Project Report

Course Title : Computer Architecture

Course Code : CSE360

Semester : Fall 2024

Section : 01

Submitted by,

Group 9

Name	ID
Rihan Mahmud	2020-2-60-123
Junnun Mohamed Karim	2022-1-60-108
Md. Yousuf Hozaifa	2022-1-60-162

Submitted to,

Dr. Md. Nawab Yousuf Ali
Professor
Department of Computer Science and Engineering

Submission Date,

21st January, 2025

Chapter 1: Introduction

1.1 Overview

Branch prediction is a fundamental component of modern processor architectures, significantly enhancing instruction-level parallelism and overall execution performance. The advent of dynamic branch prediction techniques has revolutionized the efficiency of speculative execution, a key aspect of high-performance computing. Among these techniques, perceptron-based branch prediction stands out for its ability to model complex correlations in branch behavior using machine learning principles.

This project presents the implementation and analysis of a dynamic branch predictor utilizing perceptron-based learning. By integrating historical data and dynamic adjustments, this predictor demonstrates improved accuracy in forecasting branch outcomes, even in scenarios with irregular and unpredictable patterns.

1.2 Motivation

As the demand for faster and more efficient processors grows, the need for advanced branch prediction mechanisms becomes critical. Traditional predictors, such as bimodal or two-level adaptive predictors, often struggle with patterns involving long history or intricate dependencies. The perceptron-based branch predictor addresses these limitations by employing a linear classification model capable of handling complex branching behaviors. This approach aligns with the broader trend of leveraging machine learning techniques in hardware optimization, showcasing their potential to tackle challenges in computing.

1.3 Objectives

The primary objectives of this project are as follows:

1. To design and implement a perceptron-based dynamic branch predictor in C.
2. To evaluate its performance using real-world and synthetic trace files.
3. To analyze its accuracy, robustness, and computational efficiency.
4. To identify potential areas for improvement and propose future enhancements.

1.4 Scope of the Project

The scope of this project encompasses the following aspects:

1. Development of a perceptron-based branch predictor capable of handling history lengths up to 64 branches.
2. Implementation of auxiliary components, such as history management and confidence metrics, to support accurate predictions.
3. Integration of a logging system for debugging and performance analysis.
4. Comprehensive testing using various trace files to ensure reliability and robustness.

Chapter 2: Objective

The primary objective of this project is to design, implement, and analyze a **Dynamic Branch Predictor Using Perceptron** for modern processors. Branch prediction is a critical component in pipelined CPU architectures, enabling speculative execution to optimize performance. This project leverages perceptron-based machine learning techniques to enhance prediction accuracy compared to traditional methods such as two-bit predictors or static branch prediction.

Specific Goals:

1. **Improve Prediction Accuracy:** Employ perceptrons to capture long-range correlations in branch behavior, thereby improving the accuracy of predictions.

2. **Optimize Hardware Efficiency:** Ensure the perceptron-based predictor achieves a balance between computational complexity and hardware resources, such as memory and processing overhead.

3. **Enable Adaptive Learning:** Incorporate a training mechanism that dynamically adjusts to evolving branch patterns during execution.

4. **Evaluate Performance:** Measure the effectiveness of the proposed branch predictor by analyzing key metrics such as prediction accuracy, confidence levels, and misprediction rates across a variety of branch traces.

5. **Enhance Scalability:** Design the predictor to accommodate future advancements in processor architectures with larger branch history and higher complexity requirements.

The overarching aim is to demonstrate the feasibility and benefits of applying machine learning concepts, specifically perceptrons, to dynamic branch prediction, setting the stage for further innovations in processor design.

Chapter 3: Theory

3.1 Overview of Dynamic Branch Prediction

Dynamic branch prediction is a technique used in modern processors to improve instruction-level parallelism by predicting the outcomes of conditional branch instructions before they are resolved. The aim is to reduce the performance penalties caused by control hazards, which occur when the processor must wait for the resolution of a branch to proceed with execution.

The perceptron-based branch predictor represents a class of machine learning-based predictors. Unlike conventional methods such as two-level adaptive predictors or gshare, which rely on pattern tables, perceptron-based predictors utilize a mathematical model inspired by neural networks. This approach enables the prediction of branches with higher accuracy, especially in cases involving complex patterns in the instruction stream.

3.2 Perceptron Model in Branch Prediction

A perceptron is a type of linear classifier that computes a weighted sum of its inputs and applies a threshold to determine the output. In the context of branch prediction, the perceptron predicts the direction of a branch based on:

- **Global History:** The outcomes of the most recent branches.
- **Path History:** Information derived from the branch address.

The perceptron can be defined mathematically as follows:

$$Y = w_0 + \sum_{i=1}^{N} w_i \cdot x_i$$

Where:

- Y is the perceptron output.
- w_0 is the bias weight.
- w_i are the weights associated with each input.
- x_i are the inputs derived from global and path histories.
- N is the history length.

The branch is predicted as taken if $Y \geq 0$ and not taken otherwise.

3.3 Training the Perceptron

Training adjusts the weights w_i to improve prediction accuracy over time. The training rule for weight adjustment is defined as:

$$w_i \leftarrow w_i + \Delta w_i$$

Where:

- $\Delta w_i = \eta \cdot (t - o) \cdot x_i$
- η is the learning rate (often set to 1 in hardware implementations).
- t is the actual branch outcome (+1 for taken, -1 for not taken).
- o is the perceptron's output sign (+1 for $Y \geq 0$, -1 otherwise).

