COMPUTER ARCHITECTURE

Project #1

Effect of varying Branch Prediction Parameters on Different Benchmarks

Akash Biswal (axb200166)
Mayank Kumar Singhal (mks200001)

Part 1 : Introduction

- **Branch prediction** is a technique used in CPU design that attempts to guess the outcome of a conditional operation (Branches)and prepare for the most likely result. A digital circuit that performs this operation is known as a Branch Predictor.
- **Gem5** is a simulation platform that is used to simulate different computer architecture designs or processor architecture without having to build the actual hardware
- In this project we explore and draw inferences from the effects of different Branch Predictors with varying parameters when tested against 2 benchmarks

Part 1: Types of Branch Predictors (in GEM5)

1. 2bit_local_Predictor

- This predictor changes prediction only on two successive mispredictions.
- Two bits are maintained in the prediction buffer and there are four different states.
- Two states corresponding to a taken state and two corresponding to not taken state.

Drawback : Causes pipeline bubble as it fails to determine the next fetch address when the branch is fetched.

2. Bi_mode Predictor

- Aimed at the elimination of destructive aliasing in global history indexed schemes.

Drawback: It can still suffer from interference between the weakly biased and strongly biased substreams.

3. Tournament Predictor

- Combination of one global and one local predictor.

Part 1: Simulation Setup

- We used UTD server to run the Gem5 simulator as installing natively would require a linux distro
- It is easier to use UTD server as it comes with all the dependencies pre-installed (we copied Gem5 to run on a local directory)
- With the help of ssh we could run all the commands from our machine and verify results with help of NoMachine interface
- After realising the number of different architectures that could be created by changing the Branch predictor type and it's a parameters, an automation script was created in python to automate the whole process

Part 2: Config.ini

```
[system.cpu.branchPred]
type=TournamentBP
BTBEntries=2048
BTBTagSize=16
RASSize=16
choiceCtrBits=2
choicePredictorSize=4096
eventq_index=0
globalCtrBits=2
globalPredictorSize=4096
instShiftAmt=2
localCtrBits=2
localHistoryTableSize=2048
localPredictorSize=1024
numThreads=1
```

- This image shows one of the configurations with the TournamentBP which is a Combined/Hybrid Predictor
- This information is stored in the config.ini file at the respective output location for each simulation
- The parameters used in this configuration were
  - Branch Table Buffer Entries: 2048
  - Local Predictor Size: 1024
  - Global Predictor: 4096
  - Choice Predictor Size: 4096
- The branch predictor type is change by passing TournamentBP() as an argument to the **BaseSimpleCPU.py** file
- The parameters are changed by editing the **BranchPredictor.py** file

Part 3: Changes to the Source Code

- Two new parameters were added to the source files
  - BTB Miss Percentage
  - Branch Misprediction Percentage

- The BTB Miss Percentage is calculated as

  BTBMissPct = (1 - (BTBHits/BTBLookups)) * 100

- The Branch Misprediction Percentage/Rate is calculated as

  BranchMispredPercent = (numBranchMispred / numBranches) * 100

Part 3: Changes to the Source Code

- To ensure these new parameters are generated on the 'stats.txt' files
- The formulas are updated in specific files of the 'src' folder in the local Gem5 directory
- The formula for the Branch Misprediction Percentage is added by
  - Adding the formula as a function in "$gem5/src/cpu/simple/base.cc"
  - Declaring this function in the header "$gem5/src/cpu/simple/exec_content.hh"

- The Formula for the BTB Miss Percent is added by
  - Adding the formula as a function in "$gem5/src/cpu/pred/bpred_unit.cc"
  - Declaring this function in the header "$gem5/src/cpu/pred/bpred_unit.hh"

Part 3: Changes for “BranchMispredPercent”

At “$gem5/src/cpu/simple/base.cc”

base.cc

```
        /*-- Extra Parameter Added --*/
        t_info.BranchMispredPercent =
            (t_info.numBranchMispred / t_info.numBranches) * 100;
        t_info.BranchMispredPercent
            .name(thread_str + ".BranchMispredPercent")
            .desc("Percent of Branch Mispredict")
            .prereq(t_info.BranchMispredPercent);
    }
}
```

At “$gem5/src/cpu/simple/exec_content.hh”

base.cc

```
// Instruction mix histogram by OpClass
Stats::Vector statExecutedInstType;

//Percent of Branch Mispredict
Stats::Formula BranchMispredPercent;
```

