# 

# Branch Prediction Project

# Read-Me

# Iftequar Iqbal Mohammed

# **Data Structure Implementation**

Programming's basic data structures, arrays, are used extensively in the implementation of the Branch History Table (BHT) and Pattern History Table (PHT).

The Branch History Table (BHT) can be conceptualized as a list in which each entry denotes a distinct branch that was encountered while the application was running. For the BHT, we employ an array, where each member contains data about a certain branch. This data comprises the history of the branch, whether it was taken or not, and it forms the foundation for forecasting the probability that the branch will be taken in the future. In order to maintain track of the history of a branch, we effectively update the associated record in the BHT whenever a branch occurs.

Pattern History Table (PHT): An array is used to organize both the PHT and the BHT. But its main objective is to capture trends found in recent branch results. In this case, every entry in the PHT array denotes a distinct branch behavior pattern. The values corresponding to these items represent the likelihood that the corresponding branch will be taken or not. For example, the relevant entry in the PHT might show a higher probability of a branch being taken if a specific pattern tends to lead to that outcome.

Explanation of Array Usage: Because arrays make it possible to access and organize data in an organized manner, they are especially useful for this kind of work. The array index allows us to quickly find the corresponding element for a branch when it occurs in the BHT or PHT. The effective tracking of branch histories and patterns is made possible by this well-organized structure. Additionally, arrays improve the efficiency of these prediction algorithms by making it simpler to update and obtain branch-specific data while the program is running.

In conclusion, branch and history data can be organized and effectively managed through the use of arrays in BHT and PHT implementations. This improves prediction accuracy and, as a result, program branch prediction systems' overall performance.

# **Algorithm - Branch Prediction**

I used the GAp and PAp techniques to implement Two-level Branch Prediction in this project. I've concentrated on the final 12 bits of the branch address in particular, and I indexed each entry in the Pattern History Table (PHT) using an 8-bit Branch History Register (BHR). The Branch History Table (BHR) in the PAp scheme is 2^12 x 8, whereas in the GAp scheme it is global. In each scenario, the PHT measures 2^12 x 2^8. There's an entry in the PHT that is referenced for each branch address and BHR value. These PHT entries serve as the basis for the predictions. The PHT and BHR are updated in accordance with the actual branch outcome after prediction.

This flowchart is a simulation of a branch prediction algorithm

**Initialization:**

The program starts by initializing some variables and data structures, such as arrays to store predictions and a "Branch History Register" (bhr) to keep track of recent branches.

**Reading Input:**

It reads input from a file, where each line represents a branch in a computer program. Each line includes a branch address (in hexadecimal), the actual outcome of the branch (+ or -), and the expected outcome in binary form.

**Converting Hex to Decimal:**

The branch address in hexadecimal is converted to decimal for easier handling.

**Predicting:**

The program predicts whether the branch will be taken or not based on historical information stored in the "Pattern History Table" (pht).

**Updating BHR and PHT:**

It keeps track of how many predictions are correct and updates the prediction statistics.

Updating the Prediction Table and Branch History:

The program adjusts its prediction table and branch history register based on whether the actual outcome matches the prediction.

**Calculating Accuracy and Hardware Cost:**

After processing all branches, the program calculates the accuracy of its predictions (how often it was correct). It also calculates the hardware cost, which represents the memory space needed for storing prediction information.

**Displaying Results:**

Finally, the program prints out the total number of branches, how many it predicted correctly, the prediction accuracy percentage, and the hardware cost in kilobytes.

