ECSE 425 – Computer Organization and Architecture

Software Simulation of Branch Prediction Strategies

SIMON FOUCHER

APRIL 2ND, 2009

1. HARDWARE SIMULATED

1.1 SOFTWARE FLOW

All four code files were designed using a similar architecture. When the software is ran, the main function opens a file pointer to the first trace, and passes it to the function readTrace, as well as the number of bits for the size of the predictor (if applicable). This function is called 10 times/trace, for 2-bit, 4-bit, …, 20-bit predictors.

Once in the readTrace function, the first step is to initialize and format the arrays. We also calculate two boundary variables named “Nmask” and “threshold”. Nmask = 2^n - 1 and serves as a maximum boundary for the predictor (the minimum boundary is always 0). Threshold = (Nmask + 1)/2 and serves as a predictor decision threshold. As a convention, low value predictors favor branch not taken. For example, on an 8 bit test, Nmask = 255, which is the maximum value that the predictor can take, and threshold is 128, meaning that if the predictor < 128, it predicts that the branch will be taken.

Afterwards, we fetch one line of data at a time from the trace file which got opened (hence the huge processing time due to massive I/O wait). The address is extracted and a lookup is made on the “addresses” array to see if it is the first time that this particular branch address is encountered. The array is scanned from index k = 0, until the system finds either a previous entry of that branch, or an empty spot on the array. If the entry is found, k is recorded and used as an index for selecting predictors. If the system encounters an empty spot on the array before finding a match, the address is recorded at that spot and the index k will now be associated with this branch for future occurrences.

Note: due to the memory address limitations of my 32-bit PC, I was not able to run the correlating predictor with the server trace file (required at least 10300 x 256 addresses) and constantly got segmentation fault. To counter this, a Boolean variable called “tooBig” was introduced to inform the predictor that it needed to “compress” the data if it was analyzing the server trace. In order to do so, the first two decimal bits of the addresses were ignored, which generated a decrease in performance for the server traces, since multiple addresses might have been sharing the same LSBs (least significant bits). The same scheme was used for every type of predictors on the server trace, therefore the data collected is still usable (we can adequately comment on the performance of different predictors, but we cannot compare server traces with INT traces, since the INT traces were fully indexed, whereas the server traces were indexed using a subset of address bits).

After extraction of the data and computation of index k, these parameters are fed to the predictors and statistical data is gathered based on whether a good prediction was made or not. At the same time, the predictors adjusted themselves to reflect the new branch outcome.

The predictors were implemented using different algorithms (described below). In order to interpret the prediction of a branch, if the predictor < threshold, the prediction is branch not taken, and if predictor > threshold, then we predict branch taken. Whether the prediction was right or wrong, the predictor got modified with accordance to the branch result (decremented for not taken and incremented for taken). The predictors were never decremented below 0 or above “Nmask”. Whenever a threshold was passed, the predictors jumped at the other edge of the prediction (see book p.83). For example, on a 3-bit predictor (Nmask = 7, threshold = 4, therefore 0-3 predict not taken and 4-7 predict taken), if the predictor was at 4 and the branch was not taken, the predictor was set to 0.

2. PREDICTOR ALGORITHMS

2.1 STATIC BRANCH PREDICTION POLICIES

Code file: /code/static.c
Data collected: See Appendix II-A)

In order to analyze the result of a simple static prediction policy, the trace files were read and a basic statistical analysis was performed on the number of branched taken and not taken. This information is sufficient to discuss the implementation of a static always predict taken or not taken algorithm, since their performance are complementary.

For example, by knowing that 18% of branches were taken on the FP traces, we know that a "predict taken" mechanism would have a failure rate of 82%, whereas a "predict not taken" algorithm would result in a failure rate of 18%.

2.2 BRANCH PREDICTION BUFFER (N-BIT PREDICTOR)

Code file: /code/buffer.c
Data collected: See Appendix II-B

A branch prediction buffer was implemented and results were gathered for 2, 4, 8, …, 20-bit buffers. Due to the non-dynamic nature of this project, we were able to generate a unique predictor for every single branch address present on the trace files (see section 1.1 Software flow). After figuring out the branch index value (k), this value and the branch result were passed on to the predictor.

The predictor consists of an array of 10 000 integers called "predictorLocal". The index k was used to select which predictor to use, and the integer present was used as a predictor. In order to minimize memory use, the predictors mimicked the behavior of binary strings, but were manipulated using integers. To demonstrate the methodology used we will use in this report (h) to imply hardware implementation and (s) for software implementation.

Here is the explanation of an 8-bit predictor. Here, Nmask = 255 and threshold = 128.

Range:	(h) [0000 0000, 1111 1111]	(s) [0, 255]
Range for predict not taken:	(h) [0000 0000, 0000 1111]	(s) [0, 127]
Range for predict taken:	(h) [0001 0000, 1111 1111]	(s) [128, 255]

After interpreting the predictor, the system read the branch result, and "totalGoodPredictions" (an integer keeping track of the number of good predictions) got incremented in the event of a good prediction. Also, the local predictor gets incremented in the event of a branch taken and decremented in

the event of a branch not taken. As described in the book, whenever a threshold is reached, the predictor jumps to the other end of the scale.

For example:

Predictor value:	(h) [0000 1111]	(s) [127]
New value if branch Taken:	(h) [1111 1111]	(s) [255]
New value if branch not Taken	(h) [0000 1110]	(s) [126]

After these updates, the system fetches the next line of the trace and continues until the trace is done. All four traces are analyzed in this manner.

2.2 CORRELATING PREDICTOR

Code file: /code/correlator.c
Data collected: See Appendix II-B

The correlating predictor uses similar predictors with one major modification. Every branch address has 256 local predictors. As above, once the branch address is known, the address array is scanned to see if there already exists a history for this branch. A slight modification had to be made for the server trace. Because of the large number of different branch addresses, the amount of memory allocated at run time to account for every single address caused a Heap/Stack collision and forced the OS (operating system) to terminate the process. To counter this, the variable “tooBig” was implemented to notify that the server address space was too big. In such a case, the two most significant digits of the branch address were ignored, which resulted in a reduced accuracy of predictions when evaluating the server traces.

The branch enters the predictor with an index k to select which predictor to choose. Once predictor[k] is selected, the globalPredictor is used to select among 256 local predictors for a single branch. If a good prediction is made, a counter is incremented to keep statistical data. Within the same thread, both the local predictor selected as well as the global predictors get updated (in the same manner described in section 2.1 Branch prediction buffer) to reflect the outcome of the branch.

This provides an extra advantage to the predictor, because the global history is used to reflect similar loops and counters encountered within a program.

2.3 TOURNAMENT PREDICTOR

Code file: /code/tournament.c
Data collected: See Appendix II-B

The tournament predictor implements both the buffer predictor and the correlating predictor. For every branch encountered, both predictors make a prediction based on their state in the same manner described above. In addition, a 2-bit history table is kept as a record of which predictor was the most accurate at the last iteration. The 2 bits of history are stored in an integer array called "chooser" and every branch address has its own history table. In the last few iterations, if the buffer predictor is more accurate, it is selected to make this prediction, and vice versa for the correlating predictor.

Once the outcome of the branch is known, the buffer predictor, local predictors and global predictors that are present within the correlating predictor get updated. Based on whether one of the two predictors was more accurate than the other, the history table associated with this particular branch also got updated to reflect this information.

For every address of every trace, statistical data was gathered based on the performance of the tournament predictor.

3. Results

The following table was built using the results gathered while running static.c (see Appendix II for raw data).

