SIMULATION OF DYNAMIC BRANCH PREDICTION SCHEME

Simulation of 2bit and 3bit dynamic branch prediction schemes

By
Swapnil Acharya

Table of Contents

OBJECTIVE

The object of this project is to implement an algorithm and simulate the use of dynamic branch prediction schemes. Correct functioning of both the base algorithm and the dynamic branch prediction schemes are to be demonstrated.

PROJECT REQUIREMENTS

1. Implement one of the following types of algorithms:
    a. Search
    b. Solution of a set of linear equations
    c. a fast sort method (Note: selection sort and insertion sort methods are not acceptable).

    The main requirements for your selected algorithm are that it result in a minimum of 50 lines of code and there are at least 3 distinct conditional branch statements within it, one of these branch statements should be dependent on data associated with the problem.

    This assignment is to be done in C/C++. You are to substitute if-statements and goto-statements for higher level conditional constructs such as: for, while, do-while and switch. This is so that the use of conditional branches will be obvious in your solution.

2. Create a version of your code in which you simulate a 2-bit dynamic branch prediction scheme. For each branch keep track of the following statistics:
    a. number of taken branches
    b. number of not-taken branches
    c. number of predictions
    d. number of miss-predictions

    In addition, you may need additional statistics for each branch in order to implement the prediction scheme.

3. Create another version of your code in which you simulate a (3,1) predictor scheme. Keep track of the same four required statistics in addition to any additional statistics for each branch in order to implement the prediction scheme.

4. Execute the base algorithm along with each prediction scheme. Use a problem size that is large enough so that the number of predictions per branch are greater than 10; for inner loop branches this number should be on the order of 100. You may include the data for the base algorithm directly in the data section of the program instead of inputting it interactively or via a file.

5. In your report, compare the two prediction schemes and show which was more accurate.

Turn in a summary report that includes the following materials:

- a written description that describes your base algorithm and predicts what the output of your base algorithm should be when it is applied to your supplied data.
- screen shots that demonstrate the base algorithm works.
- screen shots that demonstrate the dynamic branch prediction schemes work.
- describe your branch prediction schemes.
- for each branch in the program, identify the branch and display the corresponding statistics (see item 2 above). Comment on the results.
- a listing (source code) of your programs.
- the names of the team members are to be included on the first page of the report.

DESCRIPTION OF BASE ALGORITHM: MERGE SORT

The base algorithm used in this project is Merge Sort. Merge Sort is based on dive and conquer approach. Merge sort partitions an input array into two halves, then recursively calls itself for the two partitions, and then merge the two sort partitions. The merge() function is used for sorting and merging the portioned halves at each step. Since this project requires simulation of branch prediction scheme. For loops and while loops were implemented via goto and if statement. The merge sort algorithm used here has 8 branches. Among 8 branches, there are two branches that depend on data associated with the problem (input array to be sorted).

INPUT DATA FOR THE MERGE SORT ALGORITHM

A program (gen_numbers.cpp) that generates a text file that consists of randomly generated numbers was created. The contents of generated txt are as follows:

42 95 77 21 93 29 17 70 38 21 54 63 93 26 18 90 77 5 29 8 5 66 8 26 25 15 76 23 50 70 68 91 65 43 11 56 69 27 24 6 45 77 68 39 3 85 27 79 87 55 84 92 19 91 16 43 7 91 63 54 59 29 44 22 71 54 78 39 78 2 44 24 77 12 60 79 97 86 57 84 40 41 74 58 31 90 99 37 79 60 88 39 89 32 58 58 85 36 96 64 36 40 85 13 52 45 89 47 29 46 29 68 85 2 25 16 91 24 50 70 83 38 7 70 70 63 29 53 96 25 15 32 65 99 42 15 42 30 59 71 75 85 37 58 86 61 73 77 82 21 46 66 59 50 34 27 13 62 79 10 85 94 39 50 91 79 62 34 9 21 5 81 4 39 40 90 1 11 67 82 32 11 46 88 59 80 13 71 40 89 78 25 81 16 72 73 94 34 5 3 52 9 83 56 45 21 46 45 32 11 26 61 20 69 49 78 47 59 47 87 48 24 10 30 39 82 1 33 14 5 33 66 11 14 20 55 35 63 98 66 71 24 27 90 90 73 66 38 32 14 23 77 37 32 7 75 14 5 6 26 9 36 91 19 49 9 72 83 71 70 49 40 91 73 29 81 44 94 17 73 8 39 51 42 69 55 15 80 60 18 6 66 53 94 83 3 3 55 83 71

DEMONSTRATION OF BASE MERGE SORT ALGORITHM

The implementation of base merge sort algorithm is in MergeSort.cpp. Figure 1 sows the output of this base merge sort implementation.

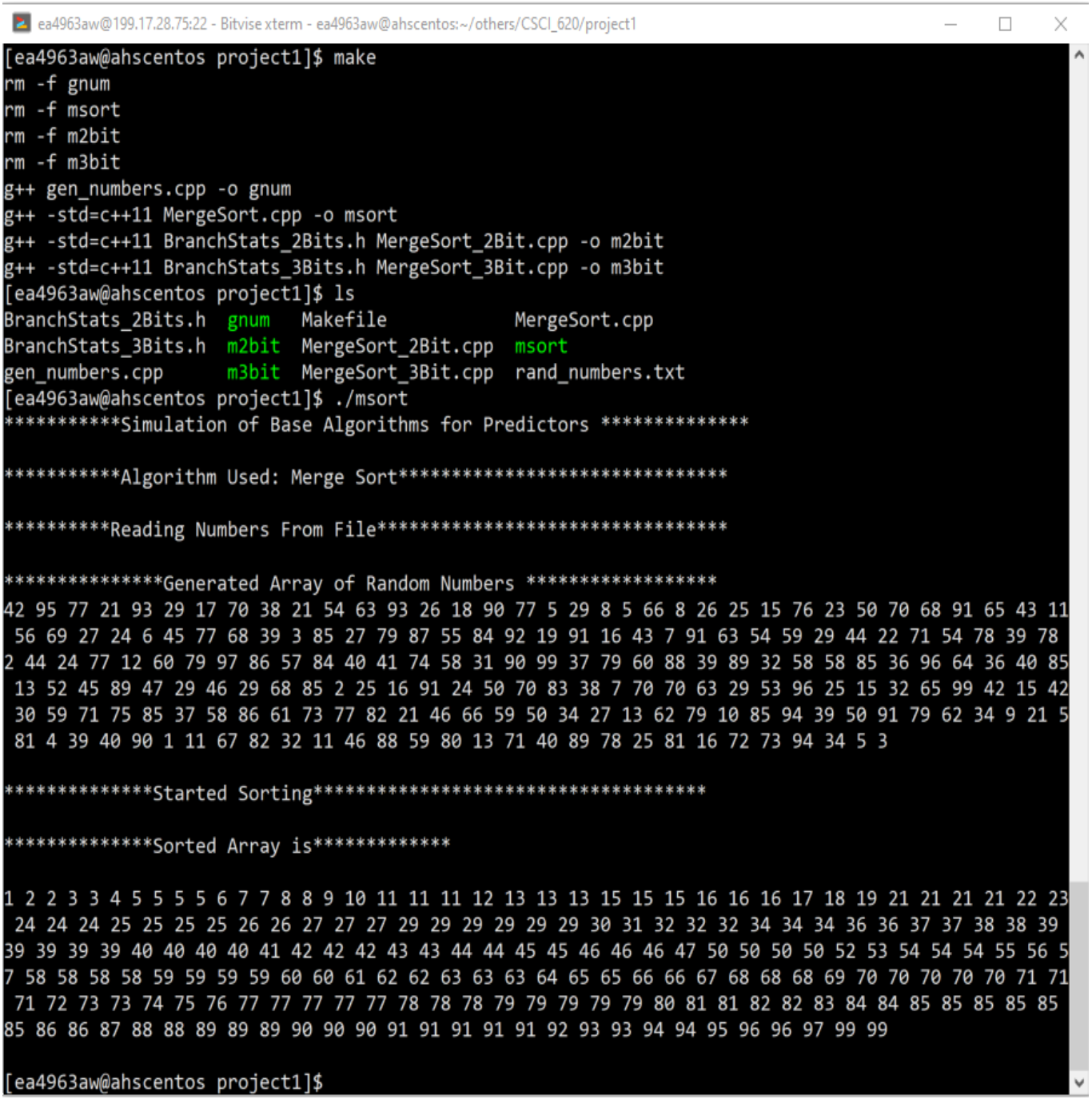


Figure 1: Output of Base Merge Sort Algorithm

DESCRIPTION OF 2-BIT PREDICTOR SCHEME

The state diagram in Figure 2, shows two-bit prediction scheme. This state diagram shows that in STATE_00 the prediction by the 2-bit predictor is NOT TAKEN. If the program action was NOT TAKEN then the predictor stays in same state else if the program action was TAKEN then the predictor moves to STATE_01. The predictions and transitions for all states are done in similar manner as discussed here and shown by Figure 2.

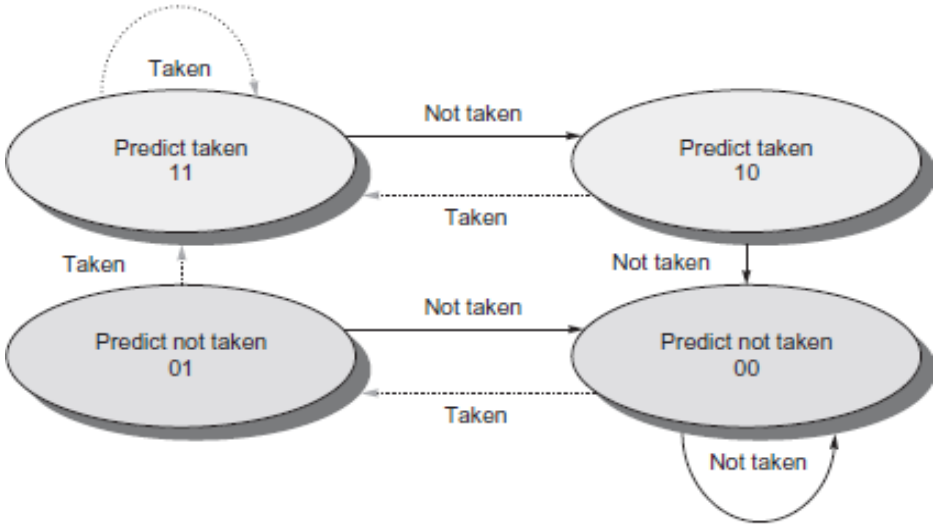


Figure 2: The 2 Bit prediction scheme sate diagram [1]

The algorithm for 2-bit predictor scheme keeps track of some parameters for all 8 branches in the merge sort algorithm. The default state for all 8 branches are initialized to STATE_00. Figure 3 shows the parameters and a default constructor that initializes all the predictors to STATE_00. An array of objects (length = number of branches) for class in Figure 3 is instantiated as global variable in the merge sort algorithm and individual array components are place as needed in the vicinity of the branches in the merge sort algorithm.

```
#ifndef _BRANCH_STATS_2_BITS_H_
#define _BRANCH_STATS_2_BITS_H_


#include <stdlib.h>
#include <stdio.h>

//representations for program actions and predictions
const int NOT_TAKEN = 0;
const int TAKEN = 1;

//representations for states
const int STATE_00 = 1000;
const int STATE_01 = 2000;
const int STATE_10 = 3000;
const int STATE_11 = 4000;

class BranchStats_2Bits{

    private:
        //variable to hold number of taken branches
        int number_of_taken_branches;

        //variable to hold number of not taken branches
        int number_of_not_taken_branches;

        //variable to hold branch predictor state for this particular branch
        int branch_predictor_state;

        //variabel to hold number of correct predictions
        int number_of_correct_predictions;

        //variabel to hold number of incorrect predictions
        int number_of_miss_predictions;

    public:

        //Default constructor, initializes 2-nit predictor to STATE_00
        BranchStats_2Bits(){
            this->number_of_taken_branches = 0;

            this->number_of_not_taken_branches = 0;

            this->branch_predictor_state = STATE_00;

            this->number_of_correct_predictions = 0;

            this->number_of_miss_predictions = 0;
        }
```

Figure 3: The parameters and default constructor for 2-bit prediction scheme.

```
for_loop_left:

    if( i > nl-1){ //BRANCH 1

        //update taken branch stats
        branch_stats[1].increase_num_taken_branches();
        branch_stats[1].update_predictions(TAKEN);

        goto done_for_loop_left;
    }

    //update not taken branch stats
    branch_stats[1].increase_num_not_taken_branches();
    branch_stats[1].update_predictions(NOT_TAKEN);


    larr[i] = array[l + i];

    i++;

    goto for_loop_left;

done_for_loop_left:
```

Figure 4: Code Snippet of how and where the Branch statistics are computed.

