Branch Predictor Selection for Complex Workloads

Singi Maharshi (25CS06021)　　Oliver Kandir (25CS06006)

Department of Computer Science and Engineering
Indian Institute of Technology Bhubaneswar

Course: Advanced Computer Architecture Lab
Prof. Dr. Devashree Tripathy

Contents

Abstract . . . 1
0.1 Introduction . . . 1
Key Tasks . . . 1
Expected Outcome . . . 2
0.2 Evaluation Parameters . . . 2
0.2.1 Branch Misprediction Rate . . . 2
0.2.2 CPI (Cycles Per Instruction) . . . 2
0.2.3 IPC (Instructions Per Cycle) . . . 2
0.2.4 Total Cycles . . . 3
0.2.5 Static vs. Dynamic Predictors . . . 3
0.2.6 Optimization Level (-O2) . . . 3
0.3 Background and Tools . . . 3
0.3.1 Description of `-O2` Optimization Flag . . . 3
0.3.2 Description: SimpleScalar . . . 4
0.3.3 Description of `sim-outorder` . . . 4
0.4 Branch Predictors Evaluated . . . 4
0.4.1 Taken Predictor . . . 4
0.4.2 Not-Taken Predictor . . . 4
0.4.3 Bimodal Predictor . . . 4
0.4.4 Two-Level Adaptive Predictor . . . 4
0.4.5 Combined (Hybrid) Predictor . . . 5
0.5 Benchmark Programs . . . 5
0.6 Experimental Setup and Commands . . . 12
0.7 Results and Analysis . . . 12
Aggregate Predictor Performance (Across All Programs) . . . 22
0.8 Discussion . . . 24
0.9 Conclusion . . . 24
References . . . 25

Abstract

This report evaluates multiple branch predictors available in the SimpleScalar sim-outorder simulator across compute-intensive benchmarks designed to reflect control-flow patterns commonly seen in large-scale machine learning workloads. We collect misprediction rate, CPI, IPC, and total cycles for taken, not-taken, bimodal, two-level, and combined (hybrid) predictors, and recommend a balanced predictor configuration for high-compute environments. Results consistently show the hybrid (comb) predictor offering the best trade-off between accuracy and performance.

0.1 Introduction

Proposed Project Title

"Comprehensive Evaluation and Recommendation of Branch Predictors for High-Compute Workloads using SimpleScalar"

Problem Statement

Modern processors increasingly execute workloads that resemble the computational patterns of Large Language Models (LLMs), including matrix multiplications, attention-like operations, expert routing, normalization, softmax evaluation, sampling, and gradient-style updates.

This project evaluates the branch prediction performance of multiple predictors in the SimpleScalar `sim-outorder` simulator using seven custom C programs, each designed to mimic a specific component of LLM computation. These programs collectively generate diverse branch patterns—ranging from highly predictable arithmetic loops to stochastic sampling, threshold-based gating, and deeply nested routing logic.

The objective is to analyze how different branch predictors respond to these compute-intensive and irregular control-flow patterns, and to identify the most balanced predictor that offers the best trade-off between prediction accuracy and overall execution performance (CPI, IPC, and runtime) when running LLM-like workloads.

Key Tasks

1. Configure and execute `sim-outorder` with multiple branch predictors (bimodal, static, 2level, hybrid, etc.) on 7 benchmarks.
2. Collect detailed metrics—misprediction rate, CPI, IPC, and total cycles.
3. Analyze results quantitatively and produce a combined performance score for each predictor.
4. Recommend the most balanced predictor for high computational workloads.

Expected Outcome

A data-driven recommendation identifying the best branch predictor configuration for compute-intensive programs, supported by experimental analysis and visualization.

0.2 Evaluation Parameters

To evaluate the performance of various branch prediction techniques (taken, not-taken, bimodal, 2-level, and comb), several microarchitectural and performance parameters were measured using the SimpleScalar `sim-outorder` simulator. These parameters reflect key aspects of processor efficiency, instruction throughput, and control-flow prediction accuracy.

0.2.1 Branch Misprediction Rate

Definition: The fraction of branch instructions that were incorrectly predicted by the branch predictor.

$$\text{Misprediction Rate} = \frac{\text{Number of Incorrect Predictions}}{\text{Total Branch Instructions}}$$

Interpretation: A lower misprediction rate indicates a more accurate predictor. Fewer mispredictions reduce pipeline flushes, leading to better performance.
Use in project: Compared across all predictors to determine their relative prediction accuracy for different program behaviors.

0.2.2 CPI (Cycles Per Instruction)

Definition: The average number of clock cycles required to execute each instruction.

$$\text{CPI} = \frac{\text{Total Cycles}}{\text{Number of Instructions Executed}}$$

Interpretation: Lower CPI values indicate higher efficiency and reduced pipeline stalls.
Relation to predictors: High branch mispredictions increase CPI due to speculative execution rollbacks. Thus, predictors with lower misprediction rates (e.g., comb) show lower CPI.

0.2.3 IPC (Instructions Per Cycle)

Definition: The number of instructions completed per clock cycle.

$$\text{IPC} = \frac{\text{Instructions Executed}}{\text{Total Cycles}}$$

Interpretation: Higher IPC values indicate better instruction throughput and pipeline utilization. It is inversely related to CPI.

Use in project: Used to measure how effectively the processor issues and retires instructions under different branch prediction schemes.

0.2.4 Total Cycles

Definition: The total number of clock cycles taken to complete execution of a benchmark program.
Interpretation: Indicates the overall execution time (in simulated cycles). Lower total cycles imply better performance and higher efficiency.
Use in project: Served as a primary performance metric for comparing overall runtime impact of different branch predictors.

0.2.5 Static vs. Dynamic Predictors

Static Predictors (taken, not-taken): Assume fixed branch outcomes for all executions—useful as baseline references. High misprediction rates show their inability to adapt to dynamic program flow.
Dynamic Predictors (bimod, 2lev, comb): Learn branch behavior based on runtime history. These significantly reduce misprediction rates and improve CPI/IPC.

0.2.6 Optimization Level (-O2)

All benchmark programs were compiled using the `-O2` optimization level of the SimpleScalar GCC toolchain, ensuring realistic and performance-optimized binaries for accurate branch prediction analysis.

0.3 Background and Tools

0.3.1 Description of `-O2` Optimization Flag

The `-O2` flag enables a balanced set of performance optimizations that improve runtime efficiency without significantly increasing compile time or binary size. It includes instruction-level optimizations (common subexpression elimination, constant propagation, strength reduction, dead code elimination), loop optimizations (partial unrolling, invariant code motion, induction variable simplification), branch/function optimizations (predictable branch rearrangement, small function inlining, cross-jumping, tail calls), and register allocation improvements (better register usage and instruction scheduling). In SimpleScalar context, compiling with:

```
sslittle-na-sstrix-gcc -O2 <program>.c -o <program>
```

improves IPC, reduces stalls and mispredictions, and provides realistic workloads for experiments.

0.3.2 Description: SimpleScalar

SimpleScalar is a computer architecture simulation toolset for modeling and evaluating modern microprocessors at the instruction level. It supports multiple simulation modes (`sim-safe`, `sim-cache`, `sim-bpred`, `sim-outorder`), configurable microarchitectural parameters, and detailed statistics (CPI, IPC, miss rates, cycles). It is widely used for pipeline performance, branch prediction, cache behavior, and ILP studies.

0.3.3 Description of `sim-outorder`

`sim-outorder` models a speculative, out-of-order superscalar pipeline with realistic components: fetch/decode/issue/commit stages; out-of-order execution; RUU/LSQ; speculative execution with branch prediction (taken, nottaken, bimod, 2lev, comb); and cache/memory hierarchy. It reports cycle-accurate performance statistics: CPI, IPC, executed instructions, and misprediction rate.

