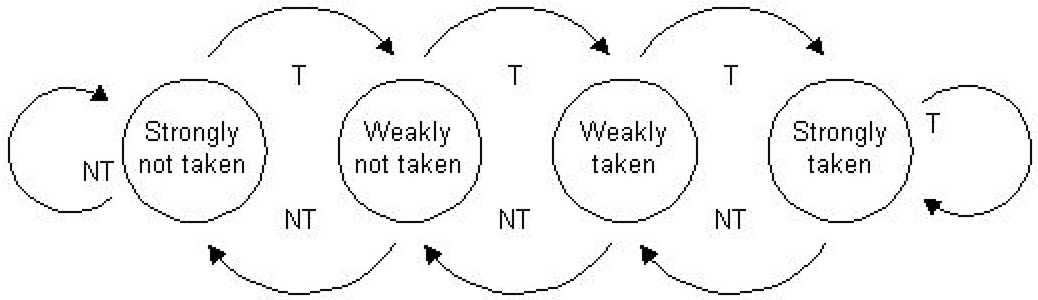
ECEN/CSCI 5593
Advanced Computer Architecture



Checkpoint 1A: Branch Prediction Simulation with PIN

Diana Southard

Summer 2016

Introduction

The goal for this checkpoint was to provide us students with experience using the PIN tool while also learning how to implement various branch prediction schemes. The prediction implementations were checked against four program application benchmarks, with the results compared to see which branch prediction algorithm provided the greatest accuracy, as determined using the number of branches correctly predicted compared to the total number seen.

Work

There were five different branch prediction schemes implemented and tested:

- **Single-bit predictor**: a 1-bit predictor storing (taken or not-taken last time). For this type of scheme, each branch is predicted based solely on the feedback from the last branch. If the previous branch was taken, then the next branch will be predicted as 'Taken,' and vice versa. This is shown below in Fig. 1.
- **Bi-model bit predictor**: a 2-bit predictor saturating counter. In this scheme, the prediction will only change if a branch is mispredicted twice. This is shown on the figure displayed on the coversheet. Going from predicting 'Strongly taken' to weakly taken' will require two misses in a row, and vice versa.
- **Two-level (GAg) predictor** using a 8-bit history register and a 2^8 entry pattern history table. Each pattern table entry has a 2-bit saturating counter. This two-level scheme no longer relies on just a single branch's history -- instead, the global history of all taken/not-taken patterns is used to determine what the next prediction should be. This is show below in Fig. 2.
- **Two-level (PAg) predictor** using a 12-bit local branch history and a 2^12 entry pattern history table. Each pattern table entry has a 2-bit saturating counter. Similar to the GAg scheme, each prediction is based on a global history pattern table. However, there is a single history register for each branch, making each branch's history independant. This is shown below in Fig. 3.

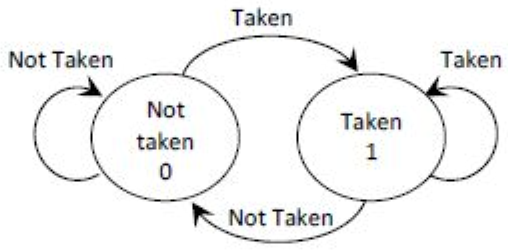
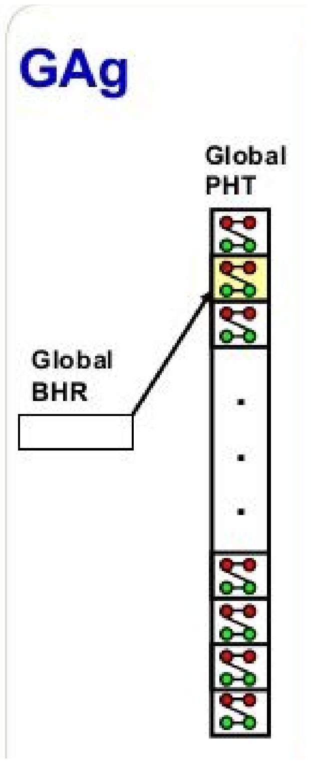