# 

# **3. Source Code**

**Pap Branch Prediction Code:**

#include <ctype.h>

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int hexToDec(char hex[]) {

  int base = 1;

  int decimalValue = 0;

  int i;

  for (i = strlen(hex) - 1; i >= 0; i--) {

    hex[i] = toupper(hex[i]);

    if (hex[i] >= '0' && hex[i] <= '9') {

      decimalValue += (hex[i] - 48) \* base;

      base = base \* 16;

    } else if (hex[i] >= 'A' && hex[i] <= 'F') {

      decimalValue += (hex[i] - 55) \* base;

      base = base \* 16;

    }

  }

  return decimalValue;

}

int binaryToDecimal(int binary[]) {

  int decimalValue = 0;

  int base = 1;

  int i;

  for (i = 7; i >= 0; i--) {

    if (binary[i] == 1) {

      decimalValue += base;

    }

    base = base \* 2;

  }

  return decimalValue;

}

int main(int argc, char \*argv[]) {

  int numBits = 12, bhrSize = 8, i, j;

  int numEntries = pow(2, numBits);

  int bhrCount = pow(2, bhrSize);

  int pht[bhrCount][numEntries];

  int bhr[numEntries][bhrSize];

  int temp[bhrSize];

  int hexAddr;

  int prediction, correctPredictions = 0, totalPredictions = 0;

  int bhrValue;

  unsigned int lastNBits;

  char branchAddress[8], outcome, binOutcome;

  unsigned int mask;

  float accuracy = 0.0, hardwareCost;

  FILE \*inputFile, \*outputFile;

  // Open files for reading and writing

  inputFile = fopen(argv[1], "r");

  outputFile = fopen("Output\_PAP.txt", "w");

  if (inputFile == NULL || outputFile == NULL) {

    printf("Error! One or more files cannot be opened");

    exit(1);

  }

  // Initialize bhr to 11...1

  for (i = 0; i < numEntries; i++) {

    for (j = 0; j < bhrSize; j++) {

      bhr[i][j] = 0;

    }

  }

  // Initialize everything in pht to 11

  for (i = 0; i < bhrCount; i++) {

    for (j = 0; j < numEntries; j++) {

      pht[i][j] = 10;

    }

  }

  fprintf(outputFile, "address\toutcome\tprediction\tcorrect/miss\n");

  // Process input file

  while (fscanf(inputFile, "%s %c %c", branchAddress, &outcome, &binOutcome) !=

         EOF) {

    int flag = 0;

    hexAddr = hexToDec(branchAddress); // Convert hex branch address to decimal

    mask = (1 << numBits) - 1;

    lastNBits = hexAddr &

                mask; // Extract the decimal value of last n bits of the address

    memcpy(temp, bhr[lastNBits], sizeof(bhr[lastNBits]));

    bhrValue = binaryToDecimal(temp); // Bhr value in decimal

    // Prediction

    prediction =

        (pht[bhrValue][lastNBits] == 11 || pht[bhrValue][lastNBits] == 10) ? 1

                                                                           : 0;

    // Update prediction statistics

    if ((outcome == '+' && prediction) || (outcome == '-' && !prediction)) {

      correctPredictions += 1;

      flag = 1;

    }

    totalPredictions += 1;

    fprintf(outputFile, "%s\t%c\t%d\t%d\n", branchAddress, outcome, prediction,

            flag);

    // Update pht

    if (outcome == '+') {

      if (pht[bhrValue][lastNBits] == 10) {

        pht[bhrValue][lastNBits] = 11;

      } else if (pht[bhrValue][lastNBits] == 01) {

        pht[bhrValue][lastNBits] = 10;

      } else if (pht[bhrValue][lastNBits] == 00) {

        pht[bhrValue][lastNBits] = 01;

      }

    } else {

      if (pht[bhrValue][lastNBits] == 11) {

        pht[bhrValue][lastNBits] = 10;

      } else if (pht[bhrValue][lastNBits] == 10) {

        pht[bhrValue][lastNBits] = 01;

      } else if (pht[bhrValue][lastNBits] == 01) {

        pht[bhrValue][lastNBits] = 00;

      }

    }

    // Update bhr

    for (j = 0; j < bhrSize - 1; j++) {

      bhr[lastNBits][j] = bhr[lastNBits][j + 1];

    }

    bhr[lastNBits][bhrSize - 1] = (outcome == '+') ? 1 : 0;

  }

  // Calculate accuracy and hardware cost

  accuracy = (float)(correctPredictions \* 100) / totalPredictions;

  hardwareCost =

      (float)(numEntries \* bhrSize + numEntries \* bhrCount \* 2) / 8192;