0.4 Branch Predictors Evaluated

In this project, five predictors available in SimpleScalar were evaluated: taken, not-taken, bimodal, two-level adaptive (2lev), and combined (hybrid).

0.4.1 Taken Predictor

A static predictor that always predicts branches as taken; good for backward loop branches, poor for rarely-taken forward branches. Predicts every branch will jump. Very poor for conditional exits or forward jumps.
Limitation: High misprediction on complex logic.

0.4.2 Not-Taken Predictor

A static predictor that always predicts not-taken; simple but suffers on loop-heavy programs.
Use Case: Acts as a performance baseline, not used in modern CPUs.

0.4.3 Bimodal Predictor

A dynamic predictor with a single-level table of 2-bit saturating counters (PHT) per branch; adapts to per branch local behavior.
Strength: Works well when branches are consistent.
Limitation: Struggles with patterns like alternating branches (e.g., 010101...).

0.4.4 Two-Level Adaptive Predictor

Uses a history register (branch outcomes) as the first level and a PHT indexed by this history as the second level; captures inter-branch correlations.

Can be **global**, **local**, or **combined**:
Global: One shared history for all branches.
Local: Each branch has its own history.
Strength: Catches correlated branches and sequences.
Limitation: More complex; can be sensitive to history length and aliasing.

0.4.5 Combined (Hybrid) Predictor

Combines local and global behavior (e.g., bimodal + two-level) with a meta-predictor to choose the best source per branch; highest accuracy with modest hardware cost.
Strength: Best average performance across diverse programs.
Limitation: Higher hardware complexity and power consumption.

0.5 Benchmark Programs

Per-program characterization and relation to Large Machine Learning Models (LMMs): This section explains each benchmark program used in the project, its relationship to a component of Large Machine Learning Models (LMMs), and the branch behavior it ex-hibits. These micro-programs mimic the control-flow patterns of deep-learning workloads such as Transformers, Mixture-of-Experts models, and optimizer steps.

Program 1 — Matrix Multiply + Piecewise Activation

LLM Correspondence:
Simulates the **Feed-Forward Neural Network (FFN)** layer of Transformers, where every token embedding is processed through multiple matrix multiplications and activation functions (ReLU, GELU).

What it does:
Initializes two 256×256 matrices using sinusoidal patterns, performs full matrix multiplication, and applies a multi-branch piecewise activation (tanh, log, square, linear, sinusoidal) followed by small modulo-based adjustments before storing final outputs.

Detailed Operation:

i. Initializes matrices `A` and `B` with trigonometric expressions to produce structured numeric patterns.

ii. Performs complete 256×256 matrix multiplication by computing dot products for each (i, j) position.

iii. Applies a piecewise activation function to the computed sum, choosing among:

   - `tanh()` for large positive values,
   - `log()` for moderate positives,

  - squaring for small positives,
  - linear scaling for small negatives,
  - `sin()` for large negatives.

iv. Adds deterministic adjustments based on modulus operations to slightly modify selected outputs.

v. Writes the final processed value into matrix `C` at the corresponding location.

Branch behavior:
Regular nested loops but highly data-dependent activation branches; the piecewise nonlinearity and modulus checks create non-uniform branch patterns. Useful for analyzing branch predictor behavior where dynamic predictors outperform static heuristics.

Program 2 — Sparse Attention (Mask + Gating)

LLM Correspondence:
Mimics the Attention Mechanism used in Transformers. Real LLMs compute attention weights and apply masks to ignore irrelevant tokens.

What it does:
Builds Q/K/V matrices, computes attention scores, and applies threshold-based masks and gating with modular conditions before combining with V.

Detailed Operation:

i. Initializes the Query (`Q`), Key (`K`), and Value (`V`) matrices using sinusoidal and trigonometric expressions to generate structured patterns.

ii. Computes dot products between each row of `Q` and each row of `K`, forming pairwise similarity scores.

iii. Scales each attention score by dividing with $\sqrt{N}$, following the standard scaled dot-product attention formulation.

iv. Applies a piecewise transformation to each score:
  - Sigmoid-like mapping for large positives,
  - Exponential amplification for large negatives,
  - Linear scaling for intermediate values.

v. Modifies attention scores through deterministic gating rules based on indices and modular conditions (`i % 7 == 0`, `j % 5 == 0`, etc.).

vi. Combines the transformed attention value with the corresponding element of the `V` matrix to produce the output.

Branch behavior:
Mix of data-dependent thresholds and periodic modular branches (e.g., i%7, j%5). Produces interleaved predictable and unpredictable outcomes, stressing predictor adaptability; hybrid predictors typically perform best.

Program 3 — Mixture-of-Experts Router

LLM Correspondence:

Represents the **Mixture-of-Experts (MoE)** architecture used in large LLMs (e.g., Switch Transformer, GPT-MoE). Each input token is dynamically routed to one or more experts.

What it does:

Routes tokens to experts through a multi-threshold decision tree and applies balancing logic using divisibility or modulus conditions.

Detailed Operation:

i. Initializes the expert weight array and generates `N` random token values within the range 0–999.

ii. Normalizes each token value using `val = route[i] / 1000.0`, preparing it for threshold-based expert assignment.

iii. Routes each token to one of eight experts by evaluating a descending sequence of threshold conditions that divide the value range into routing intervals.

iv. Updates the corresponding expert counter using a set of rules:

   - increment by 1 when expert index is even and token value is divisible by 3,
   - increment by 2 for experts divisible by 3,
   - increment by 3 for experts divisible by 5,
   - otherwise increment by `route[i] % 7`.

v. Applies load-balancing adjustments by reducing the counter when `val > 0.8` and the counter satisfies a modulus-11 condition.

vi. Prints the final number of processed tokens for each expert, reflecting workload distribution.

Branch behavior:

The deep nested if-else routing logic simulates real MoE decision-making. This stresses **correlating and hybrid predictors**, revealing their ability to handle long branch dependencies. Excellent for evaluating how well 2-level or hybrid predictors track correlated branch outcomes.

Program 4 — LayerNorm + Clipping

LLM Correspondence:

Layer Normalization, used after attention and feed-forward layers to stabilize activations. Calculate the mean and variance for the embedding of each token.

What it does:

Computes the mean and variance per-row, normalizes values, conditionally clips to ± 3, and enhances very small magnitudes.

Detailed Operation:

i. Initializes the `X` matrix with deterministic pseudo-random values and prepares the learnable parameters `gamma` and `beta`.

ii. Computes the mean of each row by averaging all feature values across the `DIM`-dimension.

iii. Computes the variance of each row and adds a small ϵ term to ensure numerical stability before taking the square root.

iv. Normalizes each element using the standard LayerNorm formula:

$$\text{norm} = \frac{X[i][j] - \text{mean}}{\text{std}}$$

v. Applies conditional modifications:

   i. clip values above 3 or below -3,

   ii. amplify very small magnitudes ($|\text{norm}| < 0.01$) by a factor of 1.5.

vi. Applies learnable affine transformation to each value using:

$$Y[i][j] = \gamma[j] \cdot \text{norm} + \beta[j]$$

vii. Applies a small deterministic correction to selected elements using a modulus-based rule.

Branch behavior:

Skewed conditions (most values remain unmodified), resulting in highly predictable branches. The bimodal predictor performs almost as well as a hybrid predictor in such low-entropy scenarios.

Program 5 — Stable Softmax + Clamps

LLM Correspondence:

Simulates **Softmax computation** in attention and output probability layers. Normalizes logits using exponentials and division.

What it does:

Performs max-subtraction, exponentiation with saturation limits, optional mod-based perturbation, normalization, and final probability capping.

