Fall 2021-EEL 5764 Computer Architecture

Performance Evaluation of Branch Prediction Strategies using gem5

Meghana Koduru, Vidhya Hari, Vijayasai Jalagam
megahanakoduru@ufl.edu; vidhya.hari@ufl.edu; v.jalagam@ufl.edu
Department of Electrical and Computer Engineering
University of Florida

INTRODUCTION

Branch prediction is a technique utilized by modern processors to guess which direction a branch will take before knowing it definitively. The purpose of technique is to improve performance by increasing the flow in the processor pipeline. In this project, we have assessed the working of select branch predictors in the x86 ISA using the gem5 simulator. We have utilized a set of programs as well as SPEC benchmarks for our analysis. We have also demonstrated how the concept of predication works in comparison to branch prediction.

We have utilized the following three predictors which are available in gem5:

2-bit local predictor: This is a simple two-bit saturating counter. It stores the last two results of the current branch to predict how the branch will move subsequently. This is an effective method if there is no or very minimum correlation among different branches in a program.

Bi-Mode predictor: This predictor uses the history of branches that are surrounding the current branch and hence follows a global prediction scheme. The branch address and branch history are XORed to determine which entries are most likely to be taken and which not taken. This is updated in the two halves of the Pattern History Table (PHT). Finally, a choice predictor chooses which path should be taken.

Tournament predictor: This combines local and global branch histories indexed by the branch address and branch history respectively. Current branch address is used to index the choice predictor which determines the predicted path.

For the purpose of this project, we have added the two custom parameters shown in Fig 1 to the gem5 statistics file (stats.txt) by specifying these parameters in the source code.

BTBMisses = BTBLookups – BTBHits

BTBMissRatio = BTBMisses/BTBLookups

```
226 system.cpu.branchPred.lookups                 5123       # Number of BP lookups (Count)
227 system.cpu.branchPred.condPredicted           3583       # Number of conditional branches predicted (Count)
228 system.cpu.branchPred.condIncorrect            832       # Number of conditional branches incorrect (Count)
229 system.cpu.branchPred.BTBLookups              2174       # Number of BTB lookups (Count)
230 system.cpu.branchPred.BTBHits                  1261       # Number of BTB hits (Count)
231 system.cpu.branchPred.BTBHitRatio          0.580037       # BTB Hit Ratio (Ratio)
232 system.cpu.branchPred.BTBMisses                913       # BTB Misses (Count)
233 system.cpu.branchPred.BTBMissRatio          0.419963       # BTB Miss Ratio (Ratio)
234 system.cpu.branchPred.RASUsed                  379       # Number of times the RAS was used to get a target. (Count)
235 system.cpu.branchPred.RASIncorrect              0       # Number of incorrect RAS predictions. (Count)
236 system.cpu.branchPred.indirectLookups         396       # Number of indirect predictor lookups. (Count)
237 system.cpu.branchPred.indirectHits             41       # Number of indirect target hits. (Count)
238 system.cpu.branchPred.indirectMisses          355       # Number of indirect misses. (Count)
239 system.cpu.branchPred.indirectMispredicted      115         # Number of mispredicted indirect branches. (Count)
```

Fig 1. Custom Parameters in stats.txt file

We have studied the effect of varying the Branch Target Buffer (BTB) Entries, predictor size and bits per counter on two parameters defined below:

BTB Miss% = [(BTB Lookups – BTB hits)/BTB Lookups] * 100

Conditional branch misprediction % = (Conditional branches predicted incorrectly/Conditional branches predicted) * 100