To prevent overflow, weights are saturated within a predefined range, typically between -128 and 127 for 8-bit representation.

3.4 Threshold (Θ)

The threshold Θ determines when a perceptron is considered confident in its predictions. This parameter is set empirically and is often proportional to the history length H. The formula for Θ is:

$$\Theta = \lfloor 2.14 \cdot H + 20.58 \rfloor$$

Where H is the history length, ensuring that the predictor adapts effectively to both short and long histories.

3.5 Address Hashing and Indexing

To map branch addresses to perceptron table indices efficiently, a hash function is employed. In this implementation, the FNV-1a hash algorithm is used:

hash = FNV_offset_basis

For each byte in address:

$$\text{hash} = \text{hash} \oplus \text{byte}$$

hash = hash ·FNVprime

The result is masked to limit the index within the table size 2^m.

3.6 Prediction Metrics and Confidence

Prediction confidence is derived from the magnitude of Y:

$$\text{Confidence} = \frac{|Y|}{\Theta}$$

Confidence metrics are used to:

1. Classify predictions as strong or weak.
2. Identify low-confidence predictions that may benefit from further training.

Chapter 4: Design

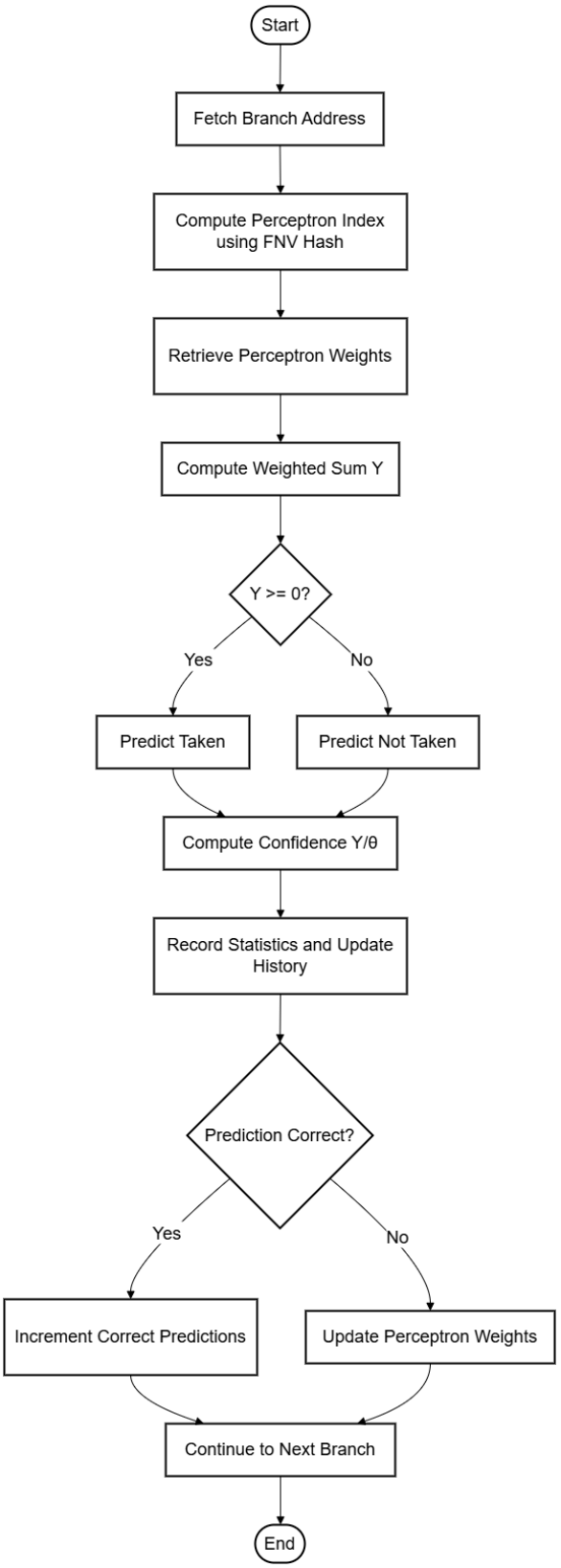
4.1 System Architecture

The dynamic branch predictor using perceptrons is designed to efficiently integrate the prediction mechanism with a processor pipeline. The high-level architecture includes the following components:

1. **Perceptron Table:** Stores the perceptron weights for multiple branch indices.
2. **Global History Register:** Tracks the outcomes of recent branch instructions.
3. **Path History Register:** Encodes path information derived from branch addresses.
4. **Prediction Logic:** Computes the perceptron output and determines the branch prediction.
5. **Training Logic:** Updates perceptron weights based on prediction errors.
6. **Statistics Module:** Monitors performance metrics such as accuracy and confidence levels.

