.

In this setup, the processor does not utilize any statically computed prediction information from the binary. All prediction logic is adaptive and updated at runtime, enabling the BPU to learn program behavior over repeated executions. However, this also means that cold-start branches (those not previously encountered) and branches with irregular

or input-dependent patterns may incur more mispredictions, as the BPU requires several executions to build up an accurate history.

This approach is robust for most general-purpose code, but can struggle with cold or highly biased branches where dynamic adaptation is slow or unnecessary. The absence of hint bits in the pipeline means that all prediction improvements must come from hardware structures and runtime feedback, with no support for compiler- or profile-inserted guidance.

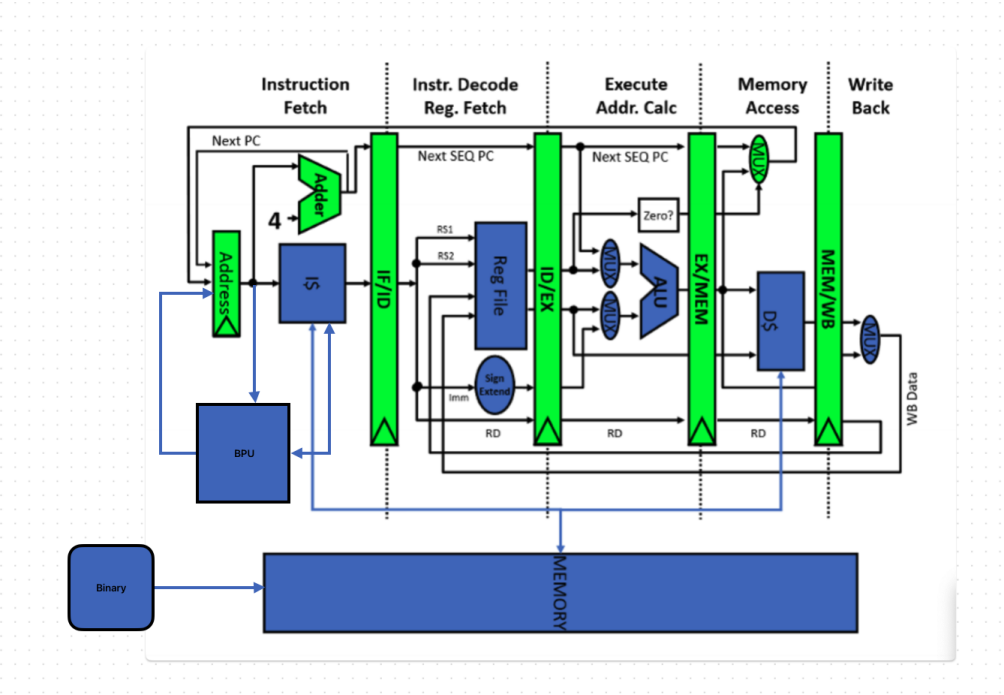


Figure 2.2: Pipeline architecture relying solely on dynamic hardware branch prediction (BPU) without static hint bits.

2.4.3 Static Program-Based Prediction with Hint Bits

An alternative to purely dynamic prediction is to embed statically computed predictions directly into the program binary using *hint bits*. This static approach leverages compile-time analysis, profiling, or machine learning techniques to determine the likely outcome of each branch before execution [7, 8, 20]. The prediction is encoded as an additional bit (or bits) associated with each branch instruction.

Hardware support for hint bits typically requires minimal modification to the processor pipeline:

- When a branch instruction is fetched, the hint bit is read alongside the instruction from the instruction cache.

- A *multiplexer* (MUX) at the input to the PC selection logic allows the processor to choose between the static prediction (hint bit) and the prediction from the dynamic BPU.
- A control mechanism (e.g., set by the processor mode, or by software) determines which source is used for branch prediction on a per-branch or per-program basis.

This structure is shown in Figure 2.3, where the MUX selects between the static hint path and the traditional dynamic prediction path.

2.4.4 Comparison: With and Without Hint Bits

Without hint bits, the pipeline relies exclusively on dynamic predictors in the BPU, which adapt to program behavior at runtime but may require several mispredictions to "learn" new patterns or handle cold-start branches.

With hint bits, the processor can exploit prior static analysis or profiling to guide branch prediction from the very first execution. This is particularly effective for highly biased branches, where behavior is consistent and predictable across runs.

A system may be designed to:

- Use hint bits for branches statically determined to be highly biased.
- Fall back to dynamic prediction for branches with uncertain or input-dependent outcomes.
- Combine both strategies using a MUX, as in Figure 2.3, for maximum flexibility and accuracy.

2.4.5 Illustrative Pipeline with Hint Bits

Figure 2.3 illustrates a pipeline that integrates both dynamic branch prediction and static hint bits to improve prediction accuracy and flexibility. At the instruction fetch stage, a multiplexer (MUX) selects between two sources of prediction: the static hint bit embedded within the instruction (typically read from the instruction cache or memory) and the output of the dynamic branch prediction unit (BPU). This hybrid mechanism supports four possible prediction scenarios (00, 01, 10, 11), depending on the combination of outputs from the static hint bit and the dynamic BPU. When the static prediction mode is enabled, the branch outcome is determined directly from the static hint bit, offering fast, low-latency prediction with minimal overhead—ideal for highly predictable branches. If static prediction is not used, the pipeline defers to the dynamic BPU, which adapts over

time based on execution history. By incorporating both mechanisms, the processor architecture achieves a balance between speed and adaptivity, leveraging static hints for simple control flows while still accommodating complex or data-dependent branches through dynamic learning.

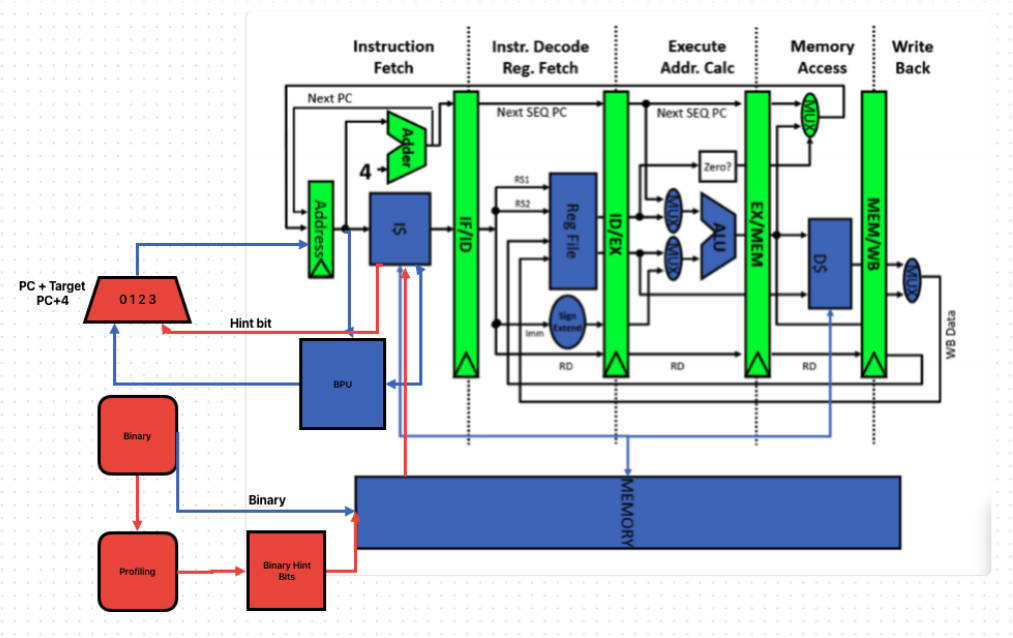


Figure 2.3: Pipeline architecture supporting both dynamic and static (hint bit) branch prediction using a multiplexer to select the prediction source.

2.4.6 Discussion

Hardware support for static branch prediction using hint bits provides a practical mechanism for leveraging offline program analysis and machine learning within traditional processor pipelines. By augmenting or bypassing the BPU with statically computed predictions, processors can reduce misprediction rates for highly biased or easily analyzed branches, reduce pipeline stalls, and improve overall instruction throughput.

This approach is particularly relevant in environments where branch behavior is stable across executions, or where profile-guided compilation can accurately identify bias at compile time. The combined use of dynamic and static predictors, facilitated by a simple MUX structure, offers the best of both worlds: rapid adaptation to runtime conditions and robust prediction for statically predictable branches.

Chapter 3

A Hardware Static Program-Based Mechanism for Predicting Highly Biased Branches

3.1 Overview

Program-based prediction aims to anticipate the behavior of branches solely through static analysis, without executing the code. This methodology is particularly focused on highly biased branches—those that are almost always taken or not taken. In this framework, machine learning models are trained offline on statically extracted features to predict whether a given conditional branch is likely to be taken.

The long-term goal of this mechanism is to enable its integration into future hardware systems. Such integration could take the form of compiler-inserted hint bits, static prediction units alongside traditional branch predictors, or hybrid designs that merge static and dynamic prediction capabilities, particularly in scenarios where runtime data is insufficient or unavailable.

3.2 Proposed Mechanism and Deployment Idea

The central idea of this mechanism is to leverage statically trained models to perform profile-based prediction for branches that exhibit extreme bias. This prediction mechanism is envisioned as a lightweight hardware-level enhancement designed to operate without requiring execution history.

In the proposed hardware implementation, the code cache loader (or an auxiliary analysis unit) is augmented with a prediction engine that processes code as it is loaded into the instruction cache. Upon detecting conditional branches in the newly loaded cache block,

the prediction unit will extract relevant static features from the surrounding instructions and classify the branches using a pre-trained model. This process can be conducted in one of two modes:

- **Offline Mode:** A predictive model is trained in advance on a large set of branches across benchmarks, then embedded in hardware as a fixed classifier. Upon loading code into the cache, the classifier predicts branch directions based on extracted static features.
- **Online Mode:** A simplified predictor is trained or updated incrementally using runtime feedback, allowing adaptation to application-specific patterns. In this case, the hardware maintains and evolves a local model.

This static prediction engine is invoked at cache load time and is optimized for low-latency operation. Because it targets only highly biased branches, which tend to exhibit predictable structural patterns, the classifier complexity can remain low while maintaining accuracy.

The predicted branch direction (taken or not taken) can then be encoded as a branch hint bit or passed to the existing branch prediction unit to influence initial prediction decisions. This process is especially useful in cold-start conditions where dynamic predictors have no history available.

3.3 Feature Set for Static Prediction

To enable prediction in the absence of profiling data, a comprehensive set of features is extracted for each conditional branch instruction. These features are derived from the instruction itself, its operands, and the surrounding static context.

Instruction-level characteristics include the `Branch Opcode`, which identifies the specific type of branch instruction, and the `Direction`, denoting whether the target is forward or backward relative to the instruction's location. Operand-specific features such as `Operand Opcode`, `Operand Function`, and `Operand Type` provide metadata on the computation being performed and how operands are compared.

Further context is captured by tracing the origins of the branch's operands, specifically the `RA` and `RB` sources, capturing their opcode, function, and operand types. The presence of a `Flag Setter`—an instruction that affects condition flags used by the branch—is another key indicator of behavior.

The `Jump Span` feature encodes the static distance to the branch target, while `Static Before` and `Static After` provide a view into local instruction density within the function. Metadata such as `Function`, `Language`, and `Branch Address` are included for

traceability and indexing. Finally, `Taken Count` and `Not Taken Count` are used strictly for label construction and evaluation, and are excluded from the training features.

3.4 Design Rationale

The use of statically derived features allows prediction to be performed early in the compilation or binary translation process. This is particularly advantageous in several practical settings.

First, cold execution paths—such as those associated with error handling or rare condition checks—often do not execute frequently enough for dynamic predictors to learn reliable patterns. Static models, by contrast, can offer upfront predictions regardless of runtime frequency.

Second, in resource-constrained systems such as embedded environments, storage and power limitations may preclude the use of sophisticated dynamic predictors or profiling mechanisms. Here, statically embedded predictions can serve as an efficient alternative.

Third, in ahead-of-time compilation pipelines or binary translation workflows (e.g., just-in-time compilers, static binary rewriters), having a reliable prediction mechanism that does not rely on profiling simplifies the integration of branch prediction metadata. The features used in this work were selected based on prior literature, mutual information scores, and empirical forward feature selection to maximize predictive value while maintaining interpretability.

3.5 Machine Learning Framework

To realize this prediction mechanism, a machine learning framework was implemented for training classifiers based on the extracted features and corresponding branch outcome labels.

Training is conducted offline using a dataset composed of statically extracted feature vectors and corresponding labels, which indicate whether a branch was typically taken or not. In particular, a subset of branches labeled as "highly biased"—e.g., exhibiting a taken or not-taken ratio above 99%—are used to explore whether this type of behavior can be statically identified with high precision.

Several model architectures were tested under this framework, including Convolutional Neural Networks (CNNs), which process encoded feature sequences via 1D convolutions, Multi-Layer Perceptrons (MLPs) for dense feature modeling, Logistic Regression as a linear baseline, and Gradient Boosting Machines (GBMs), which combine decision trees in an additive fashion.

Each model was evaluated under both single-benchmark and cross-benchmark conditions. In the former, training and testing were conducted on disjoint subsets of the same benchmark; in the latter, models were trained on multiple benchmarks and evaluated on unseen ones to assess generalizability.

3.6 Static Program-Based Branch Prediction Unit (SPBBPU) Integration

Figure 3.1 illustrates the integration of the proposed Static Program-Based Branch Prediction Unit (SPBBPU) into the existing processor pipeline. All standard pipeline and dynamic branch prediction logic remain unchanged; the SPBBPU is introduced as a dedicated module for high-bias branches. Now the MUX instead of the previous inputs of the 4 cases (00, 01, 10, 11) also takes into consideration if the branch is highly bias because in that case it takes the prediction of SPBBPU.

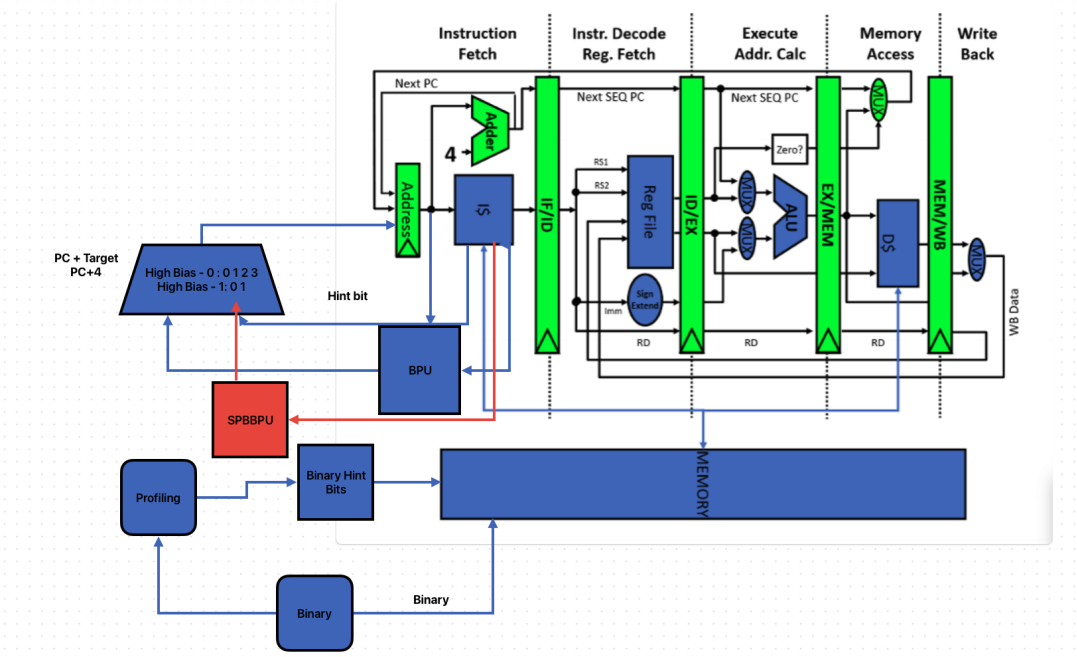


Figure 3.1: Hardware integration of the SPBBPU (red), which statically predicts high-bias branches using extracted features. The conventional dynamic BPU remains unchanged for all other branches.

Operational Flow:

- When instructions are fetched into the instruction cache, the SPBBPU monitors the instruction stream in parallel with the existing pipeline.

- For each conditional branch, the SPBBPU checks whether it qualifies as a *high-bias branch*. This determination can be made by inspecting hint bits, static features, or by predicting at run time.
- If a branch is identified as high-bias, the SPBBPU extracts relevant static features (e.g., opcode, operand types, control-flow context) and uses its internal, lightweight hardware-implemented model (e.g., a small neural network or rule-based logic) to predict the branch direction.
- The SPBBPU outputs its prediction to the pipeline control logic. For these branches, the dynamic branch predictor is either bypassed or taken into consideration.
- For all other branches, prediction proceeds as usual through the dynamic BPU, without intervention from the SPBBPU.

Key Advantages:

- **Minimal Architectural Intrusion:** Only the SPBBPU is added. All other processor components, including the dynamic branch predictor, instruction fetch, and decode logic, remain unmodified.
- **Selective Application:** The SPBBPU is only active for statically identified high-bias branches, ensuring that its impact is focused where static prediction is most effective.
- **No Hybrid Prediction:** There is a strict separation of responsibility. The SPBBPU exclusively predicts high-bias branches; the dynamic BPU predicts all others.
- **Hardware Simplicity:** The SPBBPU can be implemented as a lightweight hardware block, invoked only when relevant, with low overhead.