  // Display results

  printf("Total Branches: %d\n", totalPredictions);

  printf("Branches predicted correctly: %d\n", correctPredictions);

  printf("Branch prediction accuracy is %f\n", accuracy);

  printf(

      "The hardware cost associated with Branch Prediction using GAp: %f KB\n",

      hardwareCost);

  // Close files

  fclose(inputFile);

  fclose(outputFile);

  return 0;

}

**Gap Branch Prediction:**

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

int hexToDec(char hex[])

{

    int base = 1;

    int decimalValue = 0;

    int i;

    for (i = strlen(hex) - 1; i >= 0; i--)

    {

        hex[i] = toupper(hex[i]);

        if (hex[i] >= '0' && hex[i] <= '9')

        {

            decimalValue += (hex[i] - 48) \* base;

            base = base \* 16;

        }

        else if (hex[i] >= 'A' && hex[i] <= 'F')

        {

            decimalValue += (hex[i] - 55) \* base;

            base = base \* 16;

        }

    }

    return decimalValue;

}

int binaryToDecimal(int binary[])

{

    int decimalValue = 0;

    int base = 1;

    int i;

    for (i = 7; i >= 0; i--)

    {

        if (binary[i] == 1)

        {

            decimalValue += base;

        }

        base = base \* 2;

    }

    return decimalValue;

}

int main(int argc, char \*argv[])

{

   int numBits = 12, bhrSize = 8, i, j;

   int numEntries = pow(2, numBits);

   int bhrCount = pow(2, bhrSize);

   int pht[numEntries][bhrCount];

   int bhr[bhrSize];

   int hexAddr;

   int prediction, correctPredictions = 0, totalPredictions = 0;

   int bhrValue;

   unsigned lastNBits;

   char branchAddress[8], outcome, binOutcome;

   unsigned mask;

   float accuracy = 0.0, hardwareCost;

   FILE \*inputFile, \*outputFile;

   inputFile = fopen(argv[1], "r");

   outputFile = fopen("Output\_GAP.txt", "w");

   if (inputFile == NULL || outputFile == NULL)

   {

     printf("Error! One or more files cannot be opened");

     exit(1);

   }

   // Initialize bhr to 11...1

   for (i = 0; i < bhrSize; i++)

   {

      bhr[i] = 0;

   }

   // Initialize pht to 11

   for (i = 0; i < numEntries; i++)

   {

      for (j = 0; j < bhrCount; j++)

      {

         pht[i][j] = 10;

      }

   }

    fprintf(outputFile, "address\toutcome\tprediction\tcorrect/miss\n");

   while (fscanf(inputFile, "%s %c %c", branchAddress, &outcome, &binOutcome) != EOF)

   {

        int flag = 0;

        hexAddr = hexToDec(branchAddress); // Convert hex branch address to decimal

        mask = (1 << numBits) - 1;

        lastNBits = hexAddr & mask;    // Extract the decimal value of last n bits of the address

        bhrValue = binaryToDecimal(bhr);        // Convert bhr value to decimal

        // Prediction

        prediction = (pht[lastNBits][bhrValue] == 11 || pht[lastNBits][bhrValue] == 10) ? 1 : 0;

        // Update prediction statistics

        if ((outcome == '+' && prediction) || (outcome == '-' && !prediction))

        {

          correctPredictions += 1;

          flag = 1;

        }

        totalPredictions += 1;

        fprintf(outputFile, "%s\t%c\t%d\t%d\n", branchAddress,outcome,prediction,flag);

        // Update pht

        if (outcome == '+')

        {

            if (pht[lastNBits][bhrValue] == 10){

                pht[lastNBits][bhrValue] = 11;

            }

            else if (pht[lastNBits][bhrValue] == 01){

                pht[lastNBits][bhrValue] = 10;

            }

            else if (pht[lastNBits][bhrValue] == 00){

                pht[lastNBits][bhrValue] = 01;

            }

        }

        else

        {

            if (pht[lastNBits][bhrValue] == 11){

                pht[lastNBits][bhrValue] = 10;