Fig. 1 One-Bit Predictor Scheme Graphic



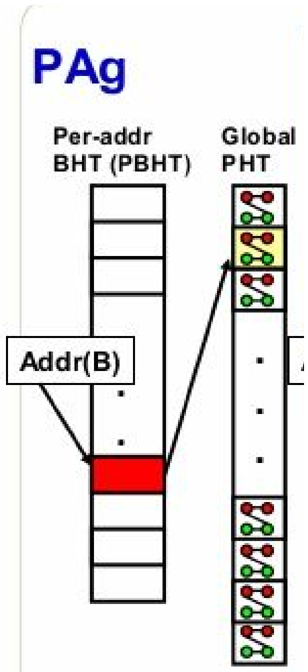


Fig. 2 GAg Predictor Scheme Graphic **Fig. 3 PAg Predictor Scheme Graphic**

To implement these prediction schemes, a direct-mapped Branch Target Buffer (BTB) was constructed in C++. The BTB was hardcoded to contain 1024 entries per the instructions of this assignment. Because there were 1024 entries, the lower 10-bit field of each instruction was used to locate its BTB entry. To determine the index of that branch in the BTB, the instruction was masked with 0x3FF. For the GAg and PAg schemes, predictions were taken by referencing a global history table's counter entries. The GAg's history table contained 2^8 entries (256) while the PAg's table contained 2^12 entries (4096), per the assignment instructions. To reference the correct entry in the global history table, both of the two-level schemes also used masking against their respectively referenced history register, 0xFF for the GAg and 0xFFF for the PAg.

Each entry in the BTB contained: the entry tag information for verifying the correct branch address, the valid bit to note whether the entry was a valid address, and prediction information.

For a 1-bit, this was a boolean set to true or false as the scheme demanded. This was initialized to false for *Not Taken*. For the 2-bit, it was a saturating counter (counting from 0 *Strongly Not Taken* up to 3 - *Strongly Taken*). This was initialized to Strongly Not Taken. For the PAg, this was a history register for that particular entry, initialized to a history of all *Not Taken* (0x0).

This BTB structure can be seen in the code snippet below.

Code Snippet: BTB Entry Structure

```
/* BTB = Branch Target Buffer
* entry_bit:
```

```
* Entry structures for the BTB Table
* 1-bit: uses valid, prediction, tag, and replaceCount variables
* 2-bit: also uses counter variable in addition to 1-bit variables
* GAg: uses BTB_History, BTB_HistoryLength, and BTB_Table
*/
struct entry_bit
{
        bool valid;        // Maintains track of valid entries in table
        bool prediction;   // Marks prediction choice for current entry
        UINT64 tag;        // Tag of current entry in table
        UINT64 ReplaceCount;     // Maintains count of total entries
replaced
        UINT8 counter;     // Used for 2-bit prediction, keep track of
current prediction counter
        UINT64 branchHistoryRegister;  // Used for PAg prediction, branch's
individual history
}BTB[BTB_SIZE];
```

In the case of the GAg scheme, a global register was used to maintain the history of all branches. This history was also initialized to a history of all *Not Taken* (0x0).

The GAg and PAg schemes both relied on using different-sized global history pattern tables. To this end, a global pointer was used, initialized depending on which prediction scheme was being used to point to a table of the right size. Each entry of the history pattern table contained a 2-bit saturating counter, with all counters being initialized to *Weakly Taken* per the assignment instructions. This can be seen below.

Code Snippet: Global History Pattern Initialization

```
if ((predictionType == GAg) | (predictionType == HYBRID)) {
        // initialize values in HistoryPatternTable
        // Dynamically assign size of HistoryPatternTable
        //     UINT8 HistorySize = 8;
        //     HistoryPatternTable = new unsigned char[pow(2.0,
HistorySize)];
        // assign space for table with required size (8-bit history, 2^8
entries in PatternTable)
        HistoryPatternTable = new unsigned char[BTB_GAG_TABLE_SIZE];
        // initialize values in table, all initialized to 'Taken'
        for (i = 0; i < BTB_GAG_TABLE_SIZE; i++){
                HistoryPatternTable[i] = 0x2;
        }
```

Other global variables included counters to track the total number of instructions seen, branches taken, branches correctly predicted, addresses missed in the BTB, and the number of BTB entries replaced.

Additionally, the code was written to be able to control, at run-time, which type of prediction scheme was being used. In order to facilitate this, an enum class named PredictionType was created and a commandline switch in the form of a pintool Knob was added to the tool. During the running of the tool, adding a -b prefix and then a choice between the available branch prediction schemes could change which scheme was used. The default was the one-bit predictor. The other choices were: two-bit [-b 1], GAg [-b 2], PAg [-b 3], and Hybrid [-b 4].

These global variables and the determination of the prediction type being used can be seen below.

Code Snippet: Global Variables

```
/* ================================================================== */
/* Global Variables */
/* ================================================================== */
UINT64 CountSeen = 0;
UINT64 CountTaken = 0;
UINT64 CountCorrect = 0;
UINT64 CountMissed = 0;
UINT64 CountReplaced = 0;
// Enum class to keep track of which branch prediction scheme will be
used,
// Default = 1-bit prediction, can be set via command line
enum PredictionType {
      ONE_BIT, TWO_BIT, GAg, PAg, HYBRID
};
// Variable to keep track of which prediction scheme is being used
PredictionType predictionType = ONE_BIT;
```