ANALYSIS OF 2-BIT LOCAL, BIMODAL AND TOURNAMENT PREDICTORS

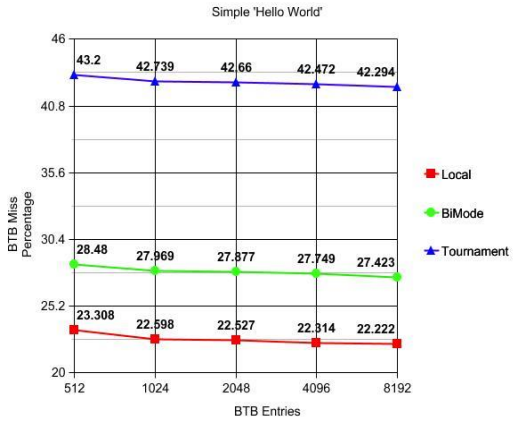
I. Analysis using a set of programs

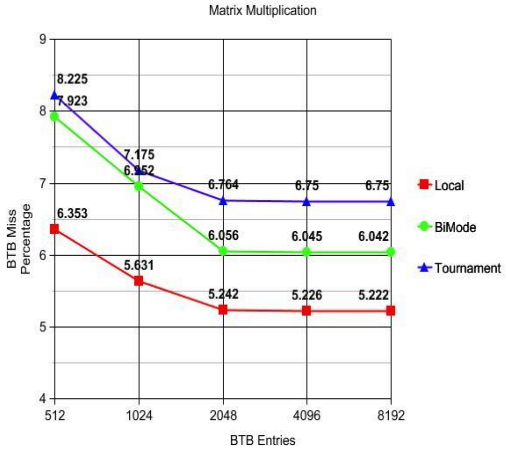
This has been studied using the out of order O3CPU and the TimingSimpleCPU. We have varied the BTB Entries to determine its effect on BTB miss percentage and conditional branch misprediction percentage. This has been tested using four simple programs – a no branch program with only a print statement, matrix multiplication, quick sort and a binary search. The matrix multiplication is produces a 12x12 matrix as the product. For quick sort, we have given an array of six elements as input and the last element is chosen as pivot. For binary search, input is a sorted array of nineteen elements. All programs have been written in the C language.

Effect of BTB Entries on BTB Miss % and Conditional Branch Misprediction %

Default Values

Parameter	Value
BTB Tag Size	16
RAS Size	16
Local predictor size (for 2-bit local and tournament predictors)	2048
Global predictor size (for BiMode and tournament predictors)	8192
Bits per counter (local and global)	2





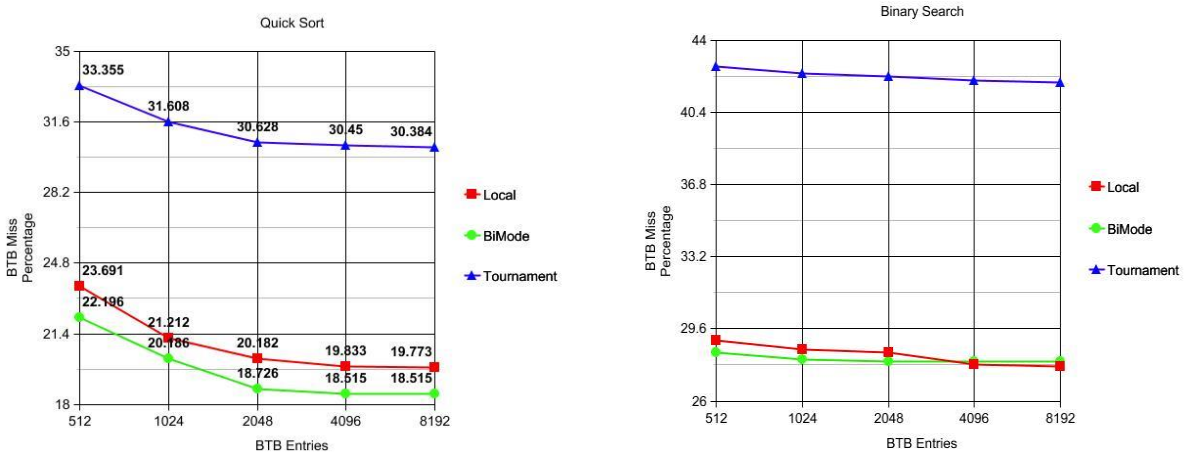


Fig 2. BTB Entries and Predictor Type vs. BTB Miss % for O3CPU

The BTB miss percentage gradually decreases as the BTB entries are swept from 512 to 8192. For all four programs, tournament predictor shows higher percentage of BTB misses compared to local and bimode predictors. The change in miss percentage is seen predominantly for matrix multiplication and quick sort while the other two programs show almost constant miss percentage irrespective of BTB entries. The BTB entries effect on miss percentage has converged around 2048 entries for all programs and higher entries do not have much effect.