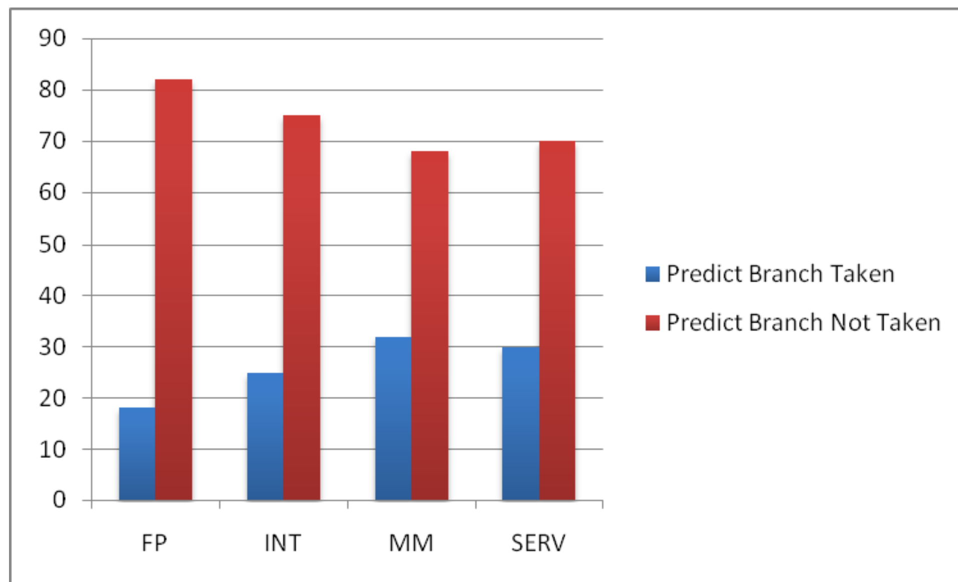
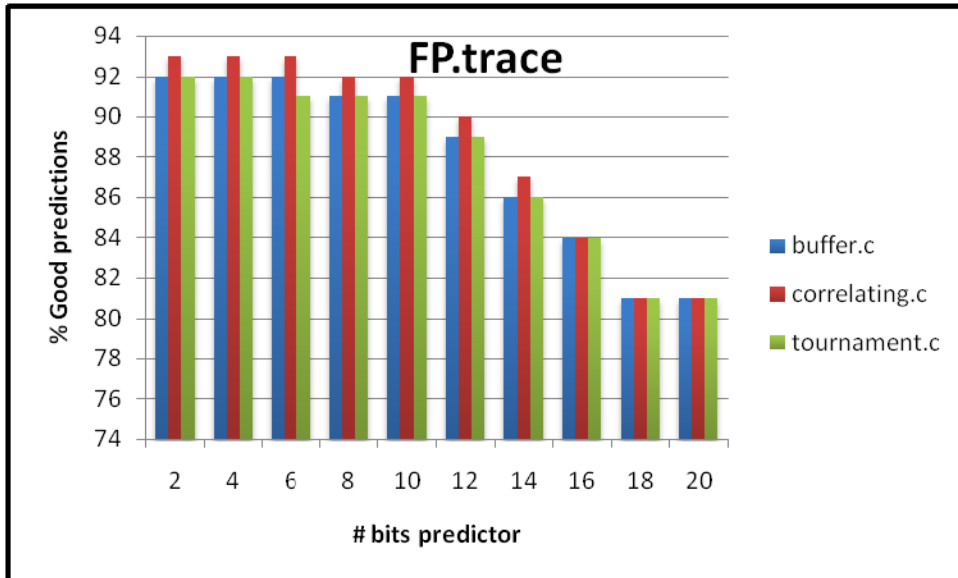


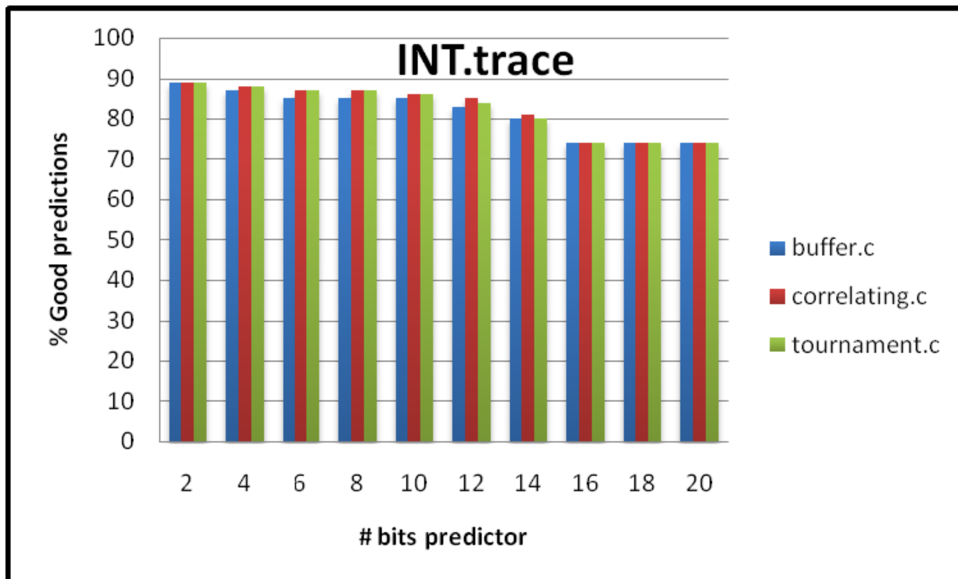
Figure 3.1: Success rate of a static Branch predictor. Predict taken in red, and predict not taken in blue.

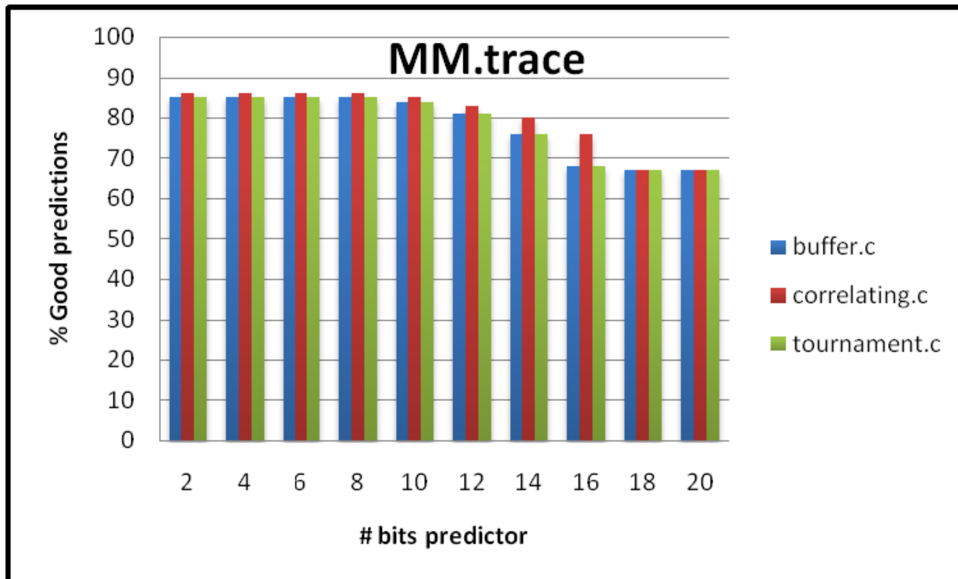
From Figure 3.1, we can easily observe that for all 4 trace files, 68% to 82% of branches end up not being taken. This is coherent with what could be expected, since the purpose of loops is usually to repeat the same task many times until it is fully accomplished. This implies that branch conditions, when evaluated, will more often be not taken such that the loop's body can be executed many times.

Based on this, the most efficient static branch prediction bias would be to predict that branches will not be taken. This would end up being right roughly ¾ of the time for general purpose processors. As further results demonstrate, there are better prediction schemes.

The following data shown in Figure 3.2 was gathered to compare all 3 “intelligent” branch prediction algorithms, i.e. branch prediction buffer, correlating predictor and tournament predictor.







