Report - Branch Predictor

1. Introduction

In computer architecture, branch instruction is a significant challenge to processor performance. Every time when processor encounters a branch, it needs to determine whether to go to the branch destination or continue with the original order. Due to the fundamental constraints of pipeline, this decision must often be made before processor gets the actual outcome. Wrong decision will make processor time cycle waste. Base on that , branch prediction emerged to make processors can speculatively execute instructions based on a prediction of the branch outcome, thereby it can help to maintain pipeline efficiency and maximising instruction throughput.

This report investigates various branch-prediction strategies and analyses their relative strengths and weaknesses across different workloads. First, it describes the design and implementation of branch-predictor simulators developed for this assignment, from the simplest “always-taken” policy to a standard two-bit predictor, gshare, and two profile-based predictors, a basic version and a combined with 2-bit counter version. Next, it outlines the experimental methodology and evaluates these predictors on a carefully selected subset of benchmark traces chosen to reflect diverse program behaviors and branch characteristics. Finally, it analyzes the results and provides insights into each predictor’s performance.

2. Predictors Design and Implementation

This section describes the design and implementation of the branch prediction strategie.

2.1 Branch Predictor Interface

All predictors implement a common interface, ensuring modularity and facilitating comparison. The core interface consists of the following components:

- Predict: Predicts a given branch, returning true for taken or false for not taken.
- Update: Updates the predictor's internal state based on the actual branch outcome.
- Reset: Re-initialize the predictor's state for a new trace file.
- GetName: Returns the predictor name for reporting.

Each predictor takes branch information containing the branch address, target, type, and flags (direct, conditional, taken). This uniform interface enables fair comparison between different prediction strategies.

2.2 Always Taken Predictor

The Always Taken predictor is the simplest approach, predicting every branch as taken. This strategy can act as a baseline for evaluating other predictors.

Design and Operation

The Always Taken predictor contains no internal state and requires no additional hardware or toolchain. When queried, it unconditionally returns "taken" for any branch. It performs no updates, as its prediction never changes regardless of actual outcomes.

2.3 Standard 2-bit predictor

The Two-Bit predictor employs a counter for each branch to track its recent behaviors. The counter has four states: Strongly Not Taken (00), Weakly Not Taken (01), Weakly Taken (10), and Strongly Taken (11). This approach allows the predictor to maintain some "confidence" in its predictions, requiring two consecutive mispredictions to change the predicted direction.

Design and Operation

The predictor maintains a Branch History Table (BHT) containing two-bit counters. When making a prediction, use the lower bits of the address to search BHT. If the counter value is 00 or 01, the branch is predicted not taken, if the counter value is 10 or 11, the branch is predicted taken. And it will update the predictor, following Figure 1.

Figure 1 2-bit predictor updating rule

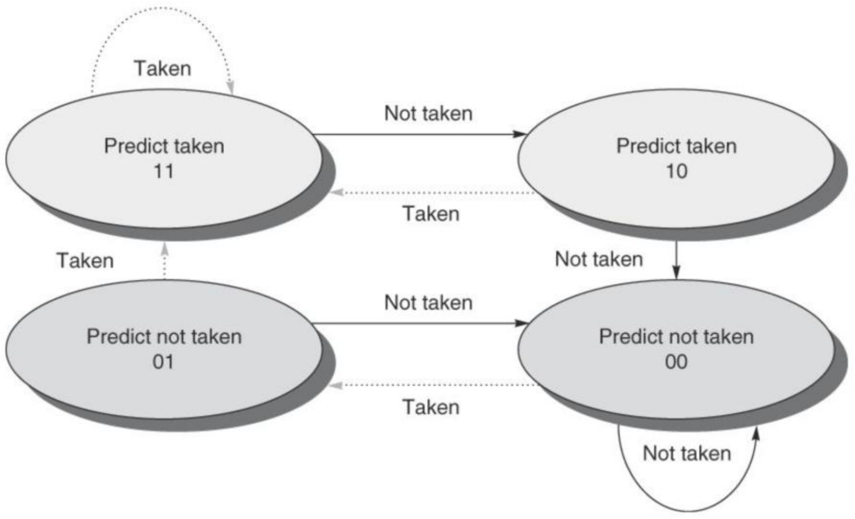


Table Indexing

This approach needs to cache the BHT, 2 bits per table entry (1KB for a 4096-entry table). And it must be faster than simply calculating the PC. Because the predictor uses the BHT indexed by the lower bits of the branch address, it may meet conflicts, one counter could be referring to a different branch. Thereby, different table sizes should have different impact, and this implementation evaluates the effect of table size by comparing tables with 512, 1024, 2048, and 4096 entries.