This approach ensures seamless integration with existing architectures while providing the accuracy and cold-start benefits of static prediction for highly predictable (high-bias) branches.

3.7 Example: High-Bias Branch in `perlbench_r`

To demonstrate the concept of a highly biased branch, consider a representative case from the `perlbench_r` benchmark, which is based on the `perl5` interpreter. The following pseudocode illustrates a branch that checks for a specific token pattern during Perl source parsing:

```
function check_comma(s: string) -> void:
    assert s is not null

    if s[0] == '␣' and s[1] == '(':
        if syntax_warning_enabled():
            w = s[2:]
            level = 1
            while w is not empty and level > 0:
                if w[0] == '(': level += 1
                elif w[0] == ')': level -= 1
                w = w[1:]

            while w[0] is space:
                w = w[1:]

            if w[0] in END_TOKENS:
                issue_warning("func-like␣construct␣detected")
```

Listing 3.1: Pseudocode of high-bias branch

Why Is This Branch Highly Biased? The condition:

```
if s[0] == '␣' and s[1] == '(':
```

is only true in rare but syntactically regular situations. Since it occurs in a parsing function, the input triggering this branch is both uncommon and consistent across runs. As a result, this branch is taken in approximately 97.35% of executions in a predictable pattern, qualifying it as highly biased.

Predictability via Static Analysis This branch relies on constant comparisons on known characters, making it a prime target for static prediction. Its opcode sequence, operand types, and contextual position within the function yield strong signals to a machine learning model trained purely on program structure. Since the outcome is deterministic and minimally affected by dynamic inputs, the static predictor can infer the direction with high confidence.

Reference Original code from `perl5/toke.c #L12019`.

Chapter 4

Evaluation of Frameworks and Experimental methodology

4.1 SPEC CPU2017 Benchmark Suite

The SPEC CPU2017 benchmark suite [25] is a standardized collection of compute-intensive workloads developed by the Standard Performance Evaluation Corporation (SPEC). It is widely used for evaluating and comparing the performance of modern CPUs and system architectures. The suite includes a diverse set of real-world applications written in C, C++, and Fortran, covering domains such as physics simulations, artificial intelligence, compiler design, and financial modeling.

The benchmarks are divided into two primary categories: `speed` and `rate`. The `speed` benchmarks measure the performance of a system running a single copy of the workload, while the `rate` benchmarks assess throughput by executing multiple instances in parallel. For this thesis, we utilize the `speed` versions of selected workloads from the SPEC CPU2017 suite to extract control flow and branch behavior features, facilitating an in-depth study of branch prediction characteristics across a broad spectrum of program behaviors.

SPEC CPU2017 is particularly suitable for this work because it reflects the complexity and variability found in real-world applications, ensuring that the evaluation of branch prediction models is both rigorous and generalizable.

Table 4.1: Branch Frequencies and Pin Overhead per Benchmark

Benchmark	Languages	Description	Taken	Not Taken	Total	Taken %	Not Taken %	Time w/ Pin (s)	Time w/o Pin (s)
507.cactuBSSN_r	C, Fortran	Numerical relativity (Einstein equations)	6,124,534,232	8,449,063,979	14,573,598,211	42.02%	57.98%	862	120
531.deepsjeng_r	C++	AI-based chess engine	78,676,666,947	86,856,574,217	165,533,241,164	47.53%	52.47%	9,157	228
519.lbm_r	C	Lattice-Boltzmann method for fluid dynamics	77,223,661	125,840,417	203,064,078	38.03%	61.97%	408	123
557.xz_r	C	Compression utility (XZ / LZMA)	19,303,804,391	13,672,973,189	32,976,777,580	58.54%	41.46%	1,516	98
508.namd_r	C++	Molecular dynamics	28,695,435,918	9,429,271,454	38,124,707,372	75.27%	24.73%	1,279	149
520.omnetpp_r	C++	Discrete event simulation (OMNeT++)	43,390,497,850	60,257,723,316	103,648,221,166	41.86%	58.14%	6,158	299
538.imagick_r	C	Based on ImageMagick (image processing)	55,842,991,084	3,515,721,023	59,358,712,107	94.08%	5.92%	12,114	881
510.parest_r	C++	Finite element analysis	157,564,094,317	36,191,377,913	193,755,472,230	81.32%	18.68%	15,774	256
544.nab_r	C, NAB	Nucleic Acid Builder (AmberTools subset)	28,993,759,798	8,876,928,847	37,870,688,645	76.56%	23.44%	6,293	206
500.perlbench_r	C, Perl	Based on the Perl interpreter (in C)	55,539,389,721	39,091,891,037	94,631,280,758	58.69%	41.31%	7,515	85
502.gcc_r	C	Based on GCC compiler code	13,594,495,015	20,230,798,504	33,825,293,519	40.19%	59.81%	1,924	26
511.povray_r	C++	Ray tracing (POV-Ray engine)	89,389,027,725	42,818,555,477	132,207,583,202	67.61%	32.39%	13,891	228
541.leela_r	C++	AI Go engine (Leela Zero)	78,659,068,304	68,357,365,974	147,016,434,278	53.50%	46.50%	9,565	371
523.xalancbmk_r	C++	XSLT processing (Xalan-C++)	92,084,737,763	24,306,524,892	116,391,262,655	79.12%	20.88%	12,068	171
505.mcf_r	C	Network optimization problem	26,266,167,190	15,798,987,287	42,065,154,477	62.44%	37.56%	5,254	266
526.blender_r	C, C++, Python	Blender rendering (core in C/C++)	23,155,087,631	33,052,828,572	56,207,916,203	41.20%	58.80%	9,372	183

4.2 Experimental Setup

All experiments in this thesis were conducted on a local x86_64 machine running a Linux environment. The system configuration is as follows:

- **CPU:** 12th Gen Intel® Core™ i7-12700 with 20 logical cores (10 performance and 10 efficiency threads), supporting simultaneous multithreading (SMT) with 2 threads per core.
- **Memory:** 32 GB of RAM, of which 30 GiB were available for user-space execution; a swap partition of 15 GiB was also available but remained unused during the experiments.
- **Storage:** The system is equipped with a non-rotational NVMe SSD (indicated by `ROTA = 0`), which provides high throughput and low latency suitable for I/O-intensive workloads.
- **Cache Hierarchy:**
  - L1 Data Cache: 512 KiB (12 instances)
  - L1 Instruction Cache: 512 KiB (12 instances)
  - L2 Cache: 12 MiB (9 instances)
  - L3 Cache: 25 MiB (shared)
- **Operating System:** Linux kernel version 5.14.0-427.18.1.el9_4.x86_64.

This configuration ensures sufficient computational and memory resources for parallel model training and feature extraction from large binary datasets.

4.3 Pin Tool API

Intel's Pin is a dynamic binary instrumentation framework widely used in research and performance analysis for x86/x86_64 binaries. In this thesis, the Pin Tool API is used to implement a custom instrumentation pass for extracting runtime features associated with conditional branches. The goal is to collect execution-level metadata that complements static analysis and enables accurate training and evaluation of machine learning models for branch prediction.

4.3.1 Instrumentation Approach

The custom Pin tool was developed using the `INST_COND_BR` category in the Pin API to identify and analyze conditional branch instructions. The tool performs the following operations for each branch encountered:

- **Address Logging:** Captures the program counter (PC) of the branch instruction to establish a mapping between dynamic behavior and static features.
- **Execution Outcome:** Records whether the branch was taken or not during execution. This binary label is used as the prediction target.
- **Execution Count:** Maintains a count of how many times each branch was taken or not-taken, enabling downstream labeling of biased branches based on runtime frequency.
- **Operand Extraction:** Extracts source operands (when applicable) by intercepting the relevant registers or memory addresses used in the comparison before the conditional branch. This allows correlation with static operand-based features.
- **Contextual Metadata:** Optionally captures function name (if symbol information is present), call site context, and basic block boundaries for enhanced static-dynamic linkage.

4.3.2 Implementation Details

The instrumentation pass was implemented using the following Pin API routines:

- `INS_IsBranch()` and `INS_IsConditionalBranch()` to identify target instructions.
- `INS_InsertIfCall()` and `INS_InsertThenCall()` for efficient predicate evaluation.
- `INS_InsertCall()` to log outcomes and addresses during execution.
- `PIN_GetContextReg()` and `INS_RegR()` to access operand-level register values at runtime.

4.3.3 Output Format

The output of the Pin tool is written in CSV format, containing the following fields:

- `Branch Address`

- `Function Name`
- `Taken Count`
- `Not Taken Count`
- `Operand Registers / Immediate Values (if extracted)`

This dataset serves as the dynamic component in the feature pipeline and is later joined with statically extracted features to construct the full training set.

4.3.4 Use Case in the Thesis

The Pin Tool was primarily used to:

1. Identify biased branches (those that are taken or not-taken with $\geq 99\%$ probability).
2. Provide the ground truth labels for machine learning classifiers (e.g., taken vs. not-taken).
3. Analyze discrepancies between static predictions and actual runtime behavior.

The collected data enables both supervised training and execution-weighted evaluation, supporting the overarching hypothesis of this work that static features can meaningfully predict branch behavior.

4.4 Machine Learning Algorithms

To evaluate the feasibility and effectiveness of program-based branch prediction, a range of machine learning algorithms were implemented and tested under a unified evaluation framework. These models are trained using static features extracted from compiled binaries, and are tasked with predicting whether a branch instruction is likely to be taken or not.

4.4.1 Classification Objective

The primary task is a binary classification problem: determining whether a given conditional branch is more frequently taken or not taken based on its static characteristics. In some experiments, this task is refined to predicting *highly biased branches*—those taken (or not) at least 99% of the time. Models are trained using labels generated from dynamic profiling (via Pin Tool) and features derived from disassembly and control-flow analysis.

4.4.2 Models Evaluated

The following models were selected to represent a diverse set of learning paradigms:

Logistic Regression

Logistic Regression (LR) serves as a simple and interpretable baseline for branch outcome prediction. Despite its linear nature, LR can often capture trends in highly separable data, such as biased branches influenced by specific opcodes or operand types.

Gradient Boosting Machines

Gradient Boosting Machines (GBMs), including implementations like `XGBoost`, are ensemble methods that construct a sequence of weak learners (typically decision trees) to correct the residuals of previous learners. GBMs are effective for tabular data and provide natural support for feature importance analysis, which is leveraged in this thesis to interpret model behavior.

Multi-Layer Perceptrons (MLPs)

MLPs are fully connected feedforward neural networks capable of modeling non-linear relationships. In this work, MLPs are used to capture complex feature interactions that may not be easily linearly separable. A standard architecture with two hidden layers and ReLU activation is employed.

Convolutional Neural Networks (CNNs)

CNNs, although typically used in image and signal processing, are adapted here to operate on structured, fixed-length feature vectors. By treating features as one-dimensional sequences, CNNs can detect local patterns and invariant substructures in feature space. This architecture has shown superior performance in capturing neighborhood-level patterns among branch-related features.

4.4.3 Training and Evaluation Protocol

Each model is trained and evaluated using the same preprocessed datasets to ensure comparability. The datasets are split using an 80/20 train-test split unless otherwise stated. Key steps include:

- **Imputation:** Missing categorical values are imputed with a constant placeholder, and numeric values with the column mean.

- **Encoding:** Categorical features are ordinally encoded. Numeric features are standardized.
- **Evaluation Metrics:** Accuracy, Precision, Recall, and F1-Score are reported for each benchmark, along with execution-weighted summaries across all benchmarks.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.2}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.3}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.4}$$

Chapter 5

Evaluation

5.1 Prevalence of Highly Biased Branches

A central hypothesis of this thesis is that many conditional branches exhibit extreme bias—that is, they are taken almost always or almost never. Such biased branches are particularly promising candidates for static prediction, especially in scenarios where hardware predictors lack sufficient runtime history or are poorly optimized for cold paths.

To validate this assumption, we performed a comprehensive analysis of conditional branch instructions using a diverse set of SPEC CPU2017 benchmarks. While this benchmark suite may not feature a large number of rarely executed (the focus of this thesis), highly biased branches, its widespread use and accessibility make it a practical choice for this initial investigation. For each benchmark, we extracted every branch instance and calculated its bias ratio

5.1.1 Branch Bias Distribution and Dynamic Impact in `perlbench_r`

This section presents a detailed analysis of branch bias and execution characteristics for the `perlbench_r` benchmark, based on eight complementary visualizations. The goal is to illustrate the distribution of branch biases, their dynamic significance, and the motivation for a static high-bias prediction mechanism. All of the trends observed for this benchmark are seen in almost all of the benchmarks.

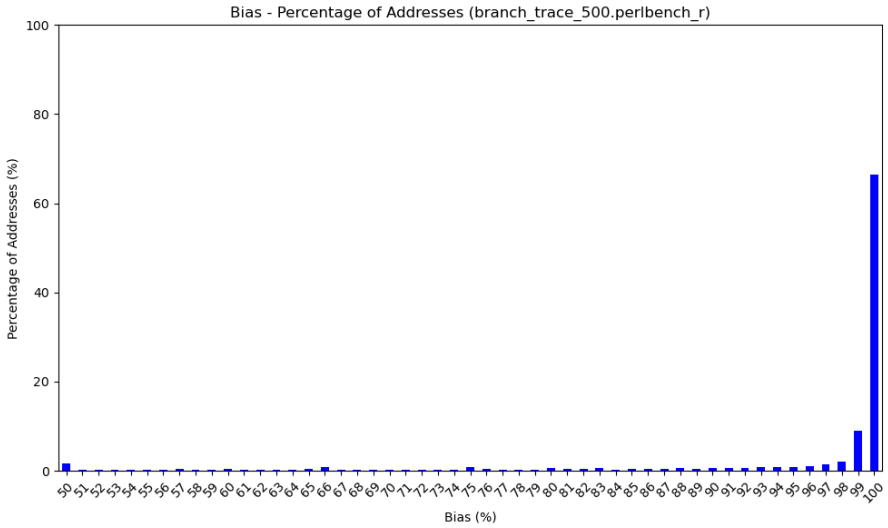


Figure 5.1: Percentage of addresses per bias percentage bin for `perlbench_r`.

1. Percentage of Addresses per Bias (Figure 5.1) This figure shows what percentage of unique branch instructions (addresses) fall into each bias bin. The vast majority of branches are clustered at very high bias values ($\sim$99–100%), indicating that most branches in this benchmark are overwhelmingly taken or not-taken. Only a small minority are distributed across moderate bias bins.

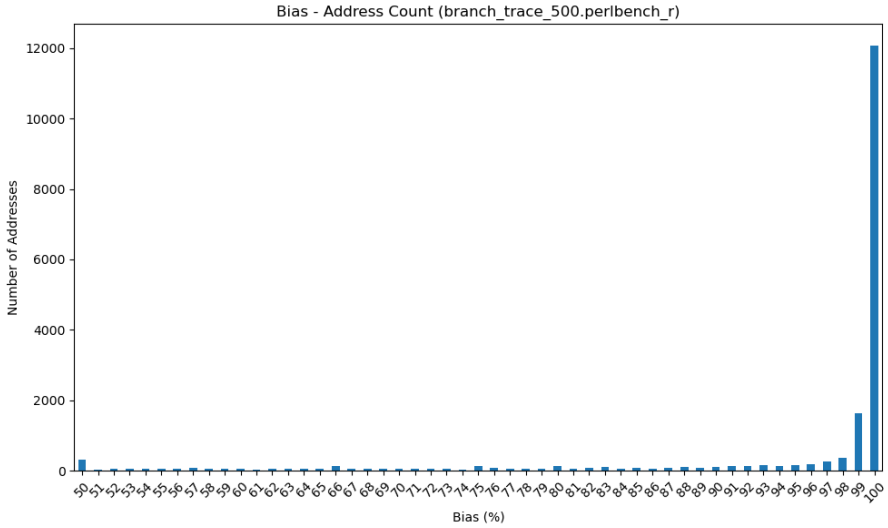


Figure 5.2: Number of addresses per bias bin for `perlbench_r`.

2. Address Count per Bias (Figure 5.2) This raw count reinforces the previous observation: thousands of static branch addresses are concentrated in the highest bias bins, while few exist elsewhere. This highlights that high-bias branches dominate the static control flow structure.

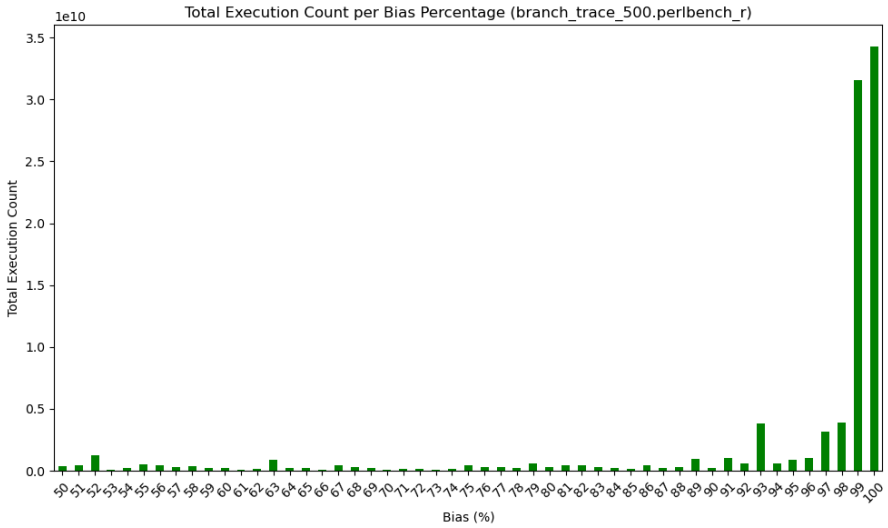


Figure 5.3: Total execution count per bias bin for `perlbench_r`.