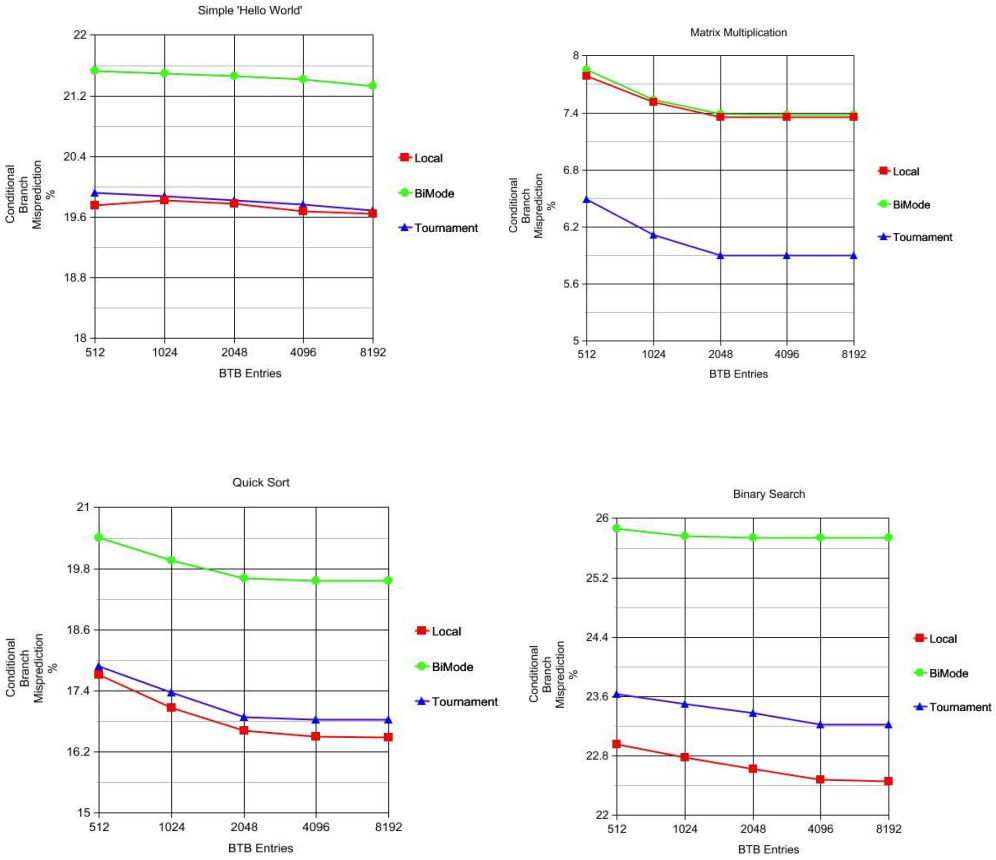


Fig 3. BTB Entries and Predictor Type vs. Conditional Branch Misprediction % for O3CPU

The conditional branch misprediction percentage decreases as the BTB entries are increased. Bimode predictor shows highest number of mispredictions for all four programs compared to the other two predictors. Similar to BTB miss percentage, the effect of BTB entries on misprediction percentage is seen clearly in matrix multiplication and quick sort while the other two do not show drastic variation. Also, around 2048 BTB entries, the misprediction percentage starts to become relatively constant.