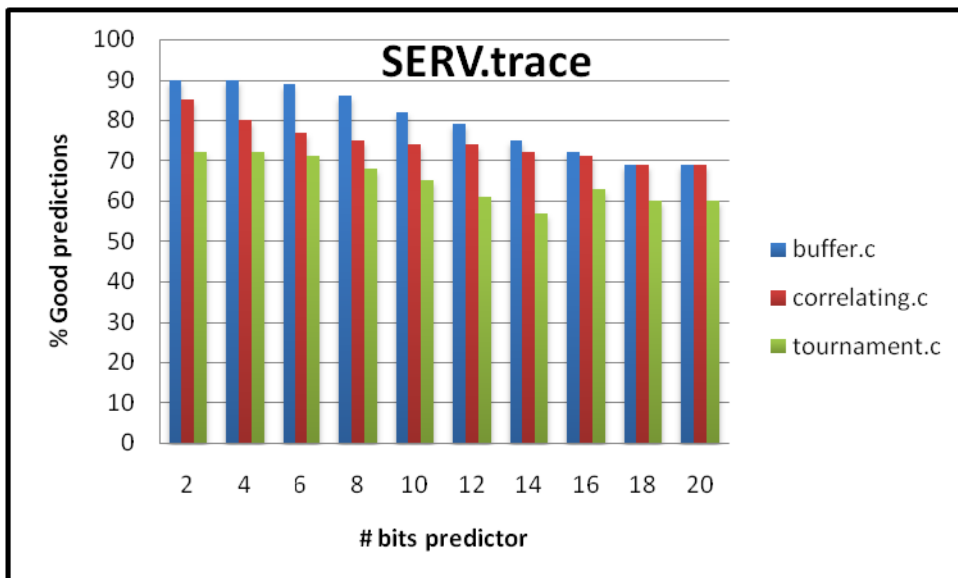


Figure 3.2: Comparing the different branch prediction algorithms using 4 trace files

From the data gathered, we can see that the optimal size for the prediction buffer is 2 bits. This is due to the fact that for larger bit size predictors, there needs to be many consecutive mistakes before the predictor changes its bias. By doing so, those predictors lose sensitivity and their performance decreases. This phenomenon is useful for the actual implementation, because a good performance from low-bit predictors reduces the quantity of hardware required.

Another observation is that, in general, the correlating predictors are better than the buffer predictors. This result was also expected since correlating predictors take advantage of global branch

information. This information provides a way to choose from an array of local predictors, so different predictors can be selected to reflect different situations for every branch. The performance gap between buffer and correlating predictors was most accentuated in the FP trace.

We can also observe that for the FP trace, the tournament prediction was not more accurate than the correlating predictor. This is most probably due to the fact that operations were repeated many times and this was captured by the tournament choosing table. Floating point operations generally require many clock cycles, so the global predictor became a good way of capturing this regularity.

In the INT trace, the best prediction algorithm was the tournament predictor. This can be attributed to the fact that the tournament makes intelligent selections of predictors. Since there are a relatively small number of integer operations, the choosing table was efficient in allocating an adequate predictor for the branch operation at hand.

Another conclusion worth pointing out is that the performance of the tournament predictor in the SERV trace was lower. This is not necessarily due to a lack in performance of the predictor. As mentioned earlier, because of the chosen architecture, the branch address array was too large to be allocated enough memory for the server trace, which forced the implementation of a predictor that is less sensitive to changes in addresses. By conglomerating different addresses within the same predictors, the performance of the predictors was greatly decreased.

We can also observe the performance of the prediction schemes on a more global scale. To do so, we use the gathered data shown in Figure 3.2 and calculate the average performance of each prediction scheme for each trace. In addition, we calculate the average performance of each prediction scheme for all traces. The data is represented in Figure 3.3.

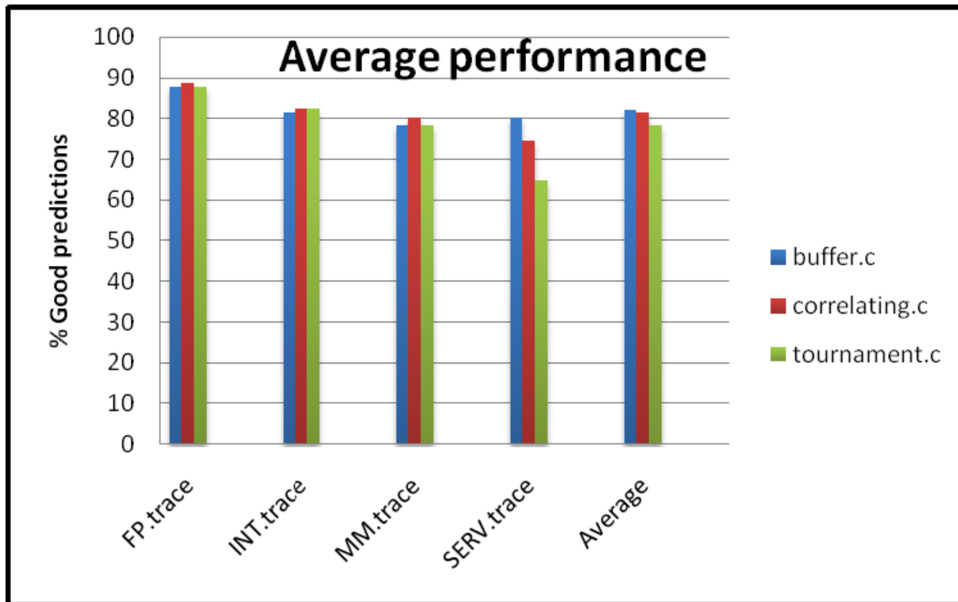


Figure 3.3: Comparing the average performance of branch prediction algorithms

Figure 3.3 shows that the performance of each prediction scheme differs from trace to trace. Specifically, the buffer predictor is significantly more accurate than the other two in the SERV trace.

The correlating predictor has the same performance with the buffer predictor in FP and MM traces, but is slightly better in the INT trace. However, its performance highly drops in the SERV trace.

The tournament predictor is slightly better in FP and MM traces, but it cannot compete with the buffer predictor in the SERV trace.

Finally, if we consider the average performance for all traces, the results show that the buffer is slightly ahead of the other two prediction schemes.

Appendix I: Code included

I-A) Overview

All the results were generated using C code on a 32 bit Linux platform. The scripts were compiled using gcc and debugged using ddd.

Notes:

- All code files are located in the subdirectory 'code/'
- The code will probably not compile and will definitely not work properly on a Windows platform

Code files included:

static.c: Gives out statistics on the number of branches taken/not taken without any prediction algorithm.

buffer.c: Implements a buffer prediction algorithm. Every trace will be tested for 2,4,…,20 byte buffers

correlating.c: Implements a correlating prediction algorithm, which uses an 8-bit global predictor to select from 256 local predictors/branch address. Every trace will be tested for 2,4,…,20 byte buffers

tournament.c: Every branch is passed through a buffer as well as a correlating predictor. Based on previous accuracies of both predictors, a 2-bit controller selects which predictor to implement.

I-B) How to compile and run

Instructions:

- To get gcc, open a terminal window in Linux and type $ ***sudo apt-get install gcc***
- In order for the trace files to be accessed, they need to be placed in a directory called '***traces/***', to which the directory containing the program is root (e.g. code/traces/).
- To compile a code file, type $ ***gcc filename.c –w***
- To execute the binary produced, type $ ***./a.out***

APPENDIX II: DATA COLLECTED

II-A) STATIC.C

FP.trace

Total BR: 2213673 Taken: 412899 Ratio: 18%

INT.trace

Total BR: 4184792 Taken: 1068045 Ratio: 25%

MM.trace

Total BR: 2229289 Taken: 713587 Ratio: 32%

SERV.trace

Total BR: 3660616 Taken: 1111595 Ratio: 30%

II-B) PREDICTION SCHEMES

			buffer.c		correlating.c		tournament.c	
	bits	**TotBR**	**GoodPr**	**% good**	**GoodPr**	**% good**	**GoodPr**	**% good**
FP.trace	2	2213673	2041014	**92**	2070849	**93**	2040070	**92**
	4	2213673	2045204	**92**	2070828	**93**	2044077	**92**
	6	2213673	2037944	**92**	2061395	**93**	2036417	**91**
	8	2213673	2034160	**91**	2056295	**92**	2030878	**91**
	10	2213673	2024693	**91**	2044533	**92**	2021957	**91**
	12	2213673	1990585	**89**	2006191	**90**	1990710	**89**
	14	2213673	1918330	**86**	1933428	**87**	1918330	**86**
	16	2213673	1869871	**84**	1881298	**84**	1869870	**84**
	18	2213673	1800774	**81**	1800774	**81**	1800774	**81**
	20	2213673	1800774	**81**	1800774	**81**	1800774	**81**