2.4 GShare

The GShare predictor improves upon the two-bit predictor by incorporating global branch history. It uses an XOR operation between the branch address and a global history register to index into the prediction table. This approach addresses pattern correlation between different branches, capturing more complex branch behaviours.

Design and Operation

The GShare predictor have a Branch History Table, same as 2-bit predictor, but it have a Global History Table (GHR) more. The GHR is a shift register that records the outcomes of recent branches

The prediction process first will do XOR operation on the branch address with the global history register. Then use the result to index into the BHT, and predict if the counter value is 10 or 11, not taken otherwise.

It will update the counter at the indexed location, same rule as 2-bit predictor. And shift the global history register left by one bit, set the least significant bit of the history register based on the actual branch outcome (1 for taken, 0 for not taken)

Global History Register

The global history register records the outcomes of recent branches as a bit pattern, where each bit represents whether a branch was taken (1) or not taken (0). The length of the history register is typically log2(table size) bits, matching the number of index bits. By introducing GHR, it can detect correlated branch patterns across different branch addresses. And the XOR indexing spreads similar addresses across the table, reducing destructive interference

2.5 Basic Profiled Predictor

I implemented two variants of profiled predictors, which use information gathered from a profiling run to initialize the prediction tables. First is a basic version.The basic profiled predictor operates in two phases: a profiling phase and a prediction phase. During profiling, it collects statistics on branch behaviours. For prediction, it uses an always-taken/not-taken policy, based on a branch history table and initialised based on the profiling data.

This predictor is simpler and requires less hardware than the 2-bit version, as it only stores a true/false per table entry for prediction. However, it doesn't adapt to changing branch behaviour during the prediction phase, as the state table is fixed after initialisation.

2.6 Profiled 2-bit Predictor

The profiled 2-bit predictor also utilizes pre-execution knowledge to enhance runtime prediction accuracy. Unlike purely dynamic predictors start with no prior information, profiled predictor leverages on pre-statistic branch behavior, and inform predictions during actual execution. This implementation combines profile-guided initialization with the adaptability of a 2-bit saturating counter scheme. Try to not only get higher prediction accuracy in the program startup and early stages, but also adapt to long-term runtime changes.

Design and Operation

This predictor should consist of BHT, an off-chip storage mechanism, and predictor Initialization Logic. The operation flow should have 2 phrase, 1) profiling phase, execute the program with instrumentation to record branch outcomes, compute the taking ratio for each branch, and generate a profile data structure; 2) execution phase, load the application's branch profile data, initialize the 2-bit counter, and do the runtime prediction and updating.

To implement in real environments, some hardware and toolchains could be changed:

1) Profile Data Loading Mechanism: Hardware support for efficient loading of profile data into the BHT.

2) Profiling Instrumentation: Compiler or runtime instrumentation to collect branch statistics.
3) Executable Format Extensions: New section in executables to store branch profile data.
4) Loader Modifications: System loader changes to extract profile data and initialize predictor.

One challenge when implementing this predictor is managing conflicts when multiple branches map to the same table entry. My approach addresses this through aggregation, for each table entry, collect statistics from all branches that map to it, and calculate a weighted aggregate taken ratio to initialize the entry. This approach tries to support the dominant branch behaviour, to decrease mispredictions for frequently executed branches.

3. Experiments

3.1 Experimental Setup

The experiments were based on the predictor designed in Section 2, simulating predictions on carefully selected subset of trace files, and calculating the misprediction rate for all algorithms. Rather than using the entire trace files, which can be extremely large, I use a muti-phase sampling approach to cut the selected trace, and get the final prepared trace file to simulate prediction.

3.2 Trace File Cutting Approach

The adopted strategy involves extracting segments from three distinct phases of each program's execution: 1)beginning phase: captures the program initialisation behaviour, 2) middle phase: represents the steady-state computation, typically involving the core algorithms, and 3) end phase: includes program finalization, results processing, and cleanup, etc. Specifically, for each selected trace (wrf, exchange2, gcc, and leela), using a python script (cut_trace.py) extracts a fixed number of branch instructions (1 million) from each of these three phases, then merges them into a single composite trace file, (saved in *trace/*).

