Advanced Branch Prediction in SimpleScalar

Implementing Combinational and Perceptron-Based Predictors

ECE 687/587 Advanced Computer Architecture I

Mohamed Ghonim, Ahliah Nordstrom, Sai Mounisha Gurram, Mahalsa Sai Dontha

Agenda

01 Intro to Branch Prediction

02 SimpleScalar Simulator Overview

03 Project Overview

04 Combinational Two-Level Adaptive Predictor

05 Perceptron-Based Predictor

06 Implementation Challenges

07 Benchmarks

08 Conclusion

09 Acknowledgements

01: Introduction to Branch Prediction

CPU Performance

Reduction of pipeline stalls and increasing instruction throughput to improve overall processing speed and efficiency

Pipeline Stalls

Guessing outcome before computation to allow continuous instruction execution

Traditional Techniques

*Evolution from static to dynamic branch prediction

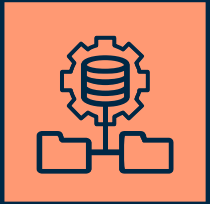
SimpleScalar

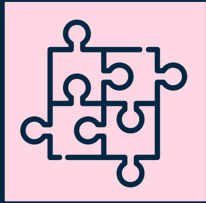
Preeminent implementation currently is single adaptive two-level branch predictor

02: SimpleScalar Simulator Overview

Relevance and Use Cases

- Widely used architectural simulator
- Versatile platform for experimenting advanced processor architectures
- Modularity and extensibility





Implementation in C

- Highly complex and comprehensive integration
- Weaving in a new BP within the established SimpleScalar
- Not as easy as only calling a BP function when needed

03: Project Overview

Motivation	Enhance simulator's predictive capabilities as much as possible **while also** contributing to the field of dynamic branch prediction
Combinational Two-Level Adaptive	This *obtainable* implementation is **highly adaptable** and designed to **enhance accuracy** while **reducing aliasing and conflict**
Perceptron	Notably *difficult* implementation that would reward us with **longer branch histories**, **complex pattern recognition**, and **superior performance**

04: Combinational Two-Level Adaptive Predictor

Duel Predictor Architecture

In parallel, requiring expansion of simulators data structures to indicate new PHT and counters

Meta-Predictor Integration

Essential for combinational approach as it utilizes saturating counters that correspond to each branch instruction to track/compare accuracy of the two predictors

History Length Management

Implemented mechanisms to dynamically manage and update history lengths with the use of algorithms that adjust based on current prediction outcomes and program behavior

Interface & Control Logic

Simulators interface modified by adding control logic to enable selection and configuration of combinational predictor.

```c
// -Project ////////////////////////////////////////////// Comb2Level //////
// BPredComb2Level: This function creates a combined two-level adaptive branch predictor.
// It integrates two distinct two-level predictors along with a meta predictor to
// dynamically select the most accurate predictor for each branch. This design aims
// to enhance prediction accuracy by leveraging the strengths of multiple prediction strategies.

struct bpred_t *bpred_create_BPredComb2Level(enum bpred_class class,  /* type of predictor to create */
                unsigned int twolev_1_l1_size,                        /* Prdictor 1 level-1 table size */
                unsigned int twolev_1_l2_size,                        /* Prdictor 1 level-2 table size */
                unsigned int twolev_2_l1_size,                        /* Prdictor 2 level-1 table size */
                unsigned int twolev_2_l2_size,                        /* Prdictor 2 level-2 table size */
                unsigned int meta_size,                               /* meta predictor table size */

                unsigned int shift_width_1,                           /* Predictor 1 history register width */
                unsigned int shift_width_2,                           /* Predictor 2 history register width */

                unsigned int xor_1,                                   /* Predictor 1 history xor address flag */
                unsigned int xor_2,                                   /* Predictor 2 history xor address flag */

                unsigned int btb_sets,                                /* number of sets in BTB */
                unsigned int btb_assoc,                               /* BTB associativity */
                unsigned int retstack_size)                           /* num entries in ret-addr stack */
```

Function Prototype for Creating Two-Level Adaptive Branch Predictor w/ Meta Predictor
(bpred.c)

```c
switch (class) {
case BPredComb2Lev:
    //Create the adaptive 2 level predictors 1 and 2, and a meta predictor that chooses which to use between them
    pred->dirpred.twolev_1  = bpred_dir_create( BPred2Level, twolev_1_l1_size, twolev_1_l2_size, shift_width_1, xor_1 );
    pred->dirpred.twolev_2  = bpred_dir_create( BPred2Level, twolev_2_l1_size, twolev_2_l2_size, shift_width_2, xor_2 );
    pred->dirpred.meta      = bpred_dir_create( BPred2bit, meta_size, 0, 0, 0);
```