	bits	TotBR	buffer.c GoodPr	buffer.c % good	correlating.c GoodPr	correlating.c % good	tournament.c GoodPr	tournament.c % good
INT.trace	2	4184792	3761918	**89**	3766125	**89**	3749958	**89**
	4	4184792	3658473	**87**	3716688	**88**	3702496	**88**
	6	4184792	3564452	**85**	3646954	**87**	3643838	**87**
	8	4184792	3573771	**85**	3656075	**87**	3655361	**87**
	10	4184792	3557480	**85**	3632158	**86**	3607969	**86**
	12	4184792	3507075	**83**	3562238	**85**	3532909	**84**
	14	4184792	3372952	**80**	3395253	**81**	3376535	**80**
	16	4184792	3126537	**74**	3126532	**74**	3126536	**74**
	18	4184792	3116747	**74**	3116747	**74**	3116747	**74**
	20	4184792	3116747	**74**	3116747	**74**	3116747	**74**
MM.trace	2	2229289	1905842	**85**	1918628	**86**	1895885	**85**
	4	2229289	1908492	**85**	1920977	**86**	1912730	**85**
	6	2229289	1905796	**85**	1922498	**86**	1910134	**85**
	8	2229289	1898798	**85**	1923038	**86**	1901594	**85**
	10	2229289	1876778	**84**	1902297	**85**	1877087	**84**
	12	2229289	1812288	**81**	1868053	**83**	1813427	**81**
	14	2229289	1703586	**76**	1799523	**80**	1703504	**76**
	16	2229289	1537199	**68**	1699376	**76**	1537199	**68**
	18	2229289	1515702	**67**	1515702	**67**	1515702	**67**
	20	2229289	1515702	**67**	1515702	**67**	1515702	**67**
SERV.trace	2	3660616	3325242	**90**	3134406	**85**	2667523	**72**
	4	3660616	3297983	**90**	2940661	**80**	2651695	**72**
	6	3660616	3261423	**89**	2846581	**77**	2616696	**71**
	8	3660616	3149829	**86**	2775663	**75**	2520838	**68**
	10	3660616	3006560	**82**	2728744	**74**	2382900	**65**
	12	3660616	2895137	**79**	2709956	**74**	2263377	**61**
	14	3660616	2751657	**75**	2667985	**72**	2113811	**57**
	16	3660616	2655985	**72**	2613431	**71**	2311200	**63**
	18	3660616	2557681	**69**	2549021	**69**	2213557	**60**
	20	3660616	2549021	**69**	2549021	**69**	2210291	**60**

		buffer.c		correlating.c		tournament.c	
	TotBR	GoodPr	% good	GoodPr	% good	GoodPr	% good
FP.trace	2213673	1956335	**87.9**	1972637	**88.6**	1955386	**87.8**
INT.trace	4184792	3435615	**81.6**	3473552	**82.5**	3462910	**82.3**
MM.trace	2229289	1758018	**78.3**	1798579	**80.2**	1758296	**78.3**
SERV.trace	3660616	2945052	**80.1**	2751547	**74.6**	2395189	**64.9**
Average	3072093	2523755	**81.975**	2499079	**81.475**	2392945	**78.325**

APPENDIX III: SOURCE C CODE

III-A) STATIC.C

```
/***********************************/
/*	Comp Arc Project
/*	By: Simon Foucher
/*	260 223 197
/***********************************/


/***********************************/
/*	includes and definitions
/***********************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>


#define TRUE 1
#define FALSE 0
typedef int BOOL;


/***********************************/
/*	Function declarations
/***********************************/


/***********************************/
/*	Global Variables & datatypes
/***********************************/


/***********************************/
/*	Functions
/***********************************/

// To clean up gathered strings from text
void stripCRLF(char array[])
{
   int i;

   for(i=0;i<500 && array[i]!='\0';i++)
     if (array[i]=='\r' || array[i]=='\n')
                    array[i]='\0';
```

```
}

void readTrace(FILE *fp)
{
	// Following arrays used to store statistical data (related elements share a common index)
	char addresses[10000][20];
	int frequency[10000];
	int taken[10000];
	int totalBR = 0, totalTak = 0;



	char temp[50];
	char tmpAddress[10];
	int i, j, k;

	// Format arrays
	for(i = 0; i < 10000; i++){
		frequency[i] = 0;
		taken[i] = 0;
	}



	// Extract lines of data; break at EOF
	while( fgets(temp, sizeof(temp), fp) != NULL ){

		stripCRLF(temp);						// Clean up the
string
		//fputs(temp, stdout);
		totalBR++;


		for(i = 0; ; i++){						//	Extract
address of branch
			tmpAddress[i] = temp[i];
			if(tmpAddress[i] == ' ')
				break;
		}

		tmpAddress[i++] = NULL;						// Add a proper
null terminator and increment index (i points at branch result)

		//for(;i<15; i++)
		//	printf("--temp[%d] = %d\n",i, temp[i]);



		for(k = 0; k < 10000; k++)					// Find out if the entry
already exists
			if(addresses[k][0] == NULL || !strncmp(addresses[k], tmpAddress, 9))
```

```
                                        break;

                    if(k != 9999){                                                                                          // If it is
a new entry, k points at the first empt spot
                              if(addresses[k][0] == NULL)
                                        //printf("\tNew Entry: %s \tat index: %d \t taken (if 1) '%c' \n\t", tmpAddress,
k, temp[i]);
                              strcpy(addresses[k], tmpAddress);
                    }


                    // At this point, i points at the branch result and k at the array index whete the element goes.
                    // record the statistics
                    frequency[k]++;

                    // ASCII 48 = 0, 49 = 1
                    if(temp[i] == 49){
                              taken[k] += 1;
                              totalTak += 1;
                    }

                    //printf("\tAddress: %s, \t used: %d, \t Taken: %d\n", addresses[k], frequency[k], taken[k]);


          }// End of while getting lines

          fclose(fp);

          printf("\n DONE!!\n\nResults:\n");

/*        for(k = 0; k < 9999; k++){
                    if(addresses[k][0] == NULL)
                              break;
                    printf("k: %d\t Add: %s\tTaken: %d\tFreq: %d\tRatio: %d\n", k, addresses[k], taken[k],
frequency[k], taken[k]*100/frequency[k]);
          }
*/
          printf("\n\nTotal BR: %d\tTaken: %d\tRatio: %d%\n\n",totalBR, totalTak, totalTak*100/totalBR);
          printf("--------------------------------------------\n");



}// End if readTrace




/*******************************************************************************/
/*        Function  main (int argc, char *argv[])
/*******************************************************************************/

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

```
FILE *fp;

// FP.trace
if((fp = fopen("traces/FP.trace","r"))==NULL)                    //Open the password file
        {
                puts("Cannot open FP.trace\n"
                        "Please make sure it is located in a directory called 'traces/'\n"
                        "having this program's directory as a root\n");
        }
else
        printf("Sucessfully opened FP.trace\nStarting analysis\n");

readTrace(fp);


// INT.trace
if((fp = fopen("traces/INT.trace","r"))==NULL)                   //Open the password file
        {
                puts("Cannot open INT.trace\n"
                        "Please make sure it is located in a directory called 'traces/'\n"
                        "having this program's directory as a root\n");
        }
else
        printf("Sucessfully opened INT.trace\nStarting analysis\n\n");

readTrace(fp);


// MM.trace
if((fp = fopen("traces/MM.trace","r"))==NULL)                    //Open the password file
        {
                puts("Cannot open MM.trace\n"
                        "Please make sure it is located in a directory called 'traces/'\n"
                        "having this program's directory as a root\n");
        }
else
        printf("Sucessfully opened MM.trace\nStarting analysis\n\n");

readTrace(fp);


// SERV.trace
if((fp = fopen("traces/SERV.trace","r"))==NULL)                  //Open the password file
        {
                puts("Cannot open SERV.trace\n"
                        "Please make sure it is located in a directory called 'traces/'\n"
                        "having this program's directory as a root\n");
        }
else
        printf("Sucessfully opened SERV.trace\nStarting analysis\n\n");

readTrace(fp);
```

```
}
```

III-B) BUFFER.C

```
/************************************/
/*	Comp Arc Project
/*	By: Simon Foucher
/*	260 223 197
/************************************/


/************************************/
/*	includes and definitions
/************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>