3. Total Execution Count per Bias (Figure 5.3) Here, the y-axis measures total dynamic executions summed for all branches in each bias bin. Again, the highest bias bins not only contain the most branches but are also responsible for nearly all dynamic branch decisions (billions of executions).

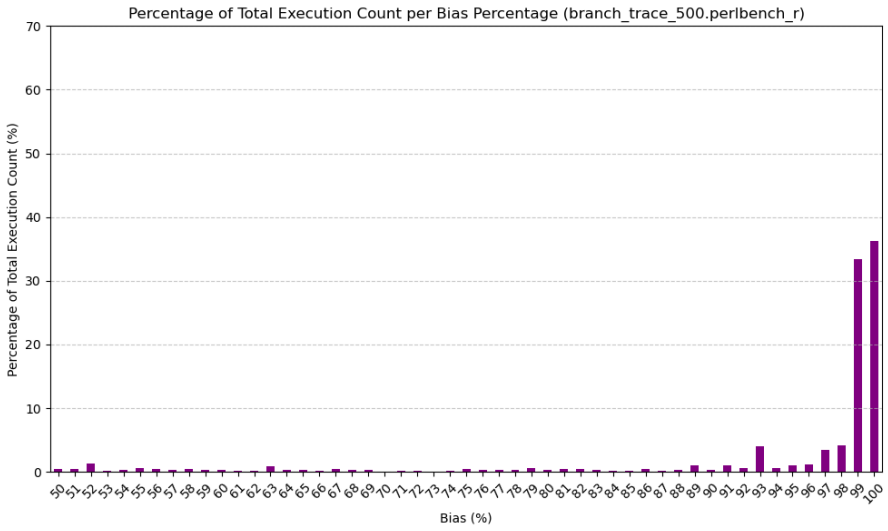


Figure 5.4: Percentage of total dynamic execution per bias bin for `perlbench_r`.

4. Percentage of Total Execution Count per Bias (Figure 5.4) This normalized view demonstrates that over 60% of all dynamic executions are attributable to the highest bias branches alone. The remaining bins collectively account for a much smaller share, underscoring the extreme dominance of high-bias branches in the program's dynamic control flow.

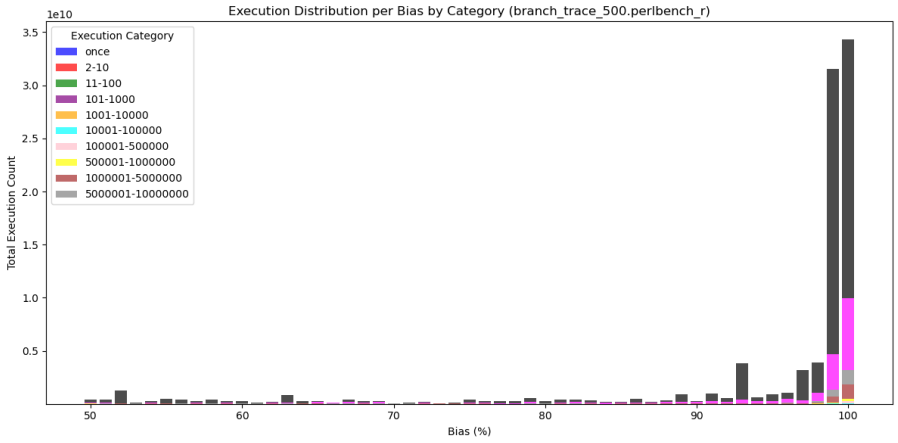


Figure 5.5: Total execution count per bias bin, subdivided by execution frequency category (`perlbench_r`).

5. Execution Distribution per Bias by Category (Figure 5.5) Each stacked bar is subdivided by how many times the branch executes (from "once" to ">5,000,000$"), showing that most dynamic activity in high-bias bins comes from branches executed millions of times. This confirms that a small set of very high-bias, high-frequency branches dominates runtime behavior .

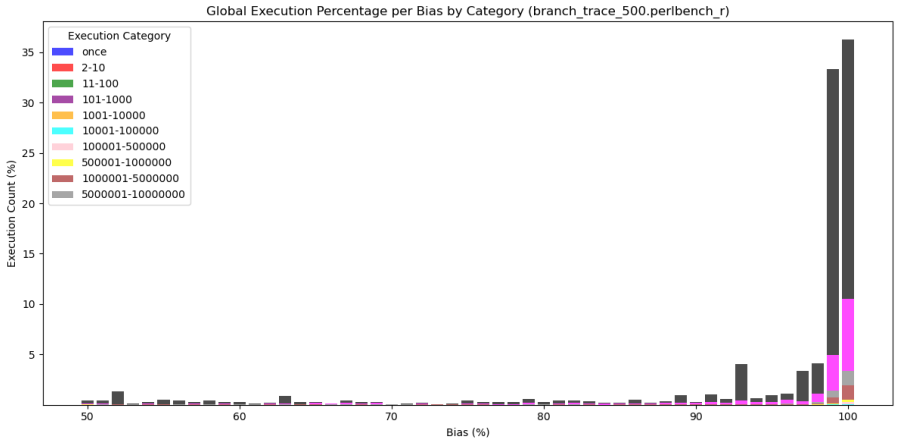


Figure 5.6: Percentage of global execution count per bias bin and execution category (`perlbench_r`).

6. Global Execution Percentage per Bias by Category (Figure 5.6) This graph further normalizes the data, confirming that the vast majority of all dynamic branch outcomes are the result of high-bias branches executed over a million times each . Lower-bias or infrequently-executed branches have minimal global effect.

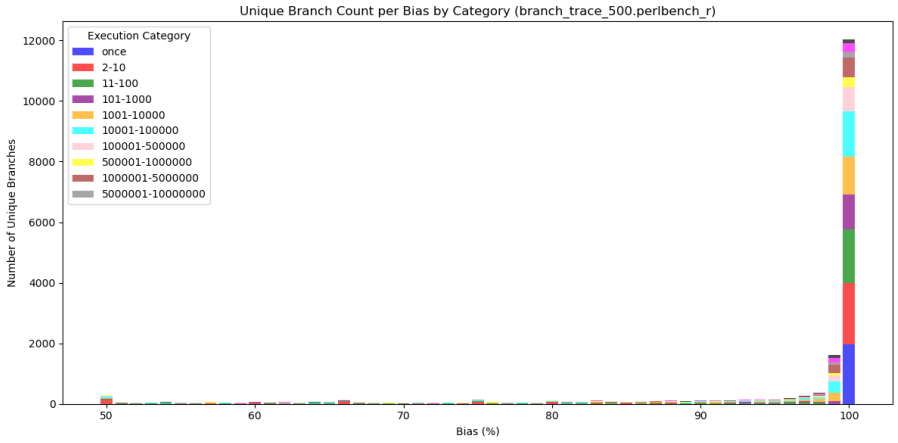


Figure 5.7: Number of unique static branches per bias bin and execution category (`perlbench_r`).

7. Unique Branch Count per Bias by Category (Figure 5.7) The final visualization reveals that, even among high-bias branches, there is a spectrum of execution frequencies: some are "cold" (executed once), while others are "hot" (executed millions of times). This justifies a prediction approach that identifies and targets only the highly biased, frequently executed branches for static prediction .

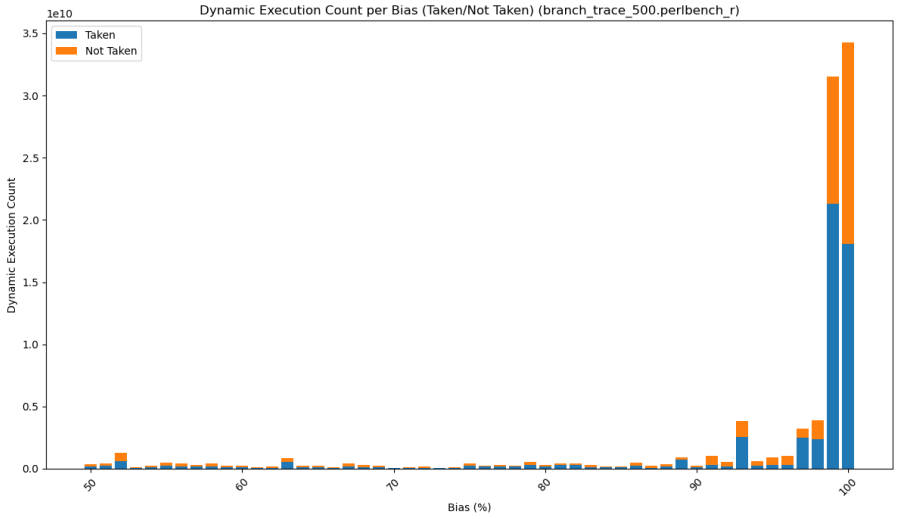


Figure 5.8: Dynamic execution count per bias bin, with separate bars for taken and not-taken outcomes in `perlbench_r`.

8. Dynamic Execution Count per Bias Bin, Split by Taken and Not Taken (Figure 5.8) This plot reveals that, for nearly all bias intervals, the majority of dynamic executions are concentrated in either the taken or not-taken class, especially for the extreme bias bins ($\sim$99–100% and $\sim$0–1%). This further illustrates the highly polarized nature of branch

behavior in this benchmark and supports the case for highly selective static prediction mechanisms.

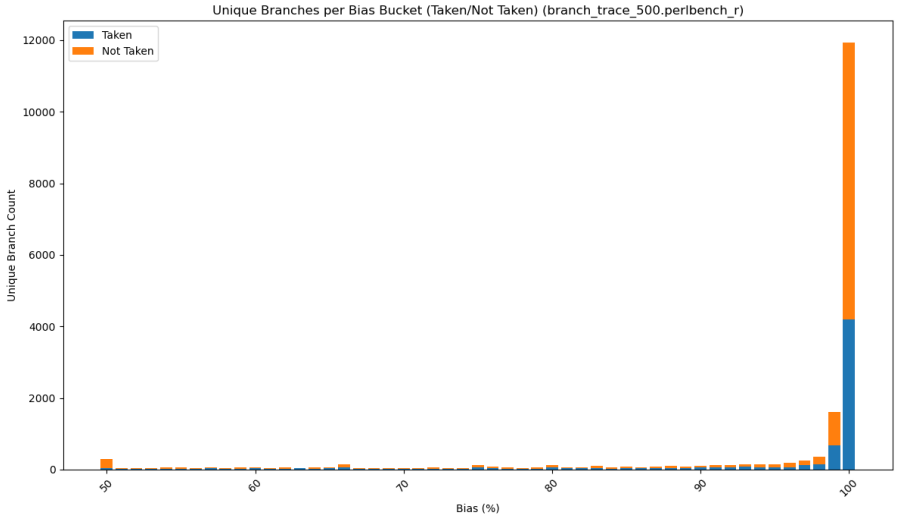


Figure 5.9: Unique branch count per bias bucket for `perlbench_r`, split by outcome (taken vs. not-taken) that there are more not taken than taken.

9. Unique Branches per Bias Bucket, Split by Taken/Not-Taken (Figure 5.9) This static analysis shows that most branch instructions are strongly biased in one direction. The last bin (99–100%) contains the majority of branches, most of which consistently yield the same outcome. This static polarity complements the dynamic execution trends seen earlier and further supports the case for selectively applying static prediction to high-bias branches.

Summary These graphs collectively show that the `perlbench_r` benchmark, like many other benchmarks, is dominated by a small set of highly biased, frequently executed branches. In this thesis we examine if static program-based prediction is particularly effective for such cases, providing motivation for the proposed hardware design focused on high-bias branches.

5.1.2 Bias Histogram per Benchmark

Figures 5.10 through 5.13 show per-benchmark histograms of branch bias, aggregated either by raw counts, execution-weighted counts, or normalized proportions. In all benchmarks, we observe that a significant proportion of branches fall into the extreme bins—e.g., $[0.0-0.05]$ or $[0.95-1.0]$. This strongly indicates the presence of statically biased branches in a wide variety of workloads.

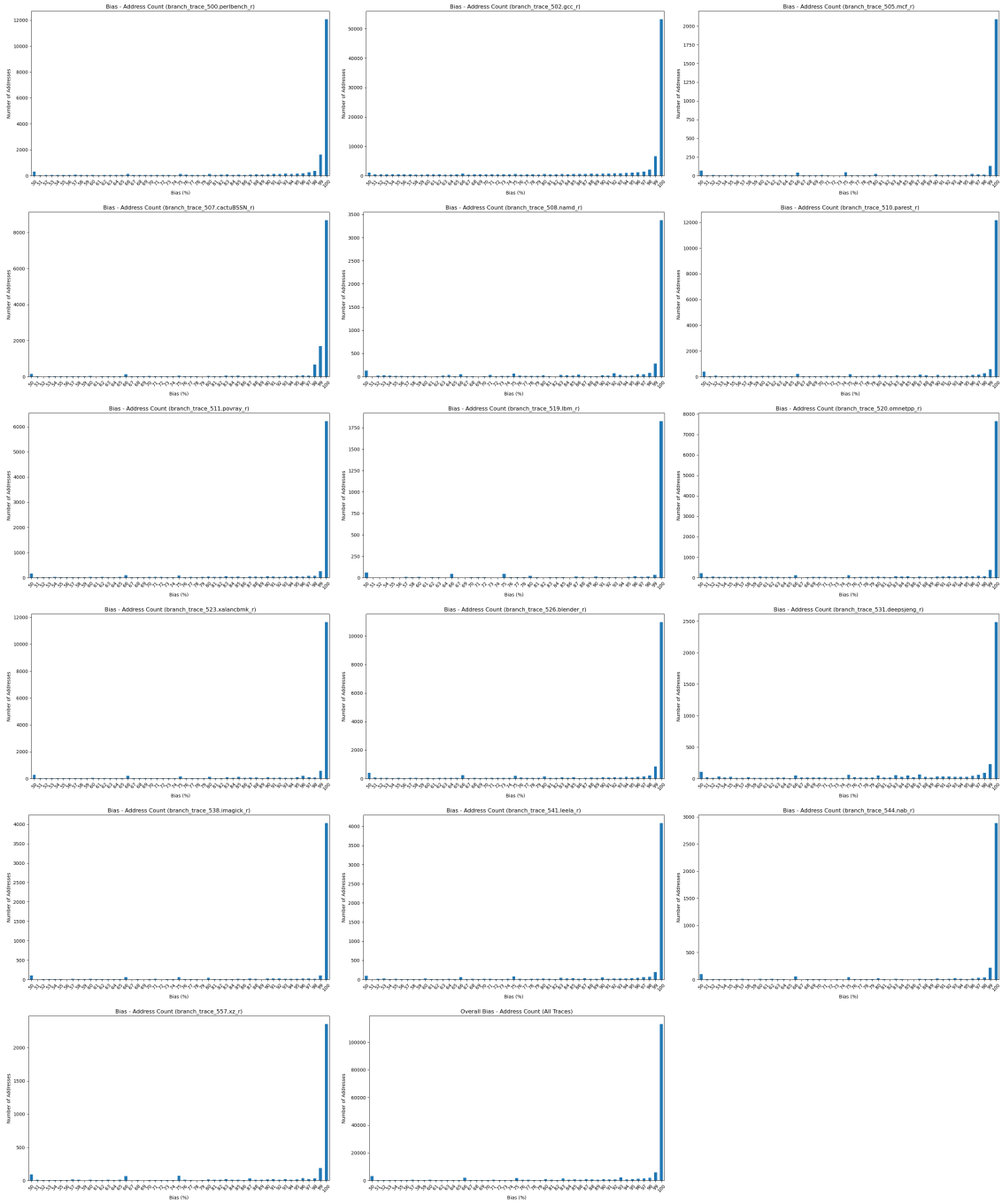


Figure 5.10: Distribution of branch address counts across bias intervals. The sharp concentration at the extremes (bias close to 0% or 100%) indicates that most branches are heavily biased, with few exhibiting neutral behavior.