3.3 Predictors to Evaluate

The following prediction strategies were evaluated:

- Always Taken
- 2-bit predictor with table sizes of 512, 1024, 2048, and 4096 entries
- Gshare predictor with 2048 entries
- Basic Profiled predictor with 2048 entries
- Profiled 2-bit predictor with 2048 entries

3.4 Trace File Analysis and Selection

Before conducting prediction experiments, I performed a comprehensive analysis of all available trace files to understand their characteristics. This analysis is used to select subsets for predictor performance experiments.

I analyzed each trace file to extract the following key metrics:

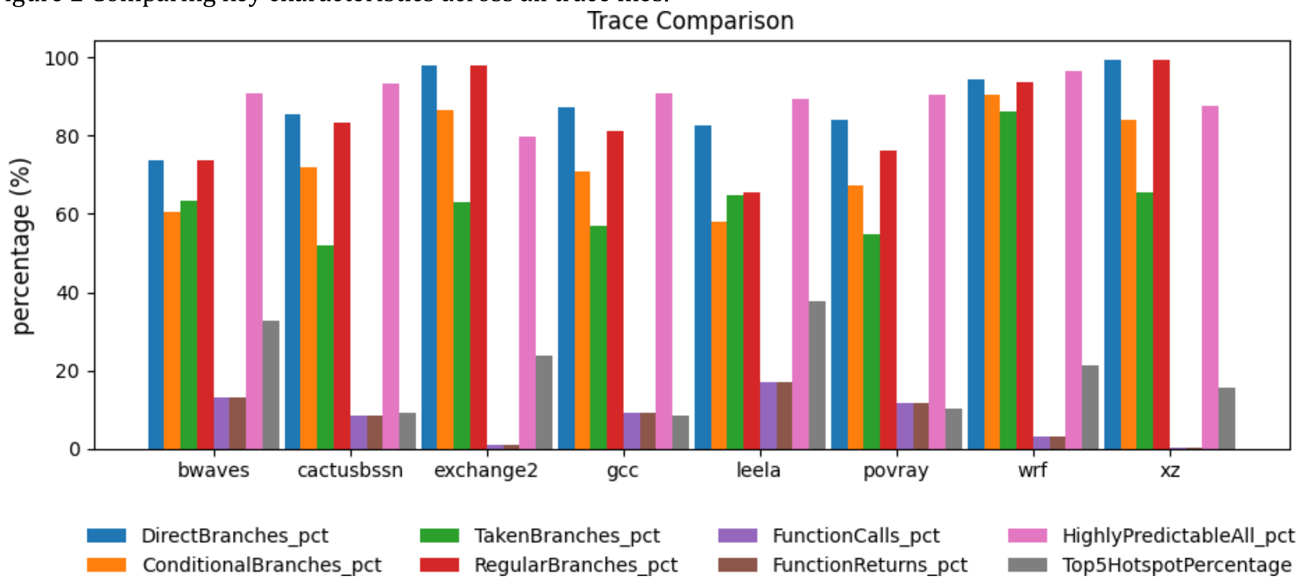
- Branch Taken Percentage: The proportion of all branches that are taken.
- Conditional Branch Percentage: The proportion of all branches that are conditional
- Branch Predictability: Percentage of unique branch locations that are highly predictable (taken >95% or <5% of the time)
- Hotspot Concentration: Most frequent executed branches locations.
- Branch Taken Patterns: How branch outcomes repeat (e.g., "TTTT" means 4 consecutive taken branches, "NNNN", 4 consecutive not taken branches, "XXXX" means branches at different locations)
- Branch PC Patterns: How branch locations repeat (e.g., "SSSS" means same PC repeated 4 times; "SDSD" means alternating between two PCs)

Table 1 summarises key characteristics of all trace files based on our analysis, Figure 2 show the visualisation of comparing key characteristics across all trace files, all analysis results saved in results/*.csv files.