Detailed Operation:

i. Initializes the `logits` matrix with deterministic pseudo-random values across the `BATCH` rows and `DIM` columns.

ii. For each row, computes the maximum logit value and subtracts it from every element. This implements the standard numerical-stability trick for softmax:

$$x'_j = x_j - \max_k(x_k),$$

which prevents overflow when computing exponentials.

iii. Computes exponentials of the shifted logits:

$$\text{expv}_j = e^{x'_j},$$

and applies stability rules:

   i. clamp excessively large exponentials to 10^3,
   ii. lift extremely small exponentials to 10^{-5},
   iii. apply a 0.95 scaling when the modulus condition on expv_j is satisfied.

iv. Accumulates the sum of all exponentials in the row:

$$S = \sum_k \text{expv}_k.$$

v. Normalizes each exponential using the softmax formula:

$$p_j = \frac{\text{expv}_j}{S},$$

producing a valid probability distribution where $\sum_j p_j = 1$.

vi. Applies final probability clamping rules:

   – cap values above 0.9 to avoid overly dominant probabilities,
   – raise very small probabilities by adding 0.001 to avoid zero-like outputs.

vii. Stores the normalized and clamped probabilities into the `probs` matrix for later use.

Branch behavior:

Mostly repetitive and consistent range-check branches with rare flips caused by mod-based perturbations. Hybrid predictors exhibit minimal mispredictions due to the stable and predictable branching pattern.

Program 6 — Top-k Sampler (Decoding)

LLM Correspondence:

Represents the **Token Sampling (Decoding)** phase in LLM inference, where the next token is chosen from probability distributions using randomization (top-k or nucleus sampling).

What it does:

Maintains a top-k array of scores, randomly selects an index, and applies bounds checks and guard conditions before returning the sample.

Detailed Operation:

i. Initializes the `logits` matrix with deterministic pseudo-random values across all `BATCH` rows and the `VOCAB` dimension.

ii. For each row, creates arrays `topk[]` and `topk_idx[]` initialized to a very low value to hold the highest K logits and their corresponding token indices.

iii. Iterates through all vocabulary logits and inserts each value into the correct position inside the top-k list using an in-place shifting mechanism:
   - compare the logit with the current top-k entries,
   - shift lower-ranked entries downward,
   - place the new value into its appropriate position,
   - maintain the list in descending order.

   This manually constructs the K largest logits without sorting the entire vocabulary.

iv. Generates a uniform random number and maps it to an integer range $[0, K-1]$, selecting one of the top-k candidates as the sampled index.

v. Applies safety and guard checks:
   - returns -1 if the selected top-k logit is too low,
   - returns a halved token index if the value is extremely high,
   - otherwise returns the index exactly as stored in `topk_idx[pick]`.

vi. Stores the final sampled token in the `sampled[]` array, representing the chosen token for that batch element.

Branch behavior:

Randomized conditions introduce stochastic, unpredictable branches, while deterministic comparisons remain stable. This combination stresses predictor robustness under partial randomness; hybrid predictors maintain consistent accuracy.

Program 7 — Gradient Update with Adaptive Learning Rate

LLM Correspondence:

Emulates the **Gradient Descent Weight Update** process during LLM training, applying adaptive learning rate updates per weight.

What it does:

Updates model weights based on gradient magnitude, dynamically adjusts the learning rate, and applies a periodic nudge when `idx % 13 == 0`.

Detailed Operation:

i. Initializes the weight matrix `W` and gradient matrix `grad` with deterministic pseudo-random values across `LAYERS` and `DIM`.

ii. Assigns a per-layer base learning rate using:

$$\texttt{lr[i]} = 0.001 + (i \bmod 5) \times 0.0001,$$

creating a repeating 5-step learning rate schedule.

iii. For each weight, reads the gradient value `g = grad[i][j]` and adjusts the learning rate adaptively:

   – multiply learning rate by 0.9 when $|g| > 1.0$,

   – multiply learning rate by 1.05 when $|g| < 0.01$.

   This models simplified adaptive learning rate behavior similar to optimizers like Adam or RMSProp.

iv. Updates each weight using gradient descent:

$$W[i][j] \leftarrow W[i][j] - \texttt{lr[i]} \cdot g.$$

v. Applies a periodic nudging rule:

$$\text{if } (\lfloor W[i][j] \times 1000 \rfloor \bmod 13 = 0) \Rightarrow W[i][j] += 0.0001.$$

This introduces a small deterministic perturbation to selected weights.

vi. Stores the updated weight back into the `W` matrix, completing one full optimizer-style update pass.

Branch behavior:

Simple threshold branches (rarely triggered) combined with regular periodic checks lead to predictable, low-entropy control flow. Bimodal predictors perform well, with hybrid predictors offering only slight improvements for the infrequent branches.

0.6 Experimental Setup and Commands

General Commands

Compile C programs

```
$IDIR/bin/sslittle-na-sstrix-gcc -O2 -o <output_object_name> <program_name>.c
```

Analyze branch predictor

```
$IDIR/simplesim-3.0/sim-outorder -bpred <predictor_type> <program_name>
```

0.7 Results and Analysis

Program 1

Table 1: Program 1 Results

	taken	Not taken	bimod	2lev	comb
bpred_dir_rate	0.0228	0.0228	0.9971	0.9857	0.9971
bpred_miss_rate	0.9772	0.9772	0.0029	0.0143	0.0029
sim_num_insn	204094264	204094264	204094264	204094264	204094264
sim_num_branches	23005235	23005235	23005235	23005235	23005235
sim_CPI	2.1316	2.1329	1.7793	1.7864	1.7793
sim_IPC	0.4691	0.4689	0.5620	0.5598	0.5620
sim_cycle	435040743	435303059	363151013	364593223	363150982

Program 2

Table 2: Program 2 Results

	taken	Not taken	bimod	2lev	comb
bpred_dir_rate	0.0768	0.0768	0.9850	0.9724	0.9911
bpred_miss_rate	0.9232	0.9232	0.0150	0.0276	0.0089
sim_num_insn	228436498	228436498	228436498	228436498	228436498
sim_num_branches	26799905	26799905	26799905	26799905	26799905
sim_CPI	1.2112	1.2134	0.8400	0.8437	0.8344
sim_IPC	0.8257	0.8241	1.1905	1.1853	1.1984
sim_cycle	276671913	277184927	191878203	192723275	190615168

Program 3

Table 3: Program 3 Results

	taken	Not taken	bimod	2lev	comb
bpred_dir_rate	0.1087	0.1087	0.9915	0.9912	0.9917
bpred_miss_rate	0.8913	0.8913	0.0085	0.0088	0.0083
sim_num_insn	13862278	13862278	13862278	13862278	13862278
sim_num_branches	2007905	2007905	2007905	2007905	2007905
sim_CPI	1.5247	1.5247	0.5712	0.5716	0.5709
sim_IPC	0.6559	0.6559	1.7507	1.7496	1.7515
sim_cycle	21135820	21136172	7918175	7923035	7914355

Program 4

Table 4: Program 4 Results

	taken	Not taken	bimod	2lev	comb
bpred_dir_rate	0.2369	0.2369	0.9226	0.9211	0.9215
bpred_miss_rate	0.7631	0.7631	0.0774	0.0789	0.0785
sim_num_insn	13254474	13254474	13254474	13254474	13254474
sim_num_branches	1018226	1018226	1018226	1018226	1018226
sim_CPI	1.0364	1.0364	0.6808	0.6814	0.6816
sim_IPC	0.9649	0.9649	1.4688	1.4677	1.4672
sim_cycle	13736530	13736530	9024039	9031078	9034075

Program 5

Table 5: Program 5 Results

	taken	Not taken	bimod	2lev	comb
bpred_dir_rate	0.1500	0.1500	0.9258	0.9621	0.9748
bpred_miss_rate	0.8500	0.8500	0.0742	0.0379	0.0252
sim_num_insn	3560040	3560040	3560040	3560040	3560040
sim_num_branches	654899	654899	654899	654899	654899
sim_CPI	1.7840	1.7932	0.9260	0.8591	0.8499
sim_IPC	0.5605	0.5577	1.0799	1.1640	1.1766
sim_cycle	6351289	6383912	3296544	3058365	3025602