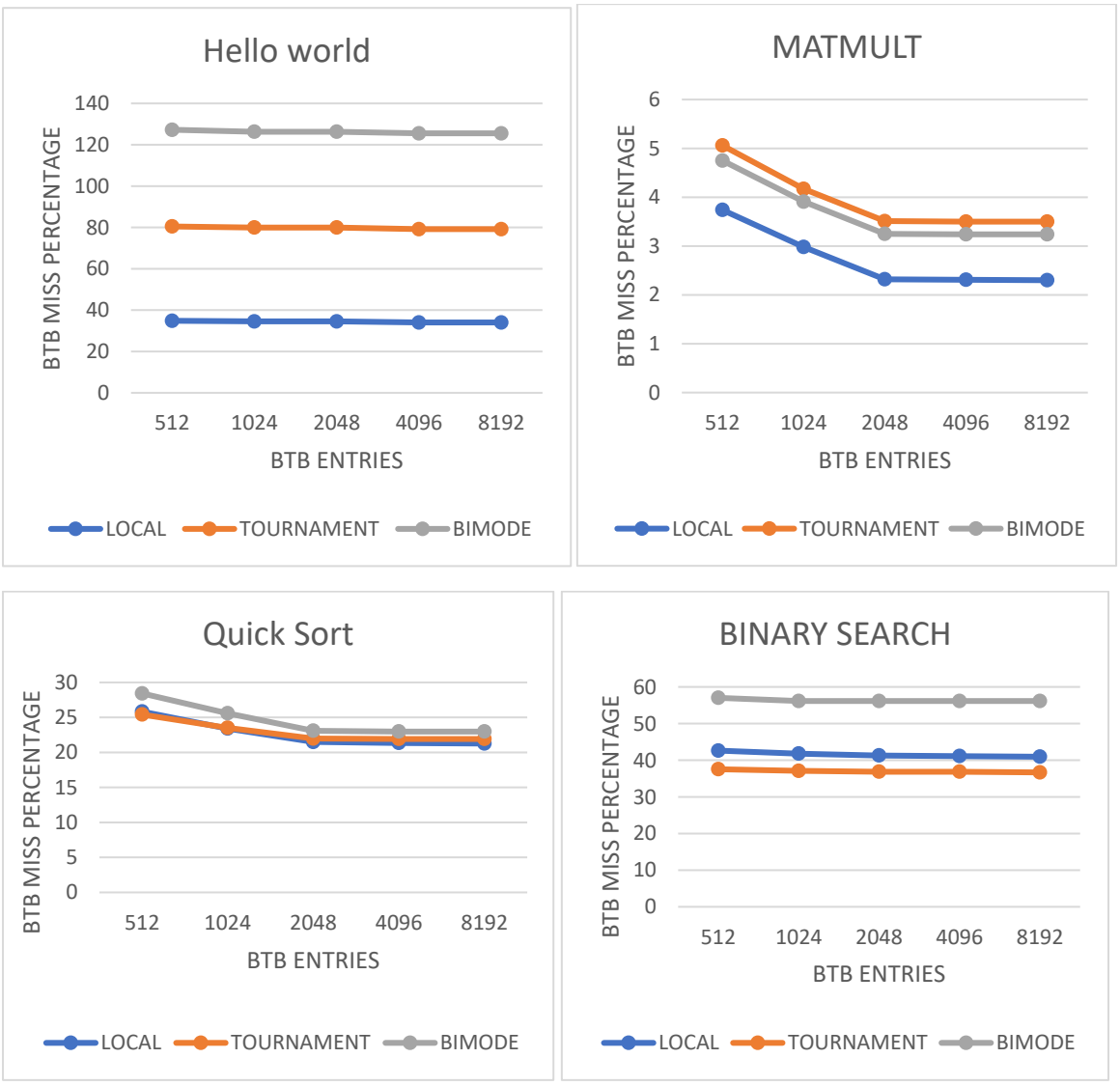


Fig 4. BTB Entries and Predictor Type vs. BTB Miss % for TimingSimpleCPU

BTB miss percentage decreases till 2048 entries for matrix multiplication and quick sort and then saturates. Other two programs have almost constant values for all entries. Bimode predictor has higher BTB miss percentage compared to the other predictors.