/************************************/
/*	Function declarations
/************************************/


/************************************/
/*	Global Variables & datatypes
/************************************/
#define TRUE 1
#define FALSE 0
typedef int BOOL;

/************************************/
/*	Functions
/************************************/

// To clean up gathered strings from text
void stripCRLF(char array[])
{
  int i;

  for(i=0;i<500 && array[i]!='\0';i++)
    if (array[i]=='\r' || array[i]=='\n')
			array[i]='\0';
}


// To compute power
int power(int x, int n){
```

```
	if(n == 0)
		return 1;

	else{
		int tmp = power(x, n/2);
		if(n%2)
			return tmp*tmp*x;
		else
			return tmp*tmp;
	}

}



void readTrace(FILE *fp, int Nbits)
{

	// Following arrays used to store statistical data (related elements share a common index)
	char addresses[100000][20];
	int predictorLocal[100000];
	int totalBR = 0, totalGoodPredictions = 0;
	int Nmask, treshold;

	char temp[50];
	char tmpAddress[10];
	int i, j, k;
	int done;



/************************************                                          INITIALIZATION
***********************************************/

	// Format arrays with preditction not taken
	for(i = 0; i < 100000; i++){
		predictorLocal[i] = 0;
	}

	// Build bit mask for N bit predictor
	Nmask = power(2, Nbits);
	treshold = Nmask/2;


// DEBUGGING
// printf("N = %d\t2^N = %d\tMask: 0x%x\n", Nbits, Nmask+1,Nmask);



/************************************                      DATA                    FETCH
***********************************************/
	// Extract lines of data; break at EOF
	while( fgets(temp, sizeof(temp), fp) != NULL ){
```

```
		stripCRLF(temp);								// Clean up the string
//printf("Line: %s", temp);
		totalBR++;
	// Increment total branch count


		for(i = 0; ; i++){								//		Extract address of branch
			tmpAddress[i] = temp[i];
			if(tmpAddress[i] == ' ')
				break;
		}

		tmpAddress[i++] = NULL;							// Add a proper null terminator and increment index (i points at branch result)

		//for(;i<15; i++)
		//	printf("--temp[%d] = %d\n",i, temp[i]);



		for(k = 0; k < 100000; k++)						// Find out if the entry already exists
			if(addresses[k][0] == NULL || !strncmp(addresses[k], tmpAddress, 9))
				break;

		if(k != 99999){										// If it is a new entry, k points at the first empt spot
			if(addresses[k][0] == NULL)
			strcpy(addresses[k], tmpAddress);
		}
		else
			printf("Warning! Reached end of buffer!!!!\n");




/************************************	PREDICTOR ************************************************/
		/* At this point, i points at the branch result and k at the array index whete the element goes
		 NOTES:
			* ASCII 48 = 0, 49 = 1
			* Example with 8 bits used for comments
			* The 4 cases evaluated are mutually exclusive (unlike in correlating predictor)
		*/
//printf("pred Was: %x\t", predictorLocal[k]);

		// If predict "NOT taken
		// So predict Local is < 0001 0000 equivalent to Renge of [0000 0000, 0000 1111]
		if(predictorLocal[k] < treshold){
			// If branch IS taken
```

```
			if(temp[i] == 49){
				totalGoodPredictions += 0;				//	Bad
prediction

				predictorLocal[k] +=1;					//
Increment the predictor
				if(predictorLocal[k] == treshold)			// If we hit the turning
point (0001 0000), jump to "strong taken"(1111 1111)
					predictorLocal[k] = Nmask - 1;			// Nmask = 1
0000 0000 so Nmask - 1 = 1111 1111 = strong taken
			}
			// If branch NOT taken
			if(temp[i] == 48){
				totalGoodPredictions += 1;				// Good
prediction
				if(predictorLocal[k] != 0)				// If predictor !=
0000 0000
					predictorLocal[k] -= 1;				//
Reflect last BR decision ourcome (LSB of 0)
				}
		}


		// If predict "IS taken"
		// If predictorLocal is in the range [0001 0000, 1111 1111]
		if(predictorLocal[k] >= treshold){
			// If branch IS taken
			if(temp[i] == 49){
				totalGoodPredictions += 1;				// Good
prediction

				if(predictorLocal[k] != Nmask - 1)			// If predictor < 1 0000
0000 = Range og [0001 0000, 1111 1111]
					predictorLocal[k] += 1;				//
increment by 1
			}
			// If branch NOT taken
			if(temp[i] == 48){
				totalGoodPredictions += 0;				//	Bad
prediction
				predictorLocal[k] -= 1;					//
Reflect last BR decision ourcome (LSB of 0)

				if(predictorLocal[k] == treshold - 1)		// If we got to 0000 1111
					predictorLocal[k] = 0;				// Jump
to predict strong not taken 0000 0000
			}
		}

//printf("pred New: %x\t", predictorLocal[k]);
		if(predictorLocal[k] == Nmask)					// Avoid overflow
(I don't think this is possible, but just in case...)
			predictorLocal[k]--;
```

```
//printf("pred Masked: %x\n", predictorLocal[k]);


		}// End of while getting lines

/*/ Counts the number of different addresses
for(k=0;;k++)
		if(addresses[k][0] == NULL)
				break;
printf("K=%d\n",k);
*/


		printf("%d\t%d\t%d\t\t%d\n", Nbits, totalGoodPredictions, totalBR, totalGoodPredictions*100/totalBR);






}// End if readTrace




/*****************************************************************************/
/*		Function  main (int argc, char *argv[])
/*****************************************************************************/

int main (int argc, char *argv[])
{
		FILE *fp;
		int Nbits;

//FP had 430 different branch address
		printf("----------------------------------------------\n");
		// Open FP.trace
		if((fp = fopen("traces/FP.trace","r"))==NULL)				//Open the password file
				{
						puts("Cannot open FP.trace\n"
								"Please make sure it is located in a directory called 'traces/'\n"
								"having this program's directory as a root\n");
				}
		else
				printf("FP.trace Analysis\n");

		printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
		for(Nbits = 2; Nbits <=20; Nbits +=2){
				rewind(fp);
				// Rewind file pointer
				readTrace(fp, Nbits);
		}
```

```
	fclose(fp);
	printf("\n---------------------------------------------\n");


	// Open INT.trace
	if((fp = fopen("traces/INT.trace","r"))==NULL)			//Open the password file
		{
			puts("Cannot open INT.trace\n"
				"Please make sure it is located in a directory called 'traces/'\n"
				"having this program's directory as a root\n");
		}
	else
		printf("INT.trace Analysis\n");

	printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
	for(Nbits = 2; Nbits <=20; Nbits +=2){
		rewind(fp);
		// Rewind file pointer
		readTrace(fp, Nbits);
	}

	fclose(fp);
	printf("\n---------------------------------------------\n");



	// Open MM.trace
	if((fp = fopen("traces/MM.trace","r"))==NULL)			//Open the password file
		{
			puts("Cannot open MM.trace\n"
				"Please make sure it is located in a directory called 'traces/'\n"
				"having this program's directory as a root\n");
		}
	else
		printf("MM.trace Analysis\n");

	printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
	for(Nbits = 2; Nbits <=20; Nbits +=2){
		rewind(fp);
		// Rewind file pointer
		readTrace(fp, Nbits);
	}

	fclose(fp);
	printf("\n---------------------------------------------\n");


//Server has 10 388 Different branch address

	// Open SERV.trace
	if((fp = fopen("traces/SERV.trace","r"))==NULL)			//Open the password file
		{
```

```
			puts("Cannot open SERV.trace\n"
				"Please make sure it is located in a directory called 'traces/'\n"
				"having this program's directory as a root\n");
		}
	else
		printf("SERV.trace Analysis\n");

	printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
	for(Nbits = 2; Nbits <=20; Nbits +=2){
		rewind(fp);
		// Rewind file pointer
		readTrace(fp, Nbits);
	}

	fclose(fp);
	printf("\n---------------------------------------------\n");




}
```

III-C) CORRELATING.C

```
/************************************/
/*	Comp Arc Project
/*	By: Simon Foucher
/*	260 223 197
/************************************/


/************************************/
/*	includes and definitions
/************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>