Program 6

Table 6: Program 6 Results

	taken	Not taken	bimod	2lev	comb
bpred_dir_rate	0.0262	0.0262	0.9748	0.9737	0.9748
bpred_miss_rate	0.9738	0.9738	0.0252	0.0263	0.0252
sim_num_insn	5568379	5568379	5568379	5568379	5568379
sim_num_branches	1336524	1336524	1336524	1336524	1336524
sim_CPI	2.2066	2.2066	0.4940	0.4954	0.4939
sim_IPC	0.4532	0.4532	2.0244	2.0186	2.0247
sim_cycle	12287143	12287379	2750614	2758555	2750176

Program 7

Table 7: Program 7 Results

	taken	Not taken	bimod	2lev	comb
bpred_dir_rate	0.0089	0.0089	0.9930	0.9915	0.9927
bpred_miss_rate	0.9911	0.9911	0.0070	0.0085	0.0073
sim_num_insn	1288630	1288630	1288630	1288630	1288630
sim_num_branches	67070	67070	67070	67070	67070
sim_CPI	1.2687	1.2688	0.8681	0.8684	0.8680
sim_IPC	0.7882	0.7881	1.1519	1.1516	1.1521
sim_cycle	1634935	1635064	1118679	1119005	1118541

Program 1 — Predictor Comparison (Graphical Analysis)

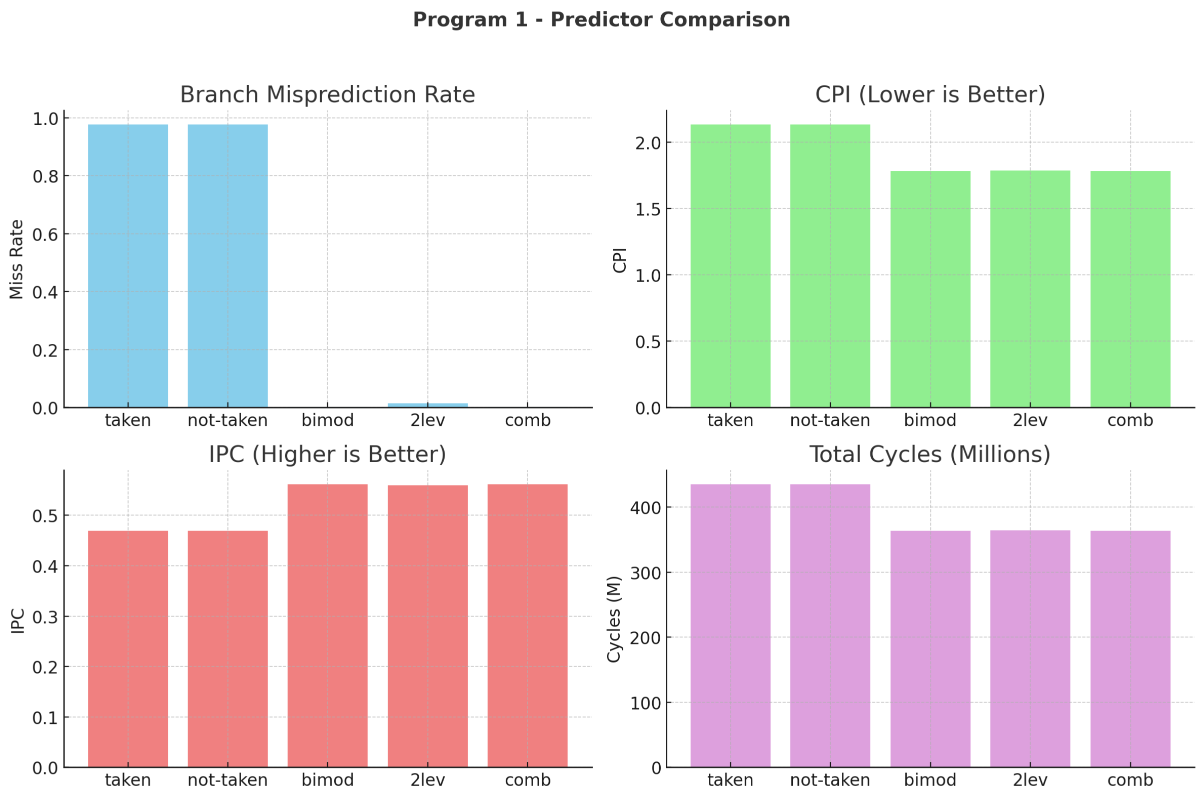

Figure 1: Program 1 — Predictor Comparison showing Branch Misprediction Rate, CPI, IPC, and Total Cycles.

Program 2 — Predictor Comparison (Graphical Analysis)

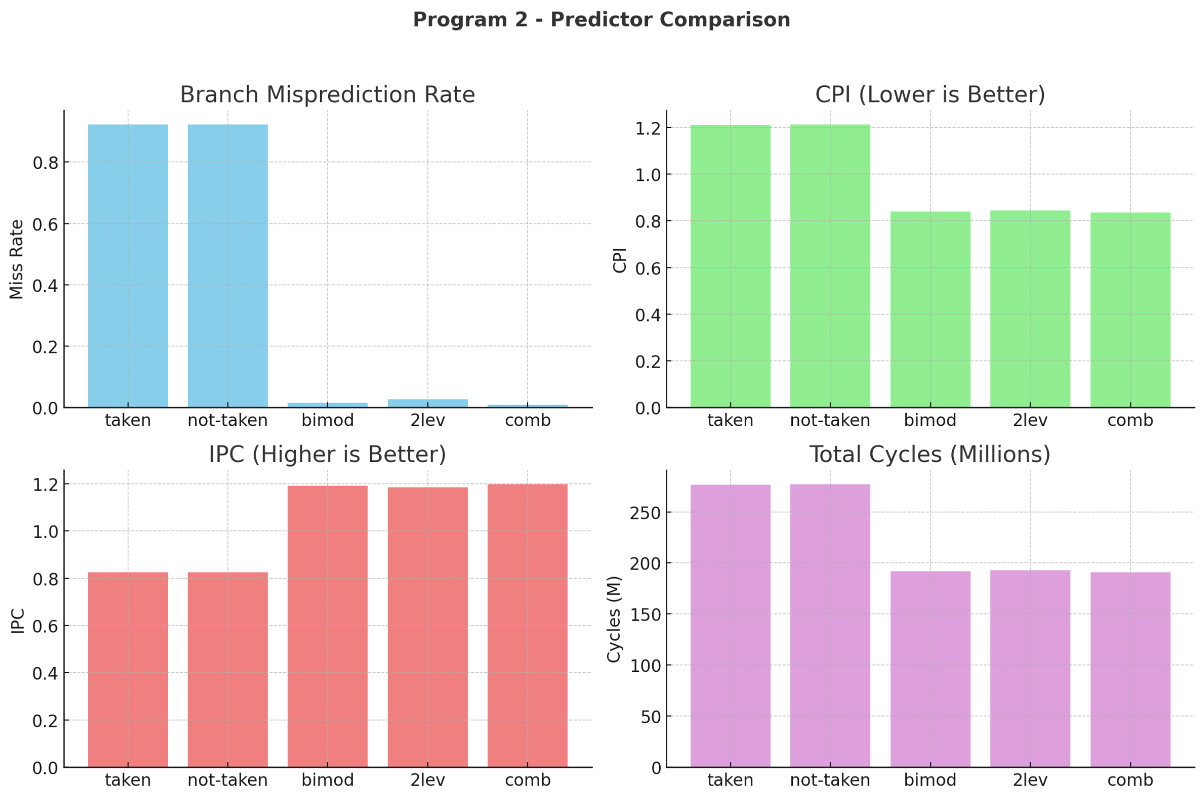

Figure 2: Program 2 — Predictor Comparison showing Branch Misprediction Rate, CPI, IPC, and Total Cycles.