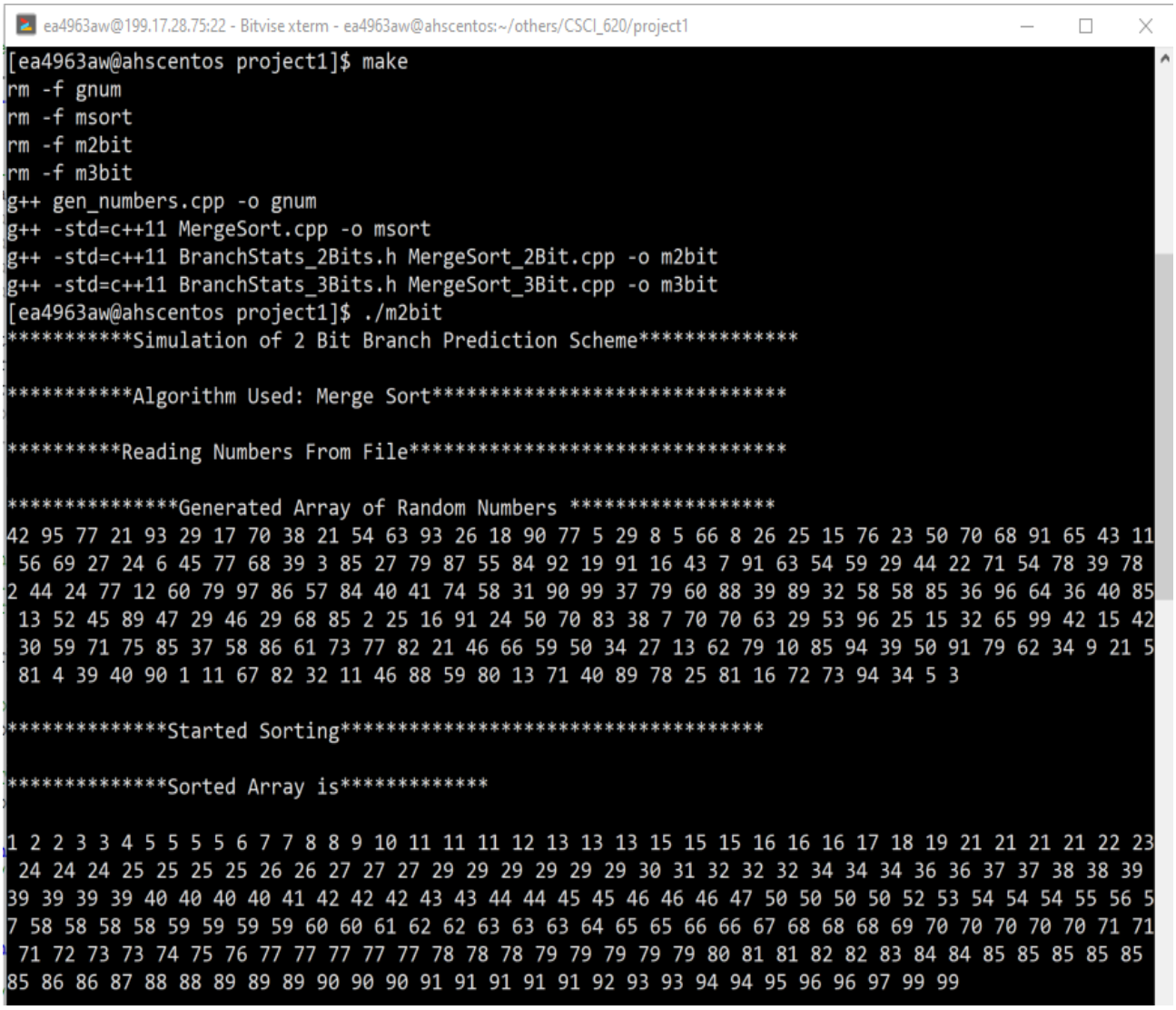
Consider code snippet in Figure 4, when the if statement becomes true then branch is considered to be TAKEN. When a branch is TAKEN the number of taken branches is increased. Then to compute predictions and state updates, the program action, TAKEN in this case is passed to the function in Figure 5.

```
void update_predictions(int program_action){

    //update predictions if it was correct or not correct
    if(this->branch_predictor_state == STATE_11){

        //STATE 11 Predicts TAKEN
        if(program_action == TAKEN){ //if program action was taken

            this->number_of_correct_predictions++;
            //state stays the same
        }
        else{ //if program action was not taken
            this->number_of_miss_predictions++;

            //update to STATE_10
            this->branch_predictor_state = STATE_10;
        }
    }
```

Figure 5, computation of state updates and predictions.

Initially the default state for all branches are set to STATE_00. For this case consider, the current state to be STATE_11. As depicted by the state diagram in Figure 2, For STATE_11, the prediction by the 2-bit predictor is TAKEN. If the program action was TAKEN and the prediction by the predictor was also TAKEN, then the number of correct predictions is increased, and the state transition is performed.

OUTPUT OF 2-BIT PREDICTION SCHEME



```
*********Branch Statistics**********

Branch: 0
Number of Taken Branches: 200
Number of Not Taken Branches: 199
Number of Correct Branch Predictions: 137
Number of Miss Branch Predictions: 262
Correct Branch Prediction : 34.34 %
Miss Branch Prediction : 65.66 %

Branch: 1
Number of Taken Branches: 199
Number of Not Taken Branches: 812
Number of Correct Branch Predictions: 812
Number of Miss Branch Predictions: 199
Correct Branch Prediction : 80.32 %
Miss Branch Prediction : 19.68 %

Branch: 2
Number of Taken Branches: 199
Number of Not Taken Branches: 732
Number of Correct Branch Predictions: 732
Number of Miss Branch Predictions: 199
Correct Branch Prediction : 78.63 %
Miss Branch Prediction : 21.37 %

Branch: 3
Number of Taken Branches: 199
Number of Not Taken Branches: 1281
Number of Correct Branch Predictions: 1281
Number of Miss Branch Predictions: 199
Correct Branch Prediction : 86.55 %
Miss Branch Prediction : 13.45 %
```

```
Branch: 4
Number of Taken Branches: 650
Number of Not Taken Branches: 631
Number of Correct Branch Predictions: 596
Number of Miss Branch Predictions: 685
Correct Branch Prediction : 46.53 %
Miss Branch Prediction : 53.47 %

Branch: 5
Number of Taken Branches: 631
Number of Not Taken Branches: 0
Number of Correct Branch Predictions: 629
Number of Miss Branch Predictions: 2
Correct Branch Prediction : 99.68 %
Miss Branch Prediction : 0.32 %

Branch: 6
Number of Taken Branches: 199
Number of Not Taken Branches: 162
Number of Correct Branch Predictions: 164
Number of Miss Branch Predictions: 197
Correct Branch Prediction : 45.43 %
Miss Branch Prediction : 54.57 %

Branch: 7
Number of Taken Branches: 199
Number of Not Taken Branches: 101
Number of Correct Branch Predictions: 184
Number of Miss Branch Predictions: 116
Correct Branch Prediction : 61.33 %
Miss Branch Prediction : 38.67 %
```

DESCRIPTION OF 3-BIT PREDICTOR SCHEME

The algorithm for 3-bit predictor scheme keeps track of some parameters for all 8 branches in the merge sort algorithm like the 2-bit predictor. The default state for all 8 branches are initialized to STATE_00. Figure 6 shows the parameters that are sued to keep track of branch statistics.

Unlike the 2-bit predictor scheme, 3-bit predictor store history of branches. If the last 3 breaches were TAKEN then the history bits are set to TAKEN, TAKEN, TAKEN. A combination table of history bits is made, of ra 3bit predictor there are total of 8 combinations. These combinates are represented by integer values as shown in Figure 6. The combination table is represented by a has table where the key is the representation of a combination and value is the prediction of that combination. The global history is maintained using a vector, the entries in this vector will be the representation of combinations. The default constructor initializes all the combinations in combination table as NOT TAKEN. In addition to that, the default constructor also adds "COMB_000" in the global history, i.e. history bits being NOT TAKEN, NOT TAKEN, NOT TAKEN.

Consider code snippet in Figure 4, when the if statement becomes true then branch is considered to be TAKEN. When a branch is TAKEN the number of taken branches is increased. Then to compute predictions and state updates, the program action, TAKEN in this case is passed to the function in Figure 7.

Initially all the entries in combinations are initialized to NOT TAKEN. For this case consider, the last entry in global history to be COMB_000, i.e. history bits being NOT TAKEN, NOT TAKEN, NOT TAKEN. The prediction entry for this history is fetched from combination table. If the prediction entry was TAKEN and the program action was NOT TAKEN then the number of incorrect predictions are update, the entry in the combination table for COMB_000 is update from TAKEN to NOT TAKEN and a new combination COMB_001 i.e. NOT TAKEN, NOT TAKEN, TAKEN is added to global history table.

In general, a combination table consisting of possible combination for 3 history bits is created. This is then initialized to either TAKEN or NOT TAKEN. Then an entry is added to global history table depending upon how the combination table was initialized. When the merge sort algorithm runs, depending upon the last entry in global history table and program action, the prediction count updates are done and entries in combination table are updated and an a new entry is made on global history table. The entries on global history table are simply left shifted, where the history bit 3's entry becomes history bit 2's entry, history bit 2's entry becomes history bit 1's entry and history bit 3's entry becomes the program action.