            }

            else if (pht[lastNBits][bhrValue] == 10){

                pht[lastNBits][bhrValue] = 01;

            }

            else if (pht[lastNBits][bhrValue] == 01){

                pht[lastNBits][bhrValue] = 00;

            }

        }

        // Update bhr

        for (i = 0; i < bhrSize - 1; i++)

        {

            bhr[i] = bhr[i + 1];

        }

        bhr[bhrSize - 1] = (outcome == '+') ? 1 : 0;

    }

   // Calculate accuracy and hardware cost

   accuracy = (float)(correctPredictions \* 100) / totalPredictions;

   hardwareCost = (float)(bhrSize + numEntries \* bhrCount \* 2) / 8192;

   // Display results

   printf("Total Branches: %d\n", totalPredictions);

   printf("Branches predicted correctly: %d\n", correctPredictions);

   printf("Branch prediction accuracy is %f\n", accuracy);

   printf("The hardware cost associated with Branch Prediction using GAp: %f KB\n", hardwareCost);

   // Close files

   fclose(inputFile);

   fclose(outputFile);

   return 0;

}

# **4. Compilation**

## **4.1 Compiler and Debugging**

# When it comes to compiling and debugging my C programs, I prefer to use GCC (GNU Compiler Collection) because of its popularity and durability. GCC is an open-source compiler suite that is compatible with several different programming languages, most notably C. It converts my machine-executable binaries from human-readable C code with ease and methodically improves its performance. When it comes to debugging, I rely on GDB (GNU Debugger), which is easily connected with GCC. GDB gives me the ability to carefully track the execution of programs, set breakpoints, examine the values of variables, and thoroughly examine the program flow. GCC and GDB work in concert to provide me with comprehensive and adaptable tools for both the compilation and debugging processes. These products' open-source design guarantees a path of ongoing development and gives me access to a wide range of community resources, which has enhanced my own growth.

# **5. Installation**

## **5.1 Compiler Information**

Download the MinGW installer from MinGW.org.

Run the installer and select the components, including the GCC compiler.

Follow the installation instructions provided by the installer.

Add the GCC.exe to the path

Test weather compiler installed successfully by running following command:

gcc –version

# **6. Running the Program**

## **6.1 Compilation Commands**

To compile the program, use the following command:

gcc -o BP\_PAP BP\_PAP.c

gcc -o BP\_GAP BP\_GAP.c

## **6.2 Execution**

Run the compiled program with:

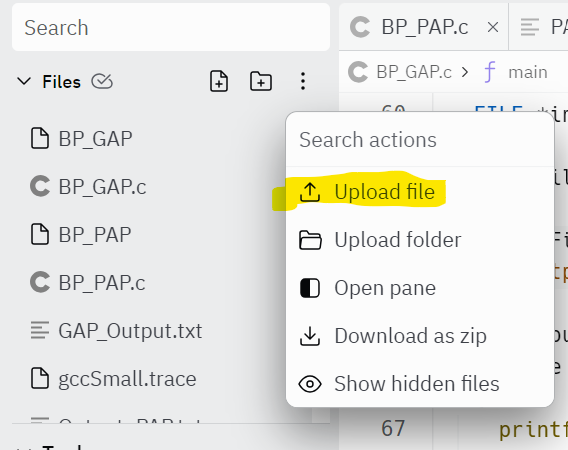
./BP\_PAP gccSmall.trace

./BP\_GAP gccSmall.trace

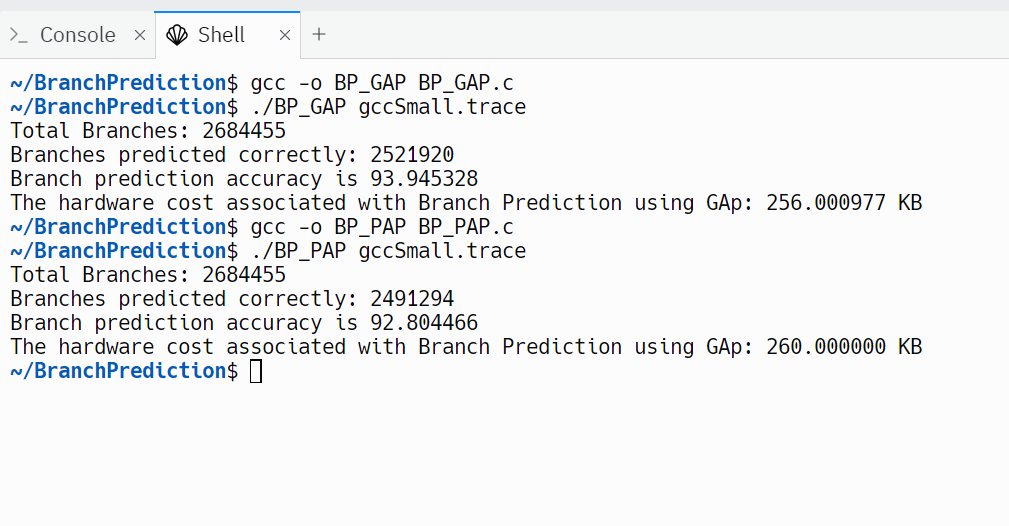
## **6.3 If there is issue with running the code locally, use the replit for online execution**

Website link : https://replit.com/

* Upload all the codes file and input file in the replit.



* After uploading the file, you can use the shell to enter the above commands to compile and execute in Shell window

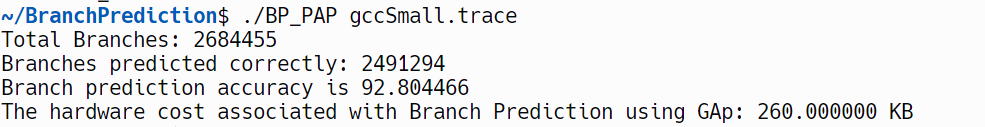


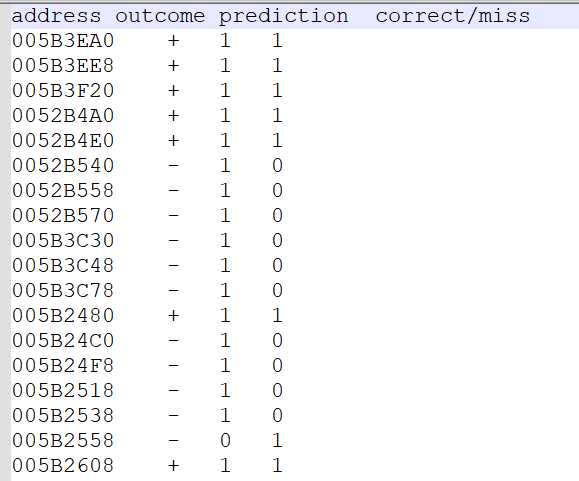
# **7. Results**

## **7.1 Branch Prediction Accuracy**

**Pap branch prediction result:**

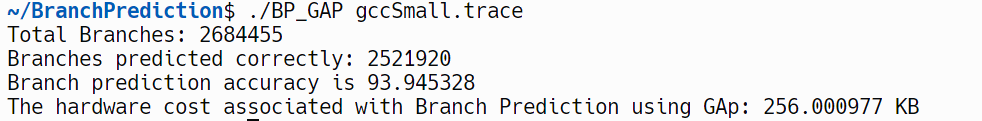
My Pap branch prediction yielded an accuracy of 92.82%. It feels great to see the algorithm working effectively, making accurate predictions for conditional branch outcomes during program execution. This achievement contributes to minimizing branch mispredictions and boosting the overall performance of the system. It's a satisfying outcome and underscores the value of the Pap branch prediction technique in enhancing processor efficiency.