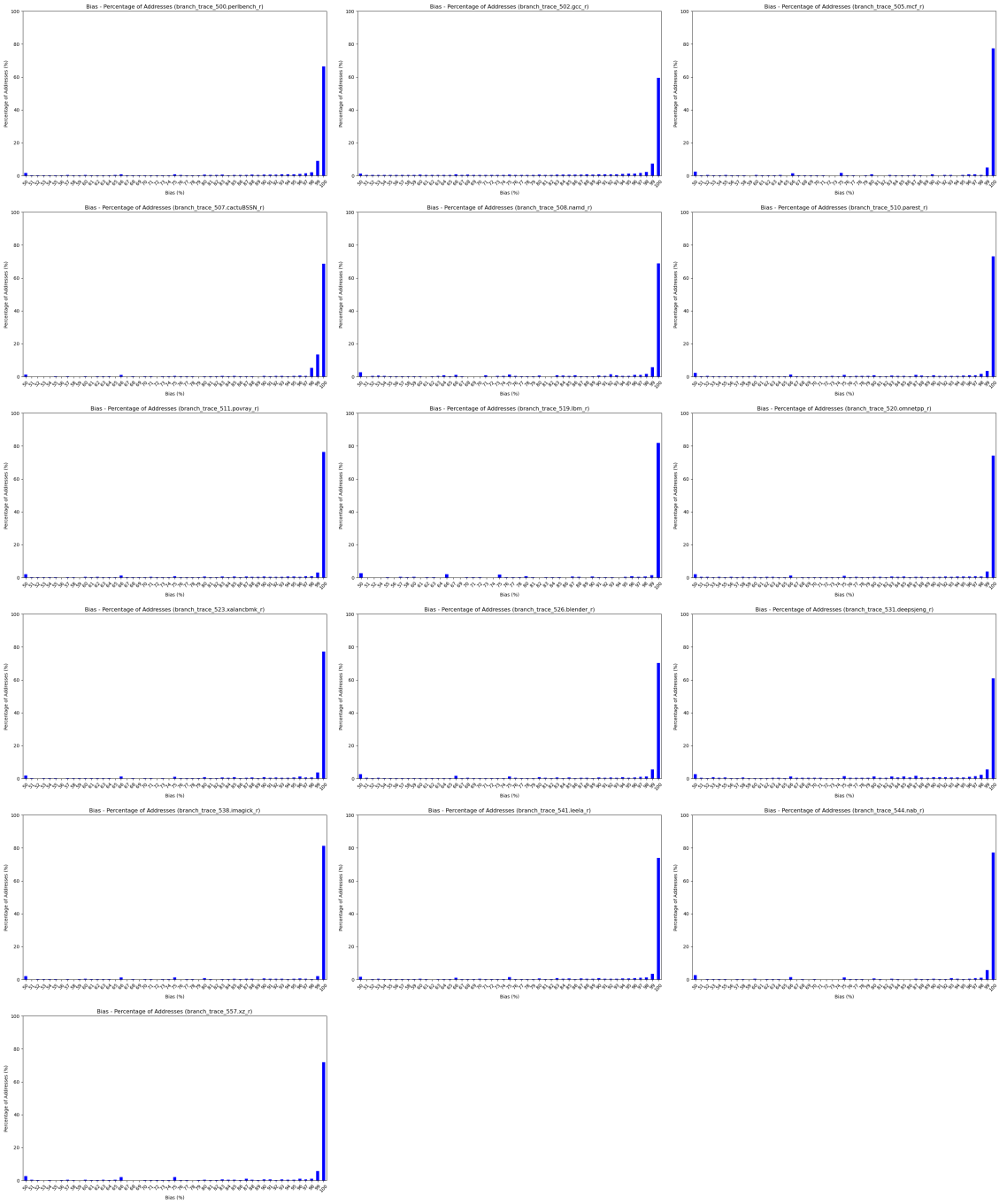


Figure 5.11: Percentage of unique branch addresses per bias bucket. This confirms that the majority of static branches are either predominantly taken or not taken, making them suitable targets for static classification.

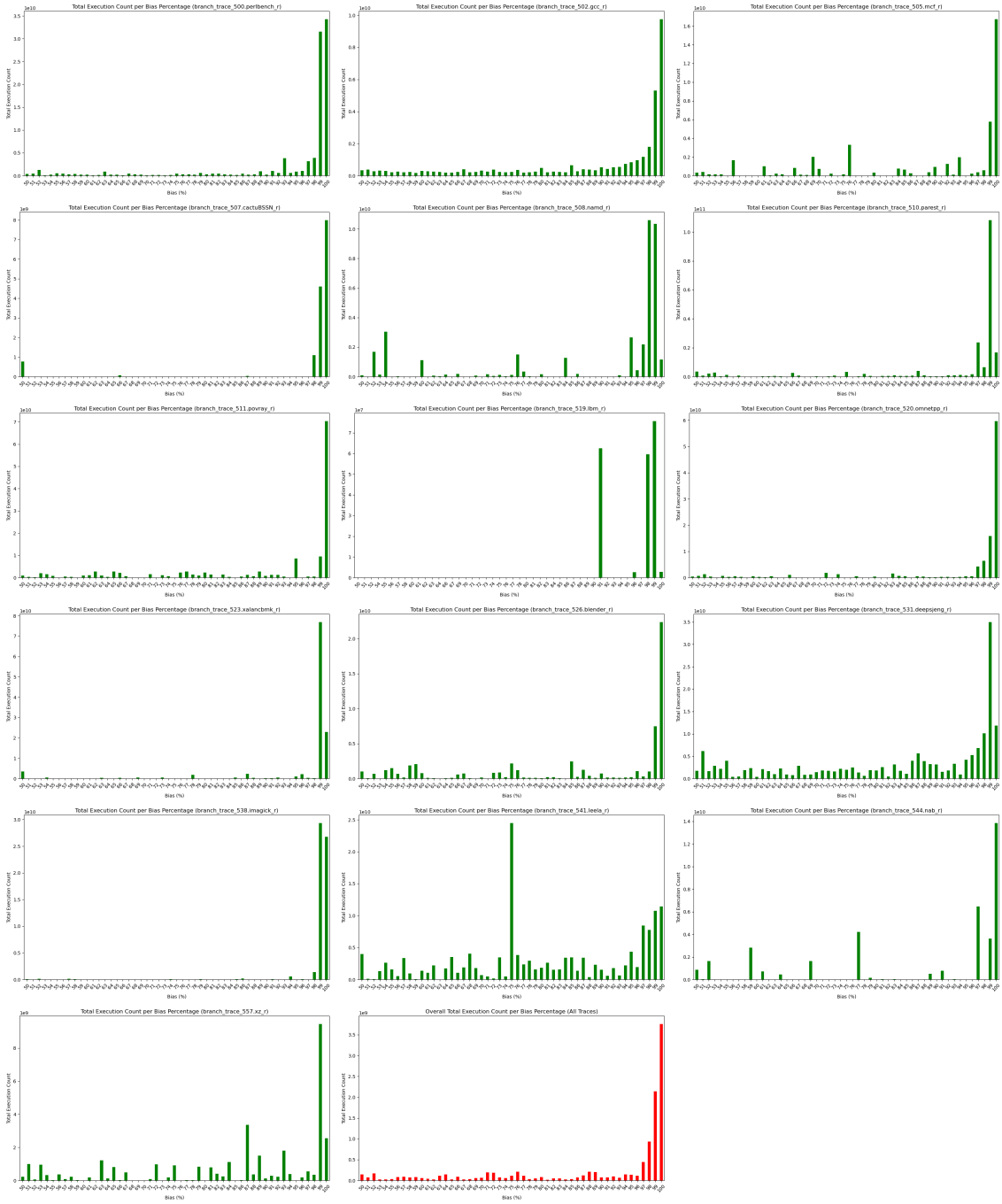


Figure 5.12: Total execution counts grouped by branch bias intervals for each benchmark. The plots show that most execution activity concentrates on highly biased branches, particularly those near 100% or 0% bias, highlighting their disproportionate influence on program performance.

Figure 5.13: Relative contribution of each bias category to total execution count. While highly biased branches dominate execution in absolute terms, this plot confirms their dominance even when normalized. The sharp peak near 0% and 100% bias illustrates the concentration of dynamic execution around these extremes.

5.1.3 Execution-Weighted and Unique Address Distribution

In addition to raw bias distribution, we also analyze branches from three complementary perspectives:

- **Total execution count per bias bin** (Figure 5.14),
- **Execution percentage grouped by frequency categories** (Figure 5.15),
- **Unique branch address distribution per bin** (Figure 5.16).

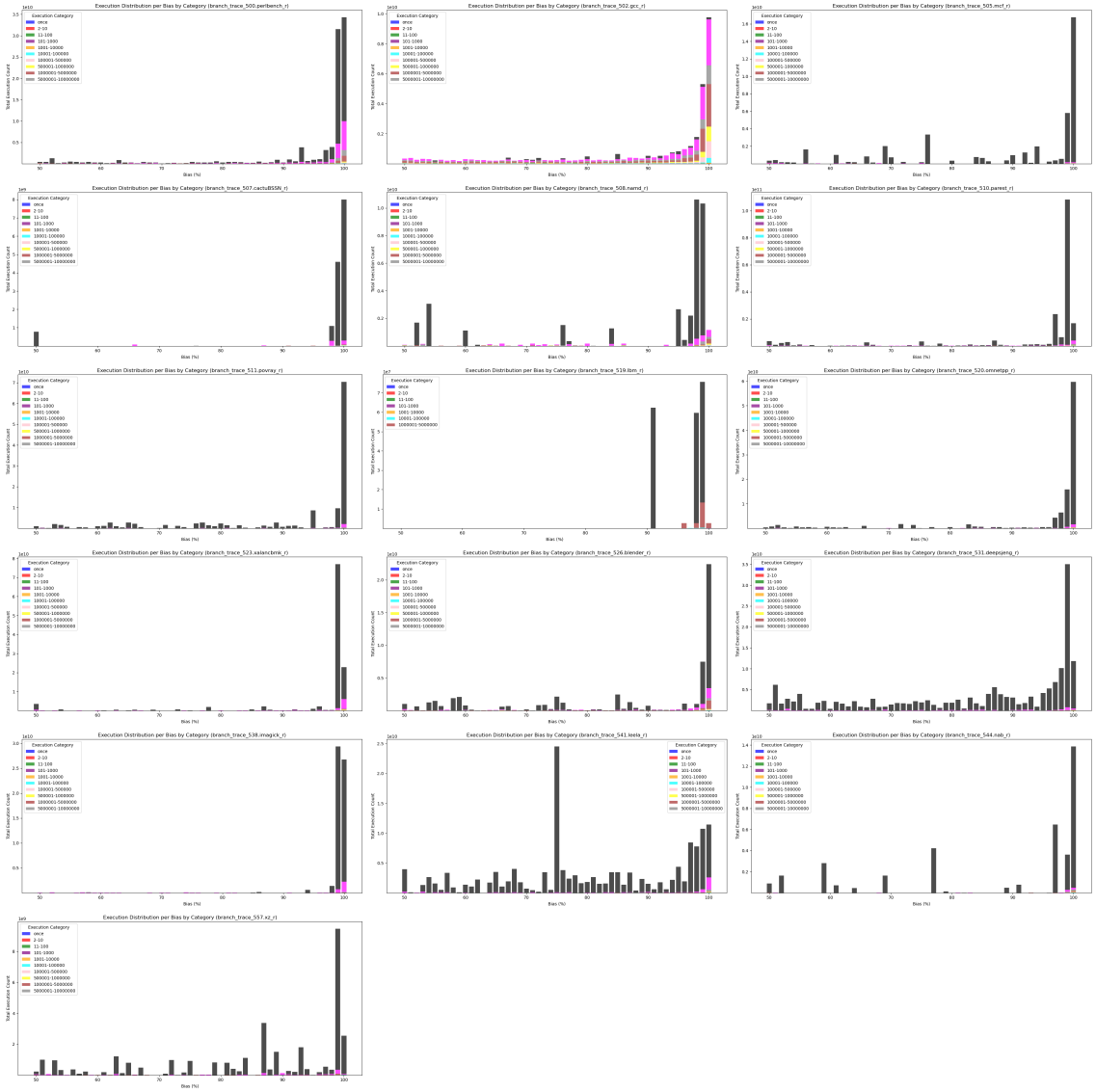


Figure 5.14: Execution distribution across bias categories segmented by execution frequency. The data reveal that most execution time is concentrated in a small number of branches that are either almost always taken or not taken. This reinforces the practical importance of accurately predicting highly biased branches in performance-critical paths. As we can see leela does not behave like the as the majority of its branches have a 75% bias and that is due to its algorithm.

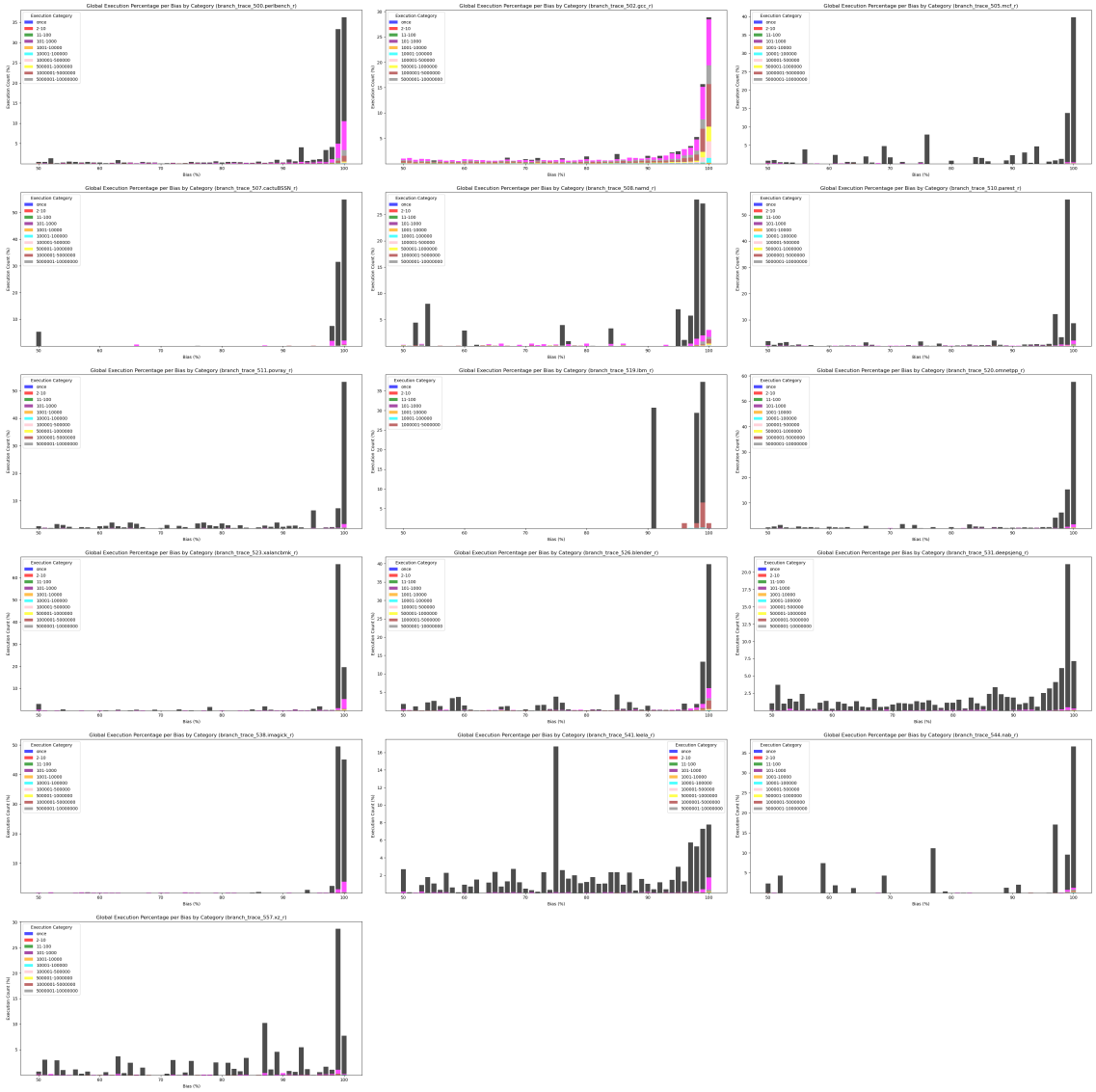


Figure 5.15: Global distribution of execution percentages across bias intervals, grouped by frequency category. The figure shows that across all benchmarks, a significant proportion of total execution is associated with branches exhibiting extreme bias. This indicates that even though some biased branches are infrequent, their cumulative execution contribution remains non-negligible, reinforcing their relevance to prediction models.