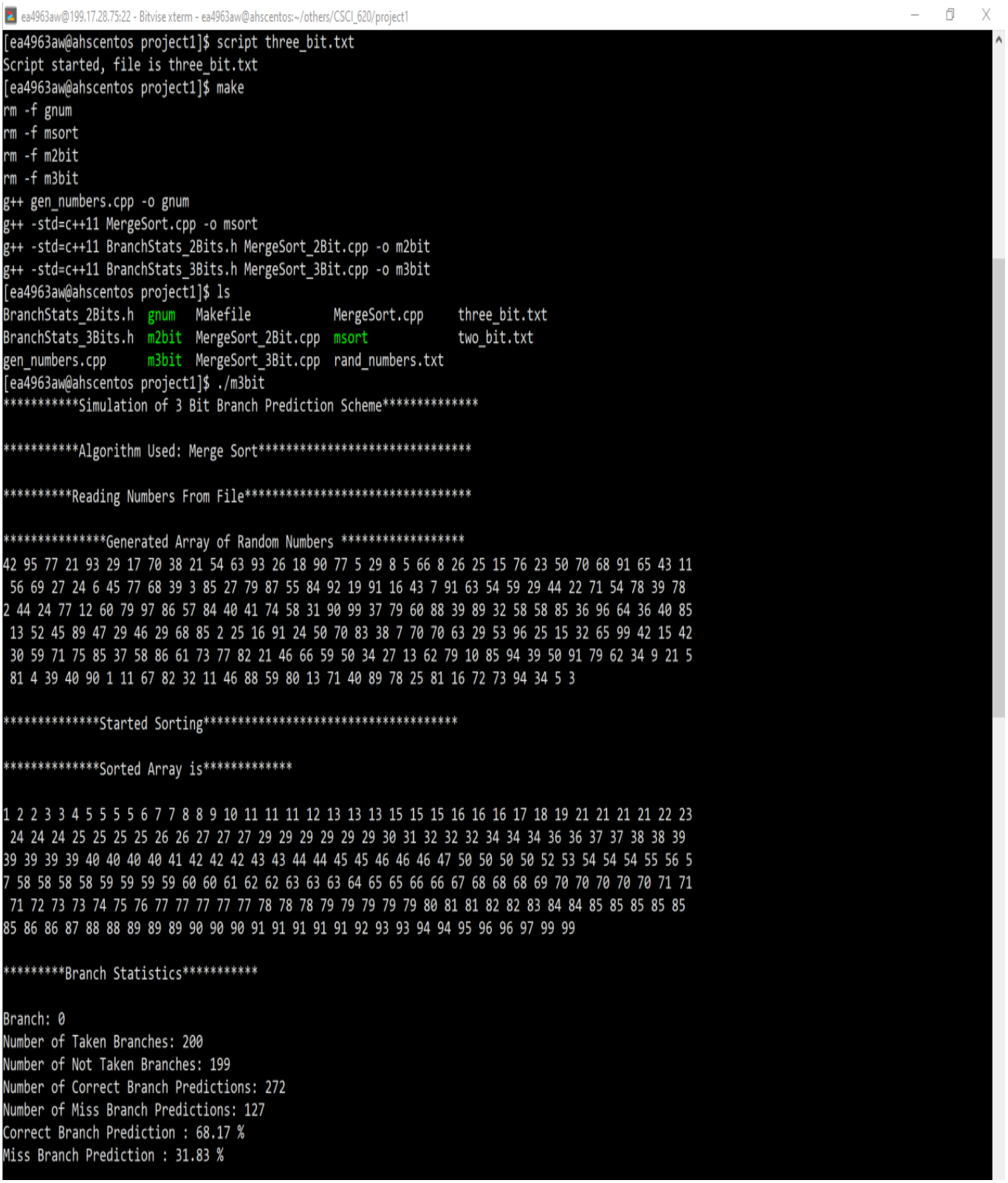
```
//representations for program actions and predictions
const int NOT_TAKEN = 0;
const int TAKEN = 1;

//representations for possible hisotry bits
const int COMB_000 = 1000; //N,N,N
const int COMB_001 = 2000; //N,N,T
const int COMB_010 = 3000; //N,T,N
const int COMB_011 = 4000; //N,T,T
const int COMB_100 = 5000; //T,N,N
const int COMB_101 = 6000; //T,N,T
const int COMB_110 = 7000; //T,T,N
const int COMB_111 = 8000; //T,T,T


class BranchStats_3Bits{

    private:
        //variable to hold number of taken branches
        int number_of_taken_branches;

        //variable to hold number of not taken branches
        int number_of_not_taken_branches;

        //variable to hold number of correct predictions
        int number_of_correct_predictions;

        //variable to hold number of incorrect predictions
        int number_of_miss_predictions;

        //a hash table to hold combination tabel of hisotry bits and current predict value
        //with combination being the key and prediction value being the value for the hash table
        std::unordered_map<int,int> combination_table;

        //a vector to hold the global history bits
        std::vector<int> global_history_table;


    public:

        //default constructor initializes all combination table for 3 bt
        //predictor to be NOT TAKEN
        BranchStats_3Bits(){

            this->number_of_taken_branches = 0;

            this->number_of_not_taken_branches = 0;

            this->number_of_correct_predictions = 0;

            this->number_of_miss_predictions = 0;

            //initialize combination tabel for 3 bit predictor
            for(int i=1000; i < 9000; i+= 1000){
                this->combination_table[i] = NOT_TAKEN;
            }

            //initialize global history table with T,T,T
            this->global_history_table.push_back(COMB_111);

        }
```

Figure 6: The parameters and default constructor for 3-bit prediction scheme.

```
133     void update_predictions(int program_action){
134
135         //get last item from history table
136         int last_history = this->global_history_table.back();
137
138         //get combination
139         if(last_history == COMB_000){
140
141             //go to combination table to get prediction
142             int prediction = this->combination_table[COMB_000];;
143
144             //if prediction and program action were taken
145             if((prediction == TAKEN) && (program_action == TAKEN)){
146
147                 //update number of correct predictions
148                 this->number_of_correct_predictions++;
149
150                 //update history table
151                 this->global_history_table.push_back(COMB_001);
152             }
153             //if prediction and program action were not taken
154             else if( (prediction == NOT_TAKEN) && (program_action == NOT_TAKEN)){
155
156                 //update number of correct predictions
157                 this->number_of_correct_predictions++;
158
159                 //update history table
160                 this->global_history_table.push_back(COMB_000);
161             }
162             else if( (prediction == TAKEN ) && (program_action == NOT_TAKEN) ){
163
164                 //update number of correct predictions
165                 this->number_of_miss_predictions++;
166
167                 //update combination_table
168                 this->combination_table[COMB_000] = NOT_TAKEN;
169
170                 //update history table
171                 this->global_history_table.push_back(COMB_000);
172             }
173             else if( ( prediction == NOT_TAKEN) && ( program_action == TAKEN)){
174
175                 //update number of correct predictions
176                 this->number_of_miss_predictions++;
177
178                 //update combination_table
179                 this->combination_table[COMB_000]= TAKEN;
180
181                 //update history table
182                 this->global_history_table.push_back(COMB_001);
183             }
184
185         }
```

Figure 7, updates of predictions count, combination table and global history table

OUTPUT OF 3-BIT PREDICTION SCHEME



ea4963aw@199.17.28.75:22 - Bitvise xterm - ea4963aw@ahscentos:~/others/CSCI_620/project1

```
Branch: 1
Number of Taken Branches: 199
Number of Not Taken Branches: 812
Number of Correct Branch Predictions: 742
Number of Miss Branch Predictions: 269
Correct Branch Prediction : 73.39 %
Miss Branch Prediction : 26.61 %

Branch: 2
Number of Taken Branches: 199
Number of Not Taken Branches: 732
Number of Correct Branch Predictions: 772
Number of Miss Branch Predictions: 159
Correct Branch Prediction : 82.92 %
Miss Branch Prediction : 17.08 %

Branch: 3
Number of Taken Branches: 199
Number of Not Taken Branches: 1281
Number of Correct Branch Predictions: 1193
Number of Miss Branch Predictions: 287
Correct Branch Prediction : 80.61 %
Miss Branch Prediction : 19.39 %

Branch: 4
Number of Taken Branches: 650
Number of Not Taken Branches: 631
Number of Correct Branch Predictions: 670
Number of Miss Branch Predictions: 611
Correct Branch Prediction : 52.30 %
Miss Branch Prediction : 47.70 %

Branch: 5
Number of Taken Branches: 631
Number of Not Taken Branches: 0
Number of Correct Branch Predictions: 629
Number of Miss Branch Predictions: 2
Correct Branch Prediction : 99.68 %
Miss Branch Prediction : 0.32 %

Branch: 6
Number of Taken Branches: 199
Number of Not Taken Branches: 162
Number of Correct Branch Predictions: 189
Number of Miss Branch Predictions: 172
Correct Branch Prediction : 52.35 %
Miss Branch Prediction : 47.65 %

Branch: 7
Number of Taken Branches: 199
Number of Not Taken Branches: 101
Number of Correct Branch Predictions: 177
Number of Miss Branch Predictions: 123
Correct Branch Prediction : 59.00 %
Miss Branch Prediction : 41.00 %

[ea4963aw@ahscentos project1]$ exit
exit
Script done, file is three_bit.txt
[ea4963aw@ahscentos project1]$
```

RESULTS

Branch	Statistics	2-Bit Predictor	3-Bit Predictor
0	Number of Taken Branches	200	200
	Number of Not Taken Branches	199	199
	Number of Correct Branch Predictions	137	272
	Number of Miss Branch Predictions	262	127
	Correct Branch Prediction	34.34%	68.17%
	Miss Branch Prediction	54.66%	31.83%
1	Number of Taken Branches	199	199
	Number of Not Taken Branches	812	812
	Number of Correct Branch Predictions	812	742
	Number of Miss Branch Predictions	199	269
	Correct Branch Prediction	80.32%	73.39%
	Miss Branch Prediction	19.68%	26.61%
2	Number of Taken Branches	199	199
	Number of Not Taken Branches	732	732
	Number of Correct Branch Predictions	732	772
	Number of Miss Branch Predictions	199	159
	Correct Branch Prediction	78.63%	82.92%
	Miss Branch Prediction	21.37%	17.08%
3	**Number of Taken Branches**	**199**	**199**
	Number of Not Taken Branches	**1281**	**1281**
	Number of Correct Branch Predictions	**1281**	**1193**
	Number of Miss Branch Predictions	**199**	**287**
	Correct Branch Prediction	**86.55%**	**80.61%**
	Miss Branch Prediction	**13.45%**	**19.39%**
4 (Data dependent)	Number of Taken Branches	650	650
	Number of Not Taken Branches	631	631
	Number of Correct Branch Predictions	596	670
	Number of Miss Branch Predictions	685	611
	Correct Branch Prediction	46.53%	52.30%
	Miss Branch Prediction	53.47%	47.70%

5 (Data dependent)	Number of Taken Branches	631	631
	Number of Not Taken Branches	0	0
	Number of Correct Branch Predictions	629	629
	Number of Miss Branch Predictions	2	2
	Correct Branch Prediction	99.68%	99.68%
	Miss Branch Prediction	0.32%	0.32%
6	Number of Taken Branches	199	199
	Number of Not Taken Branches	162	162
	Number of Correct Branch Predictions	164	189
	Number of Miss Branch Predictions	197	172
	Correct Branch Prediction	45.43%	52.35%
	Miss Branch Prediction	54.57%	47.65%
7	Number of Taken Branches	199	199
	Number of Not Taken Branches	101	101
	Number of Correct Branch Predictions	184	177
	Number of Miss Branch Predictions	116	123
	Correct Branch Prediction	61.33%	59.00%
	Miss Branch Prediction	38.67%	41.00%

Table 1: Comparison of the performance of 2-bit and 3-bit predictor schemes.