Code Snippet: Setting Prediction Scheme Using PIN Knob

```
/* ================================================================== */
/* Commandline Switches */
/*
// Add knob to change which type of branch-prediction scheme is being
used
// Variable: written once, coming from pin tool, specify the output name,
with usage comment in case of error
KNOB<UINT64> KnobBranchPrediction(KNOB_MODE_WRITEONCE, "pintool",
            "b", "0", "specify branch prediction scheme being used [0 =
1-bit, 1 = 2-bit, 2 = GAg, 3 = PAg, 4 = Hybrid (Combo of 2-bit and GAg)
[defaults to 1-bit]");
```

With these prediction schemes, the following scenarios with branches seen were encountered:

1. An instruction is located in the BTB, and it is correctly predicted
2. An instruction is located in the BTB, and it is incorrectly predicted
3. An instruction is missed in the BTB, but it was not taken (correctly predicted)
4. An instruction is missed in the BTB and was taken, resulting in a new instruction being added to the BTB.

The count of correctly predicted branches is incremented in the appropriate cases. In cases 1 and 2, the BTB entry containing the instruction has its prediction scheme updated determining on their own algorithms and the correctness of the prediction.

In case 3, aside from incrementing the count of correctly predicted branches, the global history register has a *Not Taken* if the GAg scheme is being used, with the corresponding history pattern table entry set to *Not Taken*.

In case 4, a new branch is inserted in the BTB. For the 1-bit and 2-bit schemes, entries in the BTB tables are only initially created when a branch is first taken. Whenever a new branch is inserted, its prediction is set to *Taken* or *Weakly Taken*, for 1- or 2-bit predictors. For the PAg, the branch's history register is set to 0x1, indicating that only the most recent branch (the newly inserted one) was taken. For the GAg, a *Taken* is shifted into the global branch history register. For the PAg and GAg schemes, the corresponding history pattern table entry is then set to *Taken*.

Hybrid Predictor

The assignment required us to come up with our own prediction scheme. I chose a Tournament Predictor, a hybrid mixing both local and global predictors -- the two-bit and the GAg predictor. To chose which prediction to use, I maintained a global hybridCounter variable that functioned as a saturating counter. If the hybridCounter predicted *Taken* (hybridCounter >1), then the GAg scheme was used. Otherwise, the two-bit prediction was.

All of the above BTB functions can be seen in the appendix of this report.

Results

The PIN tool was generated and the resulting .so file was run on the following Spec2006 integer programs: bzip2, sjeng, libquantum, and h264. The PIN Knob was set to run the program for only 10,000,000,000 branch instructions before exiting. This PIN Knob setting is accessed by using the -l prefix [-l 10000000000].

To generate the shell script that would run the PIN tool on the benchmark programs, the following command line template was followed:

```
bench-run.pl --bench spec-cpu2006:int:401.bzip2:train --build base --prefix
"pin -t <pin_tool_name> --" --copy output.out --log LOG
```

The Perl script would itself create another script to be run on the program line in order to compute the simulations. Because there were four different benchmarks I was running, each of those on the five different branch prediction schemes, I wrote a buildAll.sh script. This script would first set up the benchmark scripts for each of the different branch prediction schemes under the different benchmark applications, and then run each individually, storing resulting outputs in aptly named files.

The results of this execution can be seen below, with a graphical representation of the varying prediction accuracies among the multiple application runs.