Program 3 — Predictor Comparison (Graphical Analysis)

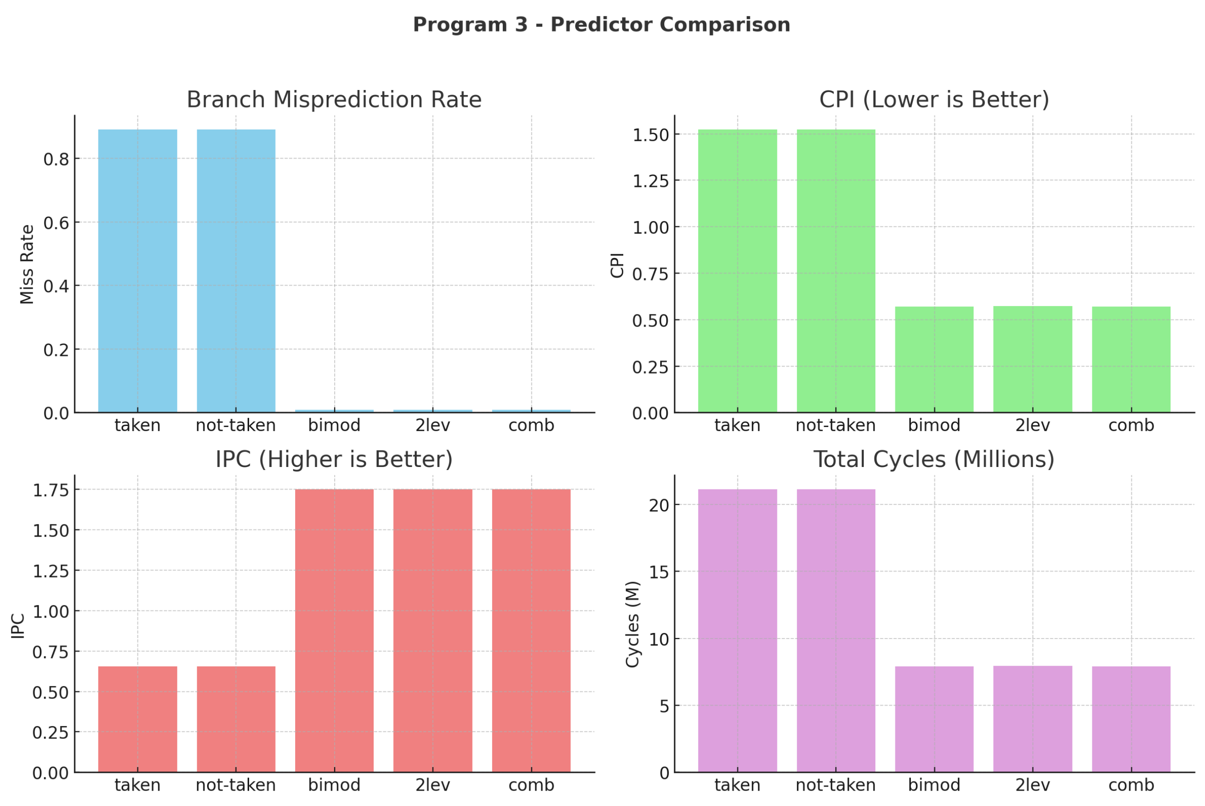

Figure 3: Program 3 — Predictor Comparison showing Branch Misprediction Rate, CPI, IPC, and Total Cycles.

Program 4 — Predictor Comparison (Graphical Analysis)

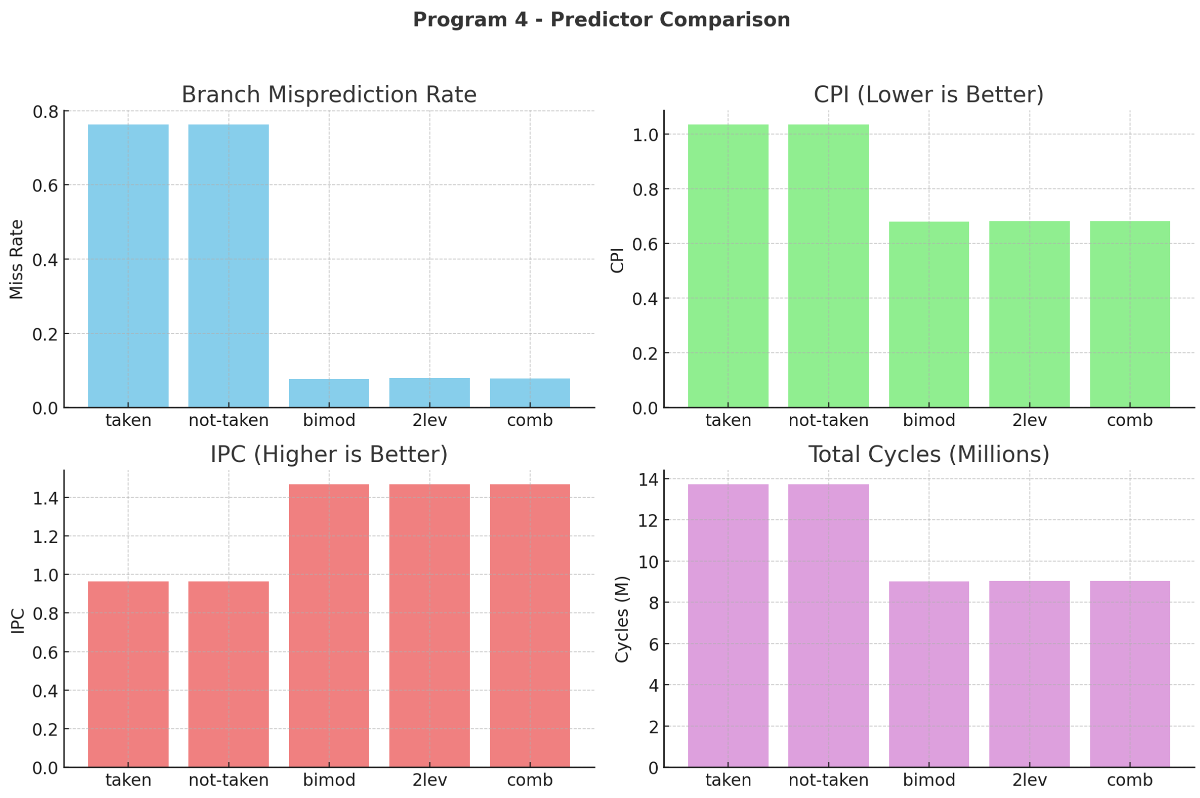

Figure 4: Program 4 — Predictor Comparison showing Branch Misprediction Rate, CPI, IPC, and Total Cycles.