Table 1 Trace File Characteristics

Trace	Taken Branches (%)	Conditional Branches (%)	Branch Predictability (%)	Hotspot Concentration (%)	Top Taken Pattern	Top PC Pattern
bwaves	63.43	60.57	90.97	32.74	XXXX (91.23%)	DDDD (94.71%) Different locations
cactusbssn	51.93	72.06	93.27	9.08	XXXX (80.51%)	DDDD (85.99) Different locations
exchange2	63.07	86.4	79.83	23.73	XXXX (43.09%)	DDDD (50.85%) Diverse, unpredictable
gcc	57.01	70.7	90.98	8.43	XXXX (82.03%)	DDDD (87.25) Different locations
leela	64.74	58.12	89.39	37.78	XXXX (60.93%)	DDDD (78.82) Moderately varied
povray	54.9	67.27	90.52	10.39	XXXX (94.87%)	DDDD (96.44) Different locations
wrf	86.07	90.37	96.38	21.44	TTTT (72.93%)	SSSS(65.91) Consecutive taken branches
xz	65.44	84.21	87.67	15.57	XXXX (90.78%)	DDDD (92.25) Different locations

Figure 2 Comparing key characteristics across all trace files.



Based on this analysis, I selected four traces that represent diverse branch behaviours:

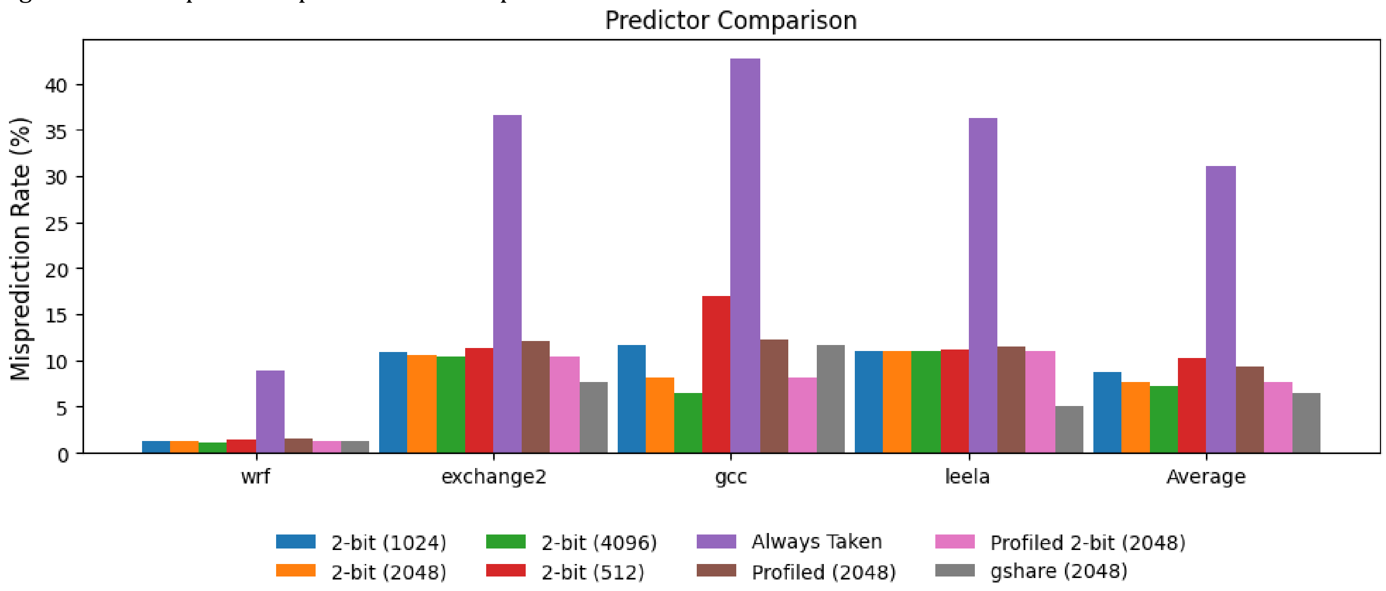
1) **wrf**: wrf has the highest conditional branch rate (90.37%), and the highest predictability (96.38%), and it has unique taken pattern TTTT, and unique pc pattern SSSS. These show wrf should be highly regular, and loop-oriented code. Simple predictors should have good performance on wrf.
2) **exchange2**: exchange2 has the lowest predictability (79.83%), and the most diverse pattern distribution. It should be the most challenging trace for prediction, and could be a good trace to test predictor's resilience to unpredictable patterns, and can contrast with wrf.
3) **gcc**: gcc has very low hotspot concentration (8.43%), and it has many unique branch locations. The widely spread branch location could be a good test for BHT table size, and the performance when meeting conflicts.
4) **leela**: leela has highest hotspot concentration(37.78%), and moderate pattern diversity. It can be a contrast test trace with gcc.