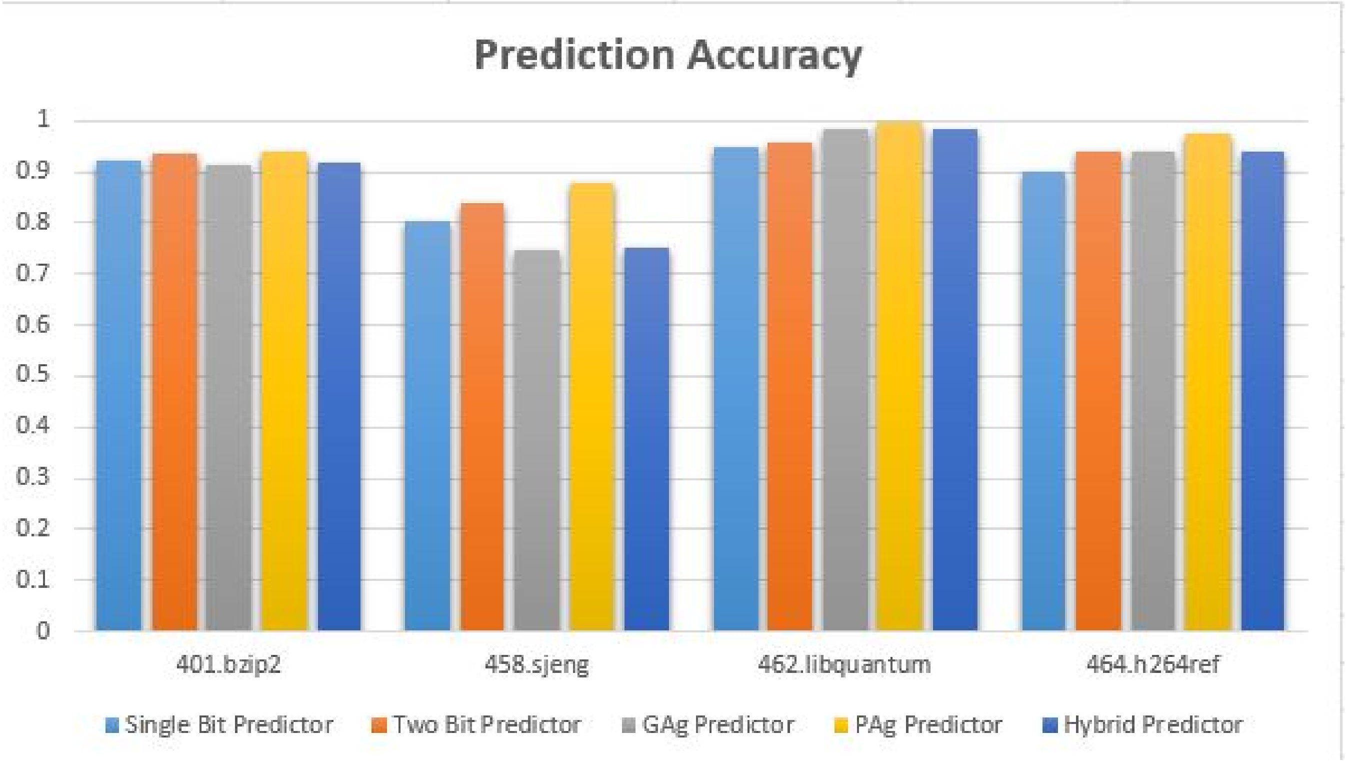
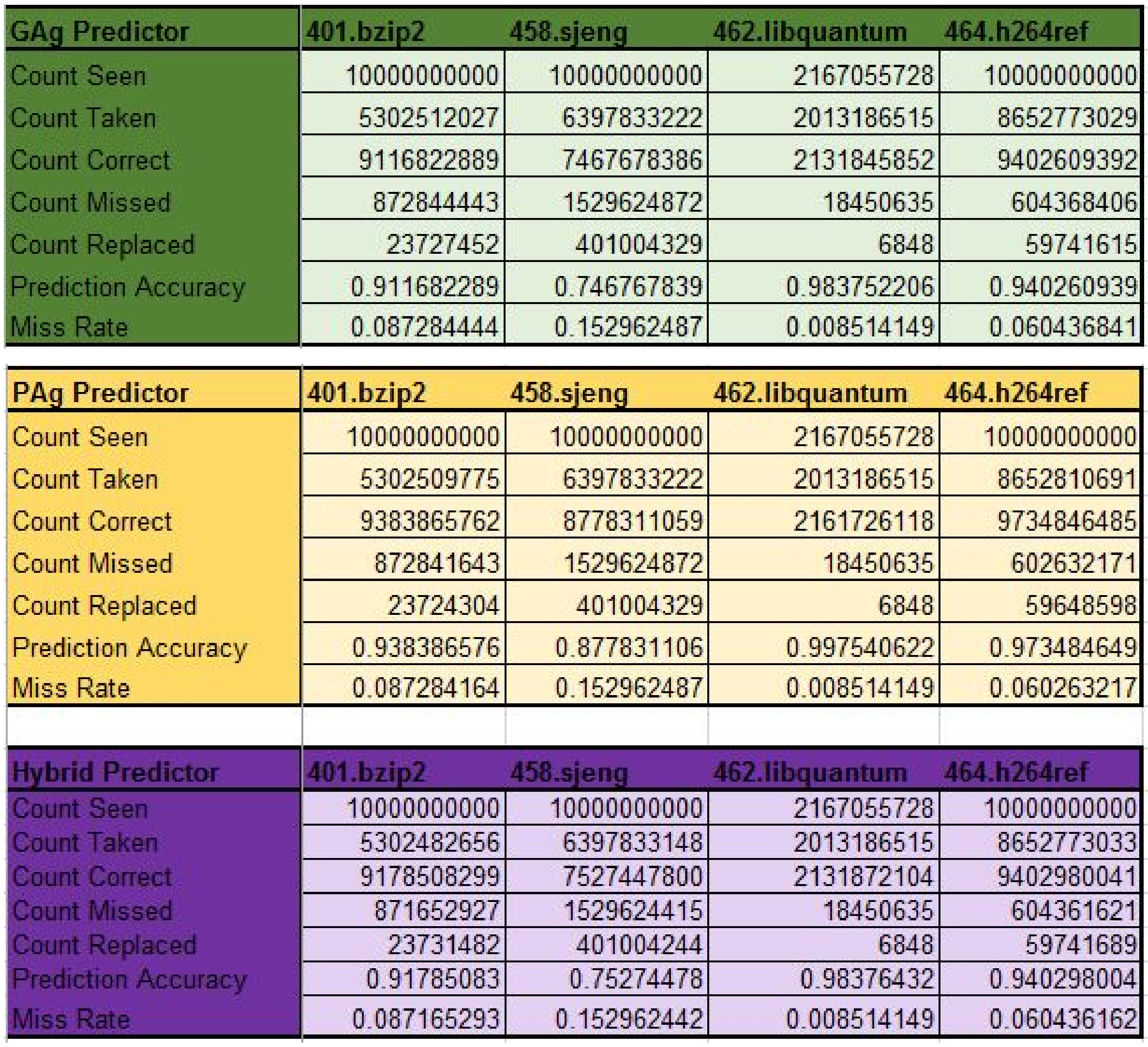