Program 5 — Predictor Comparison (Graphical Analysis)

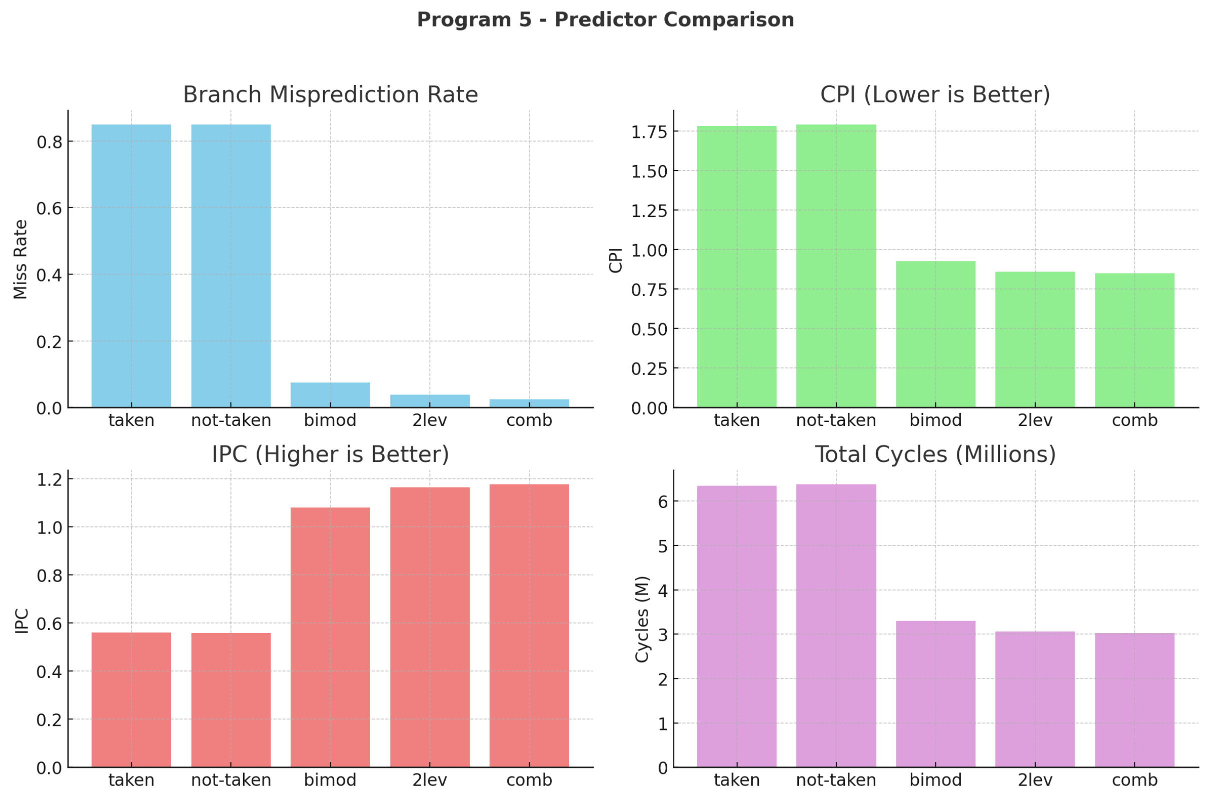

Figure 5: Program 5 — Predictor Comparison showing Branch Misprediction Rate, CPI, IPC, and Total Cycles.

Program 6 — Predictor Comparison (Graphical Analysis)

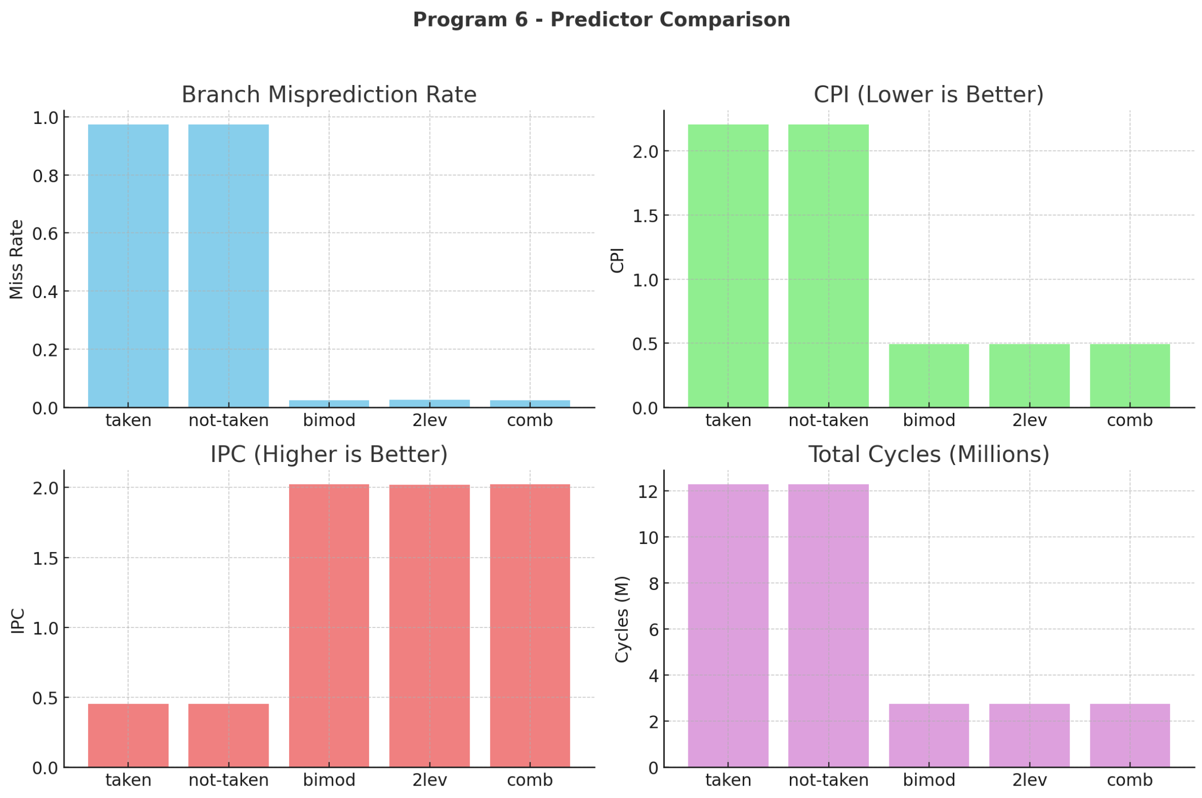

Figure 6: Program 6 — Predictor Comparison showing Branch Misprediction Rate, CPI, IPC, and Total Cycles.

Program 7 — Predictor Comparison (Graphical Analysis)