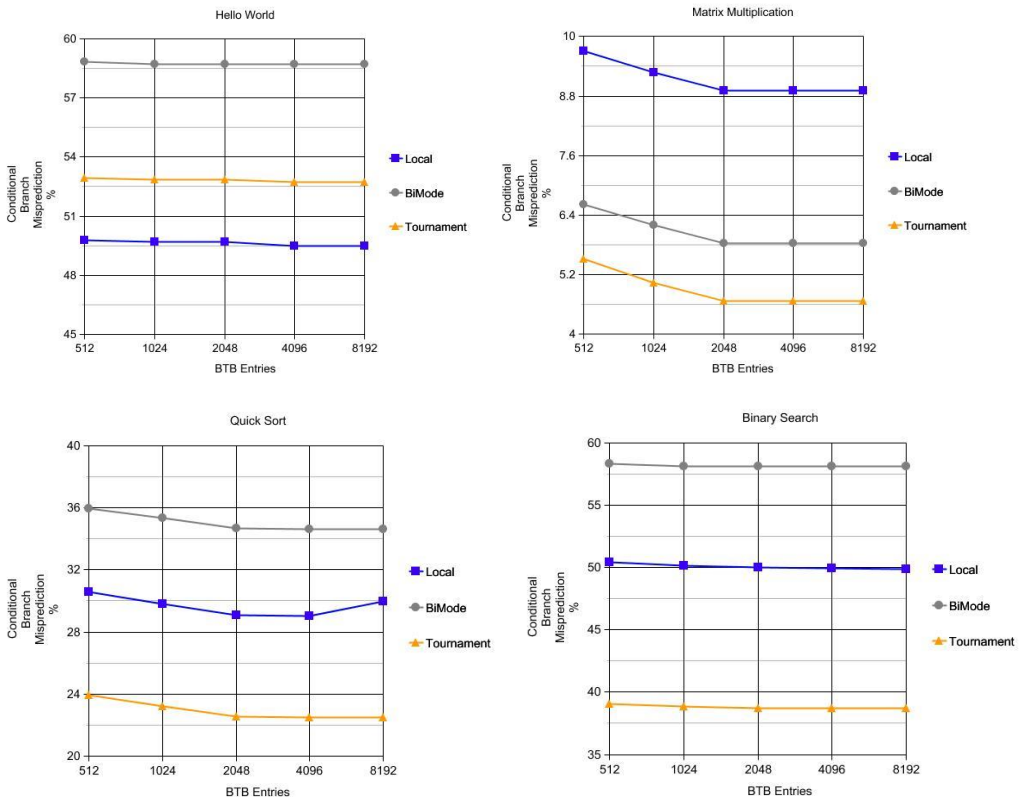


Fig 5. BTB Entries and Predictor Type vs. Conditional Branch Misprediction % for TimingSimpleCPU

Misprediction percentage tends to saturate around 2048 entries. Variation in misprediction is seen for matrix multiplication and quick sort while the other two programs have near constant values for all BTB entries. Out of the three predictors, tournament predictor exhibits lowest mispredicted branch % for matmult, search and sort programs.

II. Analysis using SPEC 2006 benchmarks

This has been studied in the O3CPU. We have utilized three SPEC 2006 benchmarks - 429.mcf, 401.bzip2 and 458.sjeng from https://github.com/timberjack/Project1_SPEC. 429.mcf is a combinatorial optimization algorithm which is written in C and uses almost exclusively integer arithmetic. 401.bzip2 is a compression algorithm written in ANSI C. 458.sjeng is based on artificial intelligence and is used for game tree searching and pattern recognition. It is also written in ANSI C. We have varied the predictor size and the bits per counter and studied the effect on the BTB miss and conditional branch misprediction percentage for each of the predictors.