/************************************/
/*	Function declarations
/************************************/


/************************************/
/*	Global Variables & datatypes
/************************************/
#define TRUE 1
#define FALSE 0
typedef int BOOL;

/************************************/
/*	Functions
/************************************/

// To clean up gathered strings from text
void stripCRLF(char array[])
{
   int i;

   for(i=0;i<500 && array[i]!='\0';i++)
     if (array[i]=='\r' || array[i]=='\n')
			array[i]='\0';
}


// To compute power
int power(int x, int n){

	if(n == 0)
```

```
                return 1;

        else{
                int tmp = power(x, n/2);
                if(n%2)
                        return tmp*tmp*x;
                else
                        return tmp*tmp;
        }

}



void readTrace(FILE *fp, int Nbits, int tooBig)
{

        char addresses[1000][20];
        // 8 bits global predictor can represent 0d numbers ranging from [0,255]
        int predictorGlobal;

        // 8 198 x 255 is the largest array that can be declared in a 32 bits Linux machine without getting a
segmentation fault.
        int predictorLocal[8100][256];
        // Tha largest file is SERV which has 10 388 different addresses

        int totalBR = 0, totalGoodPredictions = 0;
        int Nmask, treshold;
        char temp[50];
        char tmpAddress[11];
        int i, j, k;



//***********************************                                    INITIALIZATION
**********************************************

        // Format arrays with preditction not taken
        predictorGlobal = 0;
        for(i = 0; i < 8100; i++){
                for(k = 0; k < 256; k++)
                                predictorLocal[i][k] = 0;
        }

        // Build bit mask for N bit predictor
        Nmask = power(2, Nbits);
        treshold = (Nmask)/2;




// DEBUGGING
// printf("N = %d\t2^N = %d\tMask: 0x%x\n", Nbits, Nmask+1,Nmask);
```

```
//***********************************                    DATA                    FETCH
**********************************************
		// Extract lines of data; break at EOF
		while( fgets(temp, sizeof(temp), fp) != NULL ){

			stripCRLF(temp);                                        // Clean up the
string
//printf("Line: %s", temp);
			totalBR++;
		// Increment total branch count


			if(tooBig)
		// If the address space is too big, skip the first digit
				k = 1;
			else
				k = 0;
		// Otherwise copy the while string

			for(i = 0; ; i++, k++){                                          //
Extract address of branch
				tmpAddress[i] = temp[k];
				if(tmpAddress[i] == ' ')
					break;
			}

			tmpAddress[i++] = NULL;                               // Add a proper
null terminator and increment index (i points at branch result)

			if(tooBig)
				i++;

			for(k = 0; k < 1000; k++)                        // Find out if the entry
already exists
				if(addresses[k][0] == NULL || !strncmp(addresses[k], tmpAddress, 10))
					break;

			if(k != 8099){                                                // If it is
a new entry, k points at the first empt spot
				if(addresses[k][0] == NULL)
				strcpy(addresses[k], tmpAddress);
			}
			else
		// Warning if we busr out of memory
				printf("Warning! Reached end of buffer!!!!\n");
```

```
//************************************                                                    PREDICTOR
***********************************************
			// At this point, i points at the branch result and k at the array index whete the element goes
			//  NOTES:
			//		ASCII 48 = 0, 49 = 1

//printf("pred Was: %x\t", predictorLocal[k]);

			// If predict "NOT taken
			// So predict Local is < 0001 0000 equivalent to Renge of [0000 0000, 0000 1111]
			if(predictorLocal[k][predictorGlobal] < treshold){
				// If branch IS taken
				if(temp[i] == 49){
					// Update # of good predictions
					totalGoodPredictions += 0;
			// Bad prediction

					// Update Local predictor & jump if reach treshold
					predictorLocal[k][predictorGlobal] +=1;
		// Increment the predictor
					if(predictorLocal[k][predictorGlobal] == treshold)			//	Hit	turning
point (0001 0000), jump to "strong taken"(1111 1111)
						predictorLocal[k][predictorGlobal] = Nmask - 1;			//
Nmask = 1 0000 0000 so Nmask - 1 = 1111 1111 = strong taken

					// Update Global predictor if not at its max value
					if(predictorGlobal != 255)
						predictorGlobal++;
				}

				// If branch NOT taken
				if(temp[i] == 48){
					// Update # of good predictions
					totalGoodPredictions += 1;
			// Good prediction

					// Update Local predictor if we are not at edge value
					if(predictorLocal[k][predictorGlobal] != 0)					//	If
predictor != 0000 0000
						predictorLocal[k][predictorGlobal] -= 1;				//
Reflect last BR decision ourcome (LSB of 0)
					}

					// Update Global predictor if not at its max value
					if(predictorGlobal != 255)
						predictorGlobal++;

			}


			// If predict "IS taken"
			// If predictorLocal is in the range [0001 0000, 1111 1111]
			if(predictorLocal[k][predictorGlobal] >= treshold){
```

```
                    // If branch IS taken
                    if(temp[i] == 49){
                        // Update # of good predictions
                        totalGoodPredictions += 1;
                // Good prediction

                        // Update Local predictor if we are not at edge value
                        if(predictorLocal[k][predictorGlobal] != Nmask - 1)             // If predictor < 1
0000 0000 = Range og [0001 0000, 1111 1111]
                            predictorLocal[k][predictorGlobal] += 1;                        //
increment by 1

                        // Update Global predictor if not at its min value
                        if(predictorGlobal != 0)
                            predictorGlobal--;
                    }

                    // If branch NOT taken
                    if(temp[i] == 48){
                        // Update # of good predictions
                        totalGoodPredictions += 0;
                // Bad prediction
                        predictorLocal[k][predictorGlobal] -= 1;                            //
Reflect last BR decision ourcome (LSB of 0)

                        // Update Local predictor & jump if reach treshold
                        if(predictorLocal[k][predictorGlobal] == treshold - 1)  // If we got to 0000 1111
                            predictorLocal[k][predictorGlobal] = 0;
            // Jump to predict strong not taken 0000 0000

                        // Update Global predictor if not at its min value
                        if(predictorGlobal != 0)
                            predictorGlobal--;
                    }
                }

//printf("pred New: %x\t", predictorLocal[k]);
                if(predictorLocal[k][predictorGlobal] == Nmask)                              //  Avoid
overflow
                    predictorLocal[k][predictorGlobal]--;
                if(predictorGlobal == 256)
                    predictorGlobal--;
//printf("pred Masked: %x\n", predictorLocal[k]);


            }// End of while getting lines

            printf("%d\t%d\t%d\t\t%d\n", Nbits, totalGoodPredictions, totalBR, totalGoodPredictions*100/totalBR);



}// End if readTrace
```

```
/*****************************************************************************/
/*          Function  main (int argc, char *argv[])
/*****************************************************************************/

int main (int argc, char *argv[])
{
          FILE *fp;
          int Nbits;



          printf("---------------------------------------------\n");
          // Open FP.trace
          if((fp = fopen("traces/FP.trace","r"))==NULL)                     //Open the password file
                    {
                              puts("Cannot open FP.trace\n"
                                        "Please make sure it is located in a directory called 'traces/'\n"
                                        "having this program's directory as a root\n");
                    }
          else
                    printf("FP.trace Analysis\n");


          printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
          for(Nbits = 2; Nbits <=20; Nbits +=2){
                    rewind(fp);
                    // Rewind file pointer
                    readTrace(fp, Nbits, 0);
          }

          fclose(fp);
          printf("\n---------------------------------------------\n");


          // Open INT.trace
          if((fp = fopen("traces/INT.trace","r"))==NULL)                    //Open the password file
                    {
                              puts("Cannot open INT.trace\n"
                                        "Please make sure it is located in a directory called 'traces/'\n"
                                        "having this program's directory as a root\n");
                    }
          else
                    printf("INT.trace Analysis\n");

          printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
          for(Nbits = 2; Nbits <=20; Nbits +=2){
                    rewind(fp);
                    // Rewind file pointer
                    readTrace(fp, Nbits, 0);
          }
```

```
fclose(fp);
printf("\n--------------------------------------------\n");