Branch Predictor Initialization (bpred.c)

Function to Reset BP Counters Post-Priming (bpred.c)

```c
bpred_after_priming(struct bpred_t *bpred)
{
  if (bpred == NULL)
    return;

// Initializing the counter of used after priming.
// -Project ///////////////////////////////////////////// Comb2Level //////
  bpred-> used_twolev_1 = 0;
  bpred-> used_twolev_2 = 0;
// -Project ///////////////////////////////////////////// Comb2Level //////
```

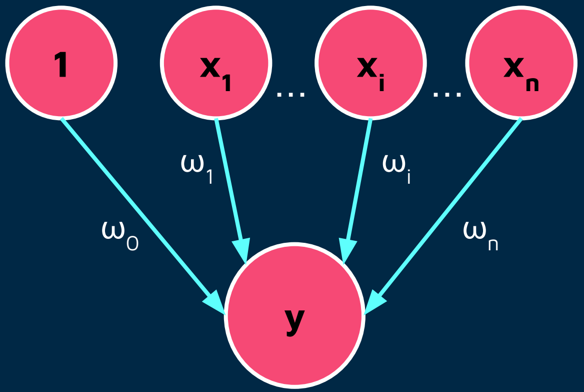
```c
// -Project ///////////////////////////////////////////// Comb2Level //////
    if (pred->class == BPredComb2Lev){
      if(dir_update_ptr->dir.meta)
        pred->used_twolev_1++;
      else
        pred->used_twolev_2++; }
// -Project ///////////////////////////////////////////// Comb2Level //////
```

BP Update Logic in Combined Predictor (bpred.c)

```c
// -Project ///////////////////////////////////////////
static int BPredComb2Lev_nelt = 9;
static int BPredComb2Lev_config[9] =
  {
  8,            /*predictor_1 l1size*/
  65536,        /*predictor_1 l2size*/
  8,            /*predictor_1 histry*/
  FALSE,        /*predictor_1 xor*/
  8,            /*predictor_2 l1size*/
  256,          /*predictor_2 l2size*/
  8,            /*predictor_2 histry*/
  FALSE,        /*predictor_2 xor*/
  1024          /*meta_table_size*/
  };
// -Project ///////////////////////////////////////////
```

Configuration Array (sim-outorder.c)

05: Perceptron-Based Predictor



$$w_0 + \sum n, i=1, x_i \; w_i = 0$$

Once output 'y' computed, we train it

If branch NT -> += -1

If branch T -> += 1

If sign(y_{out}) ≠ t OR $|y_{out}|$ ≤ 0

THEN for i = 0 to n DO

$w_i := w_i + tx_i$

Data Structure and Algorithm Integration

Development of efficient data structures to store weights and branch histories.

Handling Perceptron IO

Takes global branch history as inputs and outputs a prediction. Special attention w/ respect to SimpleScalars data handling conventions.

Resource Optimization

Limited potential increase in computational load by integrating w/ existing SimpleScalar BP infrastructure.

Compatibility w/ SimpleScalar Architecture

Designed for compatibility w/ different configurations of SimpleScalar, helpful to other users and uses.

Modular Design

Modular component to facilitate easy updates, debugging, and other future expansions.

Testing & Validation

Unit testing of individual components and integrated testing within the full simulator.

```c
// -Project ///////////////////////////////////////////// Perceptron //////
// Structure 'perc': Defines the perceptron predictor's internal mechanism.
// It includes the perceptron's weight table, which maps various branch history
// patterns to their corresponding weights, and a mask table used to select certain history bits.
// This structure is pivotal for the perceptron algorithm's learning and prediction processes.
  struct{
    int weight_i;                          /* weightt indices */
    int weight_bits;                       /* weight bits */
    int history;                           /* history length for global history */
    int lookup_out;                        /* output of each lookup*/
    signed int weight_table[400][400];     /* weight table, 2 dimensional array with an arbitrary large number
    signed int mask_table[100];            /* masks table */
    int i;                                 /* index */
  } perc;
// -Project ///////////////////////////////////////////// Perceptron //////
```