These four traces can complement each other. wrf and exchange2, they have similar branch rates but opposite predictability profiles. gcc and leela have similar predictability but opposite hotspot concentration. This selection ensures evaluation covers the different branch behaviors, and can help make the experiments more comprehensive.

4. Results & Analysis

4.1 Overall Predictor Performance

Figure 3 Overall predictor performance comparison



The experimental evaluation involved testing eight different branch predictors across four carefully selected trace files. Each predictor was evaluated on three million branches from each trace. Figure 3 summarizes the misprediction rates for all predictors across all traces. The results reveal significant variations in predictor performance across different traces, highlighting the importance of matching branch prediction strategies to program characteristics.

Table 2 Overall predictor performance comparison

	2-bit (512)	2-bit (2048)	2-bit (4096)	2-bit (1024)	Always Taken	Profiled (2048)	Profiled 2-bit (2048)	gshare (2048)
wrf	1.37	1.24	1.1	1.25	8.96	1.61	1.23	1.31
exchange2	11.27	10.5	10.35	10.81	36.64	12.08	10.49	7.59
gcc	16.94	8.09	6.49	11.64	42.65	12.3	8.09	11.64
leela	11.18	11	10.97	11.02	36.28	11.44	10.99	5.12
Average	10.19	7.7075	7.2275	8.68	31.1325	9.3575	7.7	6.415

4.2 Performance by Trace Type

wrf

The wrf trace demonstrates remarkably high branch predictability, all complex predictors achieve excellent performance (1-2% misprediction). Even the simplest predictor (Always Taken) performs reasonably well at 8.96%. Minimal variation between different predictor types and table sizes

This behavior aligns with earlier analysis, wrf has the highest predictability, and a distinctive TTTT pattern , which shows many same loop during runtime. The very low table size sensitivity suggests few unique branch locations with highly consistent behaviour.

exchange2

The exchange2 trace presents a more challenging case, gshare significantly outperforms all other predictors at 7.59%. An obvious gap exists between gshare and the next best predictor (2-bit 4096) at 10.35%. And table size for 2-bit predictors is moderate sensitive.

These results confirm analysis before of exchange2 having the lowest predictability among the traces. The better performance of gshare indicates the presence of correlated branch patterns that benefit from global history information, and this is consistent with exchange2's diverse pattern distribution.

gcc

The gcc trace shows the strongest sensitivity to predictor characteristics. When increase table size is increased, the accuracy has a large improvement (16.94% → 6.49%), and the 2-bit (4096) performs best at 6.49%. But surprisingly, gshare (2048) underperforms at 11.64% compared to 2-bit (2048) at 8.09%.

That may because gcc has the lowest hotspot concentration (8.43%), indicating numerous unique branch locations. The limited effectiveness of gshare suggests that gcc's branches exhibit stronger local patterns than global correlations.

leela

The leela trace shows a distinctive pattern, gshare outperforms other predictors at 5.12%. And it has very limited sensitivity to table size in 2-bit predictors. All other predictors cluster around 11% misprediction rate.