Fig. 4 Graphical Results of Benchmark Executions: Prediction Accuracy

Single Bit Predictor	401.bzip2	458.sjeng	462.libquantum	464.h264ref
Count Seen	10000000000	10000000000	2167055723	10000000000
Count Taken	5302510519	6397833222	2013186521	8652775624
Count Correct	9200091726	8041598416	2051711125	8983606868
Count Missed	872842570	1529624872	18450625	604352945
Count Replaced	23725352	401004329	6842	59749108
Prediction Accuracy	0.920009173	0.804159842	0.946773589	0.898360687
Miss Rate	0.087284257	0.152962487	0.008514144	0.060435295

Two Bit Predictor	401.bzip2	458.sjeng	462.libquantum	464.h264ref
Count Seen	10000000000	10000000000	2167055723	10000000000
Count Taken	5302512818	6397833222	2013186521	8652775622
Count Correct	9355667990	8393279459	2075185057	9393056629
Count Missed	872845404	1529624872	18450625	604357658
Count Replaced	23728502	401004329	6842	59749151
Prediction Accuracy	0.935566799	0.839327946	0.957605767	0.939305663
Miss Rate	0.08728454	0.152962487	0.008514144	0.060435766



GAg Predictor	401.bzip2	458.sjeng	462.libquantum	464.h264ref
Count Seen	10000000000	10000000000	2167055728	10000000000
Count Taken	5302512027	6397833222	2013186515	8652773029
Count Correct	9116822889	7467678386	2131845852	9402609392
Count Missed	872844443	1529624872	18450635	604368406
Count Replaced	23727452	401004329	6848	59741615
Prediction Accuracy	0.911682289	0.746767839	0.983752206	0.940260939
Miss Rate	0.087284444	0.152962487	0.008514149	0.060436841

PAg Predictor	401.bzip2	458.sjeng	462.libquantum	464.h264ref
Count Seen	10000000000	10000000000	2167055728	10000000000
Count Taken	5302509775	6397833222	2013186515	8652810691
Count Correct	9383865762	8778311059	2161726118	9734846485
Count Missed	872841643	1529624872	18450635	602632171
Count Replaced	23724304	401004329	6848	59648598
Prediction Accuracy	0.938386576	0.877831106	0.997540622	0.973484649
Miss Rate	0.087284164	0.152962487	0.008514149	0.060263217

Hybrid Predictor	401.bzip2	458.sjeng	462.libquantum	464.h264ref
Count Seen	10000000000	10000000000	2167055728	10000000000
Count Taken	5302482656	6397833148	2013186515	8652773033
Count Correct	9178508299	7527447800	2131872104	9402980041
Count Missed	871652927	1529624415	18450635	604361621
Count Replaced	23731482	401004244	6848	59741689
Prediction Accuracy	0.91785083	0.75274478	0.98376432	0.940298004
Miss Rate	0.087165293	0.152962442	0.008514149	0.060436162

Fig. 5 Results from Application Runs

From these results, the following questions can be answered:

The predictor with the highest accuracy is regularly the PAg. The GAg predictor performed the worst for the sjeng application, indicating that the sjeng application has the least amount of global correlation. It performed the best for the libquantum application, indicating that the opposite is true for libquantum.

Conclusion

There is something incorrect with my implementation, as can be seen in the miss rate chart shown below.