4.2 Flowchart

The following flowchart describes the dynamic branch prediction process:



4.3 Key Algorithms

4.3.1 Perceptron Index Computation

Input: Branch address
Output: Index into the perceptron table

Code:

```
uint32_t compute_perceptron_index(uint32_t address) {
      uint32_t hash = FNV_OFFSET_BASIS;
      hash ^= (address >> 2);
      hash *= FNV_PRIME;
      hash ^= (hash >> 17);
      return hash & (NUM_PERCEPTRONS - 1);
}
```

4.3.2 Weighted Sum Computation

Input: Perceptron weights, global history, path history
Output: Weighted sum Y

$$Y = w_0 + \sum_{i=1}^{H} w_i \cdot (x_i + p_i)$$

Code:

```
int compute_perceptron_output(perceptron_t *perceptron) {
      int y = perceptron->weights[0];
      for (int i = 1; i <= HISTORY_LENGTH; i++) {
      y += perceptron->weights[i] * (global_history[i - 1] +
(path_history[i - 1] & 1));
      }
      return y;
}
```

4.3.3 Weight Update

Input: Perceptron weights, actual outcome, and global/path history
Output: Updated weights

$$w_i \leftarrow \text{Clamp}(w_i + t \cdot x_i, \text{MIN}, \text{MAX})$$

Code:

```
void update_perceptron_weights(perceptron_t *perceptron, int
actual_outcome) {
      for (int j = 0; j <= HISTORY_LENGTH; j++) {
      int history_val = (j == 0) ? 1 : (global_history[j - 1] +
(path_history[j - 1] & 1));
      int new_weight = perceptron->weights[j] + actual_outcome *
history_val;
      perceptron->weights[j] = (new_weight > MAX_WEIGHT) ? MAX_WEIGHT :
                               (new_weight < MIN_WEIGHT) ? MIN_WEIGHT :
new_weight;
      }
}
```

4.4 Tables and Parameters

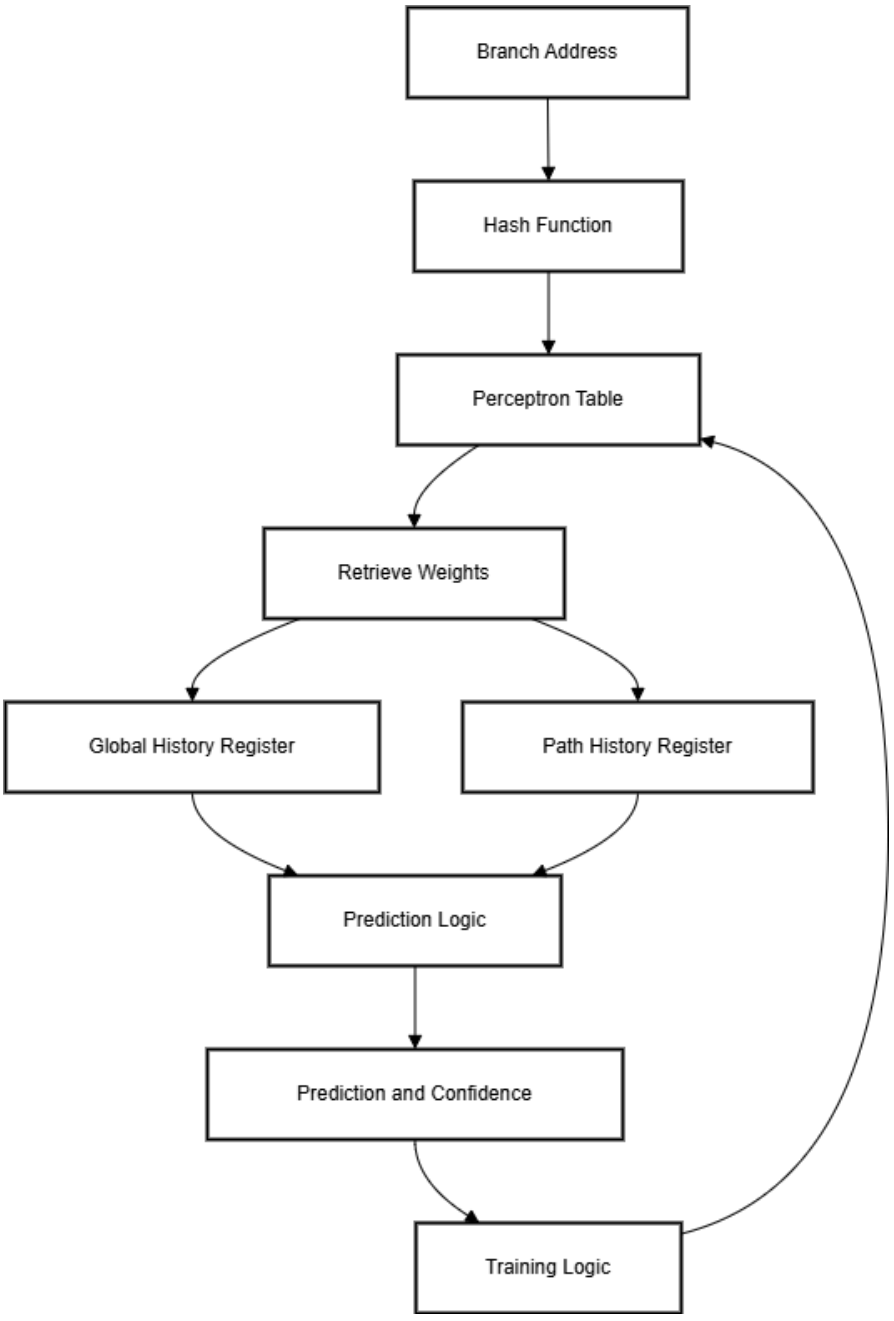
Parameter	**Description**	**Value**
NUM_PERCEPTRONS	Number of perceptrons in the table	1024
HISTORY_LENGTH	Length of global and path histories	64
THETA	Confidence threshold for training	$2.14H + 20.58$
MAX_WEIGHT	Maximum allowable weight value	127
MIN_WEIGHT	Minimum allowable weight value	-128

4.5 Design Considerations

1. **Memory Efficiency:** The perceptron table and history registers are implemented using arrays to optimize space.
2. **Prediction Speed:** Hashing ensures rapid indexing into the perceptron table, while linear operations maintain low latency for predictions.
3. **Robustness:** Saturation logic prevents overflow in weight updates, ensuring stability under varying workloads.