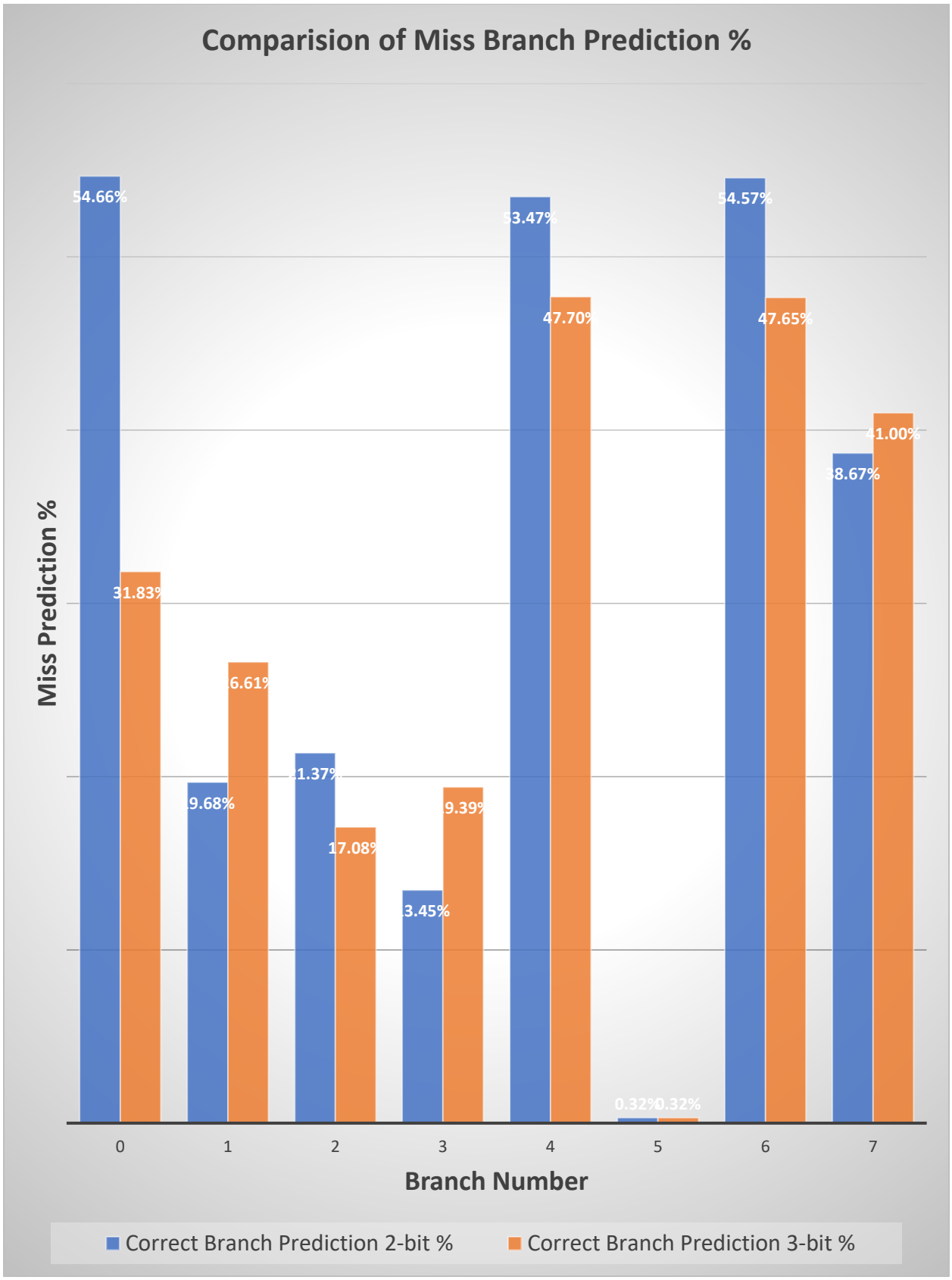


Figure 8: Comparison of Correct Branch Prediction Percentage of both 2-bit and 3-bit predictors.

DISCUSSIONS

The results in table 1 and figure 8 are for a problem size N = 200. Table 1 shows that, for the outer loop branches number of predictions per branches are higher 10 and for inner loop branches the number of predictions are in the order of hundred. Branch 0,1,2,3,6,7 are not data dependent branches where are branch 4 and 5 are data dependent. From figure 8 for both data dependent branches the 3-bit predictor perform better than 2-bit predictor since 3-bit prediction's miss prediction % is less when compared to that of 3-bit predictor. This is also true for data independent branches. The miss prediction % of 3-bit prediction is higher than 2-bit predictor in only 3 branches (branch 1,3 and 7) whereas the miss prediction % of 2-bit predictor is higher than 3-bit predictor in 4 branches. (branch 0,2,4,6). The branches where 2-bit predictor has higher miss prediction than 3-bit predictor is significantly higher. Therefore, based on these results the 3-bit predictor performs slightly better than 2-bit predictor.

SUMMARY

Merge sort algorithm was implemented in C++. The loops in the merge sort code were implemented using if statements and goto statements in order to simulate assembly code. 2-bit and 3-bit dynamic branch predictor schemes were implemented and integrated with the merge sort algorithm. The results indicated that 3-bit predictor scheme performed slightly better than 2-bit predictor scheme.

REFERENCES

[1] John L. Hennessy and David A. Patterson, *Computer Architecture A Quantitative Approach*, Elsevier Inc., Sixth Edition, 2019, ISBN: 978-0-12-811905-1.

RUNNING THE SOURCE FILES

1. In a Unix/Linux terminal navigate to where the source files are stored
2. Type make in terminal to trigger the makefile to run.
3. The makefile will have created 3 executables
4. To run base merge sort type ./msort in terminal
5. To run merge sort with 2-bit predictor simulation type ./m2bit in terminal
6. To run merge sort with 3-bit predictor simulation type ./m3bit in terminal

SOURCE CODE LISTINGS

PROGRAM USED TO GENERATE RANDOM NUMERS IN A FILE

```
1. /*
2. *Date: 11/22/2020
3. *File: gen_numbers.cpp
4. *Description: This program generates random numbers and writes them in
   a file.
5. *             The numbers of random numbers to be generated must be
   supplied as argument
6. *             to the executable
7. */
8.
9.
10.  #include <stdio.h>
11.  #include <stdlib.h>
12.  #include <time.h>
13.
14.  /*
15.  * This
16.  *@param min the minimum value for the random number generation
17.  *@param max the maximum value for the random bumber generation
18.  *@returns a dynamic integer array
19.  */
20.  int main(int argc, char * argv[]){
21.
22.      if(argc < 2){
23.
24.          printf("Specify The numbers of random numbers to be generated
   \n");
25.
26.          return 0;
27.      }
28.
29.      int N = atoi(argv[1]);
30.
31.      FILE *fp;
32.
33.      fp = fopen("rand_numbers.txt","w+");
34.
35.      //intialize random seed
36.      srand(time(NULL));
37.
38.      //generate random number between
39.      for(int i = 0; i < N; i++){
40.          int final_rand_num = (rand()% (100-1)) + 1;
41.          fprintf(fp,"%d ",final_rand_num);
42.      }
43.
44.      fclose(fp);
45.  }
```

MERGE SORT BASE ALGORITHM

```
1. /*
2. *Date: 11/21/2020
3. *File: MergeSort.cpp
4. */
5.
6.
7. // C++ program for Merge Sort
8. #include <stdlib.h>
9. #include <stdio.h>
10.
11.  using namespace std;
12.
13.  //Problem size
14.  const int N = 200;
15.
16.
17.  /*
18.  *This function displays the contents of a dynamic array.
19.  *@param *ar the dynamic integer array whose contents are to be
   displayed
20.  *@param ar_size the number of elements inside the array or the lengh
   of the array
21.  *@returns None
22.  */
23.  void display_array_contents(int * ar, int ar_size){
24.
25.      //initialize count
26.      int count = 0;
27.
28.      display_loop:
29.          if (count > ar_size - 1){
30.
31.              printf("\n");
32.
33.              return;
34.          }
35.
36.          printf("%d ", *(ar + count) );
37.
38.          count = count + 1;
39.
40.          goto display_loop;
41.
42.
43.  }
44.
45.
46.  // Merges two subarrays of arr[].
47.  // First subarray is arr[l..m]
48.  // Second subarray is arr[m+1..r]
49.  void merge(int *array, int l, int m, int r)
50.  {
51.      int i , j , k, nl, nr;
52.
```

```
53.     //size of left sub-arrays
54.     nl = m-l+1;
55.
56.     //size of right sub-arrays
57.     nr = r-m;
58.
59.     // Create temp arrays
60.     int larr[nl];
61.
62.     int rarr[nr];
63.
64.      //copy to left temp array
65.      i = 0;
66.
67.      for_loop_left:
68.
69.          if( i > nl-1){ //BRANCH 1
70.
71.              goto done_for_loop_left;
72.          }
73.
74.          larr[i] = array[l + i];
75.
76.          i++;
77.
78.          goto for_loop_left;
79.
80.      done_for_loop_left:
81.
82.      j = 0;
83.
84.      //copy to right temp array
85.      for_loop_right:
86.
87.          if( j > nr-1){ //BRANCH 2
88.
89.              goto done_for_loop_right;
90.          }
91.
92.          rarr[j] = array[m + 1 + j];
93.
94.          j++;
95.
96.          goto for_loop_right;
97.
98.      done_for_loop_right:
99.
100.     // Merge the temp arrays back into arr[l..r]
101.
102.     // Initial index of first subarray
103.     i = 0;
104.
105.     // Initial index of second subarray
106.     j = 0;
107.
108.     // Initial index of merged subarray
109.     k = l;
```

```
110.
111.
112.
113.     //merge arrays
114.     merge_array_while_loop:
115.
116.         //converting while to if, took 2 hours but nedded to change
   logic from && to ||
117.         if((i > nl-1) || (j > nr-1)){ //BRANCH 3
118.
119.             goto done_merge_array_while_loop;
120.         }
121.
122.         if(larr[i] <= rarr[j]){ //BRANCH 4, data dependent branching
123.
124.             array[k] = larr[i];
125.             i++;
126.             k++;
127.
128.             goto merge_array_while_loop;
129.         }
130.
131.         if(larr[i] > rarr[j]){ //BRANCH 5, data dependent branching
132.
133.             array[k] = rarr[j];
134.             j++;
135.             k++;
136.
137.             goto merge_array_while_loop;
138.         }
139.
140.     done_merge_array_while_loop:
141.
142.     // Copy the remaining elements of
143.     // L[], if there are any
144.     copy_remaining_left_while_loop:
145.
146.         if(i > nl-1){ //BRANCH 6
147.
148.             goto done_copy_remaining_left_while_loop;
149.         }
150.
151.         array[k] = larr[i];
152.         i++;
153.         k++;
154.
155.         goto copy_remaining_left_while_loop;
156.
157.
158.     done_copy_remaining_left_while_loop:
159.
160.     // Copy the remaining elements of
161.     // R[], if there are any
162.
163.     copy_remaining_right_while_loop:
164.
165.         if (j > nr-1) { //BRANCH 7
```

```
166.
167.             goto done_copy_remaining_right_while_loop;
168.         }
169.
170.         array[k] = rarr[j];
171.         j++;
172.         k++;
173.
174.         goto copy_remaining_right_while_loop;
175.
176.     done_copy_remaining_right_while_loop:
177.         return;
178. }
179.
180.
181. /*
182. * This function sort an input array based on merge sort Algorithm.
183. * Merge Sort is based on dive and conquer approach. Merge sort
184. * partitions an input array into two halves, then recursively calls
185. * itself for the two partitions, and then merge the two sort
   partitions.
186. * @param array the input integer array that is to be sorted
187. * @param l the left or the lower index of the array or the section of
   the
188. *  array to be sorted
189. * @param r the right or the upper index of the array or the section of
   the
190. *  array to be sorted
191. * @returns None the input array is sorted when the function is
   complete
192. */
193. void mergeSort(int *array, int l, int r)
194. {
195.     int m;
196.
197.
198.     if(l > r - 1){ //BRANCH 0
199.
200.         goto done_merge_sort;
201.     }
202.
203.     //get the middle index for array partition
204.     // Same as (l+r)/2, but avoids overflow for large l and h
205.     m = l + (r-l)/2;
206.
207.     //Recursively call the firt half of the array for sorting
208.     mergeSort(array, l, m);
209.
210.     //Recursively call the firt half of the array for sorting
211.     mergeSort(array, m + 1, r);
212.
213.     //Merge the sorted halves
214.     merge(array, l, m, r);
215.
216.     done_merge_sort:
217.         //just a dummy bariabel so that the label "done_merge_sort"
   works
```

```
218.         int done = 0;
219.
220. }
221.
222.
223.
224. // Driver code
225. int main()
226. {
227.     printf("***********Simulation of Base Algorithms for Predictors
   *************\n\n");
228.
229.     printf("***********Algorithm Used: Merge
   Sort*****************************\n\n");
230.
231.     printf("**********Reading Numbers From
   File*****************************\n\n");
232.     FILE *fp;
233.     fp = fopen("rand_numbers.txt","r");
234.     int *arr = new int[N];
235.     for(int i = 0; i < N ; i++){
236.         fscanf(fp,"%d",(arr+i));
237.     }
238.
239.
240.     printf("***************Generated Array of Random Numbers
   *****************\n");
241.     display_array_contents(arr, N);
242.     printf("\n");
243.
244.     printf("*************Started
   Sorting***********************************\n\n");
245.     mergeSort(arr, 0, N - 1);
246.
247.     printf("*************Sorted Array is*************\n\n");
248.     display_array_contents(arr, N);
249.     printf("\n");
250.
251.
252.     return 0;
253. }
```