// Open MM.trace
if((fp = fopen("traces/MM.trace","r"))==NULL)                     //Open the password file
        {
                puts("Cannot open MM.trace\n"
                        "Please make sure it is located in a directory called 'traces/'\n"
                        "having this program's directory as a root\n");
        }
else
        printf("MM.trace Analysis\n");

printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
for(Nbits = 2; Nbits <=20; Nbits += 2){
        rewind(fp);
        // Rewind file pointer
        readTrace(fp, Nbits, 0);
}

fclose(fp);
printf("\n--------------------------------------------\n");



// Open SERV.trace
if((fp = fopen("traces/SERV.trace","r"))==NULL)                   //Open the password file
        {
                puts("Cannot open SERV.trace\n"
                        "Please make sure it is located in a directory called 'traces/'\n"
                        "having this program's directory as a root\n");
        }
else
        printf("SERV.trace Analysis\n");

printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
for(Nbits = 2; Nbits <=20; Nbits +=2){
        rewind(fp);
        // Rewind file pointer
        readTrace(fp, Nbits, 1);
}

fclose(fp);
printf("\n--------------------------------------------\n");
```

}

III-D) Tournament.c

```
/************************************/
/*          Comp Arc Project
/*          By: Simon Foucher
/*          260 223 197
/************************************/


/************************************/
/*          includes and definitions
/************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>





/************************************/
/*          Function declarations
/************************************/


/************************************/
/*          Global Variables & datatypes
/************************************/
#define TRUE 1
#define FALSE 0
typedef int BOOL;

/************************************/
/*          Functions
/************************************/

// To clean up gathered strings from text
void stripCRLF(char array[])
{
  int i;

  for(i=0;i<500 && array[i]!='\0';i++)
   if (array[i]=='\r' || array[i]=='\n')
                array[i]='\0';
}


// To compute power
int power(int x, int n){

        if(n == 0)
                return 1;
```

```
	else{
		int tmp = power(x, n/2);
		if(n%2)
			return tmp*tmp*x;
		else
			return tmp*tmp;
	}

}



void readTrace(FILE *fp, int Nbits)
{
	// Data gathering variables & common
	char addresses[8000][20];
	char temp[50];
	char tmpAddress[10];
	int i, j, k;

	// Buffer (n bit) Predictor data
	int BufferPredictor[8000];
	int NmaskBuffer, tresholdBuffer;
	int bufferPrediction;

	// Correlating Predictor data
	int predictorGlobal;
	int correlatingPredictor[1000][255];
	int NmaskCorrelator, tresholdCorrelator;
	int correlatorPrediction;

	// Chooser table
	int chooser[8000];

	// Statistical variables
	int totalBR = 0, totalGoodPredictions = 0;




/***********************************                                    INITIALIZATION
**********************************************/

	// Format arrays with preditction not taken
	predictorGlobal = 0;
	for(i = 0; i < 8000; i++){
		BufferPredictor[i] = 0;
	// Buffer predictor
		chooser[i] = 0;
	// Chooser
		for(k = 0; k < 256; k++)
```

```
					correlatingPredictor[i][k] = 0;					//
Correlating predictor
		}

		// Build bit mask for predictors
		NmaskBuffer = power(2, Nbits);
		tresholdBuffer = NmaskBuffer/2;

		NmaskCorrelator = power(2, Nbits);
		tresholdCorrelator = (NmaskCorrelator)/2;




/***********************************			DATA			FETCH
***********************************************/
		// Extract lines of data; break at EOF
		while( fgets(temp, sizeof(temp), fp) != NULL ){

			stripCRLF(temp);						// Clean up the
string
//printf("Line: %s", temp);
			totalBR++;
		// Increment total branch count


			for(i = 0; ; i++){						//	Extract
address of branch
				tmpAddress[i] = temp[i];
				if(tmpAddress[i] == ' ')
					break;
			}

			tmpAddress[i++] = NULL;					// Add a proper
null terminator and increment index (i points at branch result)

			//for(;i<15; i++)
			//	printf("--temp[%d] = %d\n",i, temp[i]);



			for(k = 0; k < 8000; k++)					// Find out if the entry
already exists
				if(addresses[k][0] == NULL || !strncmp(addresses[k], tmpAddress, 9))
					break;

			if(k != 7999){							// If it is
a new entry, k points at the first empt spot
				if(addresses[k][0] == NULL)
				strcpy(addresses[k], tmpAddress);
			}
			else
```

```
                              printf("Warning! Reached end of buffer!!!!\n");



/***********************************                    BUFFER                    PREDICTOR
**********************************************/

                    // If predict "NOT taken
                    // So predict Local is < 0001 0000 equivalent to Renge of [0000 0000, 0000 1111]
                    if(BufferPredictor[k] < tresholdBuffer){
                              bufferPrediction = 0;
          // Save result of the prediction
                              // If branch IS taken
                              if(temp[i] == 49){
                                        BufferPredictor[k] +=1;
          // Increment the predictor
                                        if(BufferPredictor[k] == tresholdBuffer)                    // Hit the turning
point (0001 0000), jump to "strong taken"(1111 1111)
                                                  BufferPredictor[k] = NmaskBuffer - 1;                    //
NmaskBuffer = 1 0000 0000 so NmaskBuffer - 1 = 1111 1111 = strong taken
                              }
                              // If branch NOT taken
                              if(temp[i] == 48){
                                        if(BufferPredictor[k] != 0)
          // If predictor != 0000 0000
                                                  BufferPredictor[k] -= 1;
          // Reflect last BR decision ourcome (LSB of 0)
                                        }
                    }


                    // If predict "IS taken"
                    // If BufferPredictor is in the range [0001 0000, 1111 1111]
                    if(BufferPredictor[k] >= tresholdBuffer){
                              bufferPrediction = 1;
          // Save result of the prediction

                              // If branch IS taken
                              if(temp[i] == 49){
                                        if(BufferPredictor[k] != NmaskBuffer - 1)                    // If predictor < 1
0000 0000 = Range og [0001 0000, 1111 1111]
                                                  BufferPredictor[k] += 1;
          // increment by 1
                              }
                              // If branch NOT taken
                              if(temp[i] == 48){
                                        BufferPredictor[k] -= 1;
          // Reflect last BR decision ourcome (LSB of 0)

                                        if(BufferPredictor[k] == tresholdBuffer - 1)          // If we got to 0000 1111
                                                  BufferPredictor[k] = 0;
          // Jump to predict strong not taken 0000 0000
```

```
			}
		}


		if(BufferPredictor[k] == NmaskBuffer)					//  Avoid
overflow (I don't think this is possible, but just in case...)
			BufferPredictor[k]--;





//***********************************		CORRELATING		PREDICTOR
*********************************************

		// If predict "NOT taken
		// So predict Local is < 0001 0000 equivalent to Renge of [0000 0000, 0000 1111]
		if(correlatingPredictor[k][predictorGlobal] < tresholdCorrelator){
			correlatorPrediction = 0;
	// Save result
			// If branch IS taken
			if(temp[i] == 49){
				// Update Local predictor & jump if reach tresholdCorrelator
				correlatingPredictor[k][predictorGlobal] +=1;
			// Increment the predictor
				if(correlatingPredictor[k][predictorGlobal] == tresholdCorrelator)
	// Hit turning point (0001 0000), jump to "strong taken"(1111 1111)
					correlatingPredictor[k][predictorGlobal] = NmaskCorrelator - 1;
	// NmaskCorrelator = 1 0000 0000 so NmaskCorrelator - 1 = 1111 1111 = strong taken

				// Update Global predictor if not at its max value
				if(predictorGlobal != 255)
					predictorGlobal++;
			}

			// If branch NOT taken
			if(temp[i] == 48){
				// Update Local predictor if we are not at edge value
				if(correlatingPredictor[k][predictorGlobal] != 0)
	// If predictor != 0000 0000
					correlatingPredictor[k][predictorGlobal] -= 1;
	// Reflect last BR decision ourcome (LSB of 0)
				}

				// Update Global predictor if not at its max value
				if(predictorGlobal != 255)
					predictorGlobal++;
		}


		// If predict "IS taken"
		// If correlatingPredictor is in the range [0001 0000, 1111 1111]
```