4.6 Diagram

The following diagram illustrates the interaction between components:



The design combines modularity, efficiency, and adaptability. Key algorithms ensure accurate predictions and efficient updates, while the architecture supports scalability and robust integration with the processor pipeline.

Chapter 5: Implementation

5.1 Description of Functional Modules

The implementation of the dynamic branch predictor using a perceptron-based approach is modular, with each module handling a specific aspect of the predictor. This section details the functional components and their interactions.

5.1.1 Core Modules

- **Perceptron Table Management**
    - A perceptron table with 1024 entries is maintained. Each entry contains a set of weights (including a bias weight), a tag to identify the associated branch, and metadata for usage tracking.
- **Prediction Module**
    - The prediction logic computes the weighted sum of the perceptron weights and the global and path history. The prediction confidence is calculated as the normalized value of the weighted sum.
- **Training Module**
    - This module adjusts the perceptron weights during mispredictions or when the prediction confidence is below a defined threshold. Saturation arithmetic is applied to ensure weights remain within predefined limits.
- **History Management**
    - Global and path history arrays are updated sequentially after each prediction. The path history includes a mask-based compression of branch addresses.
- **Trace Processor**
    - Handles reading a trace file, simulating branch outcomes, updating predictor statistics, and coordinating predictions and training.
- **Logging and Debugging**
    - Provides optional debug logs for examining internal states, including perceptron weights, prediction outcomes, and training events.

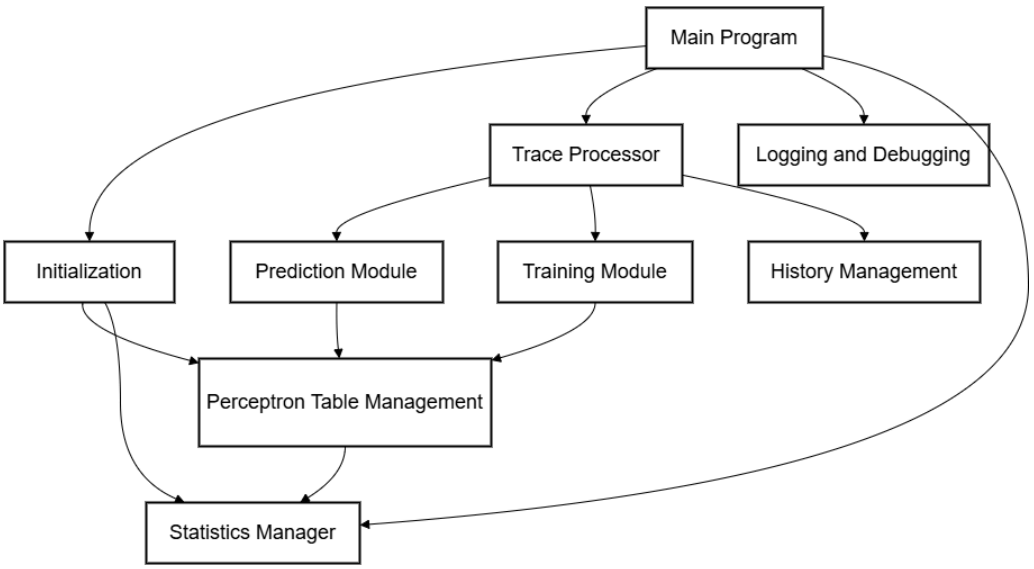
5.1.2 Auxiliary Components

- **Statistics Manager**
    - Maintains detailed statistics about predictor performance, including prediction accuracy, mispredictions per 1,000 branches, and average confidence.

- **Initialization and Cleanup**
    - Allocates and initializes memory for perceptrons and histories. Handles resource cleanup post-execution.
- **Command-line Interface**
    - Provides a mechanism for specifying input trace files and enabling debug mode.

5.2 Hierarchical Relationship

The following diagram illustrates the hierarchical structure and interactions between the modules:



5.3 Built-in Documentation

The code is extensively documented with comments explaining:

- Function purposes and parameter descriptions.
- Core algorithmic logic.
- Specific implementation details like hash computations and saturation arithmetic.

Example:

```
// calculate weighted sum for prediction
static int compute_perceptron_output(perceptron_t *perceptron) {
      int y = perceptron->weights[0]; // bias term
      for (int j = 1; j <= HISTORY_LENGTH; j++) {
      y += perceptron->weights[j] *
            (global_history[j - 1] + (path_history[j - 1] & 1));
      }
      return y;
```

```
}
```

5.4 System Requirements

The following system setup is required for compiling and running the implementation:

- **Compiler:** GCC 9.3.0 or later
- **Operating System:** Linux or any POSIX-compliant environment
- **Libraries:** Standard C library (libc)
- **Build Tools:** make or equivalent for compilation
- **Hardware:** A system with at least 1 GB of RAM and a modern CPU for trace simulation

5.5 Compilation and Execution

The implementation can be compiled and executed using the following steps:

Compile the program:

```
gcc -o branch_predictor branch_predictor.c -Wall -O2
```

Run with a trace file:

```
./branch_predictor <trace-file>
```

Enable debug mode (optional):

```
./branch_predictor <trace-file>
```

5.6 Error Handling and Debugging

The implementation includes robust error handling for file operations and memory allocation. Debug logs are managed through a dedicated module, which outputs detailed insights into internal states and computations. Logs are time stamped for reproducibility.