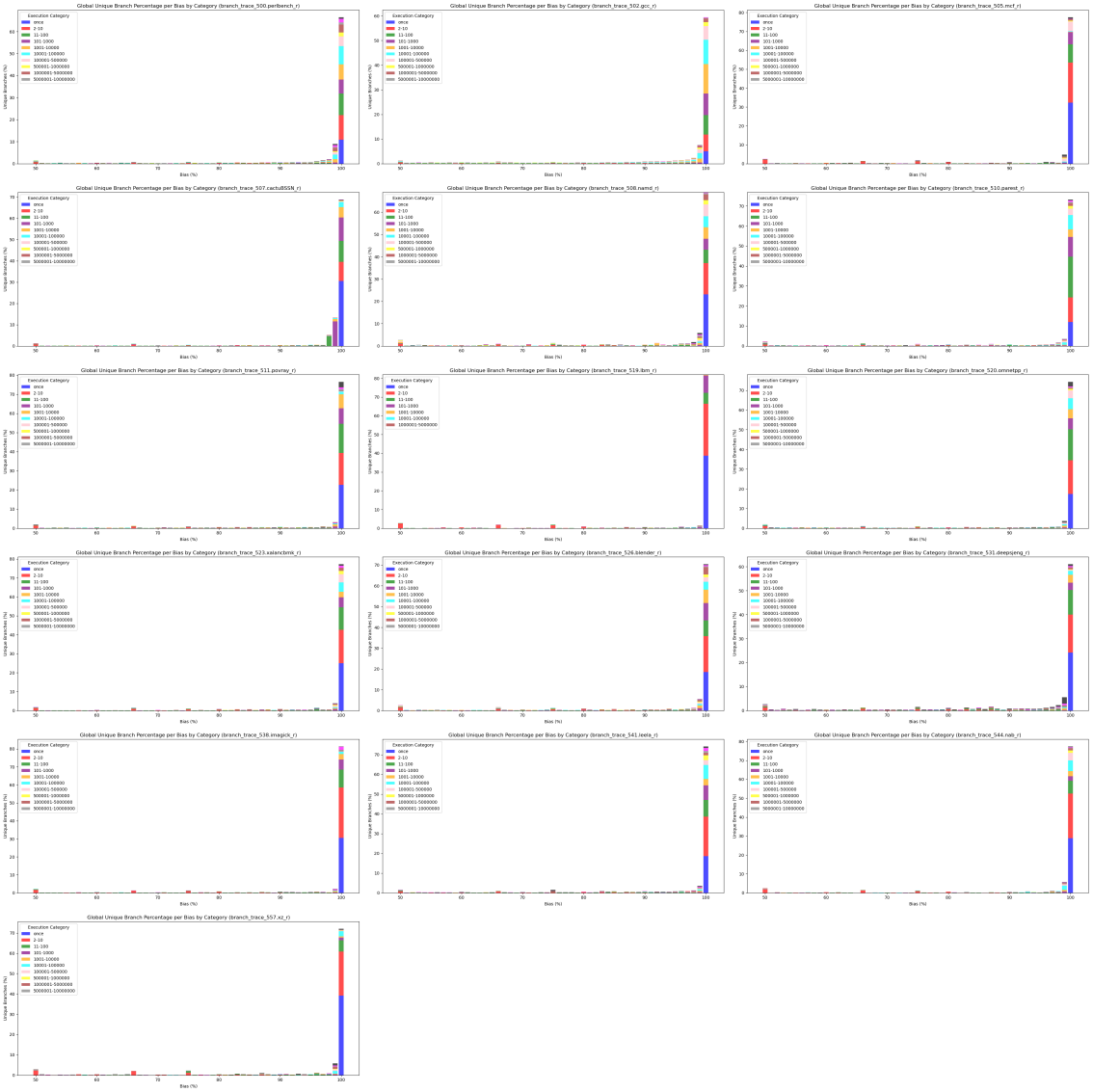


Figure 5.16: Global distribution of unique branches across bias intervals and frequency categories. The figure highlights that a substantial number of distinct branch instructions exhibit extreme bias, particularly in the 0–5% and 95–100% intervals. This supports the notion that many static branches demonstrate highly skewed behavior, making them suitable targets for static prediction mechanisms.

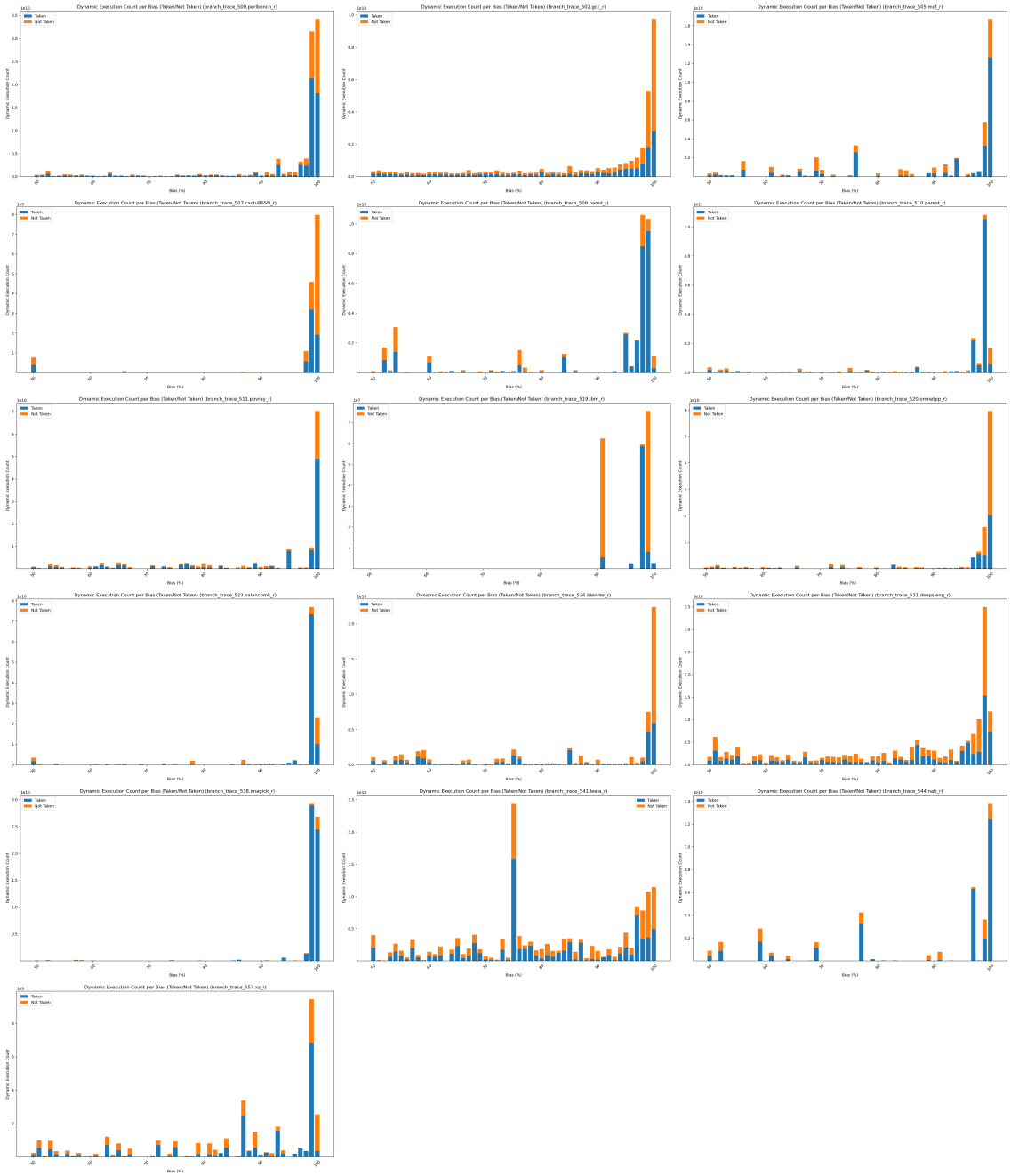


Figure 5.17: Stacked bar plots of dynamic execution counts for each bias interval, separated by taken and not taken outcomes for every benchmark. These plots reveal that for most benchmarks, the bulk of dynamic executions occur at the extreme bias intervals and are predominantly from either taken or not-taken branches, confirming the highly polarized nature of real-world branch behavior. This further emphasizes the motivation for static prediction mechanisms focused on highly biased branches.

Figure 5.18: Unique static branch counts per bias interval, split by taken and not taken outcomes. This global view across all benchmarks confirms that the vast majority of statically observed branches fall in the extreme bins (i.e., always taken or always not taken), and that each bin tends to be dominated by a single outcome. This further reinforces the feasibility and usefulness of statically identifying and optimizing for high-bias branches.

5.1.4 Conclusion: Bias is Widespread and Predictable

Across all metrics—static frequency, execution count, and unique instruction count—we observe a consistent trend: branches are overwhelmingly skewed toward highly biased behavior. This justifies the design decision to focus on bias-aware static prediction.

These insights establish that:

- High-bias branches are not rare edge cases; they are statistically dominant.
- These branches offer meaningful opportunities for static prediction to complement hardware prediction.
- Any prediction model that leverages bias can generalize well across benchmarks and workloads.

Having established this, we next evaluate whether program-based prediction models can accurately identify these high-bias branches using static features alone.

5.2 Can the Model Predict High Bias and Direction?

To assess whether the model can effectively determine if a branch is highly biased and whether it is likely to be taken or not, we trained a binary classifier using the complete feature set and evaluated its performance across all benchmarks. Both for bias prediction and direction we used XGBoost as it had the best results for both categories. The results are visualized in Figure 5.19 and summarized in Table 5.1.

Confusion Matrix Terms

- **TP (True Positives)**: Instances where the model correctly predicted taken.
- **TN (True Negatives)**: Instances where the model correctly predicted not taken.
- **FP (False Positives)**: Instances where the model incorrectly predicted taken (actual direction is not taken).
- **FN (False Negatives)**: Instances where the model incorrectly predicted not taken (actual direction is taken).

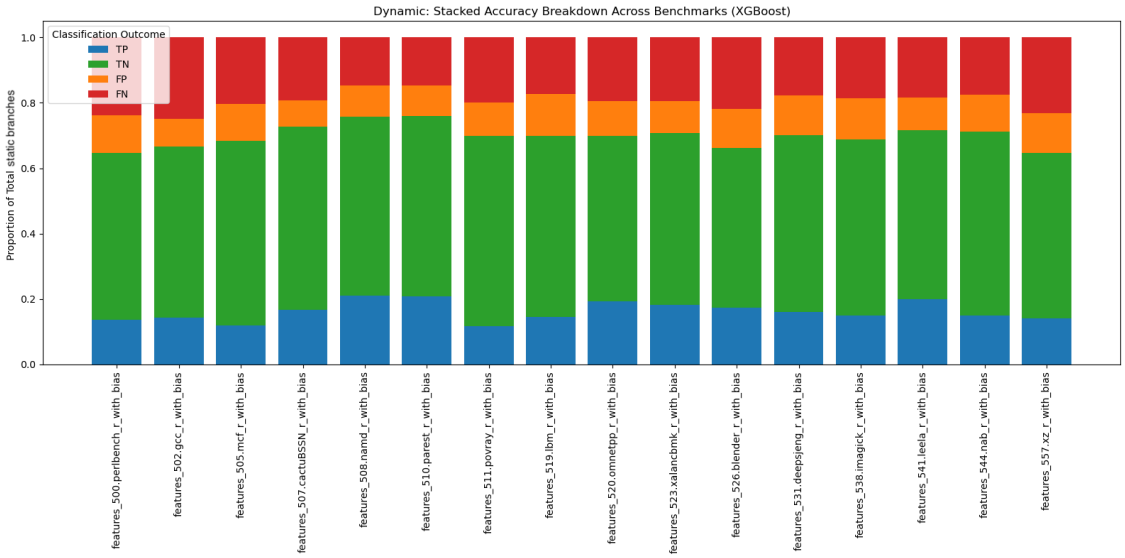


Figure 5.19: Stacked Accuracy Breakdown Across Benchmarks For Direction Prediction

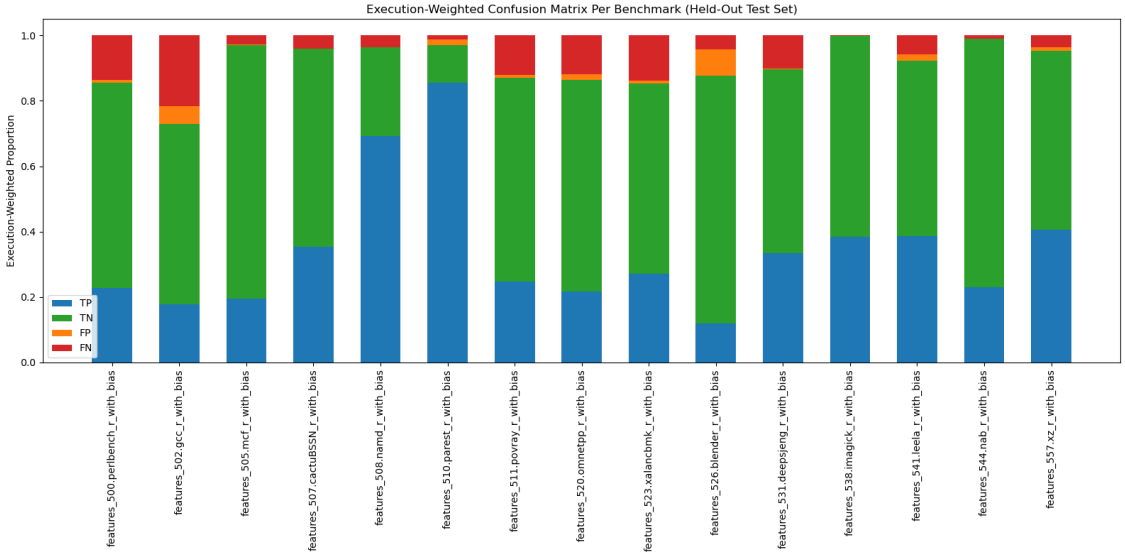


Figure 5.20: Dynamic: Stacked Accuracy Breakdown Across Benchmarks For Direction Prediction

Evaluation Summary. As indicated by the overall confusion matrix:

	Predicted Not Taken	Predicted Taken
Actual Not Taken	22,691	4175
Actual Taken	9,315	6795

The classifier is heavily biased toward predicting the majority class (not taken). This is reflected in the low recall value of 42.18% for the taken class, despite a moderate overall

accuracy of 68.61%. Precision (61.94%) and F1 score (50.18%) remain low, indicating poor performance in detecting taken branches, which are in the minority.

Per-Benchmark Trends.

Table 5.1 presents the individual benchmark metrics. Many benchmarks show low recall and F1 scores, suggesting the model fails to correctly classify taken branches in those cases.

Table 5.1: Per-Benchmark Direction Classification Performance

Benchmark	Accuracy	Precision	Recall	F1-Score
features_500.perlbench	0.6457	0.5396	0.3632	0.4342
features_502.gcc	0.6656	0.6249	0.3640	0.4600
features_505.mcf	0.6828	0.5120	0.3699	0.4295
features_507.cactuBSSN	0.7281	0.6792	0.4618	0.5498
features_508.namd	0.7572	0.6879	0.5874	0.6337
features_510.parest	0.7602	0.6944	0.5853	0.6352
features_511.povray	0.6990	0.5324	0.3677	0.4350
features_519.lbm	0.6996	0.5328	0.4577	0.4924
features_520.omnetpp	0.6992	0.6460	0.4981	0.5625
features_523.xalancbmk	0.7086	0.6544	0.4829	0.5557
features_526.blender	0.6616	0.5883	0.4423	0.5050
features_531.deepsjeng	0.7021	0.5702	0.4762	0.5190
features_538.imagick	0.6871	0.5382	0.4458	0.4876
features_541.leela	0.7158	0.6637	0.5225	0.5847
features_544.nab	0.7109	0.5657	0.4590	0.5068
features_557.xz	0.6468	0.5349	0.3786	0.4434

Discussion.

The stacked bar plot clearly visualizes a predominance of true negatives (green), indicating that the classifier is mostly identifying branches as not taken. However, the relatively small amount of true positives (blue) and the dominance of false negatives (red) suggest a major challenge in detecting highly biased branches that are frequently taken.

This result points to two important conclusions:

- The model currently struggles to predict whether a branch is taken, especially when taken branches are underrepresented.
- Class imbalance may be significantly impairing learning. Resampling or focusing only on highly biased branches (instead of all) could yield better results.

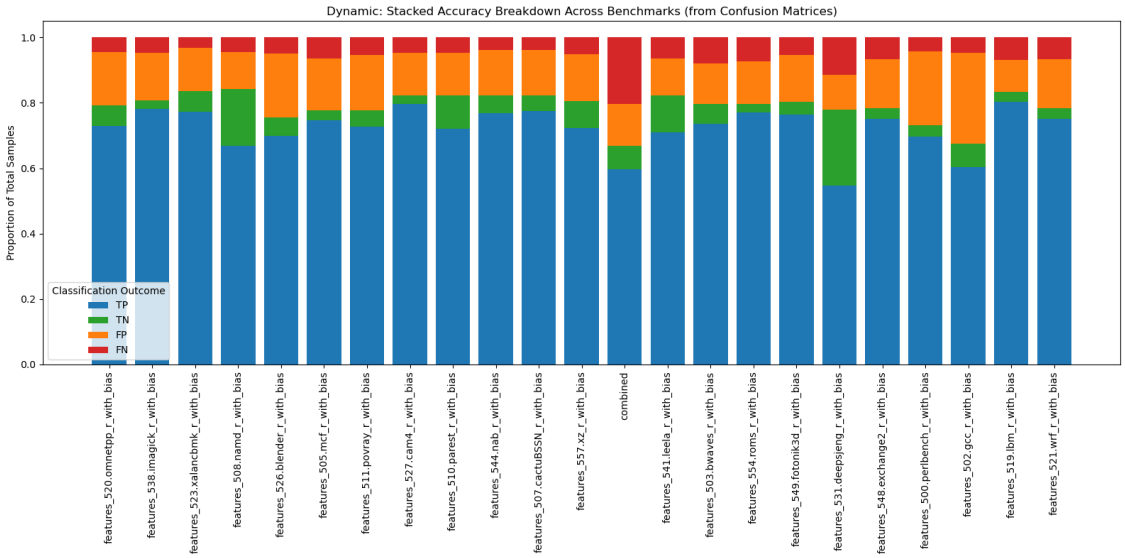


Figure 5.21: Execution-weighted stacked classification outcomes for bias prediction across benchmarks. TP: True Positive, TN: True Negative, FP: False Positive, FN: False Negative.

Observation: Figure 5.21 presents the execution-weighted classification outcome for predicting whether branches are highly biased. In contrast to the taken/not-taken prediction task, this task achieves significantly higher performance across all benchmarks. The true positive (TP) and true negative (TN) proportions dominate the stack for nearly every benchmark, while false positives (FP) and false negatives (FN) are minimal. This indicates that the model is both precise and sensitive in identifying biased behavior.

Performance Summary: As shown in Table 5.2, the bias prediction task achieves an overall execution-weighted accuracy of 79.48% and a precision of 76.76%, with near-perfect recall of 93.42%. The resulting F1-score of 84.28% demonstrates that the model is not only detecting biased branches accurately, but doing so with strong class balance. Moreover, most benchmarks exhibit both high individual accuracy and perfect recall, confirming the model's robustness in this setting.