```
			if(correlatingPredictor[k][predictorGlobal] >= tresholdCorrelator){
				correlatorPrediction = 1;
			// Save result
				// If branch IS taken
				if(temp[i] == 49){
					// Update Local predictor if we are not at edge value
					if(correlatingPredictor[k][predictorGlobal] != NmaskCorrelator - 1)
		// If predictor < 1 0000 0000 = Range og [0001 0000, 1111 1111]
						correlatingPredictor[k][predictorGlobal] += 1;
		// increment by 1

					// Update Global predictor if not at its min value
					if(predictorGlobal != 0)
						predictorGlobal--;
				}

				// If branch NOT taken
				if(temp[i] == 48){
					correlatingPredictor[k][predictorGlobal] -= 1;
		// Reflect last BR decision ourcome (LSB of 0)

					// Update Local predictor & jump if reach tresholdCorrelator
					if(correlatingPredictor[k][predictorGlobal] == tresholdCorrelator - 1)	// If we
got to 0000 1111
						correlatingPredictor[k][predictorGlobal] = 0;
		// Jump to predict strong not taken 0000 0000

					// Update Global predictor if not at its min value
					if(predictorGlobal != 0)
						predictorGlobal--;
				}
			}


			if(correlatingPredictor[k][predictorGlobal] == NmaskCorrelator)
		// Avoid overflow
				correlatingPredictor[k][predictorGlobal]--;
			if(predictorGlobal == 256)
				predictorGlobal--;



//**************************************	CHOOSER
****************************************************
// Here, 'bufferPrediction' and 'correlatorPrediction' contain the predict results of both algorithms
// The branch result is evaluating a second time (redundent, but we are not aiming for performance)
// The chooser are 2 bit numbers, ranginf from 00 = 0 to 11 = 3. Their treshold is 2.
// Chooser 0,1 prefers Buffer
// Chooser 2,3 prefers Correlator

		switch(chooser[k]) {
			// case 0, 1 select Buffer predictor
			case 0:
```

```
				if(temp[i] == 49){					// If branch IS taken
					if(bufferPrediction == 1)		// Predict IS taken
						totalGoodPredictions++;		// Good prediction = update good
prediction counter
					if(bufferPrediction == 0)		// Predict NOT taken
						chooser[k] = 1;				//	Bad	prediction	=
Update chooser
				}
				if(temp[i] == 48){					// If branch NOT taken
					if(bufferPrediction == 1)		// Predict IS taken
						chooser[k] = 1;				//	Bad	prediction	=
Update chooser
					if(bufferPrediction == 0)		// Predict NOT taken
						totalGoodPredictions++;		// Good prediction = update good
prediction counter
				}
				break;

			case 1:
				if(temp[i] == 49){					// If branch IS taken
					if(bufferPrediction == 1){		// Predict IS taken
						totalGoodPredictions++;		// Good prediction = update good
prediction counter
						chooser[k] = 0;				//	Good	prediction	=
update chooser to stable
					}
					if(bufferPrediction == 0)		// Predict NOT taken
						chooser[k] = 3;				//	Bad	prediction	=
Update chooser
				}
				if(temp[i] == 48){					// If branch NOT taken
					if(bufferPrediction == 1)		// Predict IS taken
						chooser[k] = 3;				//	Bad	prediction	=
Update chooser
					if(bufferPrediction == 0){		// Predict NOT taken
						totalGoodPredictions++;		// Good prediction = update good
prediction counter
						chooser[k] = 0;				//	Good	prediction	=
update chooser to stable
					}
				}
				break;

			// case 2,3 select correlating predictor
			case 2:
				if(temp[i] == 49){					// If branch IS taken
					if(correlatorPrediction == 1){	// Predict IS taken
						totalGoodPredictions++;		// Good prediction = update good
prediction counter
						chooser[k] = 3;				//	Good	prediction	=
update chooser to stable
					}
					if(correlatorPrediction == 0)	// Predict NOT taken
```

```
                                        chooser[k] = 0;                         //     Bad     prediction     =
Update chooser
                        }
                        if(temp[i] == 48){                              // If branch NOT taken
                                if(correlatorPrediction == 1)           // Predict IS taken
                                        chooser[k] = 0;                         //     Bad     prediction     =
Update chooser
                                if(correlatorPrediction == 0){          // Predict NOT taken
                                        totalGoodPredictions++;         // Good prediction = update good
prediction counter
                                        chooser[k] = 3;                         //     Good     prediction     =
update chooser to stable
                                }
                        }
                        break;


                case 3:
                        if(temp[i] == 49){                              // If branch IS taken
                                if(correlatorPrediction == 1){          // Predict IS taken
                                        totalGoodPredictions++;         // Good prediction = update good
prediction counter
                                        chooser[k] = 3;                         //     Good     prediction     =
update chooser to stable
                                }
                                if(correlatorPrediction == 0)           // Predict NOT taken
                                        chooser[k] = 2;                         //     Bad     prediction     =
Update chooser
                        }
                        if(temp[i] == 48){                              // If branch NOT taken
                                if(correlatorPrediction == 1)           // Predict IS taken
                                        chooser[k] = 2;                         //     Bad     prediction     =
Update chooser
                                if(correlatorPrediction == 0){          // Predict NOT taken
                                        totalGoodPredictions++;         // Good prediction = update good
prediction counter
                                        chooser[k] = 3;                         //     Good     prediction     =
update chooser to stable
                                }
                        }
                        break;
                default:
                        break;
        }// End of switch chooser

        }// End of while getting lines


        printf("%d\t%d\t%d\t\t%d\n", Nbits, totalGoodPredictions, totalBR, totalGoodPredictions*100/totalBR);
```

```
}// End if readTrace



/**************************************************************************/
/*	Function  main (int argc, char *argv[])
/**************************************************************************/

int main (int argc, char *argv[])
{
	FILE *fp;
	int Nbits;


	printf("---------------------------------------------\n");
	// Open FP.trace
	if((fp = fopen("traces/FP.trace","r"))==NULL)			//Open the password file
		{
			puts("Cannot open FP.trace\n"
				"Please make sure it is located in a directory called 'traces/'\n"
				"having this program's directory as a root\n");
		}
	else
		printf("FP.trace Analysis\n");

	printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
	for(Nbits = 2; Nbits <=20; Nbits +=2){
		rewind(fp);
		// Rewind file pointer
		readTrace(fp, Nbits);
	}

	fclose(fp);
	printf("\n---------------------------------------------\n");


	// Open INT.trace
	if((fp = fopen("traces/INT.trace","r"))==NULL)			//Open the password file
		{
			puts("Cannot open INT.trace\n"
				"Please make sure it is located in a directory called 'traces/'\n"
				"having this program's directory as a root\n");
		}
	else
		printf("INT.trace Analysis\n");

	printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
	for(Nbits = 2; Nbits <=20; Nbits +=2){
```

```
		rewind(fp);
		// Rewind file pointer
		readTrace(fp, Nbits);
	}

	fclose(fp);
	printf("\n--------------------------------------------\n");



	// Open MM.trace
	if((fp = fopen("traces/MM.trace","r"))==NULL)				//Open the password file
		{
			puts("Cannot open MM.trace\n"
				"Please make sure it is located in a directory called 'traces/'\n"
				"having this program's directory as a root\n");
		}
	else
		printf("MM.trace Analysis\n");

	printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
	for(Nbits = 2; Nbits <=20; Nbits +=2){
		rewind(fp);
		// Rewind file pointer
		readTrace(fp, Nbits);
	}

	fclose(fp);
	printf("\n--------------------------------------------\n");



	// Open SERV.trace
	if((fp = fopen("traces/SERV.trace","r"))==NULL)				//Open the password file
		{
			puts("Cannot open SERV.trace\n"
				"Please make sure it is located in a directory called 'traces/'\n"
				"having this program's directory as a root\n");
		}
	else
		printf("SERV.trace Analysis\n");

	printf("Nbits\tGoodPr\tTotBR\t\tAccuracy(%)\n");
	for(Nbits = 2; Nbits <=20; Nbits +=2){
		rewind(fp);
		// Rewind file pointer
		readTrace(fp, Nbits);
	}

	fclose(fp);
	printf("\n--------------------------------------------\n");
```

```
}
```