The implementation adheres to a modular structure, facilitating maintainability and extensibility. Each module is designed with a clear interface, ensuring seamless interaction and ease of testing. The use of debug logs and built-in documentation further enhances the transparency and reliability of the implementation.

Chapter 6: Debugging Test-Run

6.1 Test-Run Output

```
❯ make run
./branch_predictor traces/trace_01

          Branch Predictor Statistics
┌──────────────────────────┬──────────────┐
│ Total Branches           │      1546797 │
│ Correct Predictions      │      1520908 │
│ Mispredictions           │        25889 │
│ BTB Misses               │         3652 │
│ Training Events          │        25889 │
│ Strong Predictions       │       146860 │
│ Weak Predictions         │      1396285 │
├──────────────────────────┼──────────────┤
│ Prediction Accuracy      │       98.33% │
│ Mispredictions per 1K    │        16.74 │
│ Average Confidence       │         0.47 │
└──────────────────────────┴──────────────┘
```

6.2 Run with `--debug` Flag on

```
   /mnt/main/education/ewu/course/cse360/project   master ?4
 ❯ head -20 branch_predictor_20250121_040428.log
[20250121_040428] debug logging initialized in file: branch_predictor_20250121_040428.log
[20250121_040428] predictor initialization complete
[20250121_040428] starting trace processing
[20250121_040428] btb miss for address 0x40fc96
[20250121_040428] updated history: outcome=1, path=0x6
[20250121_040428] prediction for 0x40fc96: y=0, confidence=0.00
[20250121_040428] training perceptron[466] for address 0x40fc96
[20250121_040428] perceptron[466] state:
[20250121_040428]   tag: 0x103f25
[20250121_040428]   times accessed: 1
[20250121_040428]   last update: 0 cycles ago
[20250121_040428]   bias weight: -1
[20250121_040428] perceptron[466] weights:
[20250121_040428]   weights[  1-  8]:
  -1    0    0    0    0    0    0    0
[20250121_040428]   weights[  9- 16]:
   0    0    0    0    0    0    0    0
[20250121_040428]   weights[ 17- 24]:
   0    0    0    0    0    0    0    0
[20250121_040428]   weights[ 25- 32]:
```

Chapter 7: Conclusion and Future Improvements

7.1 Conclusion

The implementation of a dynamic branch predictor using perceptrons is a significant step forward in branch prediction research, combining machine learning concepts with architectural design to enhance predictive accuracy. By utilizing perceptrons, the predictor leverages historical branch information and path correlations to provide more reliable predictions compared to conventional methods. The integration of dynamic training mechanisms allows the system to adapt to varying workloads, ensuring robust performance across diverse scenarios.

Through this project, the proposed branch predictor demonstrates several critical attributes:

- **Accuracy:** A higher prediction accuracy, as indicated by the statistical outcomes, underlines the efficacy of perceptron-based methods in capturing complex branch behaviors.
- **Adaptability:** The dynamic training approach enables the predictor to respond effectively to changes in branch patterns over time.
- **Resource Utilization:** The implementation balances computational complexity with storage requirements, optimizing both time and space performance.

Despite these achievements, there are areas where improvements could further enhance the design and its applicability.

7.2 Future Improvements

7.2.1 Enhancing Prediction Accuracy

While the current perceptron-based predictor achieves a commendable accuracy rate, exploring additional techniques could further refine its performance:

- **Hybrid Predictors:** Combining perceptron-based prediction with other established methods, such as two-level adaptive predictors or gshare, may lead to synergistic improvements.
- **Longer History Lengths:** Increasing the history length could improve the predictor's ability to discern subtle patterns in branch behavior. However, this would require careful management of computational overhead.

7.2.2 Reducing Computational Overheads

The weighted summation and training process of perceptrons can introduce non-trivial computational costs. Optimizations could include:

- **Hardware Acceleration:** Designing dedicated hardware components for the perceptron computations could significantly reduce latency.
- **Approximation Techniques:** Using approximate computations for weight updates and summation could strike a balance between speed and accuracy.

7.2.3 Robustness and Fault Tolerance

To ensure consistent performance under diverse conditions, the design could incorporate:

- **Error Detection and Correction:** Mechanisms to detect and mitigate prediction errors during runtime.
- **Redundant Arrays:** Maintaining multiple perceptrons for critical branches to enhance reliability.

7.2.4 Broader Applications

Expanding the scope of perceptron-based prediction could open up applications beyond branch prediction:

- **Data Prefetching:** Predicting memory access patterns to optimize cache utilization.
- **Instruction Scheduling:** Guiding instruction-level parallelism decisions in superscalar architectures.