Effect of predictor size on BTB Miss % and Conditional Branch Misprediction %

Default Values

Parameter	Value
BTB Entries	4096
BTB Tag Size	16

RAS Size	16
Bits per counter (local and global)	2
Instruction Count	1000000

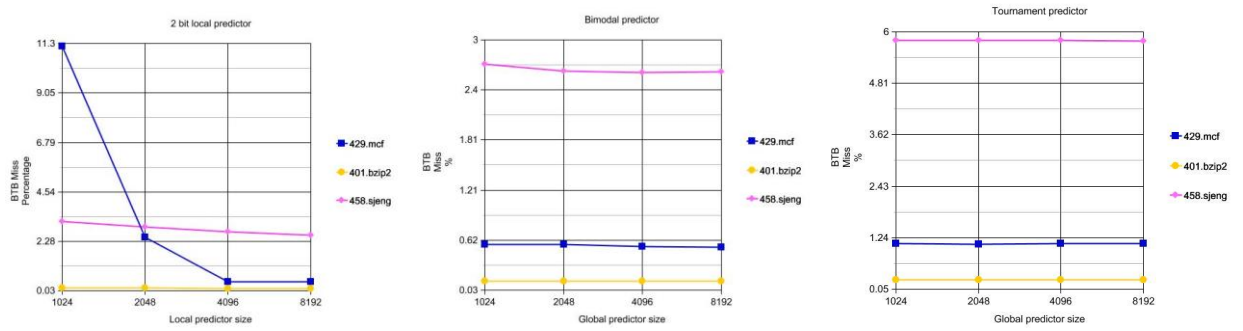


Fig 6. Predictor size and benchmark vs. BTB Miss %

For 429.mcf, miss percentage shows drastic variation between predictor size of 1024 and 2048 while using local predictor. However, it remains almost constant for the other two irrespective of predictor size. For 401.bzip2, miss percentage is least and nearly constant compared to the other benchmarks and is the case for with all predictors. For 458.sjeng, not much variation in miss percentage is observed for different predictor sizes and it shows nearly similar values for all predictors.

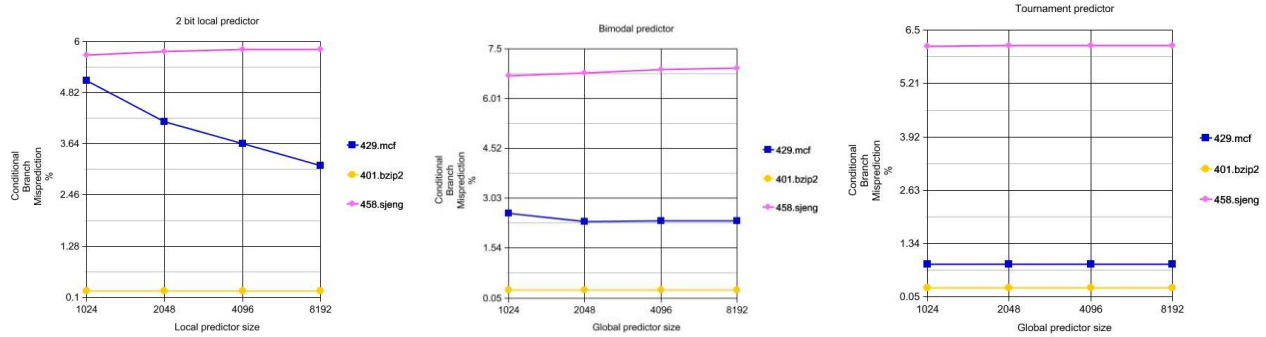


Fig 7. Predictor size and benchmark vs. Conditional branch misprediction %

For 429.mcf, misprediction percentage is least while using tournament predictor and does not change with size. The percentage is highest for local predictor and decreases gradually as the local predictor size is increased. It does not show much variation with bimode predictor size. Misprediction percentage is least for 401.bzip2 and it remains near constant for all predictors irrespective of predictor size. 458.sjeng benchmark shows slight increase in misprediction for local and bimode predictors between sizes of 2048 and 8192. It is, however, almost constant for tournament predictor. Among the three benchmarks, sjeng shows highest misprediction percentage for any predictor.