Perceptron Predictor Data Structure Definition (bpred.c)

```c
// -Project ////////////////////////////////////////////// Perceptron //////
// Here we implement the way the look up works in the perceptron predictor.
// This implementation is mimicing the way the perceptron predictor is implemented in the paper.
  case BPredPerc:
    {
  int index, j;
  signed int product[100], sum = 0;
  signed int output = 0;
  int *entry; // pointer to the perceptron entry

  product[0] = 0; // initialize the product to zero
  index = (baddr >> MD_BR_SHIFT) % pred_dir->config.perc.weight_i; // index = (baddr >> MD_BR_SHIFT) % perceptrons number
  pred_dir->config.perc.i = index; // set the perceptron index to the index of the branch address

  // set the first [0] bit of the history to 1 always to provide a bias (given in paper)
  pred_dir->config.perc.mask_table[0] = 1;

// calculating the output of the perceptron = w[0] + sum (w[i]*x[i]) where x[i] is the history of the branch outcomes
// and w[i] is the weight of the branch outcome history and w[0] is the bias
// sum (w[i]*x[i]) is calculated by multiplying each weight with the corresponding history bit and adding them together

      for (j=0; j < pred_dir->config.perc.history; j++) {// for each bit in the history

        // multiply the weight with the corresponding history bit and add it to the product
        product[j] = (pred_dir->config.perc.weight_table[index][j]) * (pred_dir->config.perc.mask_table[j]);
        output += product[j]; } // add the product to the output

      pred_dir->config.perc.lookup_out = output;            // set the perceptron output to the calculated output
      p = &pred_dir->config.perc.weight_table[index][j];    // set the pointer to the perceptron entry
      // & means the address of the perceptron entry (&variable means the address of the variable)
    }
    break;
```

Perceptron Lookup in BP (bpred.c)

Perceptron BP Weight Update (bpred.c)

```c
if (pred->class == BPredPerc)
{
  int j;
  signed int t, x[200];
  int theta;
  // This equation below is from the paper , page 5 threshold (theta) = [1.93h + 14] where h is the history length//
  theta = (1.93 * (pred->dirpred.bimod->config.perc.history) + 14);
  int index = pred->dirpred.bimod->config.perc.i;              // index = (baddr >> MD_BR_SHIFT) % perceptrons number
  int lookup_out = pred->dirpred.bimod->config.perc.lookup_out; // lookup_out = perceptron output
  if (taken)
    t = 1;  else  t = -1;

  if (lookup_out < 0) // if lookup_out is negative, make it positive
  lookup_out = (-1)*lookup_out;

  // The sign of the lookup_out is different from the sign of t means that the actual outcome is different from the predicted outcome
  // and we need to update the weights
```

This section updates weights of the perceptron-based branch predictor. It's executed after the actual branch direction is resolved.

Steps:

1. Calculate threshold (theta) for weight updating using formula from referenced paper.

2. Determine actual branch outcome (t) as 1 (taken) or -1 (not taken).

3. Adjust perceptron output (lookup_out) to always be positive for comparison.

4. Check if weights need to be updated based on the perceptron output and actual outcome.

5. Update weights based on history and actual outcome. Increment or decrement weights depending on the match with actual outcome. Clamp weights within the range [-128, 127] to avoid overflow.

6. Update the history of branch outcomes, shifting in the most recent outcome.

```c
  if (lookup_out <= theta || (lookup_out < 0 && t > 0) || (lookup_out >= 0 && t < 0)) {
    for (j=0; j < pred->dirpred.bimod->config.perc.history; j++)  // for each bit in the history
    {
      if (pred->dirpred.bimod->config.perc.mask_table[j] == 0)    // if the bit is 0, set x[j] to -1
      x[j] = -1;  else  x[j] = 1;                                  // else set x[j] to 1

      if (t == x[j])                                               // if the actual outcome is the same as the history bit, increment the weight
      pred->dirpred.bimod->config.perc.weight_table[index][j]++;
      else                                                         // else decrement the weight
      pred->dirpred.bimod->config.perc.weight_table[index][j]--;
      // clamp the weight for this specific implementation to avoid overflow
      if (pred->dirpred.bimod->config.perc.weight_table[index][j] > 127)
      pred->dirpred.bimod->config.perc.weight_table[index][j] = 127;

      // clamp the weight for this specific implementation to avoid overflow
      if (pred->dirpred.bimod->config.perc.weight_table[index][j] < -128)
      pred->dirpred.bimod->config.perc.weight_table[index][j] = -128;
    }

    for (j=1; j < pred->dirpred.bimod->config.perc.history; j++)          // shift the history bits to the right
    pred->dirpred.bimod->config.perc.mask_table[j-1] = pred->dirpred.bimod->config.perc.mask_table[j];
    //This operation effectively discards the oldest history bit and moves every other bit one step towards the start of the array.

    pred->dirpred.bimod->config.perc.mask_table[pred->dirpred.bimod->config.perc.history-1] = taken; // shift in the most recent outcome
    // After the shift, the newest branch outcome (taken) is placed at the end of the history array
    // This updates the history to include the most recent branch result while removing the oldest one.
  }
}
```

```c
// -Project /////////////////////////////////////////////// Perceptron //////
/* Perceptron predictor config (<l1size> <l12size> <shift_width>) */
static int perceptron_nelt = 3;
static int perceptron_config[3] =
  {
  128,           /* Index size weight */
  8,             /* Number of BHR history bits (1 bit for the sign + 7 (for 2^7 = 128) = 8 bits)*/
  27             /* history */
  };
```