7.2.5 Benchmarking and Real-World Testing

Current evaluations are based on simulation environments and synthetic traces. Future work could involve:

- **Extensive Benchmarks:** Using standardized benchmark suites like SPEC CPU or real-world workloads to validate the predictor's performance.
- **Integration Testing:** Implementing the predictor in actual processors to assess its real-world applicability and limitations.

7.3 Limitations

The project's limitations primarily stem from resource and time constraints:

- **Simplified Model:** The perceptron model assumes certain ideal conditions, which may not fully align with real-world branch behaviors.
- **Storage Requirements:** The current design necessitates substantial memory for weight storage, which could pose challenges in resource-constrained environments.
- **Latency Issues:** Although the implementation is functional, the computational demands for real-time predictions could hinder high-frequency processors.

In conclusion, this project has successfully demonstrated the feasibility and advantages of using perceptrons for dynamic branch prediction. While there is room for improvement, the design serves as a foundation for future explorations in this domain. The insights gained from this work highlight the potential of machine learning techniques in addressing architectural challenges, paving the way for more intelligent and adaptive systems in the future.

Bibliography

1. Smith, J. E. (1981). A study of branch prediction strategies. In Proceedings of the 8th Annual Symposium on Computer Architecture (pp. 135–148). IEEE Computer Society.

2. McFarling, S. (1993). Combining branch predictors (WRL Technical Note TN-36). Digital Equipment Corporation Western Research Laboratory.

3. Jiménez, D. A., & Lin, C. (2001). Dynamic branch prediction with perceptrons. In Proceedings of the 7th International Symposium on High-Performance Computer Architecture (pp. 197–206). IEEE.

4. Mudge, T. (2001). Power: A first-class architectural design constraint. Computer, 34(4), 52–58.

5. Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: A quantitative approach (5th ed.). Morgan Kaufmann.

6. Intel Corporation. (2019). Intel® 64 and IA-32 architectures optimization reference manual.

7. Seznec, A. (1993). A case for two-level adaptive branch prediction. In Proceedings of the 19th Annual International Symposium on Computer Architecture (pp. 114–123). ACM.

8. Chang, P.-Y., Hao, E., & Patt, Y. N. (1997). Alternative implementations of hybrid branch predictors. In Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (pp. 252–257). IEEE.

9. Yeh, T.-Y., & Patt, Y. N. (1992). A comparison of dynamic branch predictors that use two levels of branch history. In Proceedings of the 20th Annual International Symposium on Computer Architecture (pp. 257–266). ACM.