2-BIT PREDICTOR IMPLEMENTATION

```
1. /*
2. * Date: 11/21/2020
3. * File: BranchStats_2Bits.h
4. * Description: The BranchStats_2Bits class in this file implements 2-
   bit prediction scheme
5. */
6.
7.
8. #ifndef _BRANCH_STATS_2_BITS_H_
9. #define _BRANCH_STATS_2_BITS_H_
10.
11.
12.  #include <stdlib.h>
13.  #include <stdio.h>
14.
15.  //representations for program actions and predictions
16.  const int NOT_TAKEN = 0;
17.  const int TAKEN = 1;
18.
19.  //representations for states
20.  const int STATE_00 = 1000;
21.  const int STATE_01 = 2000;
22.  const int STATE_10 = 3000;
23.  const int STATE_11 = 4000;
24.
25.
26.  class BranchStats_2Bits{
27.
28.      private:
29.          //variable to hold number of taken branches
30.          int number_of_taken_branches;
31.
32.          //variable to hold number of not taken branches
33.          int number_of_not_taken_branches;
34.
35.          //variable to hold branch predictor state for this particular
   branch
36.          int branch_predictor_state;
37.
38.          //variable to hold number of correct predictions
39.          int number_of_correct_predictions;
40.
41.          //variable to hold number of incorrect predictions
42.          int number_of_miss_predictions;
43.
44.      public:
45.
46.          //Default constructor, initializes 2-nit predictor to STATE_00
47.          BranchStats_2Bits(){
48.              this->number_of_taken_branches = 0;
49.
50.              this->number_of_not_taken_branches = 0;
51.
52.              this->branch_predictor_state = STATE_00;
```

```
53.
54.             this->number_of_correct_predictions = 0;
55.
56.             this->number_of_miss_predictions = 0;
57.         }
58.
59.         //Overloaded constructor, initializes 2-nit predictor to
   supplied state as argument
60.         BranchStats_2Bits(int state){
61.             this->number_of_taken_branches = 0;
62.
63.             this->number_of_not_taken_branches = 0;
64.
65.             this->branch_predictor_state = state;
66.
67.             this->number_of_correct_predictions = 0;
68.
69.             this->number_of_miss_predictions = 0;
70.         }
71.
72.         //method to increase number of taken branches
73.         void increase_num_taken_branches(){
74.             this->number_of_taken_branches++;
75.         }
76.
77.
78.         //method to fetch  number of taken branches
79.         int get_num_taken_branches() const{
80.             return this->number_of_taken_branches;
81.         }
82.
83.         //method to increase number of not taken branches
84.         void increase_num_not_taken_branches(){
85.             this->number_of_not_taken_branches++;
86.         }
87.
88.         //method to fetch  number of not taken branches
89.         int get_num_not_taken_branches() const{
90.             return this->number_of_not_taken_branches;
91.         }
92.
93.         //method to fetch  number of correct branch predictions
94.         int get_num_correct_predictions() const{
95.             return this->number_of_correct_predictions;
96.         }
97.
98.         //method to fetch  number of incorrect branch predictions
99.         int get_num_miss_predictions() const{
100.            return this->number_of_miss_predictions;
101.        }
102.
103.        //method to display statis for this branch
104.        void print_statistics(){
105.            printf("Number of Taken Branches: %d\n",this-
   >number_of_taken_branches);
106.            printf("Number of Not Taken Branches: %d\n",this-
   >number_of_not_taken_branches);
```

```
107.             printf("Number of Correct Branch Predictions: %d\n",this-
   >number_of_correct_predictions);
108.             printf("Number of Miss Branch Predictions: %d\n",this-
   >number_of_miss_predictions);
109.
110.             double total_predictions = this-
   >number_of_correct_predictions + this->number_of_miss_predictions;
111.             double cpr = this->number_of_correct_predictions /
   total_predictions;
112.             double mpr = this->number_of_miss_predictions /
   total_predictions;
113.             printf("Correct Branch Prediction : %0.2f %%\n", cpr*100);
114.             printf("Miss Branch Prediction : %0.2f %% \n", mpr*100);
115.         }
116.
117.         /*
118.         * This method depending upon current stae of the 2-bit
   predictor and
119.         * action taken by the program , update number of
   correct.incorrect predictions and
120.         * state transitions for the 2-bit preditor
121.         * @param program_action the action trak ny program i.e. if the
   branch was TAKEN or NOT TAKEN
122.         */
123.         void update_predictions(int program_action){
124.
125.             //update predictions if it was correct or not correct
126.             if(this->branch_predictor_state == STATE_11){
127.
128.                 //STATE 11 Predicts TAKEN
129.                 if(program_action == TAKEN){ //if program action was
   taken
130.
131.                     this->number_of_correct_predictions++;
132.                     //state stays the same
133.                 }
134.                 else{ //if program action was not taken
135.                     this->number_of_miss_predictions++;
136.
137.                     //update to STATE_10
138.                     this->branch_predictor_state = STATE_10;
139.                 }
140.             }
141.             else if(this->branch_predictor_state == STATE_10){
142.
143.                 //STATE 10 Predicts TAKEN
144.                 if(program_action == TAKEN){ //if program action was
   taken
145.                     this->number_of_correct_predictions++;
146.                     this->branch_predictor_state = STATE_11;
147.                 }
148.                 else{ //if program action was not taken
149.                     this->number_of_miss_predictions++;
150.
151.                     //update to STATE_00
152.                     this->branch_predictor_state = STATE_00;
153.
```

```
154.                 }
155.             }
156.             else if(this->branch_predictor_state == STATE_00){
157.
158.                 //STATE 00 Predicts NOT TAKEN
159.
160.                 if(program_action == TAKEN){ //if program action was
    taken
161.                     this->number_of_miss_predictions++;
162.                     this->branch_predictor_state = STATE_01;
163.                 }
164.                 else{ //if program action was not taken
165.                     this->number_of_correct_predictions++;
166.
167.                     //STATE DOES NOT CHANGE
168.                 }
169.
170.             }
171.             else{ //STATE_01
172.
173.                 //STATE 01 Predicts NOT TAKEN
174.
175.                 if(program_action == TAKEN){ //if program action was
    taken
176.
177.                     this->number_of_miss_predictions++;
178.
179.                     this->branch_predictor_state = STATE_11;
180.                 }
181.                 else{ //if program action was not taken
182.                     this->number_of_correct_predictions++;
183.
184.                     //update to STATE_00
185.                     this->branch_predictor_state = STATE_00;
186.                 }
187.             }
188.         }
189. };
190.
191.
192.
193.
194. #endif
```