Conclusion: These results support the claim that program-based learning can effectively classify highly biased branches. Given that such branches are static in behavior, they are particularly well-suited for this kind of prediction, and performance suggests that bias labels could reliably be inferred without runtime profiling.

Table 5.2: Per-Benchmark Bias Classification Performance

Benchmark	**Accuracy**	**Precision**	**Recall**	**F1-Score**
features_500.perlbench	0.7311	0.7548	0.9426	0.8383
features_502.gcc	0.6751	0.6842	0.9277	0.7876
features_503.bwaves	0.7961	0.8550	0.9032	0.8784
features_505.mcf	0.7780	0.8247	0.9217	0.8705
features_507.cactuBSSN	0.8217	0.8474	0.9526	0.8969
features_508.namd	0.8412	0.8547	0.9369	0.8939
features_510.parest	0.8235	0.8481	0.9380	0.8908
features_511.povray	0.7762	0.8108	0.9310	0.8667
features_519.lbm	0.8341	0.8928	0.9203	0.9063
features_520.omnetpp	0.7915	0.8175	0.9407	0.8748
features_521.wrf	0.7829	0.8321	0.9194	0.8736
features_523.xalancbmk	0.8360	0.8531	0.9616	0.9041
features_526.blender	0.7552	0.7805	0.9354	0.8509
features_527.cam4	0.8224	0.8582	0.9453	0.8996
features_531.deepsjeng	0.7787	0.8371	0.8262	0.8316
features_538.imagick	0.8068	0.8426	0.9429	0.8899
features_541.leela	0.8235	0.8626	0.9181	0.8895
features_544.nab	0.8236	0.8477	0.9523	0.8970
features_548.exchange2	0.7829	0.8321	0.9194	0.8736
features_549.fotonik3d	0.8026	0.8406	0.9355	0.8855
features_554.roms	0.7961	0.8540	0.9141	0.8830
features_557.xz	0.8058	0.8342	0.9348	0.8816

Table 5.3: Aggregate Performance Summary for Bias Prediction

Metric	Value
Overall Accuracy (confusion matrix based)	0.7474
Overall Precision	0.7676
Overall Recall	0.9342
Overall F1-Score	0.8428
True Average Accuracy (across benchmarks)	0.7948

5.3 Comparison: General vs. High-Bias Branch Outcome Prediction

This section compares two models trained to predict whether a branch will be `taken` or `not taken`. In both cases XGBoost was used to have a fair comparisson:

- A **general model** trained on all branches, regardless of their bias.
- A **high-bias-only model** trained exclusively on branches with bias > 0.995.

5.3.1 Evaluation Metrics

Table 5.4 summarizes the aggregated metrics across all benchmarks.

Table 5.4: Overall Performance: General vs. High-Bias Branch Prediction

Model Type	Accuracy	Precision	Recall	F1-Score
General Model (All branches)	0.6861	0.6194	0.4218	0.5018
High-Bias Model (Bias > 0.995)	0.7174	0.6401	0.4419	0.5229

While the high-bias model exhibits higher accuracy than the general model, it also has substantially better recall and F1-score. This is expected, as the model trained explicitly on high-bias datasets, leading the model to predominantly predict better than the other model).

5.3.2 Per-Benchmark Comparison

Figures 5.22 and 5.24 show stacked confusion matrix breakdowns across benchmarks for the two models. Both models reflects a diverse classification profile, while at the

same time True Negatives (Green) dominate the graphs, suggesting difficulty capturing minority `taken` cases.

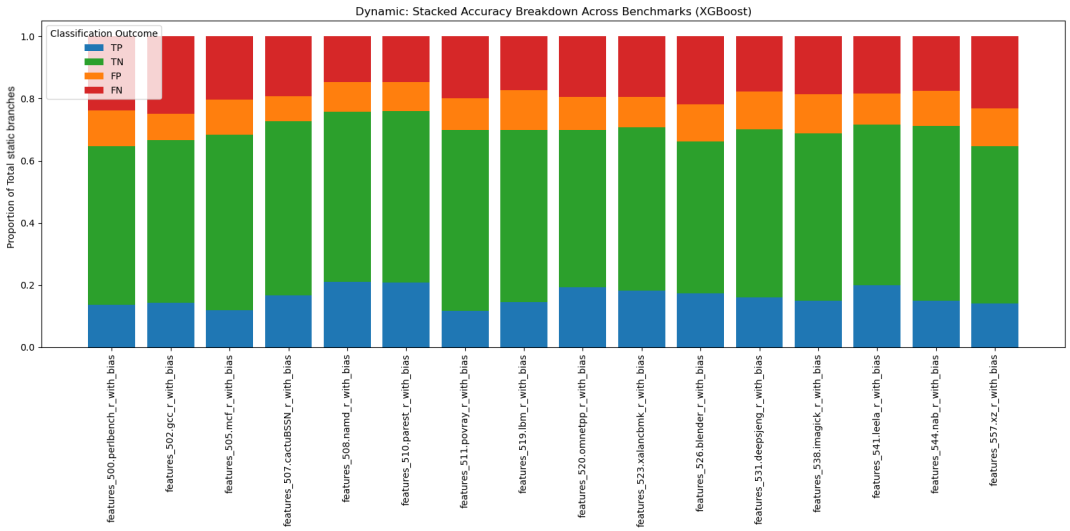


Figure 5.22: Static confusion matrix breakdown (general model).

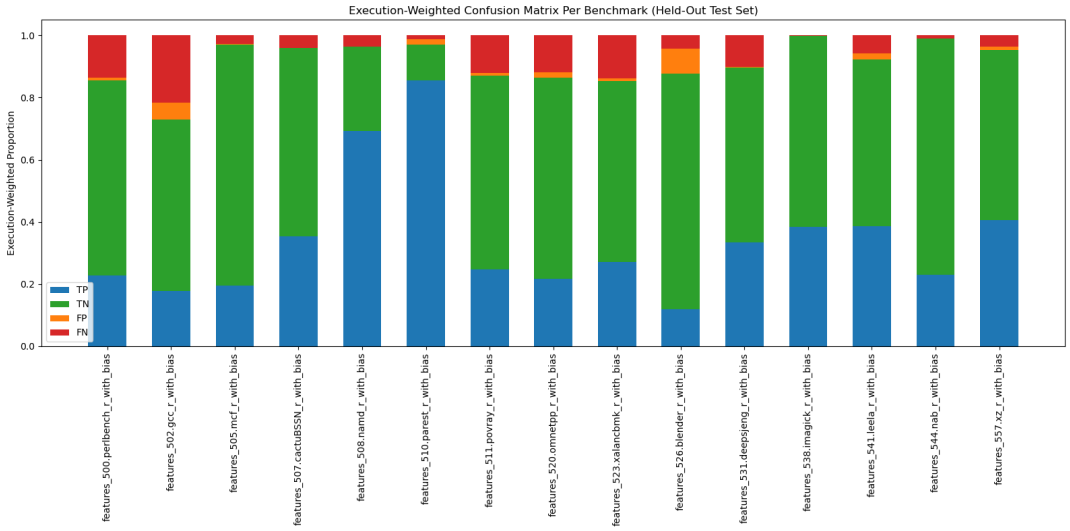


Figure 5.23: Execution-weighted confusion matrix breakdown (general model).

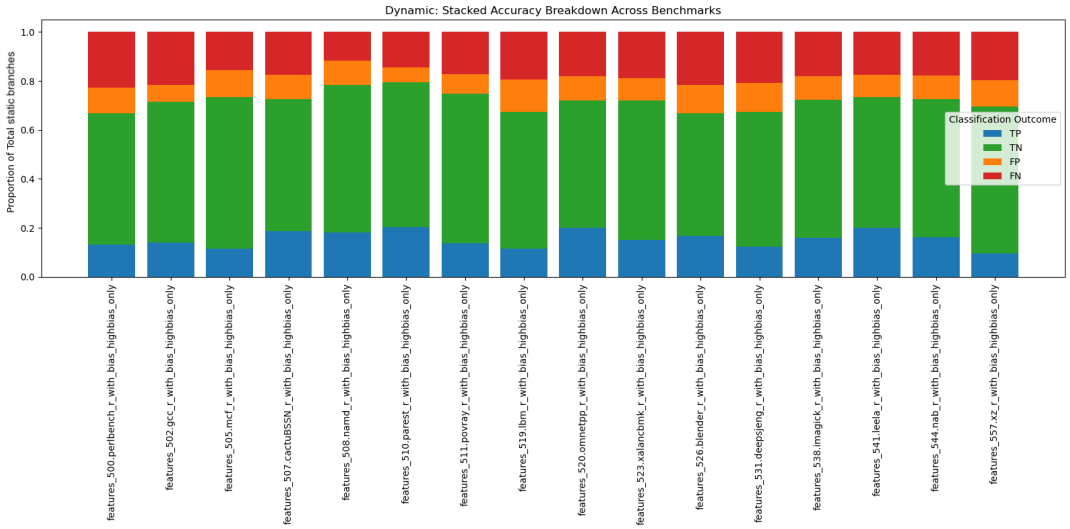


Figure 5.24: Staic confusion matrix breakdown (high-bias-only model).

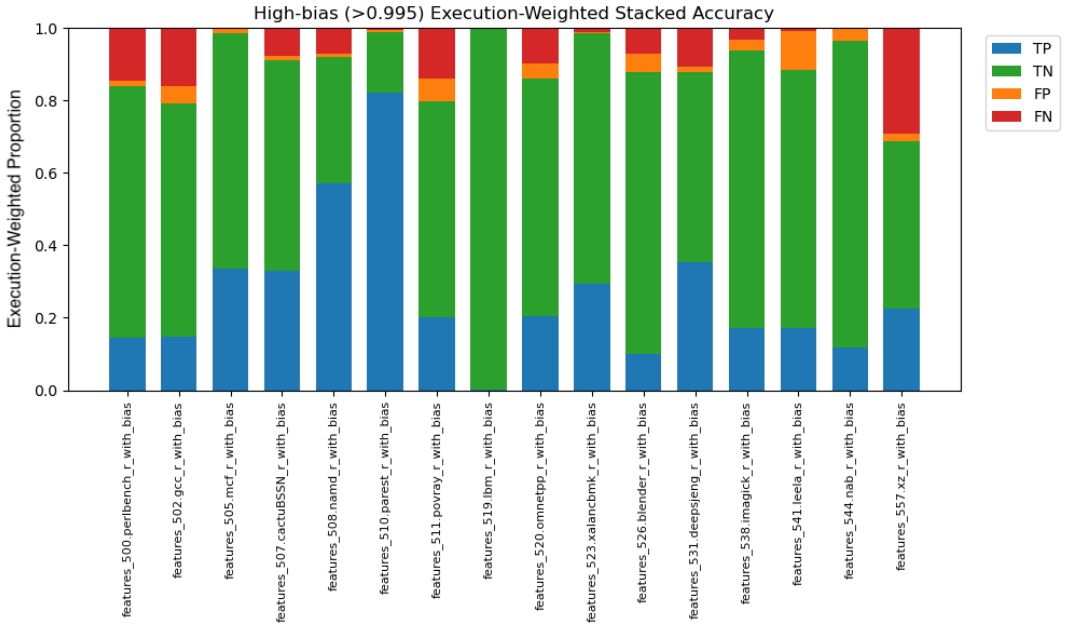


Figure 5.25: Execution-weighted confusion matrix breakdown (high-bias-only model).

5.3.3 Discussion

The general model benefits from exposure to a broader range of branch behaviors, enabling similar recall and F1-score despite lower precision. In contrast, the high-bias model demonstrates better precision and accuracy signifying that a model specifically trained for high bias branches outperforms the genera model.

This supports the hypothesis that while bias-targeted prediction may be feasible in terms of accuracy, it requires careful class balancing or alternative strategies (e.g., cost-

sensitive learning) to handle imbalanced distributions effectively.

5.4 Feature Importance Analysis

Understanding which static program features most influence branch outcome prediction is essential for building efficient and interpretable models, as well as for informing hardware implementation strategies. In this section, we analyze the relative importance of individual features using two complementary approaches: Random Forest feature importance metrics and sequential forward feature selection with XGBoost.

Random Forest Feature Importances

Figure 5.26 presents the feature importance scores assigned by a Random Forest classifier trained on the static feature set. The results reveal that `Jump Span` (the distance between the branch and its target) is by far the most informative feature, followed by local instruction density features (`Static After` and `Static Before`). These three features together account for the majority of predictive power in the model. Additional contributors include the `Bias` (dynamic execution bias used as a label during evaluation), `Branch Opcode`, and several operand metadata features. In contrast, lower-level architectural details such as operand types and register types appear to have limited standalone impact.

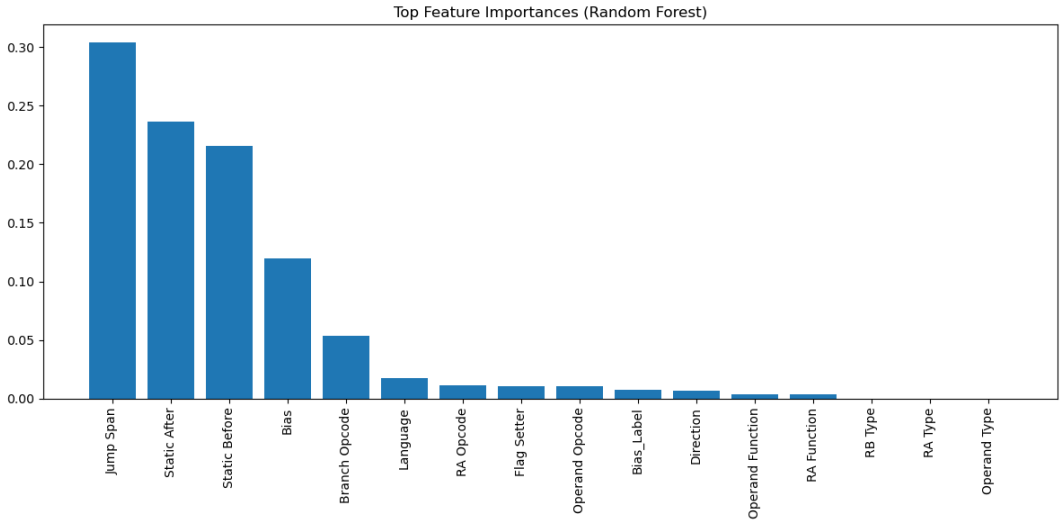


Figure 5.26: Random Forest feature importance scores for static branch prediction. The most influential features are `Jump Span`, `Static After`, and `Static Before`, while operand types and register information are less important.

Sequential Feature Selection

To further assess the minimum feature subset required for strong predictive performance, we conducted sequential forward feature selection using an XGBoost classifier. Figure 5.27 shows model accuracy as a function of the number of features included. The results indicate that the majority of accuracy gains are achieved with the first 6–8 features. Beyond this point, adding more features yields only marginal improvements, and in some cases, may lead to diminishing returns or slight overfitting.

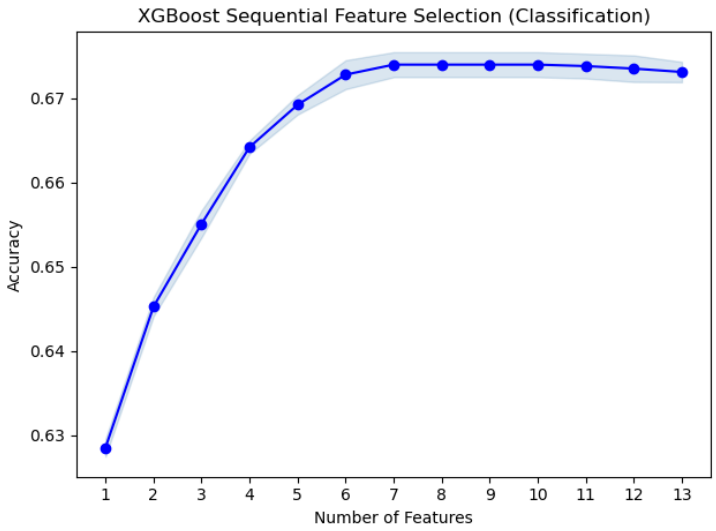


Figure 5.27: XGBoost sequential feature selection: accuracy as a function of the number of features. Most gains are realized by the first several features, supporting the design of lightweight static predictors.

This trend confirms that a compact subset of static features—primarily jump span, local instruction density, and a few branch/operand characteristics—is sufficient for high prediction accuracy. These findings have direct implications for hardware design: they enable practical, low-overhead branch prediction units that operate on a small set of easily extractable features. The features elected were: - Branch Opcode - Operand Opcode - Operand Function - Static Before - Static After - Jump Span - Bias confirming their importance shown in the graph above.

Summary

Together, these analyses show that only a handful of carefully chosen static features are necessary to approach the upper bounds of prediction accuracy. This supports the core thesis that static, program-based prediction is both feasible and hardware-friendly for high-bias branch prediction, as it avoids the complexity and overhead associated with large, profile-based feature sets.

5.5 Per-Benchmark vs. Cross-Benchmark Generalization in Taken/Not-Taken Prediction

In this analysis, we evaluate the accuracy of static program-based models in predicting whether each conditional branch will be `taken` or `not taken`, using only features derived from pin tool. XGBOOST was used for the two models as it offers both a fair comparison and the highest accuracy. The classification performance is measured in both per-benchmark and cross-benchmark training/testing scenarios.

