HW4 Report

20246471 Wen-Tzu(Joko), Chang

A. Characterize the Benchmark Workloads

Benchmark Overview

Benchmark	Problem Description	Algorithms and Data Structures	Characteristics and Performance Impact
gcc	Simulates the behavior of a C compiler, involving many branch instructions and pointer operations.	Uses **tree-based structures** (syntax trees) and recursive algorithms.	**Branch-intensive**: Frequent control flow changes make branch prediction accuracy critical. Mis-predictions result in pipeline flushes, impacting performance.
mcf	Solves the minimum-cost flow problem, commonly used for network traffic and logistics planning.	Utilizes **graph algorithms** (minimum flow path calculations).	**Memory-access intensive**: Frequent data accesses lead to high L1 and L2 cache miss rates, slowing memory access speed.
bzip2	Data compression tool based on the Burrows-Wheeler Transform (BWT) algorithm.	Implements **sorting**, frequency counting, and bit-level encoding.	**Mixed compute and memory-intensive**: Requires significant computation for compression, with high memory access demands.
swim	Used for numerical fluid dynamics simulations, focusing on matrix operations.	Involves **matrix iteration algorithms** (high-dimensional array operations).	**Floating-point intensive**: Highly dependent on floating-point ALU utilization, requiring efficient computation resource usage.
art	Simulates neural network models, applicable to image recognition and automated classification tasks.	Employs **sparse matrix operations** and neural network algorithms.	**Memory-access and compute intensive**: Sparse matrix operations increase memory access latency, while performance is constrained by floating-point operations.

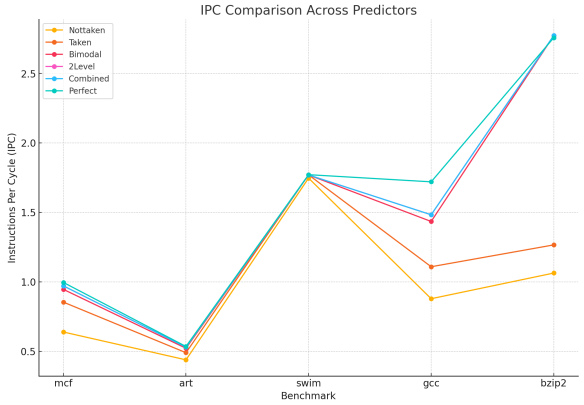
Workload Characteristics and Performance Impact

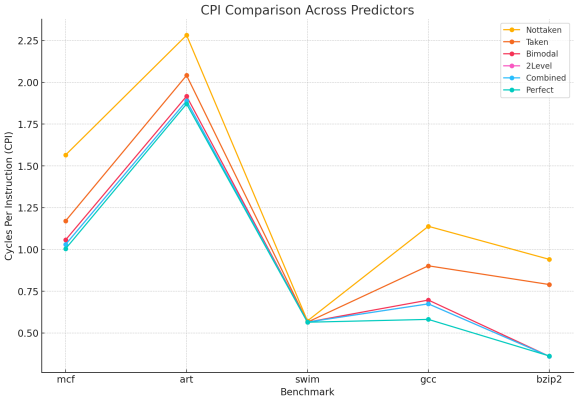
- `gcc` is a branch-intensive workload where branch prediction accuracy significantly influences pipeline performance.

  Key metrics to analyze include branch misprediction rates and CPI (Cycles Per Instruction).
- `mcf` is memory-access intensive, resulting in higher L1 and L2 cache miss rates, which increase latency and impact overall performance.
- `bzip2` is a mix of computation and memory access that demands ALU efficiency and optimized cache performance. Performance is affected by balancing computational resources and memory access.
- `swim` relies heavily on floating-point operations. Key metrics include FPALU utilization and CPI.

- `art` combines sparse matrix operations with neural network algorithms, making it memory-access intensive. Cache misses (L1/L2) and memory access latency are critical factors in its performance.