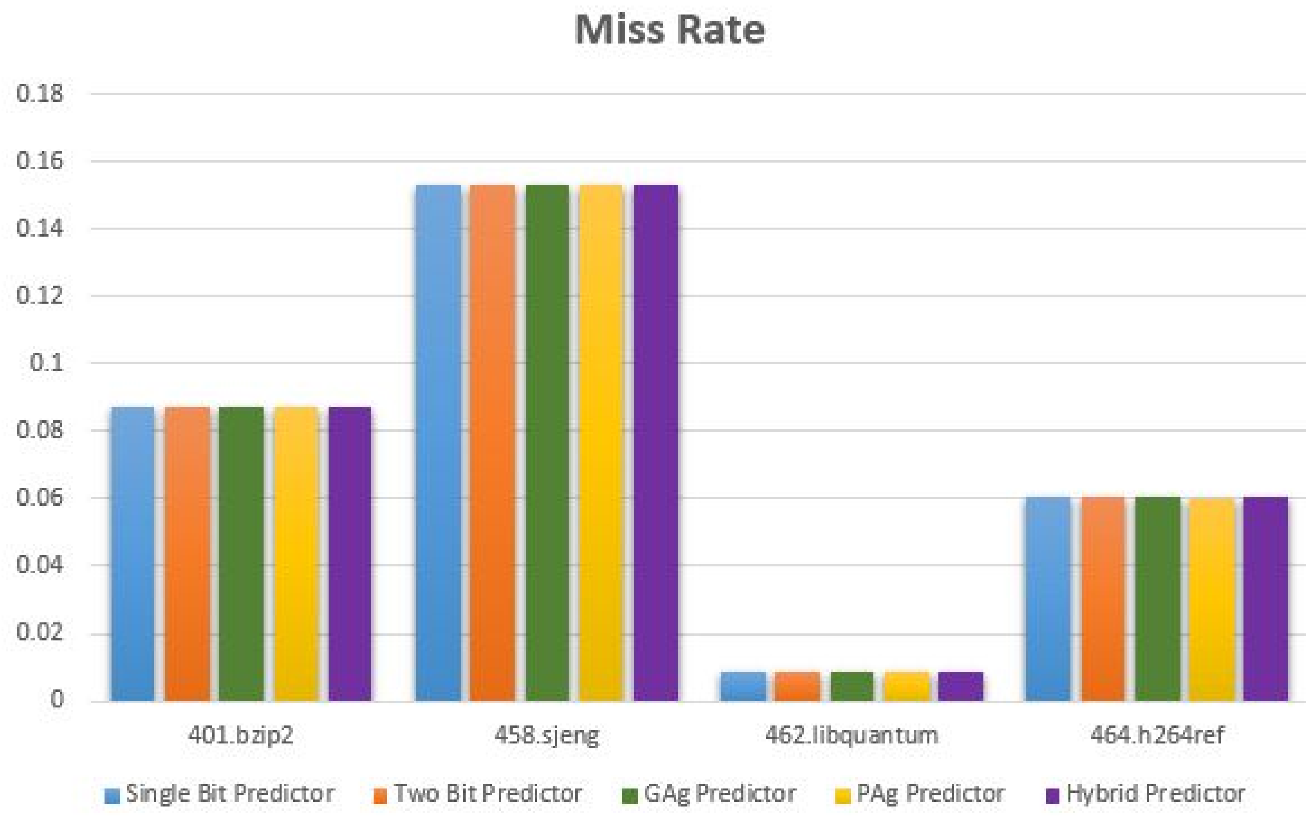


Fig. 6 Graphical Results of Benchmark Executions: Miss Rate

In Fig. 4, the prediction rates are all abnormally high. No prediction scheme accuracy falls below 70%, whereas it is expected that 1- and 2-bit predictors should have accuracy rates significantly lower than that. I was expecting an accuracy rate of ~50% for the 2-bit. In the miss rate, it looks that every type of predicting scheme had nearly identical miss rates when compared to their application runs.

After changing the implementation schemes around, however, this final presented implementation is the only one that makes sense to me. The 1-bit and 2-bit schemes follow what was either originally given or what was discussed thoroughly in lectures, and this high prediction accuracy still resulted.

The hybrid predictor obviously inherits the weaknesses of the combined prediction schemes, as can be seen in the results of the sjeng run. The GAg prediction accuracy is the lowest in that application run, followed by the hybrid, most likely due to the hybrid predictor relying on the GAg scheme. A better combination should be possible -- perhaps combining the PAg and 2-bit schemes, especially in the case of the sjeng application run where the PAg and 2-bit predictors had the highest and second-highest accuracy, respectively.