MERGE SORT WITH 2-BIT PREDICTOR INCLUDED

```
1. /*
2. *Date: 11/21/2020
3. *File: MergeSort_2Bit.cpp
4. */
5.
6.
7. // C++ program for Merge Sort
8. #include <stdlib.h>
9. #include <stdio.h>
10.  #include <time.h>
11.  #include "BranchStats_2Bits.h"
12.
13.  using namespace std;
14.
15.  //Problem size
16.  const int N = 200;
17.
18.  //Branch Prediction Parameters
19.  const int TOTAL_BRANCHES = 8;
20.
21.
22.  //declare a global stats
23.  BranchStats_2Bits branch_stats[TOTAL_BRANCHES];
24.
25.
26.  /*
27.  *This function displays the contents of a dynamic array.
28.  *@param *ar the dynamic integer array whose contents are to be
   displayed
29.  *@param ar_size the number of elements inside the array or the lengh
   of the array
30.  *@returns None
31.  */
32.  void display_array_contents(int * ar, int ar_size){
33.
34.      //initialize count
35.      int count = 0;
36.
37.      display_loop:
38.          if (count > ar_size - 1){
39.
40.              printf("\n");
41.
42.              return;
43.          }
44.
45.          printf("%d ", *(ar + count) );
46.
47.          count = count + 1;
48.
49.          goto display_loop;
50.
51.
52.  }
```

```
53.
54.
55.  // Merges two subarrays of arr[].
56.  // First subarray is arr[l..m]
57.  // Second subarray is arr[m+1..r]
58.  void merge(int *array, int l, int m, int r)
59.  {
60.      int i , j , k, nl, nr;
61.
62.     //size of left sub-arrays
63.     nl = m-l+1;
64.
65.     //size of right sub-arrays
66.     nr = r-m;
67.
68.     // Create temp arrays
69.     int larr[nl];
70.
71.     int rarr[nr];
72.
73.      //copy to left temp array
74.      i = 0;
75.
76.      for_loop_left:
77.
78.          if( i > nl-1){ //BRANCH 1
79.
80.              //update taken branch stats
81.              branch_stats[1].increase_num_taken_branches();
82.              branch_stats[1].update_predictions(TAKEN);
83.
84.              goto done_for_loop_left;
85.          }
86.
87.          //update not taken branch stats
88.          branch_stats[1].increase_num_not_taken_branches();
89.          branch_stats[1].update_predictions(NOT_TAKEN);
90.
91.
92.          larr[i] = array[l + i];
93.
94.          i++;
95.
96.          goto for_loop_left;
97.
98.      done_for_loop_left:
99.
100.     j = 0;
101.
102.     //copy to right temp array
103.     for_loop_right:
104.
105.         if( j > nr-1){ //BRANCH 2
106.
107.             //update taken branch stats
108.             branch_stats[2].increase_num_taken_branches();
109.             branch_stats[2].update_predictions(TAKEN);
```

```
110.
111.             goto done_for_loop_right;
112.         }
113.
114.         //update not taken branch stats
115.         branch_stats[2].increase_num_not_taken_branches();
116.         branch_stats[2].update_predictions(NOT_TAKEN);
117.
118.
119.         rarr[j] = array[m + 1 + j];
120.
121.         j++;
122.
123.         goto for_loop_right;
124.
125.     done_for_loop_right:
126.
127.     // Merge the temp arrays back into arr[l..r]
128.
129.     // Initial index of first subarray
130.     i = 0;
131.
132.     // Initial index of second subarray
133.     j = 0;
134.
135.     // Initial index of merged subarray
136.     k = l;
137.
138.
139.
140.     //merge arrays
141.     merge_array_while_loop:
142.
143.         //converting while to if, took 2 hours but nedded to change
    logic from && to ||
144.         if((i > nl-1) || (j > nr-1)){ //BRANCH 3
145.
146.             //update taken branch stats
147.             branch_stats[3].increase_num_taken_branches();
148.             branch_stats[3].update_predictions(TAKEN);
149.
150.             goto done_merge_array_while_loop;
151.         }
152.
153.         //update not taken branch stats
154.         branch_stats[3].increase_num_not_taken_branches();
155.         branch_stats[3].update_predictions(NOT_TAKEN);
156.
157.         if(larr[i] <= rarr[j]){ //BRANCH 4, data dependent branching
158.
159.             array[k] = larr[i];
160.             i++;
161.             k++;
162.
163.             //update taken branch stats
164.             branch_stats[4].increase_num_taken_branches();
165.             branch_stats[4].update_predictions(TAKEN);
```

```
166.
167.             goto merge_array_while_loop;
168.         }
169.
170.         //update not taken branch stats
171.         branch_stats[4].increase_num_not_taken_branches();
172.         branch_stats[4].update_predictions(NOT_TAKEN);
173.
174.         if(larr[i] > rarr[j]){ //BRANCH 5, data dependent branching
175.
176.             array[k] = rarr[j];
177.             j++;
178.             k++;
179.
180.             //update taken branch stats
181.             branch_stats[5].increase_num_taken_branches();
182.             branch_stats[5].update_predictions(TAKEN);
183.
184.             goto merge_array_while_loop;
185.         }
186.
187.         //update not taken branch stats
188.         branch_stats[5].increase_num_not_taken_branches();
189.         branch_stats[5].update_predictions(NOT_TAKEN);
190.
191.     done_merge_array_while_loop:
192.
193.     // Copy the remaining elements of
194.     // L[], if there are any
195.     copy_remaining_left_while_loop:
196.
197.         if(i > nl-1){ //BRANCH 6
198.
199.             //update taken branch stats
200.             branch_stats[6].increase_num_taken_branches();
201.             branch_stats[6].update_predictions(TAKEN);
202.
203.             goto done_copy_remaining_left_while_loop;
204.         }
205.
206.         //update not taken branch stats
207.         branch_stats[6].increase_num_not_taken_branches();
208.         branch_stats[6].update_predictions(NOT_TAKEN);
209.
210.         array[k] = larr[i];
211.         i++;
212.         k++;
213.
214.         goto copy_remaining_left_while_loop;
215.
216.
217.     done_copy_remaining_left_while_loop:
218.
219.     // Copy the remaining elements of
220.     // R[], if there are any
221.
222.     copy_remaining_right_while_loop:
```

```
223.
224.         if (j > nr-1) { //BRANCH 7
225.
226.             //update taken branch stats
227.             branch_stats[7].increase_num_taken_branches();
228.             branch_stats[7].update_predictions(TAKEN);
229.
230.             goto done_copy_remaining_right_while_loop;
231.         }
232.
233.         //update not taken branch stats
234.         branch_stats[7].increase_num_not_taken_branches();
235.         branch_stats[7].update_predictions(NOT_TAKEN);
236.
237.         array[k] = rarr[j];
238.         j++;
239.         k++;
240.
241.         goto copy_remaining_right_while_loop;
242.
243.     done_copy_remaining_right_while_loop:
244.         return;
245. }
246.
247.
248. // l is for left index and r is
249. // right index of the sub-array
250. // of arr to be sorted
251. void mergeSort(int *array, int l, int r)
252. {
253.     int m;
254.
255.     if(l > r - 1){ //BRANCH 0
256.
257.         //update taken branch stats
258.         branch_stats[0].increase_num_taken_branches();
259.         branch_stats[0].update_predictions(TAKEN);
260.
261.         goto done_merge_sort;
262.     }
263.
264.     //update not taken branch stats
265.     branch_stats[0].increase_num_not_taken_branches();
266.     branch_stats[0].update_predictions(NOT_TAKEN);
267.
268.     // Same as (l+r)/2, but avoids
269.     // overflow for large l and h
270.     //int m = (l + r - l) / 2;
271.     m = l + (r-l)/2;
272.
273.     // Sort first and second halves
274.     mergeSort(array, l, m);
275.
276.     mergeSort(array, m + 1, r);
277.
278.     merge(array, l, m, r);
279.
```

```
    done_merge_sort:
        int done = 0;

}



// Driver code
int main()
{
    printf("***********Simulation of 2 Bit Branch Prediction
   Scheme*************\n\n");

    printf("***********Algorithm Used: Merge
   Sort*****************************\n\n");

    printf("**********Reading Numbers From
   File*********************************\n\n");
    FILE *fp;
    fp = fopen("rand_numbers.txt","r");
    int *arr = new int[N];
    for(int i = 0; i < N ; i++){
        fscanf(fp,"%d",(arr+i));
    }


    printf("***************Generated Array of Random Numbers
   ******************\n");
    display_array_contents(arr, N);
    printf("\n");

    printf("**************Started
   Sorting************************************\n\n");
    mergeSort(arr, 0, N - 1);

    printf("**************Sorted Array is*************\n\n");
    display_array_contents(arr, N);
    printf("\n");

    printf("*********Branch Statistics***********\n\n");
    for(int i = 0; i < TOTAL_BRANCHES; i++){
        printf("Branch: %d\n",i);
        branch_stats[i].print_statistics();
        printf("\n");
    }


    return 0;
}
```