B. Static Predictor and Dynamic Predictor Performance Analysis





Benchmark	Nottaken_IPC	Taken_IPC	Bimodal_IPC	2Level_IPC	Combined_IPC	Perfect_IPC
mcf	0.6386	0.8536	0.9455	0.9707	0.9707	0.9953
art	0.4382	0.4897	0.5216	0.5294	0.5294	0.5345
swim	1.7465	1.77	1.7693	1.7693	1.7693	1.7712
gcc	0.8783	1.1085	1.4344	1.4826	1.4826	1.7197
bzip2	1.063	1.2657	2.7722	2.7729	2.7729	2.757

Benchmark	Nottaken_CPI	Taken_CPI	Bimodal_CPI	2Level_CPI	Combined_CPI	Perfect_CPI
mcf	1.5659	1.1714	1.0576	1.0302	1.0302	1.0047
art	2.2821	2.0414	1.917	1.889	1.889	1.8708
swim	0.5726	0.565	0.5652	0.5652	0.5652	0.5646
gcc	1.1385	0.9021	0.6971	0.6745	0.6745	0.5815
bzip2	0.9408	0.7901	0.3607	0.3606	0.3606	0.3627

1. Performance Comparison: Bimodal Predictor vs. Perfect Predictor

- The results show that the **perfect predictor** achieves the highest IPC (Instructions Per Cycle) and lowest CPI (Cycles Per Instruction) because it eliminates all branch mispredictions.
- **Performance Gain**
  The performance gain from improving a
  bimodal predictor to a perfect predictor is **relatively tiny** in some benchmarks like `swim` but **significant** in branch-intensive benchmarks like `gcc`.

  Reason

  Benchmarks like `swim` are less branch-intensive, meaning the effect of branch misprediction is minimal. For these, computation dominates execution time. In contrast, benchmarks like `gcc` involve complex control flows, and reducing branch mispredictions drastically improves performance.

Benchmark	IPC (Bimodal)	IPC (Perfect)	Improvement (%)
mcf	0.9455	0.9953	~5.3%
art	0.5216	0.5345	~2.41%
swim	1.7693	1.7712	~0.1%
gcc	1.4344	1.7197	~19.9%
bzip2	2.7722	2.7570	~-0.5% (negligible)

2. Dynamic Branch Predictors: Bimodal, 2-Level, and Combined

Key Ideas

- **Bimodal**: Uses 2-bit saturating counters to predict branches based on history at a single level.
- **2-Level**: Tracks global or local branch history and uses it to predict future branches more accurately.
- **Combined**: Combines bimodal and 2-level predictors, selecting the better Predictor dynamically.

Simulation Results

- **Combined** and **2-Level Predictors** outperform the **bimodal Predictor** due to higher prediction accuracy, especially for complex control flows (`mcf`, `gcc`, `art`).
- **Combined Predictor** shows marginal improvement over 2-Level Predictor, as it adapts dynamically to workloads.

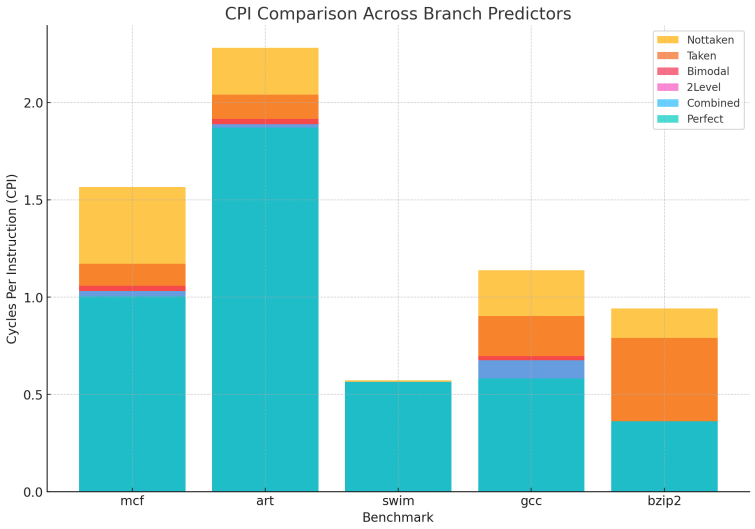
Performance Summary

Benchmark	Bimodal IPC	2-Level IPC	Combined IPC
mcf	0.9455	0.9707	0.9707
art	0.5216	0.5294	0.5294
swim	1.7693	1.7693	1.7693
gcc	1.4344	1.4826	1.4826
bzip2	2.7722	2.7729	2.7729