Configuration Array (sim-outorder.c)

06: Implementation Challenges

	PREDICTOR	CHALLENGE	SOLUTION
Resource Management	Comb 2L Adaptive	Balancing additional resource requirements of dual predictors & meta-predictor	Optimize data structures to minimize memory overhead and ensure efficient utilization of resources
Accuracy Measurement & Debugging	Comb 2L Adaptive	Measuring accuracy and performance	Debugging tools to track predictor behavior and identify issues
Computational Complexity	Perceptron	Managing computational complexity	Algorithmic optimizations and structure proposed by Jimenez and Lin
Accuracy Verification	Perceptron	Ensuring accuracy and integrity of predictor	Extensive simulation runs for analyzing and diagnosing discrepancies

08: Our 9 Benchmarked Predictors

Perceptron Predictors

- Perceptron1: Shorter branch patterns (128 entries, 8-bit, 27-bit BHR)
- Perceptron2: Extended branch patterns (256 entries, 8-bit, 54-bit BHR)

Gshare predictor: Specialized two-level predictor

- Combines global history (1024 entries) with XOR flag

Comb2lev Predictor: Hybrid two-level predictor with a 1024-entry meta predictor table.
Comb Predictor: Combines a two-level predictor (1024 l1size, 1024 l2size, 10 history bits) and a bimodal predictor (4096 table size) with a 1024-entry meta predictor.

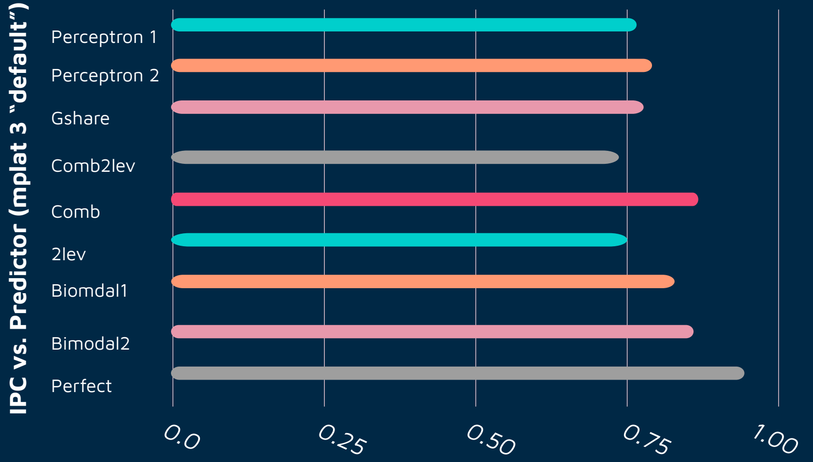
Bimodal Predictors

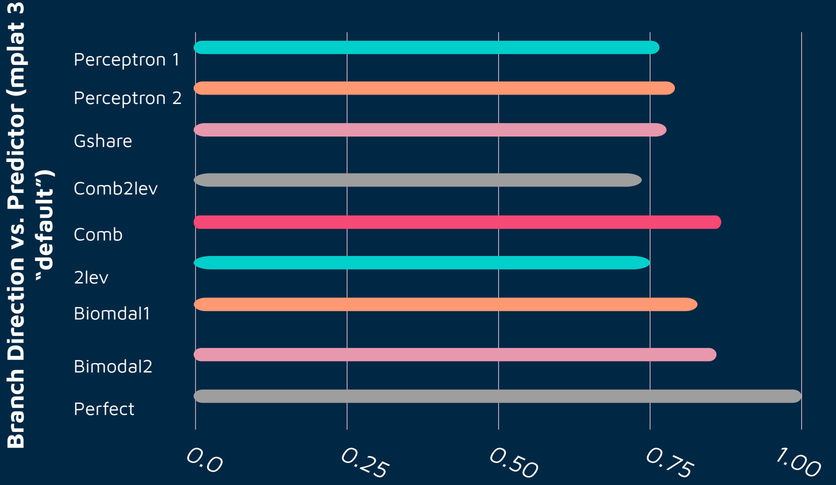
- Bimodal1: Default predictor with 2048-entry table.
- Bimodal2: Enhanced predictor with larger 16384-entry table.