5.5.1 Experimental Settings

- **Per-benchmark evaluation**: Each benchmark is split into independent training and test sets. The model is trained and evaluated using data drawn solely from the same program, capturing locally optimal patterns but potentially overfitting to benchmark-specific features.
- **Cross-benchmark evaluation**: The model is trained on a subset of benchmarks and evaluated on a disjoint set of unseen benchmarks. This simulates deployment to new code and tests the model's ability to generalize.

5.5.2 Aggregate Performance Comparison

Table 5.5 summarizes the aggregate metrics for both settings:

Table 5.5: Comparison of Prediction Metrics: Per-Benchmark vs. Cross-Benchmark

Setting	Accuracy	Precision	Recall	F1-Score
Per-Benchmark (separate)	0.6861	0.6194	0.4218	0.5018
Cross-Benchmark (complete)	0.6444	0.4421	0.2155	0.2897

The results show that while accuracy is slightly better in Per-Benchmark, precision, recall, and F1-score are far lower for the cross-benchmark scenario. This is expected: training on diverse programs introduces more variability and challenges the model's ability to generalize to unseen code.

5.5.3 Stacked Bar Plot Analysis

Figures 5.28 and 5.29 present execution-weighted, stacked bar plots of classification outcomes for the test benchmarks under each setting.

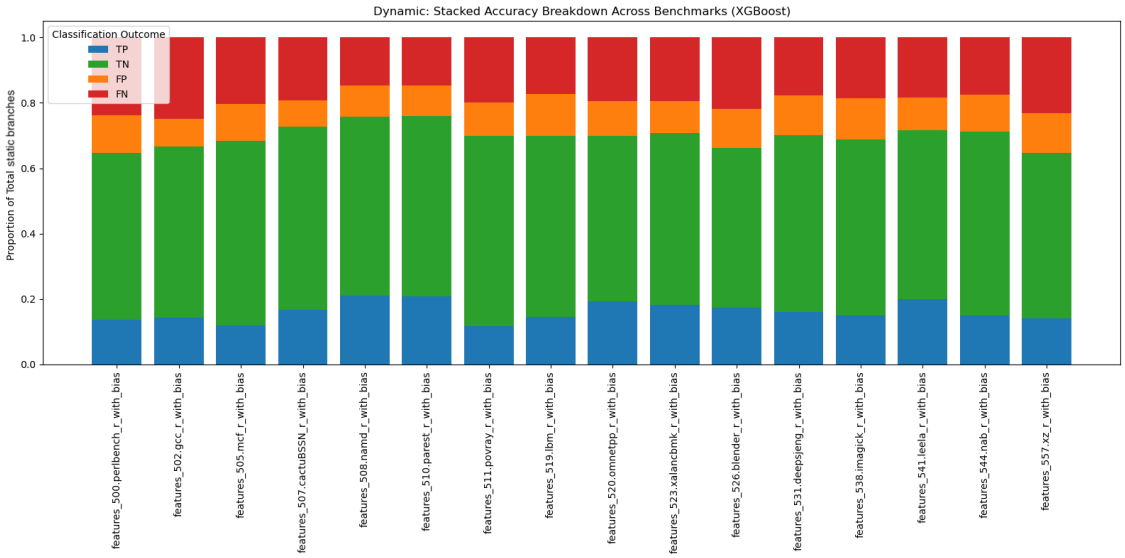


Figure 5.28: Execution-weighted stacked accuracy breakdown for per-benchmark training/testing.

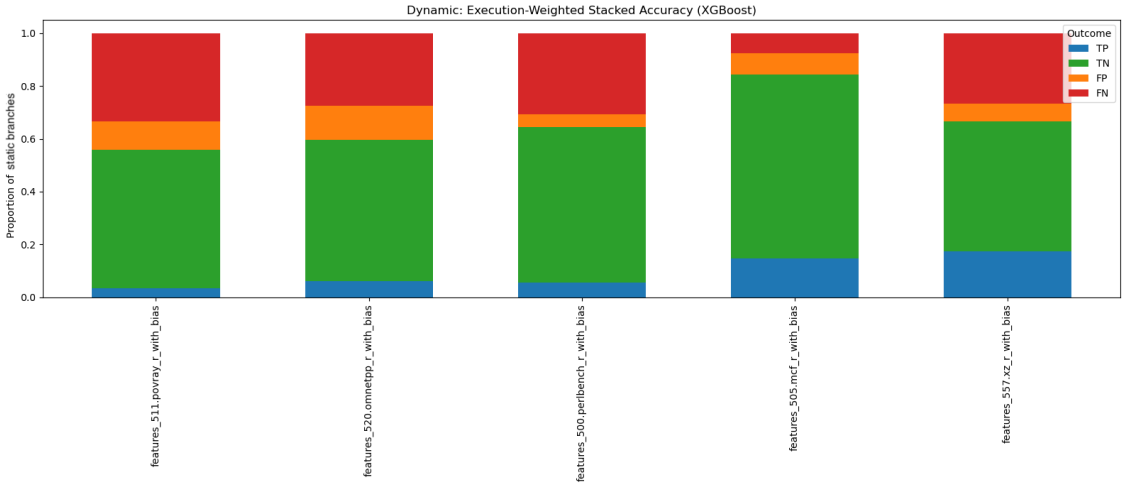


Figure 5.29: Execution-weighted stacked accuracy breakdown for cross-benchmark training/testing.

In both settings, the vast majority of dynamic executions are classified as true negatives (TN, green), with only a small proportion as true positives (TP, blue). False negatives (FN, red) make up a substantial portion of the remainder, especially in the cross-benchmark setting. The model struggles to correctly identify taken branches (TP) when deployed on unseen benchmarks, and is biased towards predicting not-taken outcomes due to the inherent imbalance of the data.

5.5.4 Interpretation and Discussion

The similarity in overall accuracy between the two settings is misleading; accuracy is dominated by the abundance of not-taken branches (the majority class). Precision, recall, and F1-score reveal the true difficulty of generalizing static predictions: recall in particular drops sharply in the cross-benchmark setting, indicating that the model rarely predicts a branch as taken unless it is extremely confident.

The stacked bar plots reinforce these findings. Cross-benchmark evaluation results in a higher fraction of false negatives, as the model fails to generalize features indicative of "taken" branches from one program to another. This highlights a key challenge in static branch prediction: the features that distinguish highly biased branches may be program-specific, and training on a broader set of benchmarks is necessary, but not always sufficient, for generalization.

5.5.5 Conclusion

While static, program-based predictors can achieve reasonable accuracy within individual benchmarks, their ability to generalize is limited. Practical deployment will require either benchmark adaptation, more sophisticated feature engineering, or hybrid approaches that incorporate runtime information or profile-guided adaptation. Nevertheless, the approach remains promising for classes of branches with strongly consistent, structurally encoded bias.

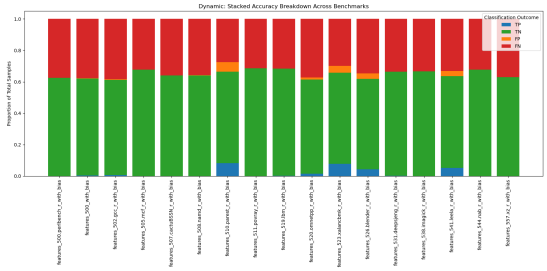
5.6 Comparison of Learning Methods

In this section we compare the four classifiers—CNN, MLP, Random Forest and XGBoost—on both the Taken/Not-Taken task and the Bias (>0.995) task. Tables 5.6 and 5.7 summarize per-benchmark performance, and Figure 5.30 / Figure 5.31 will show the execution-weighted stacked breakdowns side by side.

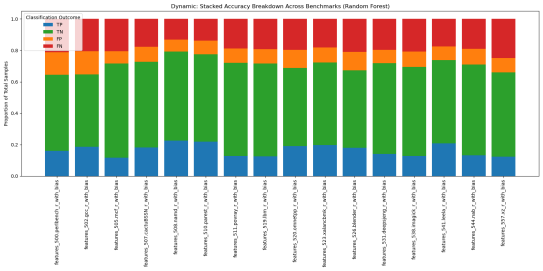
5.6.1 Taken / Not–Taken Prediction

Table 5.6: Per-Benchmark Classification Performance on Taken/Not-Taken (Dynamic Features)

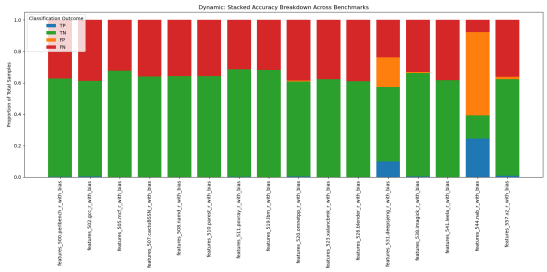
Benchmark	**CNN**				**MLP**				**RF**				**XGB**			
	Acc.	P	R	F1	Acc.	P	R	F1	Acc.	P	R	F1	Acc.	P	R	F1
500.perlbench_r	0.6257	0.0000	0.0000	0.0000	0.6262	0.7500	0.0022	0.0044	0.6451	0.5321	0.4307	0.4760	0.6457	0.5396	0.3632	0.4342
502.gcc_r	0.6117	0.6471	0.0177	0.0344	0.6109	0.6814	0.0113	0.0221	0.6467	0.5571	0.4756	0.5131	0.6656	0.6249	0.3640	0.4600
505.mcf_r	0.6772	0.0000	0.0000	0.0000	0.6772	0.0000	0.0000	0.0000	0.7164	0.6000	0.3642	0.4532	0.6828	0.5120	0.3699	0.4295
507.cactuBSSN_r	0.6405	0.0000	0.0000	0.0000	0.6405	0.0000	0.0000	0.0000	0.7269	0.6566	0.5039	0.5702	0.7281	0.6792	0.4618	0.5498
508.namd_r	0.6404	0.0000	0.0000	0.0000	0.6424	0.0000	0.0000	0.0000	0.7930	0.7492	0.6332	0.6863	0.7572	0.6879	0.5874	0.6337
510.parest_r	0.6642	0.5732	0.2280	0.3263	0.6434	0.0000	0.0000	0.0000	0.7750	0.7157	0.6123	0.6600	0.7602	0.6944	0.5853	0.6352
511.povray_r	0.6849	0.0000	0.0000	0.0000	0.6849	0.5000	0.0019	0.0039	0.7204	0.5810	0.4047	0.4771	0.6990	0.5324	0.3677	0.4350
519.lbm_r	0.6839	1.0000	0.0070	0.0140	0.6816	0.0000	0.0000	0.0000	0.7152	0.5773	0.3944	0.4686	0.6996	0.5328	0.4577	0.4924
520.omnetpp_r	0.6152	0.5636	0.0390	0.0729	0.6074	0.3043	0.0088	0.0171	0.6885	0.6252	0.4931	0.5513	0.6992	0.6460	0.4981	0.5625
523.xalancbmk_r	0.6572	0.6405	0.2084	0.3145	0.6226	0.0000	0.0000	0.0000	0.7225	0.6704	0.5207	0.5861	0.7086	0.6544	0.4829	0.5557
526.blender_r	0.6179	0.5523	0.1095	0.1828	0.6098	0.0000	0.0000	0.0000	0.6729	0.6063	0.4614	0.5240	0.6616	0.5883	0.4423	0.5050
531.deepsjeng_r	0.6638	1.0000	0.0037	0.0073	0.5735	0.3462	0.2967	0.3195	0.7194	0.6264	0.4176	0.5011	0.7021	0.5702	0.4762	0.5190
538.imagick_r	0.6660	0.0000	0.0000	0.0000	0.6630	0.3333	0.0090	0.0176	0.6952	0.5650	0.3795	0.4541	0.6871	0.5382	0.4458	0.4876
541.leela_r	0.6353	0.6064	0.1348	0.2205	0.6172	0.0000	0.0000	0.0000	0.7376	0.7034	0.5437	0.6133	0.7158	0.6637	0.5225	0.5847
544.nab_r	0.6764	0.0000	0.0000	0.0000	0.3926	0.3168	0.7582	0.4469	0.7095	0.5714	0.4098	0.4773	0.7109	0.5657	0.4590	0.5068
557.xz_r	0.6284	0.0000	0.0000	0.0000	0.6239	0.4000	0.0247	0.0465	0.6606	0.5755	0.3292	0.4188	0.6468	0.5349	0.3786	0.4434
Overall	0.6308	0.5996	0.0496	0.0916	0.6191	0.3689	0.0225	0.0425	0.6842	0.5979	0.4813	0.5333	0.6861	0.6194	0.4218	0.5018



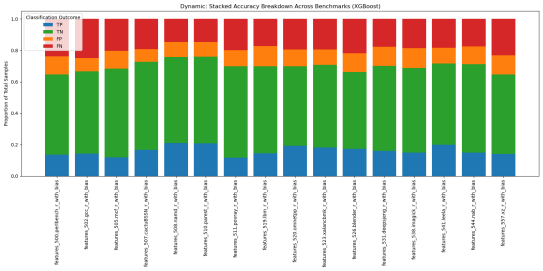
(a) CNN



(b) Random Forest



(c) MLP



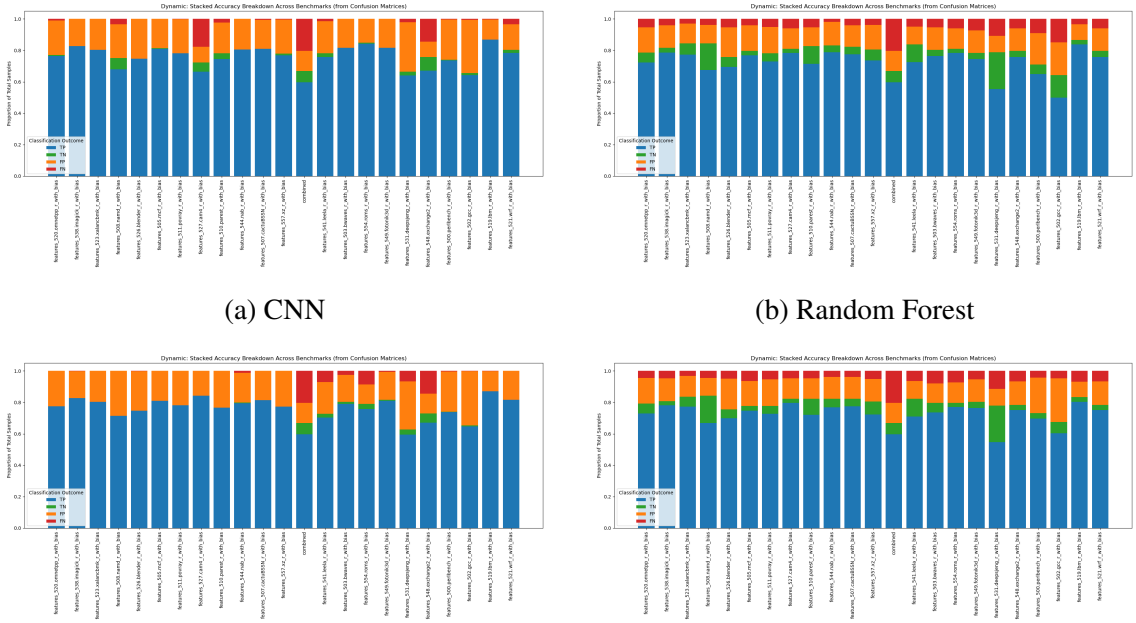
(d) XGBoost

Figure 5.30: Static stacked accuracy breakdown for Taken/Not-Taken across test benchmarks.