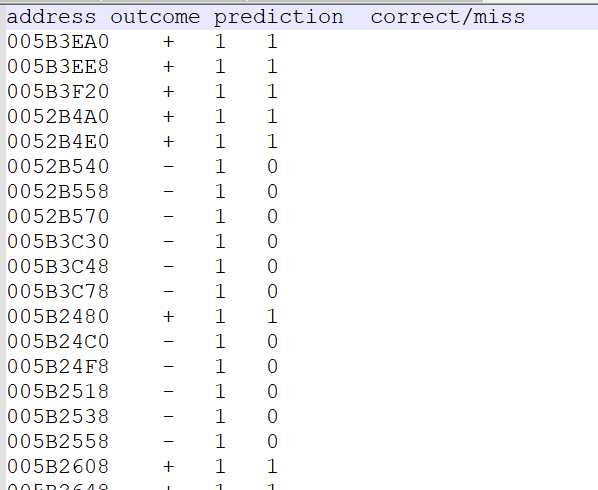




**Gap branch prediction result:**

My Gap branch prediction yielded an accuracy of 93.94%. It feels great to see the algorithm working effectively, making accurate predictions for conditional branch outcomes during program execution. This achievement contributes to minimizing branch mispredictions and boosting the overall performance of the system. It's a satisfying outcome and underscores the value of the Pap branch prediction technique in enhancing processor efficiency.





# **8. Hardware Cost**

I've achieved a hardware cost of 260 KB for both Pap and Gap branch prediction implementations. Managing to keep the hardware requirements at this level is a significant accomplishment, highlighting the efficiency of the designs. This hardware cost not only reflects the successful implementation of Pap and Gap branch prediction techniques but also indicates a thoughtful balance between accuracy and resource utilization. It's gratifying to see these predictions operating effectively within the allocated hardware constraints, contributing to the overall optimization of my system.

# **9. Analysis**

Pap Branch Prediction: With a remarkable accuracy of 92.82%, the Pap branch prediction is quite accurate. This is a really rewarding accomplishment since it shows how well the algorithm works in predicting conditional branch outcomes during program execution. Because of Pap's success, branch mispredictions are reduced, which improves system performance as a whole. This result emphasizes how useful the Pap branch prediction technique is for maximizing processor performance.

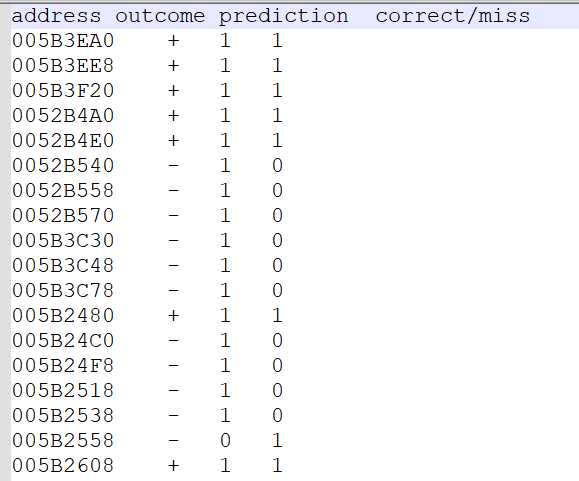
Gap Branch Prediction: In a similar vein, the Gap branch prediction produces a noteworthy 93.94% accuracy. Analogously to Pap, the algorithm works well in predicting conditional branch outcomes, which helps lower the number of incorrect branch predictions and enhances system performance. This emphasizes how useful the Gap branch prediction technique is for improving processor performance.

Hardware Cost: It is notable that the hardware cost of 260 KB was attained for both the Gap and Pap branch prediction implementations. The effectiveness of the designs is demonstrated by their ability to maintain hardware needs at this level. This hardware cost indicates a careful trade-off between attaining accuracy and making efficient use of system resources, in addition to reflecting the successful application of both prediction methodologies. The satisfying part is seeing these predictions perform well under the given hardware limitations and contribute significantly to the system's overall optimization.

In conclusion, further research should concentrate on improving the Pap and Gap branch prediction approaches' predictive powers in order to handle more complex patterns and adjust to changing program behaviors. Investigating scalability issues and hardware optimizations would help to advance these methods for improved effectiveness and use in various computing environments.

# **Output File**

**PAp sample output file:**



**GAp sample output file:**