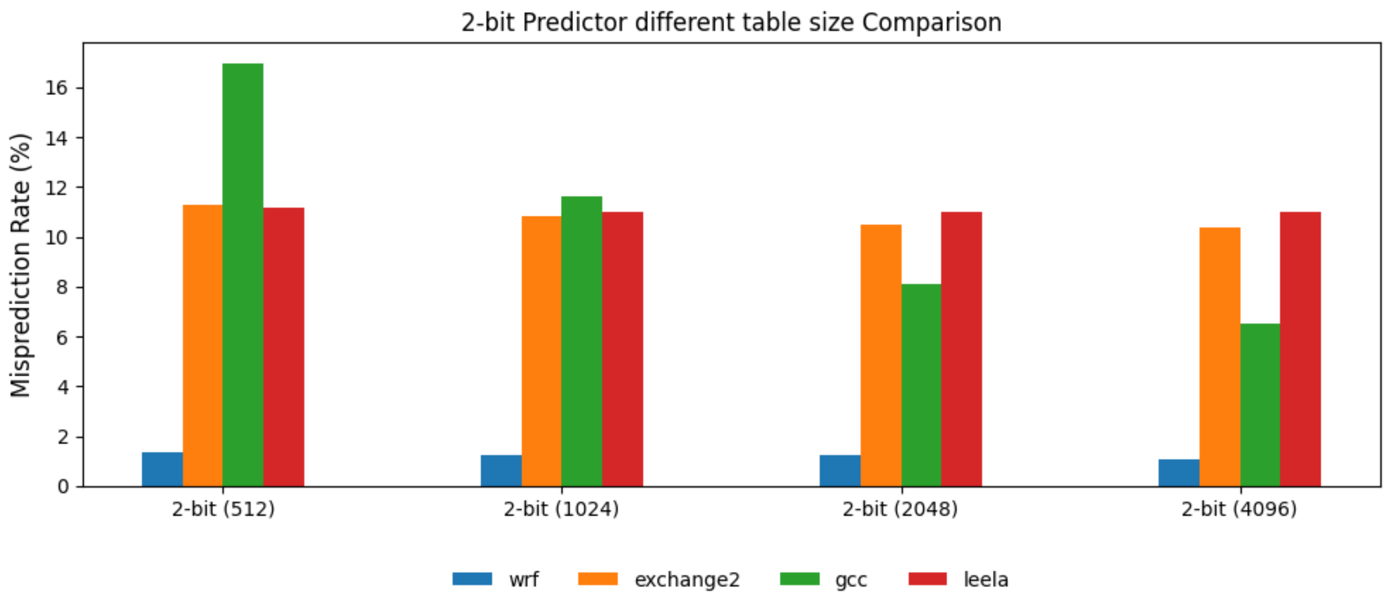
This behavior corresponds to leela's high hotspot concentration but with complex branch interdependencies. The global history information captured by gshare appears crucial for predicting branch behavior in this kind of application, suggesting that branches are strongly influenced by the program's current state or phase.

4.3 Analysis of Predictor Performance

Table Size Impact

Figure 4 shows how 2-bit predictor performance varies with table size across the four traces. gcc shows the most significant improvement (10.45% from 512 to 4096 entries), exchange2 also shows some improvement (0.92%). The variation correlates with branch diversity in each program, higher concentration less improve by increasing table size.

Figure 4 2-bit Predictor different table size comparison



Global vs. Local History

The performance gap between gshare and 2-bit predictors reveals insights about branch correlation patterns. leela and exchange2 show significant benefit from global history (5.88 and 2.91 percentage point improvements), but gcc shows significant regression with global history (3.55 percentage points).

These differences highlight the varying nature of branch correlations. Programs with interdependent decision-making benefit from global history, while programs with independent, localized decisions rely more on local patterns.

Profiled Predictors Evaluation

The two profiled predictors show interesting behavior:

Basic Profiled Predictor as a fixed predictor, same as always taken, but its performance much better than always taken, this is because branches usually make the same decision, it benefits from pre-profiling a lot. But it consistently underperforms compared to the 2-bit predictor, and its performance gap ranges from 0.37 (wrf) to 4.21 (gcc) percentage points. That is because it is limited by its inability to adapt to changing branch behavior during execution.

The profiled 2-bit predictor performs almost same to the standard 2-bit predictor, but it is always better about 0.01%, suggests that the initial state derived from profiling converges to the same state as standard 2-bit prediction. But it helps skip the cold start step.

The results indicate that simple profiling approaches provide limited benefit for these traces. The basic profiled predictor's non-adaptive design proves particularly disadvantageous for complex programs like gcc, while the 2-bit profiled predictor essentially replicates standard 2-bit behavior after sufficient execution.

5. Conclusion

Generally, the effectiveness ranking of all test predictors is gshare > 2-bit (large tables) > Profiled 2-bit > 2-bit (small tables) > Profiled > Always Taken. However, this hierarchy varies significantly by trace type.

Different programs benefit from different predictor characteristics. For programs that have regular patterns like wrf, simple predictors are sufficient due to regular patterns. For programs that have diverse branches, they have more unique branch locations, and require large prediction tables like gcc. But for state-dependent branches like leela, global history is crucial. When facing more challenges, more diverse and less predictable scenarios, consider both global history and reasonable table sizes may give better performance.

Even the best predictors face a predictability ceiling for certain programs. Exchange2 shows a minimum ~7.5% misprediction rate despite sophisticated prediction techniques. Simple profiling approaches provide limited benefits for the traces examined. The basic Profiled predictor's performance is limited by its non-adaptive design, while the Profiled 2-bit predictor converges to standard 2-bit behaviour.

The effectiveness of each predictor varies significantly across different program types. This suggests that there may be benefits to developing specialized prediction strategies. It also points to opportunities for compiler optimizations that target specific branch patterns