3-BIT PREDICTOR IMPLEMENTATION

```
1. /*
2. * Date: 11/21/2020
3. * File: BranchStats_2Bits.h
4. * Description: The BranchStats_3Bits class in this file implements 3-
   bit prediction scheme
5. */
6.
7.
8. #ifndef _BRANCH_STATS_3_BITS_H_
9. #define _BRANCH_STATS_3_BITS_H_
10.
11.
12.  #include <stdlib.h>
13.  #include <stdio.h>
14.  #include <unordered_map>
15.  #include <vector>
16.
17.  //representations for program actions and predictions
18.  const int NOT_TAKEN = 0;
19.  const int TAKEN = 1;
20.
21.  //representations for possible hisotry bits
22.  const int COMB_000 = 1000; //N,N,N
23.  const int COMB_001 = 2000; //N,N,T
24.  const int COMB_010 = 3000; //N,T,N
25.  const int COMB_011 = 4000; //N,T,T
26.  const int COMB_100 = 5000; //T,N,N
27.  const int COMB_101 = 6000; //T,N,T
28.  const int COMB_110 = 7000; //T,T,N
29.  const int COMB_111 = 8000; //T,T,T
30.
31.
32.  class BranchStats_3Bits{
33.
34.      private:
35.          //variable to hold number of taken branches
36.          int number_of_taken_branches;
37.
38.          //variable to hold number of not taken branches
39.          int number_of_not_taken_branches;
40.
41.          //variable to hold number of correct predictions
42.          int number_of_correct_predictions;
43.
44.          //variable to hold number of incorrect predictions
45.          int number_of_miss_predictions;
46.
47.          //a hash table to hold combination tabel of hisotry bits and
   current predict value
48.          //with combination being the key and prediction value being
   the value for the hash table
49.          std::unordered_map<int,int> combination_table;
50.
51.          //a vector to hold the global history bits
```

```
52.          std::vector<int> global_history_table;
53.
54.
55.      public:
56.
57.          //default constructor initializes all combination table for 3
   bt
58.          //predictor to be NOT TAKEN
59.          BranchStats_3Bits(){
60.
61.              this->number_of_taken_branches = 0;
62.
63.              this->number_of_not_taken_branches = 0;
64.
65.              this->number_of_correct_predictions = 0;
66.
67.              this->number_of_miss_predictions = 0;
68.
69.              //initialize combination tabel for 3 bit predictor
70.              for(int i=1000; i < 9000; i+= 1000){
71.                  this->combination_table[i] = NOT_TAKEN;
72.              }
73.
74.              //initialize global history table with T,T,T
75.              this->global_history_table.push_back(COMB_111);
76.
77.          }
78.
79.
80.          //method to increase number of taken branches
81.          void increase_num_taken_branches(){
82.              this->number_of_taken_branches++;
83.          }
84.
85.
86.          //method to fetch  number of taken branches
87.          int get_num_taken_branches() const{
88.              return this->number_of_taken_branches;
89.          }
90.
91.          //method to increase number of not taken branches
92.          void increase_num_not_taken_branches(){
93.              this->number_of_not_taken_branches++;
94.          }
95.
96.          //method to fetch  number of not taken branches
97.          int get_num_not_taken_branches() const{
98.              return this->number_of_not_taken_branches;
99.          }
100.
101.         //method to fetch  number of correct branch predictions
102.         int get_num_correct_predictions() const{
103.             return this->number_of_correct_predictions;
104.         }
105.
106.         //method to fetch  number of incorrect branch predictions
107.         int get_num_miss_predictions() const{
```

```
108.             return this->number_of_miss_predictions;
109.         }
110.
111.         //method to display statis for this branch
112.         void print_statistics(){
113.             printf("Number of Taken Branches: %d\n",this-
   >number_of_taken_branches);
114.             printf("Number of Not Taken Branches: %d\n",this-
   >number_of_not_taken_branches);
115.             printf("Number of Correct Branch Predictions: %d\n",this-
   >number_of_correct_predictions);
116.             printf("Number of Miss Branch Predictions: %d\n",this-
   >number_of_miss_predictions);
117.
118.             double total_predictions = this-
   >number_of_correct_predictions + this->number_of_miss_predictions;
119.             double cpr = this->number_of_correct_predictions /
   total_predictions;
120.             double mpr = this->number_of_miss_predictions /
   total_predictions;
121.             printf("Correct Branch Prediction : %0.2f %%\n", cpr*100);
122.             printf("Miss Branch Prediction : %0.2f %% \n", mpr*100);
123.         }
124.
125.         /*
126.         *This method gets the last entry from global history tabel and
   goes to the combination table,
127.         * and fetches the prediction for that entry. Then the
   prediction and the action taken by parameters
128.         * are use to update correct/incorrect predictions, update the
   entry on combination table and add
129.         * a new entry in global history.
130.         */
131.
132.         void update_predictions(int program_action){
133.
134.             //get last item from history table
135.             int last_history = this->global_history_table.back();
136.
137.             //get combination
138.             if(last_history == COMB_000){
139.
140.                 //go to combination table to get prediction
141.                 int prediction = this->combination_table[COMB_000];;
142.
143.                 //if prediction and program action were taken
144.                 if((prediction == TAKEN) && (program_action ==
   TAKEN)){
145.
146.                     //update number of correct predictions
147.                     this->number_of_correct_predictions++;
148.
149.                     //update history table
150.                     this->global_history_table.push_back(COMB_001);
151.                 }
152.                 //if prediction and program action were not taken
```

```
153.                 else if( (prediction == NOT_TAKEN) && (program_action
   == NOT_TAKEN)){
154.
155.                     //update number of correct predictions
156.                     this->number_of_correct_predictions++;
157.
158.                     //update history table
159.                     this->global_history_table.push_back(COMB_000);
160.                 }
161.                 else if( (prediction == TAKEN ) && (program_action ==
   NOT_TAKEN) ){
162.
163.                     //update number of correct predictions
164.                     this->number_of_miss_predictions++;
165.
166.                     //update combination_table
167.                     this->combination_table[COMB_000] = NOT_TAKEN;
168.
169.                     //update history table
170.                     this->global_history_table.push_back(COMB_000);
171.                 }
172.                 else if( ( prediction == NOT_TAKEN) &&
   ( program_action == TAKEN)){
173.
174.                     //update number of correct predictions
175.                     this->number_of_miss_predictions++;
176.
177.                     //update combination_table
178.                     this->combination_table[COMB_000]= TAKEN;
179.
180.                     //update history table
181.                     this->global_history_table.push_back(COMB_001);
182.                 }
183.
184.             }
185.
186.
187.
188.             //get combination
189.             else if(last_history == COMB_001){
190.
191.                 //go to combination table to get prediction
192.                 int prediction = this->combination_table[COMB_001];
193.
194.                 //if prediction and program action were taken
195.                 if((prediction == TAKEN) && (program_action ==
   TAKEN)){
196.
197.                     //update number of correct predictions
198.                     this->number_of_correct_predictions++;
199.
200.                     //update history table
201.                     this->global_history_table.push_back(COMB_011);
202.                 }
203.                 //if prediction and program action were not taken
204.                 else if( (prediction == NOT_TAKEN) && (program_action
   == NOT_TAKEN)){
```

```
205.
206.                     //update number of correct predictions
207.                     this->number_of_correct_predictions++;
208.
209.                     //update history table
210.                     this->global_history_table.push_back(COMB_010);
211.                 }
212.                 else if( (prediction == TAKEN ) && (program_action ==
    NOT_TAKEN) ){
213.
214.                     //update number of correct predictions
215.                     this->number_of_miss_predictions++;
216.
217.                     //update combination_table
218.                     this->combination_table[COMB_001] = NOT_TAKEN;
219.
220.                     //update history table
221.                     this->global_history_table.push_back(COMB_010);
222.                 }
223.                 else if( ( prediction == NOT_TAKEN) &&
    ( program_action == TAKEN)){
224.
225.                     //update number of correct predictions
226.                     this->number_of_miss_predictions++;
227.
228.                     //update combination_table
229.                     this->combination_table[COMB_001] = TAKEN;
230.
231.                     //update history table
232.                     this->global_history_table.push_back(COMB_011);
233.                 }
234.
235.             }
236.
237.
238.
239.             //get combination
240.             else if(last_history == COMB_010){
241.
242.                 //go to combination table to get prediction
243.                 int prediction = this->combination_table[COMB_010];
244.
245.                 //if prediction and program action were taken
246.                 if((prediction == TAKEN) && (program_action ==
    TAKEN)){
247.
248.                     //update number of correct predictions
249.                     this->number_of_correct_predictions++;
250.
251.                     //update history table
252.                     this->global_history_table.push_back(COMB_101);
253.                 }
254.                 //if prediction and program action were not taken
255.                 else if( (prediction == NOT_TAKEN) && (program_action
    == NOT_TAKEN)){
256.
257.                     //update number of correct predictions
```

```
258.                     this->number_of_correct_predictions++;
259.
260.                     //update history table
261.                     this->global_history_table.push_back(COMB_100);
262.                 }
263.                 else if( (prediction == TAKEN ) && (program_action ==
   NOT_TAKEN) ){
264.
265.                     //update number of correct predictions
266.                     this->number_of_miss_predictions++;
267.
268.                     //update combination_table
269.                     this->combination_table[COMB_010] = NOT_TAKEN;
270.
271.                     //update history table
272.                     this->global_history_table.push_back(COMB_100);
273.                 }
274.                 else if( ( prediction == NOT_TAKEN) &&
   ( program_action == TAKEN)){
275.
276.                     //update number of correct predictions
277.                     this->number_of_miss_predictions++;
278.
279.                     //update combination_table
280.                     this->combination_table[COMB_010] = TAKEN;
281.
282.                     //update history table
283.                     this->global_history_table.push_back(COMB_101);
284.                 }
285.
286.             }
287.
288.
289.
290.             //get combination
291.             else if(last_history == COMB_011){
292.
293.                 //go to combination table to get prediction
294.                 int prediction = this->combination_table[COMB_011];
295.
296.                 //if prediction and program action were taken
297.                 if((prediction == TAKEN) && (program_action ==
   TAKEN)){
298.
299.                     //update number of correct predictions
300.                     this->number_of_correct_predictions++;
301.
302.                     //update history table
303.                     this->global_history_table.push_back(COMB_111);
304.                 }
305.                 //if prediction and program action were not taken
306.                 else if( (prediction == NOT_TAKEN) && (program_action
   == NOT_TAKEN)){
307.
308.                     //update number of correct predictions
309.                     this->number_of_correct_predictions++;
310.
```

```
311.                     //update history table
312.                     this->global_history_table.push_back(COMB_110);
313.                 }
314.                 else if( (prediction == TAKEN ) && (program_action ==
    NOT_TAKEN) ){
315.
316.                     //update number of correct predictions
317.                     this->number_of_miss_predictions++;
318.
319.                     //update combination_table
320.                     this->combination_table[COMB_011] = NOT_TAKEN;
321.
322.                     //update history table
323.                     this->global_history_table.push_back(COMB_110);
324.                 }
325.                 else if( ( prediction == NOT_TAKEN) &&
    ( program_action == TAKEN)){
326.
327.                     //update number of correct predictions
328.                     this->number_of_miss_predictions++;
329.
330.                     //update combination_table
331.                     this->combination_table[COMB_011] = TAKEN;
332.
333.                     //update history table
334.                     this->global_history_table.push_back(COMB_111);
335.                 }
336.
337.             }
338.
339.
340.
341.             //get combination
342.             else if(last_history == COMB_100){
343.
344.                 //go to combination table to get prediction
345.                 int prediction = this->combination_table[COMB_100];
346.
347.                 //if prediction and program action were taken
348.                 if((prediction == TAKEN) && (program_action ==
    TAKEN)){
349.
350.                     //update number of correct predictions
351.                     this->number_of_correct_predictions++;
352.
353.                     //update history table
354.                     this->global_history_table.push_back(COMB_001);
355.                 }
356.                 //if prediction and program action were not taken
357.                 else if( (prediction == NOT_TAKEN) && (program_action
    == NOT_TAKEN)){
358.
359.                     //update number of correct predictions
360.                     this->number_of_correct_predictions++;
361.
362.                     //update history table
363.                     this->global_history_table.push_back(COMB_000);
```

```
364.                 }
365.                 else if( (prediction == TAKEN ) && (program_action ==
   NOT_TAKEN) ){
366.
367.                     //update number of correct predictions
368.                     this->number_of_miss_predictions++;
369.
370.                     //update combination_table
371.                     this->combination_table[COMB_100] = NOT_TAKEN;
372.
373.                     //update history table
374.                     this->global_history_table.push_back(COMB_000);
375.                 }
376.                 else if( ( prediction == NOT_TAKEN) &&
   ( program_action == TAKEN)){
377.
378.                     //update number of correct predictions
379.                     this->number_of_miss_predictions++;
380.
381.                     //update combination_table
382.                     this->combination_table[COMB_100] = TAKEN;
383.
384.                     //update history table
385.                     this->global_history_table.push_back(COMB_001);
386.                 }
387.
388.             }
389.
390.
391.
392.             //get combination
393.             else if(last_history == COMB_101){
394.
395.                 //go to combination table to get prediction
396.                 int prediction = this->combination_table[COMB_101];
397.
398.                 //if prediction and program action were taken
399.                 if((prediction == TAKEN) && (program_action ==
   TAKEN)){
400.
401.                     //update number of correct predictions
402.                     this->number_of_correct_predictions++;
403.
404.                     //update history table
405.                     this->global_history_table.push_back(COMB_011);
406.                 }
407.                 //if prediction and program action were not taken
408.                 else if( (prediction == NOT_TAKEN) && (program_action
   == NOT_TAKEN)){
409.
410.                     //update number of correct predictions
411.                     this->number_of_correct_predictions++;
412.
413.                     //update history table
414.                     this->global_history_table.push_back(COMB_010);
415.                 }
```

```
416.                 else if( (prediction == TAKEN ) && (program_action ==
   NOT_TAKEN) ){
417.
418.                     //update number of correct predictions
419.                     this->number_of_miss_predictions++;
420.
421.                     //update combination_table
422.                     this->combination_table[COMB_101] = NOT_TAKEN;
423.
424.                     //update history table
425.                     this->global_history_table.push_back(COMB_010);
426.                 }
427.                 else if( ( prediction == NOT_TAKEN) &&
   ( program_action == TAKEN)){
428.
429.                     //update number of correct predictions
430.                     this->number_of_miss_predictions++;
431.
432.                     //update combination_table
433.                     this->combination_table[COMB_101] = TAKEN;
434.
435.                     //update history table
436.                     this->global_history_table.push_back(COMB_011);
437.                 }
438.
439.             }
440.
441.
442.
443.             //get combination
444.             else if(last_history == COMB_110){
445.
446.                 //go to combination table to get prediction
447.                 int prediction = this->combination_table[COMB_110];
448.
449.                 //if prediction and program action were taken
450.                 if((prediction == TAKEN) && (program_action ==
   TAKEN)){
451.
452.                     //update number of correct predictions
453.                     this->number_of_correct_predictions++;
454.
455.                     //update history table
456.                     this->global_history_table.push_back(COMB_101);
457.                 }
458.                 //if prediction and program action were not taken
459.                 else if( (prediction == NOT_TAKEN) && (program_action
   == NOT_TAKEN)){
460.
461.                     //update number of correct predictions
462.                     this->number_of_correct_predictions++;
463.
464.                     //update history table
465.                     this->global_history_table.push_back(COMB_100);
466.                 }
467.                 else if( (prediction == TAKEN ) && (program_action ==
   NOT_TAKEN) ){
```

```
468.
469.                     //update number of correct predictions
470.                     this->number_of_miss_predictions++;
471.
472.                     //update combination_table
473.                     this->combination_table[COMB_110] = NOT_TAKEN;
474.
475.                     //update history table
476.                     this->global_history_table.push_back(COMB_100);
477.                 }
478.                 else if( ( prediction == NOT_TAKEN) &&
    ( program_action == TAKEN)){
479.
480.                     //update number of correct predictions
481.                     this->number_of_miss_predictions++;
482.
483.                     //update combination_table
484.                     this->combination_table[COMB_110] = TAKEN;
485.
486.                     //update history table
487.                     this->global_history_table.push_back(COMB_101);
488.                 }
489.
490.             }
491.
492.
493.
494.             //get combination
495.             else if(last_history == COMB_111){
496.
497.                 //go to combination table to get prediction
498.                 int prediction = this->combination_table[COMB_111];
499.
500.                 //if prediction and program action were taken
501.                 if((prediction == TAKEN) && (program_action ==
    TAKEN)){
502.
503.                     //update number of correct predictions
504.                     this->number_of_correct_predictions++;
505.
506.                     //update history table
507.                     this->global_history_table.push_back(COMB_011);
508.                 }
509.                 //if prediction and program action were not taken
510.                 else if( (prediction == NOT_TAKEN) && (program_action
    == NOT_TAKEN)){
511.
512.                     //update number of correct predictions
513.                     this->number_of_correct_predictions++;
514.
515.                     //update history table
516.                     this->global_history_table.push_back(COMB_010);
517.                 }
518.                 else if( (prediction == TAKEN ) && (program_action ==
    NOT_TAKEN) ){
519.
520.                     //update number of correct predictions
```

```
521.                     this->number_of_miss_predictions++;
522.
523.                     //update combination_table
524.                     this->combination_table[COMB_111] = NOT_TAKEN;
525.
526.                     //update history table
527.                     this->global_history_table.push_back(COMB_010);
528.                 }
529.                 else if( ( prediction == NOT_TAKEN) &&
    ( program_action == TAKEN)){
530.
531.                     //update number of correct predictions
532.                     this->number_of_miss_predictions++;
533.
534.                     //update combination_table
535.                     this->combination_table[COMB_111] = TAKEN;
536.
537.                     //update history table
538.                     this->global_history_table.push_back(COMB_011);
539.                 }
540.
541.             }
542.         }
543.
544.
545. };
546.
547.
548. #endif
```