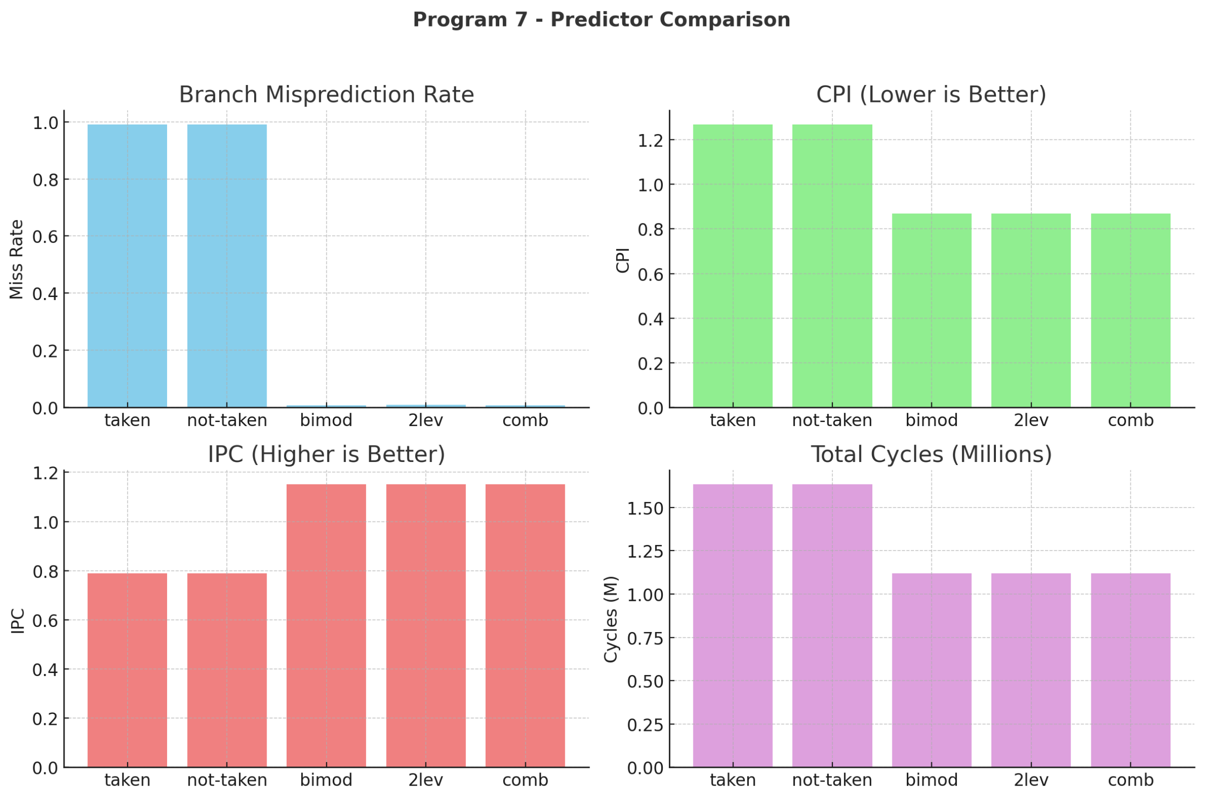

Figure 7: Program 7 — Predictor Comparison showing Branch Misprediction Rate, CPI, IPC, and Total Cycles.

Aggregate Predictor Performance (Across All Programs)

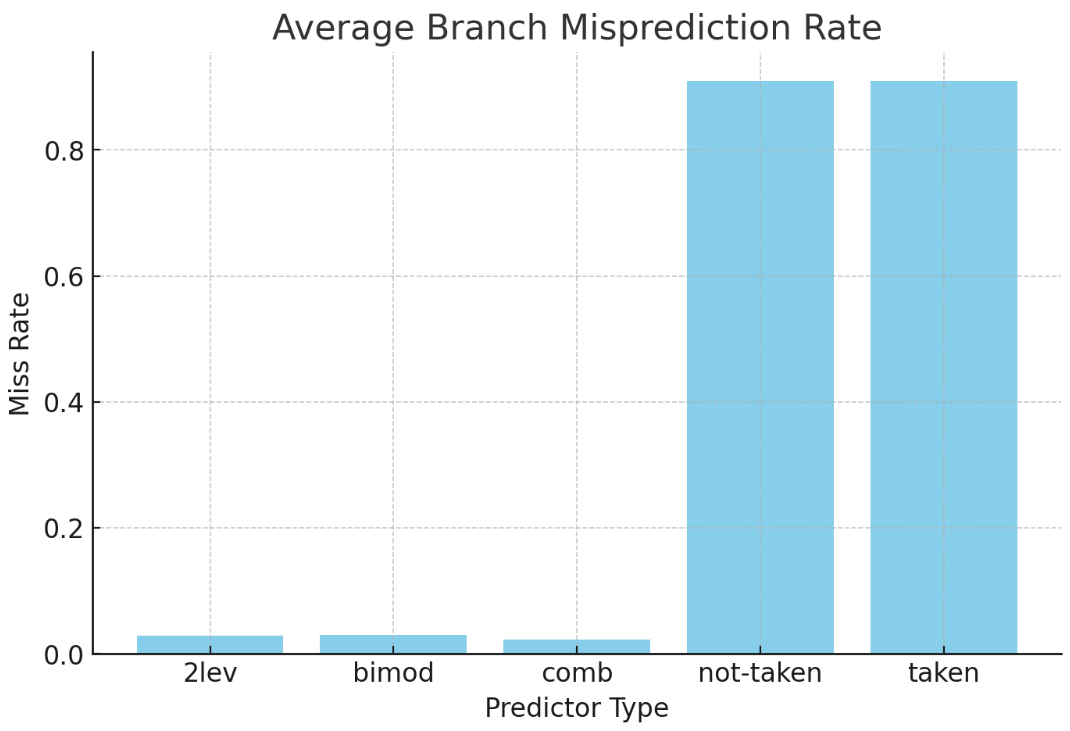

Figure 8: Average Branch Misprediction Rate across all predictors (Lower is Better).

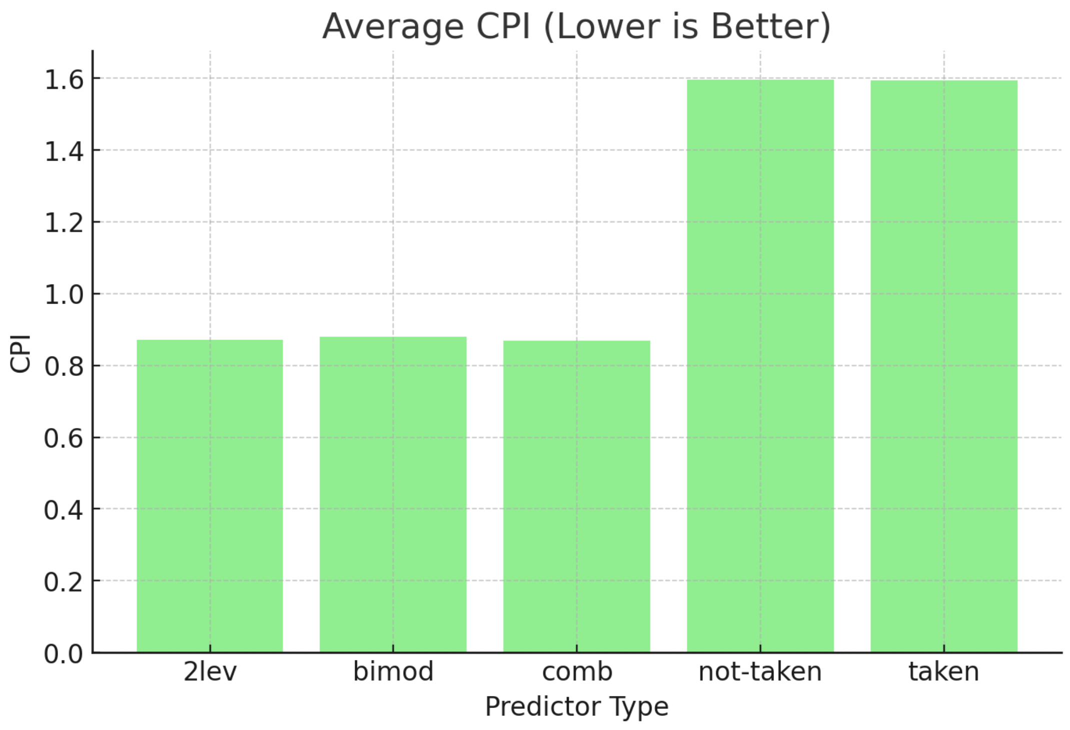

Figure 9: Average CPI across all predictors (Lower is Better).