Effect of bits per counter on BTB Miss % and Conditional Branch Misprediction %

Default Values

Parameter	Value
BTB Entries	4096
BTB Tag Size	16
RAS Size	16
Local predictor size (for 2-bit local and tournament predictors)	2048
Global predictor size (for BiMode and tournament predictors)	8192
Instruction Count	1000000

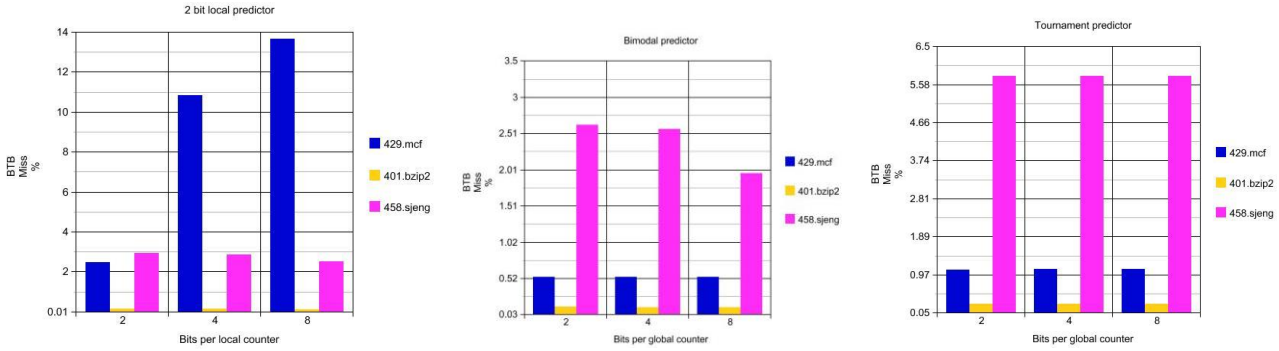


Fig 8. Bits per counter and benchmark vs. BTB Miss %

For mcf, BTB Misses show an increasing trend for local predictor as bits per counter is varied from 2 to 8. It is, however, constant for bimode and tournament predictors. Bzip2 does not show much variation as bits per counter is increased and has the least miss percentage for all predictors. For sjeng, miss percentage is least for local and highest for tournament predictor. In both these cases, the value remains near constant while sweeping across increasing counter bits. For bimode alone, there is a minor decrease in miss percentage for 8 bits per counter.

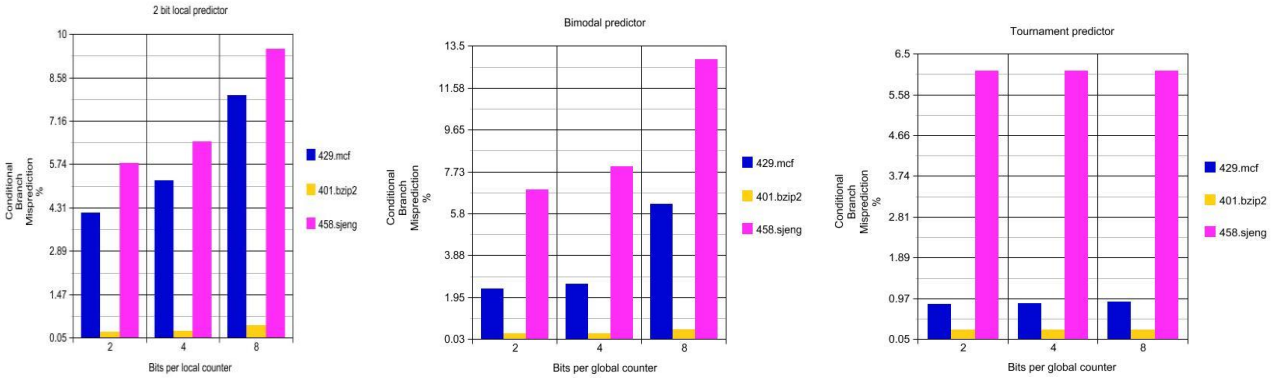


Fig 9. Bits per counter and benchmark vs. Conditional branch misprediction %

For mcf, misprediction gradually increases as bits per counter is increased from 2 to 8 for local and bimode predictors. The value is highest for local and least for tournament predictor. Sjeng also shows a similar pattern wherein misprediction percentage grows with bits per counter for local and bimode