Perfect Predictor: Ideal model with consistently correct predictions. Used as a benchmark.

07: Benchmarks – MPLAT 3

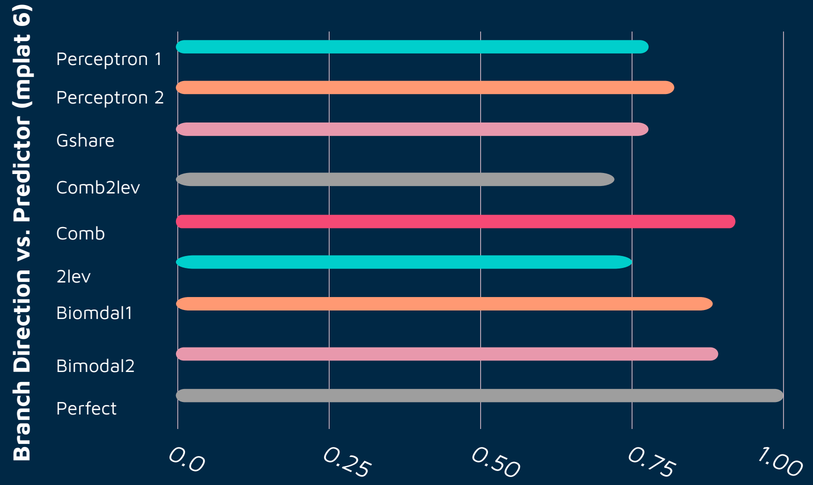
Predictor	IPC	Dir_Prediction	MPLAT	Benchmark
Perceptron1	0.7759	0.7915	3 "Default"	gcc
Perceptron2	0.8003	0.8245		
Gshare	0.7756	0.7895		
Comb2lev	0.735	0.7263		
Comb	0.8604	0.9034		
2Lev	0.7498	0.7563		
Biomdal1	0.8438	0.8832		
Bimodal2	0.8548	0.8999		
Perfect	0.9425	1		





07: Benchmarks – MPLAT 6

Predictor	IPC	Dir_Prediction	MPLAT	Benchmark
Perceptron 1	0.7001	0.7915	6	gcc
Perceptron 2	0.7303	0.8246		
Gshare	0.6989	0.7897		
Comb2lev	0.6491	0.7263		
Comb	0.8089	0.9035		
2Lev	0.6683	0.7572		
Biomdal1	0.7865	0.8832		
Bimodal2	0.8023	0.8999		
Perfect	0.9425	1		



08: Conclusions and Future Work

Performance Insight

Traditional predictors like Bimodal1 and Bimodal2 often outperform more complex models like perceptrons in terms of IPC, though the latter excel in prediction accuracy

Design Trade-Offs

Perceptron predictors, despite their lower IPC, offer high Dir_Prediction rates, illustrating a trade-off between computational intensity and prediction accuracy.

Combined Predictor Advantages

The Comb predictor showcased an impressive balance of high IPC and Dir_Prediction rates, suggesting the effectiveness of combining different prediction strategies

Optimizing Perceptron

Future work could focus on optimizing perceptron predictors for better IPC performance while maintaining their prediction accuracy.

Hybrid Predictor

Investigating hybrid predictors that integrate the strengths of various models could lead to advancements in branch prediction efficiency.

Differing Workload Adaptation

Evaluate performance of these predictors across a broader range of workloads and architectural configurations to validate generalizability and effectiveness in diverse scenarios.

09: Acknowledgments

Thank You!

We express our gratitude to our ECE 587/687 instructor, Yuchen Huang, and Adjunct Support Faculty, Venkatesh Srinivas, for their invaluable guidance and support throughout this project. Their expertise in computer architecture and practical insights significantly contributed to our learning and the successful implementation of advanced branch predictors in the SimpleScalar simulator.

References

[1] D. A. Jimenez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in Proceedings of the Seventh International Symposium on High-Performance Computer Architecture, 2001, pp. 197-206.

[2] S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.