Appendix

Code Snippet: BTB Table Initialization Function

```
/* initialize the BTB */
VOID BTB_init()
{
        int i;
        for(i = 0; i < BTB_SIZE; i++)
        {
                // All entries are initially false
                BTB[i].valid = false;
                // All entry predictions are initially 'Not Taken' (false)
                BTB[i].prediction = false;
                // All entry tags are initially zeroed-out
                BTB[i].tag = 0;
                // Initialize ReplaceCount to 0
                BTB[i].ReplaceCount = 0;
                // Initialize counter (for 2-bit predictor) to 'Not Taken' (1)
                BTB[i].counter = 1;
                // Initialize branch history (for PAg)
                /* HINT: in the case of PAg, you will need to add a field to the
BTB structure to
                 * maintain the history for the branch of that entry.
                 */
                BTB[i].branchHistoryRegister = 0x0;
        }
        // Initialize counters/registers to 0
        GlobalHistoryRegister = 0x0;
        hybridCounter = 0x2;     // Initialized to a weak taken
        /* if using GAg scheme, initialize the HistoryPatternTable
         *
         * HINT: in the case of GAg, you will need to add a history register and
history table
         * Something like:
         * unsigned int BTB_History = 0;
         * unsigned int BTB_HistoryLength = 8;
         *     unsigned char BTB_Table[256];
         *
         * The table would be initialized to 2 for weak taken, and you would
access the table using:
         *    prediction = BTB_Table[BTB_History & 0xFF];
         *    if (prediction > 1)
         *          prediction is taken
         *    else
         *          prediction is not taken
         *
         * Hybrid scheme: combines 2-bit and GAg
         */
        if ((predictionType == GAg) | (predictionType == HYBRID)) {
                // initialize values in HistoryPatternTable
                // Dynamically assign size of HistoryPatternTable
                //      UINT8 HistorySize = 8;
                //      HistoryPatternTable = new unsigned char[pow(2.0,
HistorySize)];
                // assign space for table with required size (8-bit history, 2^8
entries in PatternTable)
                HistoryPatternTable = new unsigned char[BTB_GAG_TABLE_SIZE];
                // initialize values in table, all initialized to 'Taken'
                for (i = 0; i < BTB_GAG_TABLE_SIZE; i++){
                        HistoryPatternTable[i] = 0x2;
                }
        }
        /*
         * If using PAg scheme, initialize HistoryPatternTable
         */
        if (predictionType == PAg){
                // initialize values in BTB_Table
                // Dynamically assign size of HistoryPatternTable
                //      UINT8 HistorySize = 8;
                //      HistoryPatternTable = new unsigned char[pow(2.0,
HistorySize)];
                // assign space for table with required size (12-bit history, 2^12
entries in PatternTable)
                HistoryPatternTable = new unsigned char[BTB_PAG_TABLE_SIZE];
                // initialize values in table, all initialized to 'Taken'
                for (i = 0; i < BTB_PAG_TABLE_SIZE; i++){
                        HistoryPatternTable[i] = 0x2;
                }
        }
}
```

Code Snippet: Branch Prediction Function

```
VOID br_predict(ADDRINT ins_ptr, INT32 taken)
{
        CountSeen++;
        if (taken)
                CountTaken++;
        if(BTB_lookup(ins_ptr))
        {
                if(BTB_prediction(ins_ptr) == taken) CountCorrect++;
                BTB_update(ins_ptr, taken);
        }
        else
        {
                CountMissed++;      // Keep track of miss rate
                if(!taken) {
                        CountCorrect++;    // Correctly predicted 'Not Taken'
                        // Shift in new taken history into the global history
register
                        if ((predictionType == GAg) | (predictionType == HYBRID)) {
                                GlobalHistoryRegister = (GlobalHistoryRegister << 1) |
taken;
                                HistoryPatternTable[GlobalHistoryRegister && GAgMASK] =
0x1;         // Set History Pattern Table to Not Taken
                        }
                }
                else BTB_insert(ins_ptr);
        }
        if(CountSeen == KnobBranchLimit.Value())
        {
                WriteResults(true);
                exit(0);
        }
}
```

Code Snippet: BTB Entry Insertion Function

```
/* insert a new branch in the table */
VOID BTB_insert(ADDRINT ins_ptr)
{
        UINT64 index;
        index = mask & ins_ptr;
        if(BTB[index].valid)
        {
                BTB[index].ReplaceCount++;
                CountReplaced++;
        }
        BTB[index].valid = true;
        BTB[index].tag = ins_ptr;
        switch (predictionType) {
        case TWO_BIT:
                BTB[index].counter = 2;   // Predict next branch as weakly taken
[2-bit];
                break;
        case GAg:
                // Shift in 'Taken' into the global history register
                GlobalHistoryRegister = (GlobalHistoryRegister << 1) | 0x1;
                break;
        case PAg:
            BTB[index].branchHistoryRegister = 0x1;     // Set up taken in new
history
            HistoryPatternTable[BTB[index].branchHistoryRegister && GAgMASK] =
0x2;  // Set history pattern table counter entry to Taken
            break;
      case HYBRID:
            BTB[index].counter = 2;                           // Predict next
branch as weakly taken [2-bit];
            GlobalHistoryRegister = (GlobalHistoryRegister << 1) | 0x1;
            HistoryPatternTable[GlobalHistoryRegister && GAgMASK] = 0x2;  // Set
history pattern table counter entry to Taken
            break;
      default:
            BTB[index].prediction = true;                 // Missed branches
always enter as taken/true
      }
}
```