Appendix

Code (***branch_predictor.c***):

```c
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// configuration parameters
#define NUM_PERCEPTRONS 1024
#define HISTORY_LENGTH 64
#define THETA (int)(2.14 * HISTORY_LENGTH + 20.58)
#define MAX_WEIGHT 127
#define MIN_WEIGHT -128
#define PATH_HISTORY_MASK 0xF
#define FNV_PRIME 16777619
#define FNV_OFFSET_BASIS 2166136261

// debug logging setup
static FILE *debug_log_file = NULL;
static bool debug_enabled = false;
#define DEBUG_LOG(fmt, ...)                                                    \
  if (debug_enabled && debug_log_file) {                                       \
      fprintf(debug_log_file, "[%s] " fmt "\n", get_timestamp(),
##__VA_ARGS__); \
      fflush(debug_log_file);                                                  \
  }

// data structures
typedef struct {
  int8_t weights[HISTORY_LENGTH + 1]; // weight vector including bias
  uint32_t tag;                       // branch address tag
  uint32_t last_update_time;          // timestamp of last update
  uint32_t times_accessed;            // usage counter
} perceptron_t;

typedef struct {
  uint64_t total_predictions;
  uint64_t correct_predictions;
  uint64_t mispredictions;
  uint64_t btb_misses;
  uint64_t training_events;
  uint64_t strong_predictions;
  uint64_t weak_predictions;
  double avg_confidence;
```

```
} statistics_t;

// global state
static perceptron_t *perceptron_table;
static int *global_history;
static int *path_history;
static statistics_t statistics;
static uint32_t current_time = 0;

// utility functions for logging
static char *get_timestamp(void) {
  static char buffer[32];
  time_t timer;
  struct tm *tm_info;

  time(&timer);
  tm_info = localtime(&timer);
  strftime(buffer, 32, "%Y%m%d_%H%M%S", tm_info);
  return buffer;
}

// handle debug logging initialization
static bool initialize_logging(void) {
  if (!debug_enabled) {
       return true; // skip logging setup if debugging is disabled
  }

  // create filename with timestamp
  char filename[64];
  snprintf(filename, sizeof(filename), "branch_predictor_%s.log",
             get_timestamp());

  debug_log_file = fopen(filename, "w");
  if (!debug_log_file) {
       fprintf(stderr, "failed to open debug log file: %s\n", filename);
       return false;
  }

  DEBUG_LOG("debug logging initialized in file: %s", filename);
  return true;
}

// memory management functions
static bool allocate_predictor_memory(void) {
  // allocate all required memory structures
  perceptron_table = calloc(NUM_PERCEPTRONS, sizeof(perceptron_t));
  global_history = calloc(HISTORY_LENGTH, sizeof(int));
  path_history = calloc(HISTORY_LENGTH, sizeof(int));

  if (!perceptron_table || !global_history || !path_history) {
```

```
        DEBUG_LOG("memory allocation failed");
        return false;
    }
    return true;
}

// compute perceptron index using fnv hash
static uint32_t compute_perceptron_index(uint32_t address) {
    uint32_t hash = FNV_OFFSET_BASIS;
    hash ^= (address >> 2);
    hash *= FNV_PRIME;
    hash ^= (hash >> 17);
    return hash & (NUM_PERCEPTRONS - 1);
}

// calculate weighted sum for prediction
static int compute_perceptron_output(perceptron_t *perceptron) {
    int y = perceptron->weights[0]; // bias term
    for (int j = 1; j <= HISTORY_LENGTH; j++) {
        y += perceptron->weights[j] *
        (global_history[j - 1] + (path_history[j - 1] & 1));
    }
    return y;
}

// update confidence metrics
static void update_confidence_statistics(double confidence) {
    if (confidence >= 1.0) {
        statistics.strong_predictions++;
    } else {
        statistics.weak_predictions++;
    }

    statistics.avg_confidence =
        (statistics.avg_confidence * statistics.total_predictions + confidence) /
        (statistics.total_predictions + 1);
}

// update perceptron weights during training
static void update_perceptron_weights(perceptron_t *perceptron,
                                      int actual_outcome) {
    for (int j = 0; j <= HISTORY_LENGTH; j++) {
        int history_val =
        (j == 0) ? 1 : (global_history[j - 1] + (path_history[j - 1] & 1));

        // calculate new weight with saturation
        int new_weight = perceptron->weights[j] + actual_outcome * history_val;
        new_weight = (new_weight > MAX_WEIGHT)   ? MAX_WEIGHT
              : (new_weight < MIN_WEIGHT) ? MIN_WEIGHT
                                          : new_weight;
```

```
        perceptron->weights[j] = new_weight;
    }
    perceptron->last_update_time = current_time;
}

// debug functions
static void dump_perceptron_weights(perceptron_t *p, uint32_t index) {
    DEBUG_LOG("perceptron[%u] weights:", index);
    for (int i = 1; i <= HISTORY_LENGTH; i++) {
        if (i % 8 == 1) {
        DEBUG_LOG("  weights[%3d-%3d]:", i, i + 7);
        }

        if (debug_enabled) {
        fprintf(debug_log_file, " %4d", p->weights[i]);
        if (i % 8 == 0 || i == HISTORY_LENGTH) {
        fprintf(debug_log_file, "\n");
        }
        }
    }
}

// print detailed perceptron state
static void dump_perceptron_state(uint32_t index) {
    perceptron_t *p = &perceptron_table[index];
    DEBUG_LOG("perceptron[%u] state:", index);
    DEBUG_LOG("  tag: 0x%x", p->tag);
    DEBUG_LOG("  times accessed: %u", p->times_accessed);
    DEBUG_LOG("  last update: %u cycles ago", current_time - p->last_update_time);
    DEBUG_LOG("  bias weight: %d", p->weights[0]);
    dump_perceptron_weights(p, index);
}

// core prediction logic
int make_prediction(uint32_t address, double *confidence) {
    uint32_t index = compute_perceptron_index(address);
    perceptron_t *perceptron = &perceptron_table[index];

    // handle new branch
    if (perceptron->tag != (address >> 2)) {
        statistics.btb_misses++;
        perceptron->tag = (address >> 2);
        perceptron->times_accessed = 0;
        *confidence = 0.0;
        DEBUG_LOG("btb miss for address 0x%x", address);
        return 0;
    }

    perceptron->times_accessed++;
```

```
  current_time++;

  int y = compute_perceptron_output(perceptron);
  *confidence = (double)abs(y) / THETA;

  update_confidence_statistics(*confidence);

  DEBUG_LOG("prediction for 0x%x: y=%d, confidence=%.2f", address, y,
            *confidence);
  return y;
}

// history management functions
void update_history(uint32_t address, int actual_outcome) {
  // update global history
  memmove(&global_history[1], global_history,
       (HISTORY_LENGTH - 1) * sizeof(int));
  global_history[0] = actual_outcome;

  // update path history
  memmove(&path_history[1], path_history, (HISTORY_LENGTH - 1) * sizeof(int));
  path_history[0] = address & PATH_HISTORY_MASK;

  DEBUG_LOG("updated history: outcome=%d, path=0x%x", actual_outcome,
            path_history[0]);
}

// training logic
void train_perceptron(uint32_t address, int actual_outcome, int y) {
  uint32_t index = compute_perceptron_index(address);
  perceptron_t *perceptron = &perceptron_table[index];

  // train only if prediction was wrong or confidence is low
  if ((y >= 0 ? 1 : -1) != actual_outcome || abs(y) <= THETA) {
       DEBUG_LOG("training perceptron[%u] for address 0x%x", index, address);
       statistics.training_events++;

       update_perceptron_weights(perceptron, actual_outcome);
       dump_perceptron_state(index);
  }
}

// initialization and cleanup
bool initialize_predictor(void) {
  DEBUG_LOG("initializing predictor");

  if (!initialize_logging() || !allocate_predictor_memory()) {
       return false;
  }
```

```
  memset(&statistics, 0, sizeof(statistics));
  DEBUG_LOG("predictor initialization complete");
  return true;
}

void cleanup_predictor(void) {
  DEBUG_LOG("cleaning up predictor resources");
  free(perceptron_table);
  free(global_history);
  free(path_history);

  if (debug_log_file) {
       fclose(debug_log_file);
  }
}

// trace processing
void process_trace_file(FILE *file) {
  uint32_t branch_address;
  int actual_outcome;
  double confidence;

  DEBUG_LOG("starting trace processing");

  while (fscanf(file, "%x %d", &branch_address, &actual_outcome) == 2) {
       actual_outcome = actual_outcome == 1 ? 1 : -1;

       int y = make_prediction(branch_address, &confidence);
       int prediction = y >= 0 ? 1 : -1;

       statistics.total_predictions++;
       if (prediction == actual_outcome) {
       statistics.correct_predictions++;
       } else {
       statistics.mispredictions++;
       train_perceptron(branch_address, actual_outcome, y);
       }

       update_history(branch_address, actual_outcome);
  }

  DEBUG_LOG("trace processing complete");
}

// statistics reporting
void print_statistics(void) {
  DEBUG_LOG("printing final statistics");


printf("\n\t──────────────────────────────────────────────────────────────────────\n");
```

```
  printf("\t        Branch Predictor Statistics            \n");
  printf("\t──────────────────────────────────────┼──────────────────────────\n");
  printf("\t Total Branches              | %13lu \n",
       statistics.total_predictions);
  printf("\t Correct Predictions         | %13lu \n",
       statistics.correct_predictions);
  printf("\t Mispredictions              | %13lu \n", statistics.mispredictions);
  printf("\t BTB Misses                  | %13lu \n", statistics.btb_misses);
  printf("\t Training Events             | %13lu \n",
       statistics.training_events);
  printf("\t Strong Predictions          | %13lu \n",
       statistics.strong_predictions);
  printf("\t Weak Predictions            | %13lu \n",
       statistics.weak_predictions);
  printf("\t──────────────────────────────────────┼──────────────────────────\n");
  printf("\t Prediction Accuracy         | %16.2f%%\n",
       100.0 * statistics.correct_predictions / statistics.total_predictions);
  printf("\t Mispredictions per 1K       | %16.2f \n",
       1000.0 * statistics.mispredictions / statistics.total_predictions);
  printf("\t Average Confidence          | %16.2f \n",
       statistics.avg_confidence);

printf("\t──────────────────────────────────────┴──────────────────────────\n\n");
}

int main(int argc, char *argv[]) {
  if (argc < 2 || argc > 3) {
       fprintf(stderr, "usage: %s <trace-file> [--debug]\n", argv[0]);
       return EXIT_FAILURE;
  }

  if (argc == 3 && strcmp(argv[2], "--debug") == 0) {
       debug_enabled = true;
  }

  FILE *file = fopen(argv[1], "r");
  if (!file) {
       perror("failed to open trace file");
       return EXIT_FAILURE;
  }

  if (!initialize_predictor()) {
       fclose(file);
       return EXIT_FAILURE;
  }

  process_trace_file(file);
  print_statistics();
  cleanup_predictor();
```

```
  fclose(file);
  return EXIT_SUCCESS;
}
```