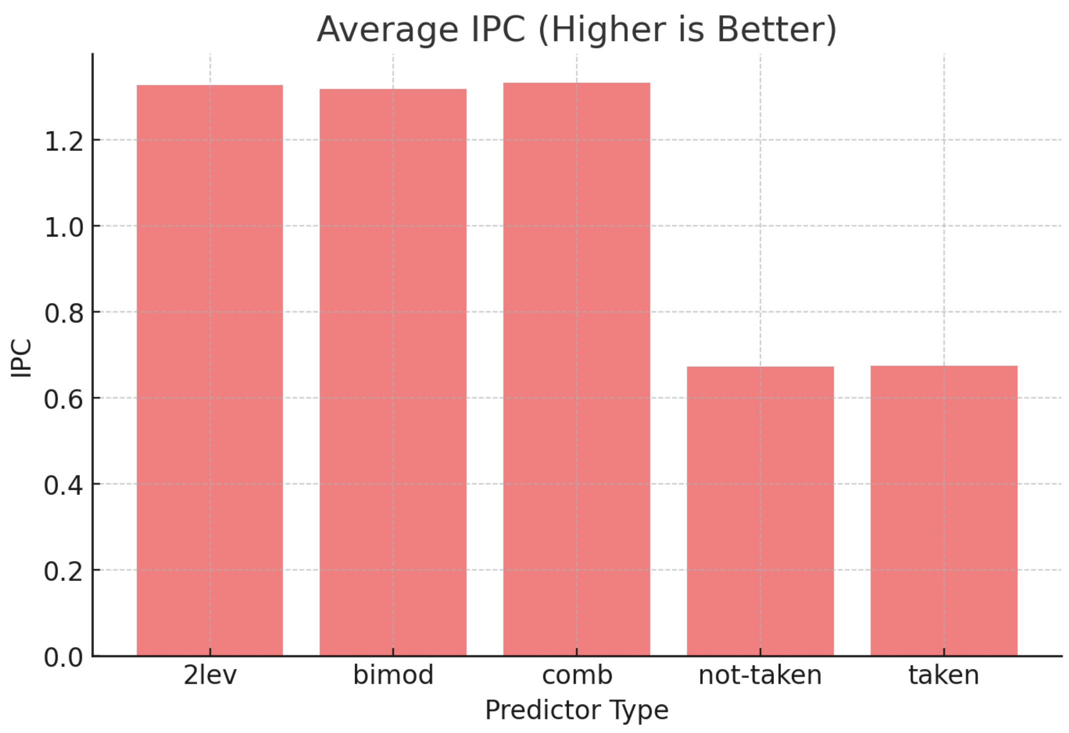

Figure 10: Average IPC across all predictors (Higher is Better).

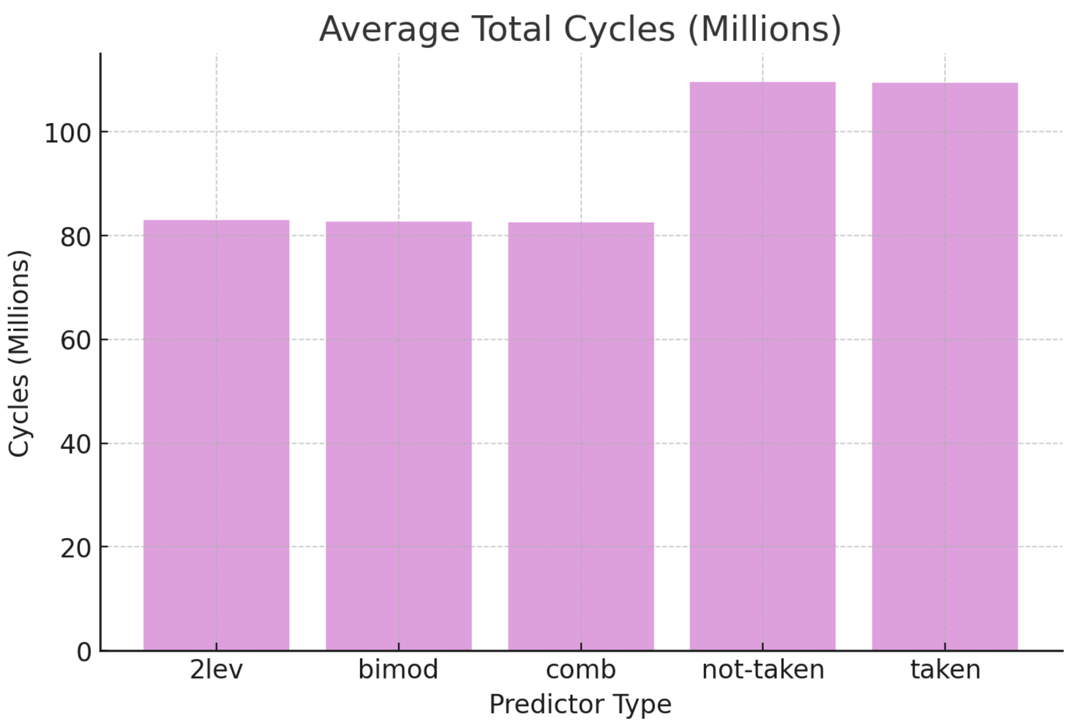

Figure 11: Average Total Cycles across all predictors (Lower is Better).

0.8 Discussion

Detailed Analysis

Branch Misprediction Rate: Static predictors (taken, not-taken) performed the worst with misprediction rates $> 90\%$ across all benchmarks. Dynamic predictors (bimod, 2lev, comb) significantly reduced mispredictions, with comb achieving the lowest average miss rate (around $\sim 1.4\%$).

CPI: Comb recorded the lowest average CPI, followed closely by 2lev; static predictors had high CPI due to frequent rollbacks.

IPC: Inversely related to CPI; comb achieved the highest IPC (around ~ 1.20).

Total Cycles: Comb consistently finished with the fewest cycles, followed by bimod; static predictors consumed roughly 2–3$\times$ more cycles.

Overall Trends

- **comb (hybrid)**: Lowest miss rate, lowest CPI, highest IPC, minimum cycles.
- **bimod**: Competitive on regular workloads; slightly weaker on complex patterns.
- **2lev**: Good accuracy; slightly higher CPI than comb overall.
- **taken / not-taken**: Baselines with poor accuracy.

0.9 Conclusion

The hybrid (`comb`) branch predictor provides the best trade-off between accuracy, throughput, and efficiency for compute-intensive workloads in the SimpleScalar `sim-outorder` environment. It should be adopted as the default configuration (`-bpred comb`) for general ACA lab experiments and architectural performance evaluations.

References

[1] Computer Architecture: A Quantitative Approach, 6th Edition By Hennessy and Patterson
https://archive.org/details/computerarchitectureaquantitativeapproach6thedition

[2] Austin, T. M., Larson, E. S., & Ernst, D. (2002). SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2), 59–67.

[3] SimpleScalar Tutorial – A Quick Guide to Using the SimpleScalar Toolset. https://cs.nju.edu.cn/swang/CA_16S/simplescalar_tutorial.pdf

[4] LLM Transformer Model Visually Explained. Polo Club. https://poloclub.github.io/transformer-explainer/

[5] Understanding Large Language Models: A Comprehensive Guide. Elastic, 2024.

[6] Understanding Transformers: The Architecture of LLMs. MLQ.ai, 2024.

[7] Transformer (deep learning architecture). Wikipedia.

[8] Program 1 (1.c) — Matrix Multiply + Piecewise Activation https://drive.google.com/file/d/1RrxFMU_zGPwzwbPAsieD4NzBG3GafbbQ/view?usp=sharing

[9] Program 2 (2.c) — Sparse Attention (Mask + Gating) https://drive.google.com/file/d/1kkSdKoOZkjDkCegQfoPYLCQsU1Dp17Zc/view?usp=sharing

[10] Program 3 (3.c) — Mixture-of-Experts Router https://drive.google.com/file/d/1rEcVWHvOAWIdf-4EZGJRLVBbV7YNFzVA/view?usp=sharing

[11] Program 4 (4.c) — LayerNorm + Clipping https://drive.google.com/file/d/1RTvORHtP3aa2w_YdJc97XqC8r-nfapEn/view?usp=sharing

[12] Program 5 (5.c) — Stable Softmax + Clamps https://drive.google.com/file/d/1jzOlX9k4NTlE5zhBv9Ta3mUtKu0zlV2J/view?usp=sharing

[13] Program 6 (6.c) — Top-k Sampler (Decoding) https://drive.google.com/file/d/12RqL5yKi7bBAauvawe2A5ISKWrf8KkVP/view?usp=sharing

[14] Program 7 (7.c) — Gradient Update with Adaptive Learning Rate https://drive.google.com/file/d/16fWVsOIlZV_GHzFzugv89SFuppRMWQp8/view?usp=sharing