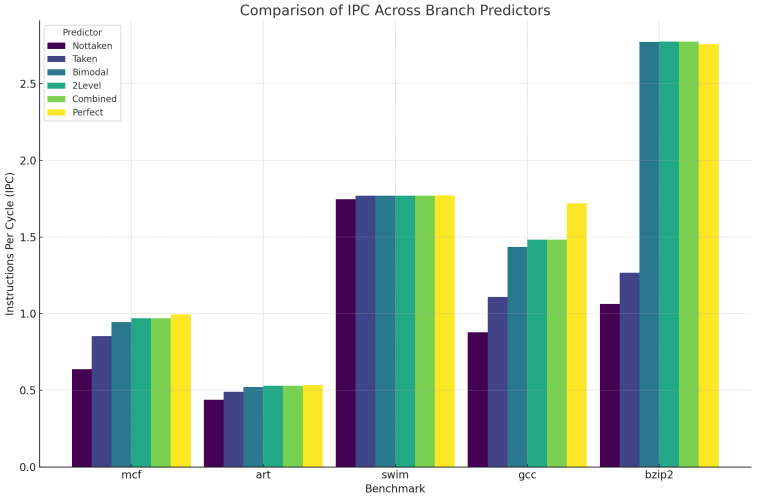
Insights

- **Bimodal predictors** perform well for simple branch patterns but suffer in workloads with correlated branches.
- **2-Level predictors** address this limitation by leveraging branch history.
- **Combined predictors** provide the best overall performance, especially for benchmarks like `gcc` and `mcf`.

Appendix



The above are the visualized charts of **branch predictors** (nottaken, taken, bimodal, 2-level, combined, perfect) showing **CPI** across different benchmarks (mcf, art, swim, gcc, bzip2).



The **IPC comparison chart** for all branch predictors (nottaken, taken, bimodal, 2-level, combined, perfect) across the benchmarks (`mcf`, `art`, `swim`, `gcc`, `bzip2`).

C. The Key Insights in Branch Predictor Design

1. XOR Operation in 2-Level Branch Prediction

In the configuration `-bpred:2lev <l1size> <l2size> <hist_size> <xor>` for the **2-level branch predictor**:

- **What is XOR Operation?**

  The `<xor>` parameter specifies whether the **branch address** and **history bits** are XORed to form the index into the **second-level table**.

  - `<xor> = 0`: Uses only branch history bits for indexing.
  - `<xor> = 1`: Combines the branch address and history bits using an XOR operation for indexing.

- **Why is XOR Operation Used?**

XOR operation helps in reducing **aliasing** or **conflicts** in the branch prediction table by

- Spreading out entries more uniformly across the table.
- Differentiating branches with similar histories but different addresses.

Without XOR, branches with identical histories could map to the duplicate table entry, causing **aliasing** and reducing prediction accuracy.

By incorporating the branch address via XOR, the predictor can distinguish between branches even if they share the same history.

XOR Operation in 2-Level Prediction: This operation reduces aliasing by containing branch address and history bits for indexing, improving accuracy.

2. Modern Branch Predictor

TAGE (TAgged GEometric predictor) is a modern branch predictor that leverages multiple tables with different history lengths and tags to improve accuracy, particularly for complex and irregular branches.

- **Structure**
  - TAGE maintains multiple predictor tables, each corresponding to a specific history length

    (e.g., shorter tables for recent history, longer tables for distant history).
  - The predictor **tags** entries in each table to avoid aliasing.
  - For each branch, TAGE **searches** the tables for matching entries, starting with the most extended history table, and chooses the most accurate predictor.
- **When and How Does TAGE Improve Accuracy?**
  - TAGE is particularly effective for **workloads with complex control flows** and **irregular branch patterns**.
  - By combining **short-term** and **long-term branch histories**, TAGE can adapt to varying branch behaviors and predict branches that depend on distant or irregular conditions.
  - Using **tags** reduces table conflicts and aliasing, further enhancing prediction accuracy.

Advantages

- TAGE significantly outperforms simpler predictors like **bimodal** and **2-level** predictors because it adapts to complex branching patterns.
- It is widely used in modern processors because it provides high accuracy while maintaining reasonable hardware costs.