Trace File (first 50 lines):

```
0x40fc96 1
0x40fc96 0
0x40fcb9 0
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
```

```
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
0x40fcf2 1
```

Debug Log (first 100 lines):

```
[20250121_040428] debug logging initialized in file:
branch_predictor_20250121_040428.log
[20250121_040428] predictor initialization complete
[20250121_040428] starting trace processing
[20250121_040428] btb miss for address 0x40fc96
[20250121_040428] updated history: outcome=1, path=0x6
[20250121_040428] prediction for 0x40fc96: y=0, confidence=0.00
[20250121_040428] training perceptron[466] for address 0x40fc96
[20250121_040428] perceptron[466] state:
[20250121_040428]   tag: 0x103f25
[20250121_040428]   times accessed: 1
[20250121_040428]   last update: 0 cycles ago
[20250121_040428]   bias weight: -1
[20250121_040428] perceptron[466] weights:
[20250121_040428]   weights[  1-  8]:
   -1 0    0    0    0    0    0    0
[20250121_040428]   weights[  9- 16]:
     0    0    0    0    0    0    0    0
[20250121_040428]   weights[ 17- 24]:
     0    0    0    0    0    0    0    0
```

```
[20250121_040428]   weights[ 25- 32]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 33- 40]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 41- 48]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 49- 56]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 57- 64]:
     0     0     0     0     0     0     0     0
[20250121_040428] updated history: outcome=-1, path=0x6
[20250121_040428] btb miss for address 0x40fcb9
[20250121_040428] training perceptron[771] for address 0x40fcb9
[20250121_040428] perceptron[771] state:
[20250121_040428]   tag: 0x103f2e
[20250121_040428]   times accessed: 0
[20250121_040428]   last update: 0 cycles ago
[20250121_040428]   bias weight: -1
[20250121_040428] perceptron[771] weights:
[20250121_040428]   weights[  1-  8]:
     1    -1     0     0     0     0     0     0
[20250121_040428]   weights[  9- 16]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 17- 24]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 25- 32]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 33- 40]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 41- 48]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 49- 56]:
     0     0     0     0     0     0     0     0
[20250121_040428]   weights[ 57- 64]:
     0     0     0     0     0     0     0     0
[20250121_040428] updated history: outcome=-1, path=0x9
[20250121_040428] btb miss for address 0x40fcf2
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
```

```
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
[20250121_040428] updated history: outcome=1, path=0x2
[20250121_040428] prediction for 0x40fcf2: y=0, confidence=0.00
```