predictors. The misprediction is highest for bimode and least for tournament predictors. Bzip2 has the least misprediction among the three benchmarks and its value remains almost constant over varying bits per counter for all three predictors. Overall, tournament predictor makes a good option for all three benchmarks as the misprediction percentage is relatively low compared to the other predictors and it remains at least constant over different number of bits per counter unlike the case for local and bimode predictors where misprediction increases as bits per counter is increased.

KEY INFERENCES

The following inferences are made with respect to the three predictors and four programs used in this project. In general, BTB miss percentage and conditional branch misoprediction percentage decrease as BTB entries are increased and tends to saturate around 2048 BTB entries. For O3CPU, tournament predictor shows very high BTB miss percent compared to 2-bit local and bimode predictors while bimode predictor shows very high conditional branch misprediction percentage compared to 2-bit local and tournament predictors. Using local or bimode will be more effective in minimizing mispredictions for these programs. For TimingSimpleCPU, bimode predictor shows high BTB miss percentage compared to the other predictors while tournament predictor provides realtively lesser misprediction percentage for three on four programs.

The following inferences are made with respect to the three predictors and benchmarks used in this project and are applicable to O3CPU. In general, BTB miss and misprediction percentages do not show much variation with predictor size except in the case of MCF benchmark executed using local predictor where a drastic reduction in values is observed. Similarly, BTB miss percent remains constant as bits per counter are increased except in the case of MCF benchmark executed using local predictor which exhibits a gradual increase in miss percentage. Misprediction percentage tends to increase with increase in bits per counter using local and bimode predictors for all the three benchmarks whereas it remains constant for tournament predictor. Local predictor exhibits decreasing BTB miss percentage and branch misprediction percentage as predictor size is increased. However, the same parameters increase as the bits per counter are increased.

Branch predication

Predication refers to the conditional execution of instructions. It helps to avoid branches by converting a control dependency to data dependency. In essence, it is a technique to replace branch prediction by execution of all possible branch paths in parallel if there is no mutual dependence among them. By eliminating branches, there is no possibility for a branch penalty which improves performance.

Here, we have considered a simple C++ program to find the first even number given an array of inputs. It has been implemented using both branched and predicated approaches. In Fig 11, Even_num is the predicate we have defined for finding an even number. This is passed as one of the parameters to the inbuilt find_if function which gives the first even number from the array.

```
int func(int arr[], int n, int x)
{
     for (int i = 0; i <= n; i++)
        {
           if ((arr[i] % 2) == 0)
                       x--;

                if (x == 0)
                   return arr[i];
        }
```

Fig 10. Branched implementation

```
bool Even_num (int i) {
  return ((i%2)==0);
}
```

```
  vector<int>::iterator it = find_if (num.begin(), num.end(), Even_num);
```

Fig 11. Predicated implementation

CONCLUSION:

Performance of three branch predictors – 2-bit local, bimode and tournament were measured in terms of BTB miss and branch misprediction percentages by varying BTB entries, predictor size and the bits per counter. We have executed this using a set of programs and a bunch of SPEC 2006 benchmarks and recorded our observations. We have also demonstrated branch predication by defining a predicate to evaluate the conditional expression which helps in avoiding potential branches in the program.

The project gave us the opportunity to further explore different functionalities of the gem5 simulator and apply them to study the concept of branch prediction. We thank Dr. Lam for his support and guidance throughout the project work.