Code Snippet: BTB Entry Update Function

```
/* update the BTB entry with the last result */
VOID BTB_update(ADDRINT ins_ptr, bool taken)
{
      UINT64 index;
      index = mask & ins_ptr;
      switch (predictionType){
      case TWO_BIT:
            // Update if branch was taken, increase strength of taken
prediction
            if (taken) {
                  if (BTB[index].counter < 3) BTB[index].counter++;
            }
            // Update if branch was not taken, increase strength of not-taken
prediction
            else {
                  if (BTB[index].counter > 0) BTB[index].counter--;
            }
            return;
      case GAg:

            // Update counter in HistoryTable
            if (taken) {
                  if (HistoryPatternTable[(GlobalHistoryRegister & GAgMASK)] <
3) {
                        HistoryPatternTable[(GlobalHistoryRegister &
GAgMASK)]++;
                  }
            }
            // Update if branch was not taken, increase strength of not-taken
prediction
            else {
                  if (HistoryPatternTable[(GlobalHistoryRegister & GAgMASK)] >
0) {
                        HistoryPatternTable[(GlobalHistoryRegister &
GAgMASK)]--;
                  }
            }
            // Shift in new taken history into the global history register
            GlobalHistoryRegister = (GlobalHistoryRegister << 1) | taken;
            return;
      case PAg:
            // Update counter in HistoryTable based on branchHistoryRegister
            if (taken) {
                  if (HistoryPatternTable[(BTB[index].branchHistoryRegister &
PAgMASK)] < 3) {
                        HistoryPatternTable[(BTB[index].branchHistoryRegister &
PAgMASK)]++;
                  }
            }
            // Update if branch was not taken, increase strength of not-taken
prediction
            else {
                  if (HistoryPatternTable[(BTB[index].branchHistoryRegister &
PAgMASK)] > 0) {
                        HistoryPatternTable[(BTB[index].branchHistoryRegister &
PAgMASK)]--;
                  }
            }
            // Shift in new taken history into branch history register
            BTB[index].branchHistoryRegister =
(BTB[index].branchHistoryRegister << 1) | taken;
            return;
      case HYBRID:
            if (hybridCounter > 1) {
                  // Update the GAg scheme

                  // Update counter in HistoryTable
                  if (taken) {
                        if (HistoryPatternTable[(GlobalHistoryRegister &
GAgMASK)] < 3) {
                              HistoryPatternTable[(GlobalHistoryRegister &
GAgMASK)]++;
                        }
                  }
                  // Update if branch was not taken, increase strength of
not-taken prediction
```

```
                    else {
                            if (HistoryPatternTable[(GlobalHistoryRegister &
GAgMASK)] > 0) {
                                    HistoryPatternTable[(GlobalHistoryRegister &
GAgMASK)]--;
                            }
                    }
                    // Shift in new taken history into the global history
register
                    GlobalHistoryRegister = (GlobalHistoryRegister << 1) | taken;
            }
            else {
                    // Update the 2-bit scheme
                    // Update if branch was taken, increase strength of taken
prediction
                    if (taken) {
                            if (BTB[index].counter < 3) BTB[index].counter++;
                    }
                    // Update if branch was not taken, increase strength of
not-taken prediction
                    else {
                            if (BTB[index].counter > 0) BTB[index].counter--;
                    }
            }
            // Now update the hybrid counter
            if (taken) {
                    if (hybridCounter < 3) hybridCounter++;
            }
            else {
                    if (hybridCounter > 0) hybridCounter--;
            }
            return;
        default:
            // Set prediction based on what was last done
            BTB[index].prediction = taken;
            break;
        }
}
```

Code Snippet: BTB Entry Prediction Determination Function

```
/* return the prediction for the given address */
bool BTB_prediction(ADDRINT ins_ptr)
{
        UINT64 index;
        index = mask & ins_ptr;
        int predict;
```

```
      switch (predictionType){
      case TWO_BIT:
            // Return based on strength of counter (0-1: Not taken, 2-3: Taken)
            return (BTB[index].counter > 1 ? 1:0);
            break;
      case GAg:
            // Use 8-bit history in HistoryRegister
            return (HistoryPatternTable[(GlobalHistoryRegister & GAgMASK)] > 1
? 1:0);
      case PAg:
            // Use 12-bit history in individual branchHistoryRegister based on
index
            predict = HistoryPatternTable[(BTB[index].branchHistoryRegister &
PAgMASK)];
            return (predict > 1 ? 1:0);
      case HYBRID:
            // if hybridCounter predicts 'Taken,' return the GAg prediction.
Otherwise, return 2-bit prediction
            if (hybridCounter > 1) {
                  return (HistoryPatternTable[(GlobalHistoryRegister &
GAgMASK)] > 1? 1:0);
            }
            else{
                  return (BTB[index].counter > 1 ? 1:0);
            }
      default:
            // Return prediction stored in individual BTB entry
            return BTB[index].prediction;
      }
}
```