MERGE SORT WITH 3-BIT PREDICTOR INCLUDED

```
1. /*
2. *Date: 11/21/2020
3. *File: MergeSort_3Bit.cpp
4. */
5.
6.
7. // C++ program for Merge Sort
8. #include <stdlib.h>
9. #include <stdio.h>
10.  #include <time.h>
11.  #include "BranchStats_3Bits.h"
12.
13.  using namespace std;
14.
15.  //Problem size
16.  const int N = 200;
17.
18.  //Branch Prediction Parameters
19.  const int TOTAL_BRANCHES = 8;
20.
21.
22.  //declare a global stats
23.  BranchStats_3Bits branch_stats[TOTAL_BRANCHES];
24.
25.
26.  /*
27.  *This function displays the contents of a dynamic array.
28.  *@param *ar the dynamic integer array whose contents are to be
   displayed
29.  *@param ar_size the number of elements inside the array or the lengh
   of the array
30.  *@returns None
31.  */
32.  void display_array_contents(int * ar, int ar_size){
33.
34.      //initialize count
35.      int count = 0;
36.
37.      display_loop:
38.          if (count > ar_size - 1){
39.
40.              printf("\n");
41.
42.              return;
43.          }
44.
45.          printf("%d ", *(ar + count) );
46.
47.          count = count + 1;
48.
49.          goto display_loop;
50.
51.
52.  }
```

```
53.
54.
55.  // Merges two subarrays of arr[].
56.  // First subarray is arr[l..m]
57.  // Second subarray is arr[m+1..r]
58.  void merge(int *array, int l, int m, int r)
59.  {
60.      int i , j , k, nl, nr;
61.
62.     //size of left sub-arrays
63.     nl = m-l+1;
64.
65.     //size of right sub-arrays
66.     nr = r-m;
67.
68.     // Create temp arrays
69.     int larr[nl];
70.
71.     int rarr[nr];
72.
73.      //copy to left temp array
74.      i = 0;
75.
76.      for_loop_left:
77.
78.          if( i > nl-1){ //BRANCH 1
79.
80.              //update taken branch stats
81.              branch_stats[1].increase_num_taken_branches();
82.              branch_stats[1].update_predictions(TAKEN);
83.
84.              goto done_for_loop_left;
85.          }
86.
87.          //update not taken branch stats
88.          branch_stats[1].increase_num_not_taken_branches();
89.          branch_stats[1].update_predictions(NOT_TAKEN);
90.
91.
92.          larr[i] = array[l + i];
93.
94.          i++;
95.
96.          goto for_loop_left;
97.
98.      done_for_loop_left:
99.
100.     j = 0;
101.
102.     //copy to right temp array
103.     for_loop_right:
104.
105.         if( j > nr-1){ //BRANCH 2
106.
107.             //update taken branch stats
108.             branch_stats[2].increase_num_taken_branches();
109.             branch_stats[2].update_predictions(TAKEN);
```

```
110.
111.             goto done_for_loop_right;
112.         }
113.
114.         //update not taken branch stats
115.         branch_stats[2].increase_num_not_taken_branches();
116.         branch_stats[2].update_predictions(NOT_TAKEN);
117.
118.
119.         rarr[j] = array[m + 1 + j];
120.
121.         j++;
122.
123.         goto for_loop_right;
124.
125.     done_for_loop_right:
126.
127.     // Merge the temp arrays back into arr[l..r]
128.
129.     // Initial index of first subarray
130.     i = 0;
131.
132.     // Initial index of second subarray
133.     j = 0;
134.
135.     // Initial index of merged subarray
136.     k = l;
137.
138.
139.
140.     //merge arrays
141.     merge_array_while_loop:
142.
143.         //converting while to if, took 2 hours but nedded to change
    logic from && to ||
144.         if((i > nl-1) || (j > nr-1)){ //BRANCH 3
145.
146.             //update taken branch stats
147.             branch_stats[3].increase_num_taken_branches();
148.             branch_stats[3].update_predictions(TAKEN);
149.
150.             goto done_merge_array_while_loop;
151.         }
152.
153.         //update not taken branch stats
154.         branch_stats[3].increase_num_not_taken_branches();
155.         branch_stats[3].update_predictions(NOT_TAKEN);
156.
157.         if(larr[i] <= rarr[j]){ //BRANCH 4, data dependent branching
158.
159.             array[k] = larr[i];
160.             i++;
161.             k++;
162.
163.             //update taken branch stats
164.             branch_stats[4].increase_num_taken_branches();
165.             branch_stats[4].update_predictions(TAKEN);
```

```
166.
167.             goto merge_array_while_loop;
168.         }
169.
170.         //update not taken branch stats
171.         branch_stats[4].increase_num_not_taken_branches();
172.         branch_stats[4].update_predictions(NOT_TAKEN);
173.
174.         if(larr[i] > rarr[j]){ //BRANCH 5, data dependent branching
175.
176.             array[k] = rarr[j];
177.             j++;
178.             k++;
179.
180.             //update taken branch stats
181.             branch_stats[5].increase_num_taken_branches();
182.             branch_stats[5].update_predictions(TAKEN);
183.
184.             goto merge_array_while_loop;
185.         }
186.
187.         //update not taken branch stats
188.         branch_stats[5].increase_num_not_taken_branches();
189.         branch_stats[5].update_predictions(NOT_TAKEN);
190.
191.     done_merge_array_while_loop:
192.
193.     // Copy the remaining elements of
194.     // L[], if there are any
195.     copy_remaining_left_while_loop:
196.
197.         if(i > nl-1){ //BRANCH 6
198.
199.             //update taken branch stats
200.             branch_stats[6].increase_num_taken_branches();
201.             branch_stats[6].update_predictions(TAKEN);
202.
203.             goto done_copy_remaining_left_while_loop;
204.         }
205.
206.         //update not taken branch stats
207.         branch_stats[6].increase_num_not_taken_branches();
208.         branch_stats[6].update_predictions(NOT_TAKEN);
209.
210.         array[k] = larr[i];
211.         i++;
212.         k++;
213.
214.         goto copy_remaining_left_while_loop;
215.
216.
217.     done_copy_remaining_left_while_loop:
218.
219.     // Copy the remaining elements of
220.     // R[], if there are any
221.
222.     copy_remaining_right_while_loop:
```

```
223.
224.         if (j > nr-1) { //BRANCH 7
225.
226.             //update taken branch stats
227.             branch_stats[7].increase_num_taken_branches();
228.             branch_stats[7].update_predictions(TAKEN);
229.
230.             goto done_copy_remaining_right_while_loop;
231.         }
232.
233.         //update not taken branch stats
234.         branch_stats[7].increase_num_not_taken_branches();
235.         branch_stats[7].update_predictions(NOT_TAKEN);
236.
237.         array[k] = rarr[j];
238.         j++;
239.         k++;
240.
241.         goto copy_remaining_right_while_loop;
242.
243.     done_copy_remaining_right_while_loop:
244.         return;
245. }
246.
247.
248. // l is for left index and r is
249. // right index of the sub-array
250. // of arr to be sorted
251. void mergeSort(int *array, int l, int r)
252. {
253.     int m;
254.
255.     if(l > r - 1){ //BRANCH 0
256.
257.         //update taken branch stats
258.         branch_stats[0].increase_num_taken_branches();
259.         branch_stats[0].update_predictions(TAKEN);
260.
261.         goto done_merge_sort;
262.     }
263.
264.     //update not taken branch stats
265.     branch_stats[0].increase_num_not_taken_branches();
266.     branch_stats[0].update_predictions(NOT_TAKEN);
267.
268.     // Same as (l+r)/2, but avoids
269.     // overflow for large l and h
270.     //int m = (l + r - l) / 2;
271.     m = l + (r-l)/2;
272.
273.     // Sort first and second halves
274.     mergeSort(array, l, m);
275.
276.     mergeSort(array, m + 1, r);
277.
278.     merge(array, l, m, r);
279.
```

```
280.     done_merge_sort:
281.         int done = 0;
282.
283. }
284.
285.
286.
287. // Driver code
288. int main()
289. {
290.     printf("***********Simulation of 3 Bit Branch Prediction
   Scheme*************\n\n");
291.
292.     printf("***********Algorithm Used: Merge
   Sort*****************************\n\n");
293.
294.     printf("**********Reading Numbers From
   File*********************************\n\n");
295.     FILE *fp;
296.     fp = fopen("rand_numbers.txt","r");
297.     int *arr = new int[N];
298.     for(int i = 0; i < N ; i++){
299.         fscanf(fp,"%d",(arr+i));
300.     }
301.
302.
303.     printf("***************Generated Array of Random Numbers
   *****************\n");
304.     display_array_contents(arr, N);
305.     printf("\n");
306.
307.     printf("**************Started
   Sorting*************************************\n\n");
308.     mergeSort(arr, 0, N - 1);
309.
310.     printf("**************Sorted Array is*************\n\n");
311.     display_array_contents(arr, N);
312.     printf("\n");
313.
314.     printf("*********Branch Statistics***********\n\n");
315.     for(int i = 0; i < TOTAL_BRANCHES; i++){
316.         printf("Branch: %d\n",i);
317.         branch_stats[i].print_statistics();
318.         printf("\n");
319.     }
320.
321.
322.     return 0;
323. }
```