5.6.2 Bias (>0.995) Prediction

Table 5.7: Per-Benchmark Performance on Predicting High-Bias Branches

Benchmark	**CNN**				**MLP**				**RF**				**XGB**			
	Acc.	P	R	F_1	Acc.	P	R	F_1	Acc.	P	R	F_1	Acc.	P	R	F_1
500.perlbench_r	0.7395	0.7395	1.0000	0.8502	0.7392	0.7405	0.9966	0.8496	0.7087	0.7633	0.8784	0.8168	0.7311	0.7548	0.9426	0.8383
502.gcc_r	0.6552	0.6551	0.9906	0.7886	0.6528	0.6523	0.9967	0.7885	0.6423	0.7063	0.7691	0.7363	0.6751	0.6842	0.9277	0.7876
503.bwaves_r	0.8158	0.8158	1.0000	0.8986	0.8026	0.8219	0.9677	0.8889	0.8026	0.8406	0.9355	0.8855	0.7961	0.8550	0.9032	0.8784
505.mcf_r	0.8097	0.8097	1.0000	0.8948	0.8097	0.8097	1.0000	0.8948	0.7966	0.8257	0.9493	0.8832	0.7780	0.8247	0.9217	0.8705
507.cactuBSSN_r	0.8161	0.8163	0.9990	0.8985	0.8145	0.8145	1.0000	0.8978	0.8229	0.8497	0.9506	0.8973	0.8217	0.8474	0.9526	0.8969
508.namd_r	0.7141	0.7141	1.0000	0.8332	0.7141	0.7141	1.0000	0.8332	0.8443	0.8525	0.9455	0.8966	0.8412	0.8547	0.9369	0.8939
510.parest_r	0.7907	0.8135	0.9435	0.8737	0.7672	0.7673	0.9996	0.8682	0.8268	0.8565	0.9301	0.8918	0.8235	0.8481	0.9380	0.8908
511.povray_r	0.7817	0.7817	1.0000	0.8775	0.7817	0.7817	1.0000	0.8775	0.7823	0.8146	0.9341	0.8703	0.7762	0.8108	0.9310	0.8667
519.lbm_r	0.8722	0.8722	1.0000	0.9317	0.8700	0.8719	0.9974	0.9305	0.8655	0.8926	0.9614	0.9257	0.8341	0.8928	0.9203	0.9063
520.omnetpp_r	0.7744	0.7744	1.0000	0.8729	0.7744	0.7744	1.0000	0.8729	0.7866	0.8179	0.9319	0.8712	0.7915	0.8175	0.9407	0.8748
521.wrf_r	0.8158	0.8158	1.0000	0.8986	0.8158	0.8158	1.0000	0.8986	0.7961	0.8394	0.9274	0.8812	0.7829	0.8321	0.9194	0.8736
523.xalancbmk_r	0.8108	0.8168	0.9855	0.8933	0.8039	0.8038	1.0000	0.8912	0.8440	0.8604	0.9620	0.9083	0.8360	0.8531	0.9616	0.9041
526.blender_r	0.7471	0.7476	0.9987	0.8551	0.7471	0.7472	0.9996	0.8552	0.7578	0.7858	0.9289	0.8514	0.7552	0.7805	0.9354	0.8509
527.cam4_r	0.8421	0.8421	1.0000	0.9143	0.8421	0.8421	1.0000	0.9143	0.8092	0.8561	0.9297	0.8914	0.8224	0.8582	0.9453	0.8996
531.deepsjeng_r	0.6613	0.6613	1.0000	0.7961	0.6267	0.6598	0.8991	0.7611	0.7874	0.8418	0.8355	0.8386	0.7787	0.8371	0.8262	0.8316
538.imagick_r	0.8280	0.8280	1.0000	0.9059	0.8260	0.8276	0.9976	0.9047	0.8169	0.8472	0.9502	0.8958	0.8068	0.8426	0.9429	0.8899
541.leela_r	0.7747	0.7750	0.9988	0.8728	0.7267	0.7762	0.9088	0.8373	0.8371	0.8641	0.9368	0.8990	0.8235	0.8626	0.9181	0.8895
544.nab_r	0.8064	0.8064	1.0000	0.8928	0.7984	0.8073	0.9852	0.8874	0.8316	0.8402	0.9770	0.9034	0.8236	0.8477	0.9523	0.8970
548.exchange2_r	0.8224	0.8212	1.0000	0.9018	0.7303	0.8430	0.8226	0.8327	0.7961	0.8394	0.9274	0.8812	0.7829	0.8321	0.9194	0.8736
549.fotonik3d_r	0.8158	0.8158	1.0000	0.8986	0.8158	0.8200	0.9919	0.8978	0.7829	0.8370	0.9113	0.8726	0.8026	0.8406	0.9355	0.8855
554.roms_r	0.8421	0.8421	1.0000	0.9143	0.7895	0.8582	0.8984	0.8779	0.8092	0.8561	0.9297	0.8914	0.7961	0.8540	0.9141	0.8830
557.xz_r	0.7813	0.7805	0.9980	0.8760	0.7737	0.7737	1.0000	0.8724	0.8043	0.8236	0.9506	0.8826	0.8058	0.8342	0.9348	0.8816
Overall	0.7294	0.7312	0.9908	0.8414	0.7234	0.7261	0.9927	0.8387	0.7351	0.7851	0.8734	0.8269	0.7474	0.7676	0.9342	0.8428



(a) CNN

(b) Random Forest

(c) MLP

(d) XGBoost

Figure 5.31: Execution-weighted stacked accuracy breakdown for High-Bias (>0.995) branches.

5.6.3 Discussion

Across the two tasks, we see that

- **CNN** yields moderate overall accuracy on Taken/Not-Taken (63%) but very low recall on rare classes; for Bias it achieves high recall but its precision and F1 drop on some benchmarks.
- **MLP** underperforms CNN on Taken/Not-Taken (62% vs. 63%) but matches its high recall on Bias, slightly lower F1 overall.
- **Random Forest** improves both precision and recall on Taken/Not-Taken (68% overall) and yields strong F1 on the Bias task (83%).
- **XGBoost** attains the best-balanced performance on Taken/Not-Taken (69% acc, 0.62 F1) and the highest recall+precision on Bias (0.84 F1).

Overall, the tree-based methods (RF, XGB) outperform the neural models on the Taken/Not-Taken task, while on the Bias task all learners achieve very high recall but differ in precision and F1. Figure 5.30 and Figure 5.31 visually confirm these trends on a per-benchmark basis.

Chapter 6

Related Work

Branch prediction has been one of the most intensively studied problems in microarchitecture for decades, due to its critical impact on instruction-level parallelism and pipeline utilization. In this chapter we survey three broad categories of prior work: (1) static, program-based heuristics; (2) evidence-based static prediction using machine learning; and (3) dynamic, hardware-based predictors including neural and tree-based approaches. We conclude with a brief discussion of more general ML-driven performance modeling efforts.

6.1 Static Program-Based Heuristics

Early "static" predictors make decisions entirely at compile time, using simple heuristics extracted from program structure. One of the most famous is the backward-taken/forward-not-taken (BTFNT) rule, which exploits the loop-branch bias to predict all backward branches as taken and forwards as not taken. McFarling and Hennessy first codified a set of such heuristics in MIPS compilers [23], and Ball & Larus [4] later showed how to combine eight intuitive rules (e.g. *loop branch*, *pointer comparison*, *call/return*) in a fixed, a priori ordering to achieve 80–90% static accuracy on SPEC programs. Wu & Larus [27] extended this work by using Dempster–Shafer theory to fuse multiple heuristics' probability estimates; however, their experiments showed only marginal improvement—if any—over the simpler fixed-order scheme.

6.2 Evidence-Based Static Prediction (ESP)

To overcome the reliance on hand-tuned heuristics, Calder et al. [6] proposed Evidence-Based Static Prediction (ESP), which uses supervised ML (neural nets or decision trees) to learn, from a corpus of programs, a mapping from static code features (e.g. opcode pat-

terns, CFG-derived attributes) to the branch take probability. ESP requires an offline training phase on existing binaries, but thereafter can predict on unseen programs purely from their static features. On a suite of 43 C and Fortran programs, ESP reduced static misprediction from 25% (best heuristic) to around 20% :contentReference[oaicite:0]index=0. Subsequent work has refined the feature sets and tree-based learners, but the core ESP idea remains the leading ML-based static predictor.

6.3 Dynamic, Hardware-Based Predictors

While static methods require no runtime support, dynamic branch predictors—implemented in hardware—continue to dominate high-performance CPUs. Beginning with two-bit counters and gshare/TAGE schemes [29], the community has explored richer structures at the cost of extra logic.

6.3.1 Perceptron and Neural Predictors

Jiménez & Lin introduced perceptron-based branch predictors that treat the global history bits and branch address bits as input neurons, learning a weight vector per branch via an on-line update rule [15]. These predictors can exploit long histories (e.g. 64 bits) and achieve higher accuracy than gshare at similar hardware cost. Follow-on work applies piecewise-linear ensembles [16], ahead-pipelined pipelines [17], and small MLPs [26] to further boost prediction of “hard-to-predict” branches.

6.3.2 Tree-Based and Boosted-Tree Predictors

More recently, researchers have begun to investigate decision trees and boosted-tree ensembles in hardware, taking advantage of their interpretability and compact look-up patterns. Early proposals used small regression trees to predict taken probability from selected global/local history patterns [1]; very-low-latency XGBoost variants have also been prototyped in FPGAs, showing comparable accuracy to perceptrons while enabling incremental retraining [10].

6.4 Other ML-Driven Performance Modeling

Beyond branch prediction, ML techniques have been used to model caches [11], nop-regression data-dependency throughput [28], and to guide DVFS or QoS controllers [19]. These works echo the ESP philosophy—learn from a corpus of measurements to predict microarchitectural behavior quickly and accurately.

6.5 Summary

In summary, static heuristics remain simple and zero-overhead but plateau around 75–80% accuracy. ESP showed that supervised ML can improve static prediction into the low-20% misprediction regime. Dynamic, hardware-based perceptron/MLP predictors push accuracy still higher at moderate cost, and tree-based methods promise additional gains with efficient implementations. Our work extends this lineage by systematically comparing four ML paradigms (CNN, MLP, Random Forest, XGBoost) on two branch prediction tasks, and demonstrating that modern tree-based learners can outperform classic neural schemes under practical budgets.

Chapter 7

Conclusion

7.1 Future Work

While the results of this thesis demonstrate the viability of static, ML-based branch prediction, several directions remain open for future exploration.

Hardware Integration.

As discussed before, the Static Program-Based Branch Prediction Unit (SPBBPU) currently exists only as a conceptual design and has not yet been integrated into hardware. All evaluation was performed offline at the software level using high-level models and profiling traces. An important avenue for future work is to implement a prototype of SPBBPU at the RTL or FPGA level, integrate it with an existing RISC-V or x86 core, and quantify the latency and area overhead versus performance benefits in a real pipeline.

Further Evaluation

Despite the extensive evaluation conducted on this approach, there are still some experiments that remain unexplored. For instance, one promising direction would be to first classify whether a branch exhibits high bias or not. Only if the branch is classified as high bias would a subsequent prediction be made for whether it is taken or not taken.

Feature Extension.

Although our current feature set—covering opcode types, jump distances, operand classes, and local instruction context—already yields useful prediction accuracy, there is clear headroom for improvement. Future work could explore additional static features such as:

- Control-flow graph (CFG) shape descriptors (e.g., loop nesting depth, post-dominators)

- Symbolic register value ranges or constant propagation indicators

The inclusion of such richer features may enable finer discrimination between hard-to-predict branches, especially in the taken/not-taken task, where recall remains relatively low even for tree-based models (e.g., 48.1% for RF and 42.2% for XGB).

7.2 Conclusion

In this thesis, we have investigated the use of machine learning (ML) to improve static branch prediction in modern wide-issue processors. We proposed a **Static Program-Based Branch Prediction Unit (SPBBPU)** that leverages learned predictions from offline profiling to assist the traditional hardware BPU, especially for branches that exhibit highly biased behavior. The core idea is to statically extract rich features from binary code and use ML models to learn predictive patterns without requiring runtime feedback.

We instantiated this framework using four representative learning algorithms—1D convolutional neural networks (CNNs), multilayer perceptrons (MLPs), Random Forests (RF), and XGBoost (XGB)—and evaluated them on two complementary tasks:

- **Taken / Not-Taken Prediction.** Predicting whether a branch is taken, using only static features.
- **High-Bias Classification.** Identifying whether a branch has a taken-probability above 99.5%, again using static features alone.

All models were trained on an 80/20 split over a suite of SPEC-style benchmarks spanning C, C++, Fortran, and numerical or symbolic domains. We evaluated both unweighted (per-benchmark) and execution-weighted (global) metrics: accuracy, precision, recall, and F_1 score. In addition, we generated stacked bar charts to visualize each model's classification breakdown (TP/TN/FP/FN) for every benchmark.

Summary of Findings

1. **Taken / Not-Taken Prediction is a skewed classification task.**

   The majority of branches in most programs are not taken, making naïve prediction trivial but unhelpful for identifying truly taken branches. The ML models were tasked with capturing both precision and recall, especially on the minority class.

2. **Tree-based methods outperform neural nets on Taken/Not-Taken.**

- CNN achieved global accuracy $\approx 63\%$, but recall on the "taken" class was only $\approx 5\%$, with $F_1 \approx 0.09$.
- MLP performed similarly (accuracy $\approx 62\%$, poor recall).
- RF improved both precision and recall (precision $\approx 60\%$, recall $\approx 48\%$, $F_1 \approx 0.53$).
- XGBoost matched RF in accuracy ($\approx 68\%$) and offered a strong balance with $F_1 \approx 0.50$.

These results suggest that tree-based learners better exploit the discrete and structural nature of static features like opcode classes, branch directionality, jump span, and register use.

3. **High-Bias Classification is easier and more separable.**

   Branches with taken-probability $\geq 99.5\%$ were reliably identified by all models:

   - All four learners reached recall ≥ 0.87 and precision ≥ 0.70.
   - CNN and MLP achieved $F_1 \approx 0.84$, with global accuracy between 72–73%.
   - RF and XGBoost also reached $F_1 \approx 0.83$–0.84, with XGBoost achieving the best accuracy at 74.7%.

   These results suggest that when bias is strong, even shallow models or basic statistical patterns suffice to classify branches correctly.

4. **Cross-Benchmark vs. Per-Benchmark Training.**

   We compared training a separate model for each benchmark versus using a cross-benchmark model trained on 80% of benchmarks and tested on the rest. The cross-benchmark models generalized well, maintaining over 72% average accuracy on High-Bias classification and improving maintainability by avoiding per-program retraining. This supports the feasibility of global SPBBPU models integrated in static compilers.

5. **Feature Importance Analysis.**

   We computed feature importances using Random Forests and XGBoost. The most influential static features included:

   - **Jump Span** — the relative distance of the jump target
   - **Surrounding Static Context** — frequencies of certain opcodes before and after the branch

- **Opcode of the Flag-Setting Instruction**
- **Bias Label (for meta learning)**
- **RA/RB Operand Types and Register Classes**

This analysis reveals that branch predictability is strongly tied to structural code patterns and local context.

6. **SPBBPU: Hardware Integration of Static Predictions.**

   Based on the encouraging results, we proposed a Static Program-Based Branch Prediction Unit (SPBBPU), a dedicated microarchitectural unit that integrates ML-derived static bias predictions with traditional BPU and hint-bit mechanisms. The SPBBPU:

   - Identifies and filters high-bias branches (those likely to be always taken or not taken)
   - Predicts bias from encoded static features embedded as binary hint bits
   - Collaborates with the BPU and compiler hints via a 3-input multiplexer
   - Applies a final rule to output prediction, giving more weight to SPBBPU in high-bias branches

   Feasibility. Our profiling showed that many programs exhibit a large number of high-bias branches (e.g., `imagick_r` with 94% taken or `parest_r` with 81% taken). Furthermore:

   - The High-Bias label can be predicted with 99% recall (MLP, CNN) and $F_1 \geq 0.84$
   - Once bias is known, taken/not-taken behavior is predicted with $\geq 75\%$ accuracy
   - This indicates that SPBBPU can offer early, accurate predictions for hardware control flow

Final Thoughts

This thesis demonstrates that static branch prediction can be significantly improved with machine learning models trained on disassembled binaries. Our proposed SPBBPU can be practically implemented with small tables, hint encodings, and compiler-based offline learning, and it complements—but does not replace—the dynamic predictor. As branch resolution remains a core performance limiter in modern out-of-order pipelines, hybrid approaches such as this may guide future microarchitectural innovations.

Bibliography

[1] T. Anderson, T. Harris, and M. J. Flynn. Perceptron-based dynamic memory prefetching. In *Proceedings of the 20th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 156–167. IEEE, 2014.

[2] J.-L. Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 2010.

[3] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313. ACM, 1993.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 46–57, 1993.

[5] B. Calder and D. Grunwald. The design and evaluation of a dynamic branch predictor. *The Journal of Supercomputing*, 11(2):141–160, 1997.

[6] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence–based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, Jan. 1997.

[7] B. Calder, D. Grunwald, D. Lindsay, M. Martin, M. Mozer, and B. Zorn. Corpus-based static branch prediction. *ACM SIGPLAN Notices*, 30(6):79–92, 1997.

[8] B. Calder, D. Grunwald, D. Lindsay, M. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 254–263. IEEE, 2001.

[9] P. P. Chang, S. A. Mahlke, and W.-m. W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 29(10):929–948, 1999.

[10] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2016.

[11] B. Dong, C. Cheng, and P. Wei. Memory cocktail therapy: Hybrid cache management for emerging non-volatile memories. In *Proceedings of the 2012 International Conference on Computer-Aided Design (ICCAD)*, pages 123–130. IEEE, 2012.

[12] P. G. Emma and E. S. Davidson. Characterization of branch and data dependencies in programs for evaluating pipeline performance. *IEEE Transactions on Computers*, 36(7):859–875, 1987.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6th edition, 2017.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2022. Available at `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[15] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.

[16] D. A. Jiménez and C. Lin. Reducing the cost of branch prediction with a sensitivity-optimized neural predictor. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 90–101, 2002.

[17] D. A. Jiménez and C. Lin. A scalable, accurate, and low-latency branch predictor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 78–88, 2003.

[18] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373. IEEE Computer Society Press, 1990.

[19] K. Kusano and V. Sarkar. Machine learning–driven power and performance management for modern data center platforms. Technical report, Rice University Department of Computer Science, 2020.

[20] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[21] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2020.

[22] A. Seznec. Analysis of the o-gehl branch predictor. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 394–405. IEEE, 2005.

[23] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[24] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Pearson Education, 10th edition, 2015.

[25] Standard Performance Evaluation Corporation. SPEC CPU 2017 Benchmark Suite. `https://www.spec.org/cpu2017/`, 2017. Accessed: 2025-05-13.

[26] L. Tarsa, G. Pekhimenko, and C. J. Rossbach. Cnn-based branch prediction for hard-to-predict branches. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 216–227, 2015.

[27] N. Wu and J. R. Larus. Feedback-directed conditional branch probability estimation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 100–110, 1994.

[28] S. Ye, M. Deisher, C. Ellis, R. Singh, and C. A. Baldwin. Ithemal: Accurate, portable basic-block throughput estimation using hierarchical rnns. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–205. ACM, 2018.

[29] T. S. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO)*, pages 51–61, 1993.

[30] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News*, 20(2):124–134, 1992.