Part 3: Changes for “BTBMissPct”

At “$gem5/src/cpu/pred/bpred_unit.cc”

bpred_unit.cc

```
RASIncorrect
        .name(name() + ".RASInCorrect")
        .desc("Number of incorrect RAS predictions.")
        ;

    /*-- Added Extra Parameters --*/
    BTBMissPct
        .name(name() + ".BTBMissPct")
        .desc("BTB Miss Percentage")
        .precision(6);
    BTBMissPct = (1 - (BTBHits/BTBLookups)) * 100;
```

At “$gem5/src/cpu/pred/bpred_unit.hh”

bpred_unit.cc

```
    Stats::Scalar RASIncorrect;

    /** Added Paramater*/
    /** Stat for percentage of time an entry in the BTB is not found. */
    Stats::Formula BTBMissPct;
```

Part 3: Changes as seen on ”stats.txt”

```
system.cpu.branchPred.usedRAS                2807587                    # Number of times the RAS
system.cpu.branchPred.RASInCorrect            375243                    # Number of incorrect RAS
system.cpu.branchPred.BTBMissPct            9.289968                    # BTB Miss Percentage
system.cpu_voltage_domain.voltage                  1                    # Voltage in Volts
system.cpu_clk_domain.clock                      500                    # Clock period in ticks
```

```
system.cpu.op_class::InstPrefetch                0       0.00%     100.00% # Class of executed instruction
system.cpu.op_class::total               823341765                         # Class of executed instruction
system.cpu.BranchMispredPercent          14.370364                         # Percent of Branch Mispredict
system.cpu.dcache.tags.replacements        8420860                         # number of replacements
system.cpu.dcache.tags.tagsinuse       2045.525716                         # Cycle average of tags in use
```

Part 4 : Branch Prediction Exploration

- As seen earlier the Branch Predictor type is changed in the "BaseSimpleCPU.py", further the parameters of the Branch Predictor in use can be changed by changing their parameters
- Changing configurations for each Branch Predictor
    - This can be done manually by changing each parameter value in "BranchPredictor.py"
    - Then this configuration is built using "scons build/X86/gem5.opt"
    - The simulation is then run using "sh. runGem5.sh", where the runGem5.sh exists in the respective benchmark folder (ex: inside the 458.sjeng)

Part 4 : Branch Prediction Exploration

For ex: in case of the TournamentBP, to run a new configuration of the BP the highlighted values must be updated

```
class TournamentBP(BranchPredictor):
    type = 'TournamentBP'
    cxx_class = 'TournamentBP'
    cxx_header = "cpu/pred/tournament.hh"

    localPredictorSize = Param.Unsigned(2048, "Size of local predictor")
    localCtrBits = Param.Unsigned(2, "Bits per counter")
    localHistoryTableSize = Param.Unsigned(2048, "size of local history table")
    globalPredictorSize = Param.Unsigned(8192, "Size of global predictor")
    globalCtrBits = Param.Unsigned(2, "Bits per counter")
    choicePredictorSize = Param.Unsigned(8192, "Size of choice predictor")
    choiceCtrBits = Param.Unsigned(2, "Bits of choice counters")
```

Part 4 : Branch Prediction Exploration

The explored configurations are:

Parameters	LocalBP()	BiModeBP()	TournamentBP()
BTBEntries	4096 -> 2048	4096 -> 2048	4096 -> 2048
localPredictorSize	2048 -> 1024		2048 -> 1024
globalPredictorSize		8192 -> 2048	8192 -> 4096
choicePredictorSize		8192 -> 2048	8192 -> 4096

Part 4 : Branch Prediction Exploration

- For all the configuration shown in the table, there are 28 total explorable iterations for all three Branch Predictors on each benchmark
- There are 2 benchmarks we run this project for which results in 56 total explorable iterations
- Doing this manually would be an extremely inefficient approach
- Therefore we wrote a Python script to automate the whole process
- The script simulated all 3 benchmarks, for all possible parameters on both the benchmark files and stored outputs of each in different folders

Part 4 : Branch Prediction Exploration

Automation algorithm:

- Libraries used: subprocess, itertools
- Change the Branch Predictor type by editing the "BaseSimpleCPU.py" file by passing the necessary type as an argument
- Change the parameters in the "BranchPredictor.py" file for all possible combinations for each predictor
- Rebuild the configuration by running "scons build/X86/gem5.opt"
- Edit the "runGem5.sh" in the particular Benchmark's folder by adding the desired output location
- Run "sh runGem5.sh" for each iteration
- To optimize this process 2 versions of the script were created and run parallely on our machines for each benchmark(both scripts have been attached)

Part 4 : Branch Prediction Exploration

Code Snippets:

- generating all the possible configurations in case TournamentBP() and iterating through it:

```
l = [BTBentries, BMBPgpsize, BMBPcpsize]
combos = list(itertools.product(*l))
for c in combos:
```

- Changing Branch Predictor Type:
  - “CPUtemplate” is a boilerplate used to edit the original file in “src”
  - “CPUtype” is the original file inside ”src” being changed

```
f3 = open(CPUtemplate, "r")
lines = f3.readlines()
f3.close()
f3 = open(CPUtype, "w")
for line in lines:
    if('BPtype' in line):
        line = line.replace('BPtype', 'BiModeBP()')
    f3.write(line)
f3.close()
```

Part 4 : Branch Prediction Exploration

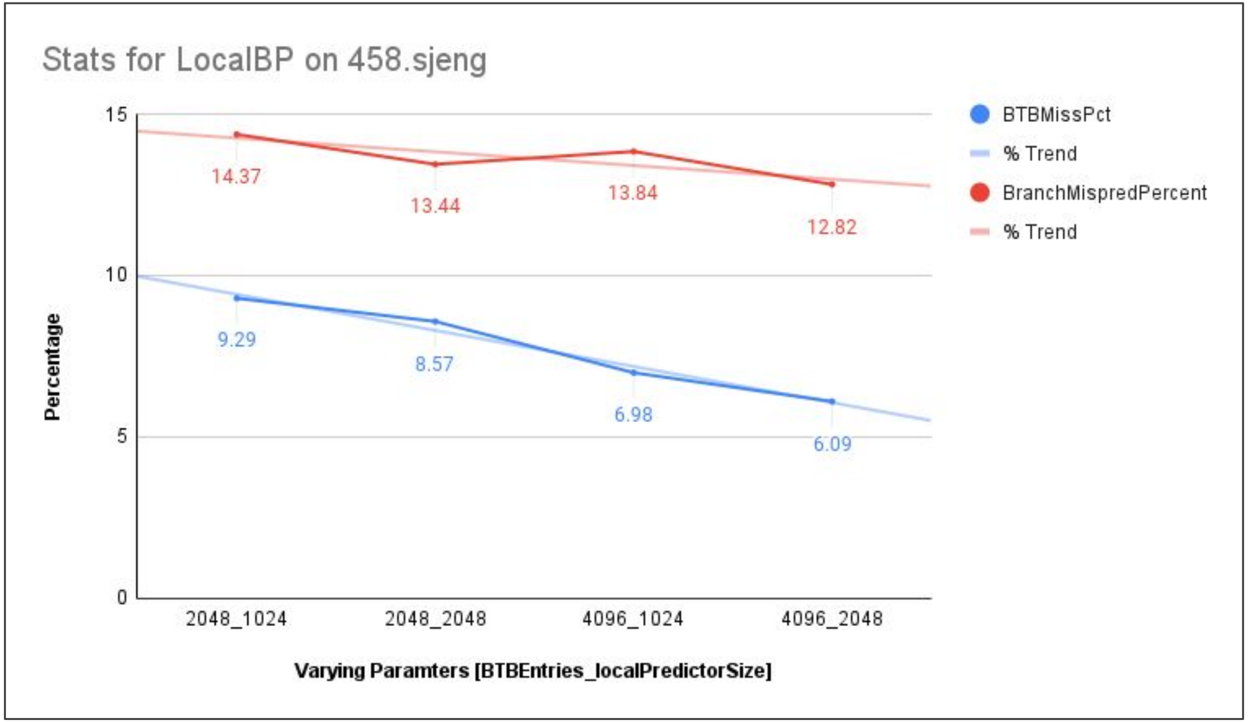
- Changing the Branch predictor parameters
  Ex: for TournamentBP()

```
if('PH5' in line):
    line = line.replace('PH5', str(c[1]))
if('PH6' in line):
    line = line.replace('PH6', str(c[2]))
if('PH7' in line):
    line = line.replace('PH7', str(c[3]))
```

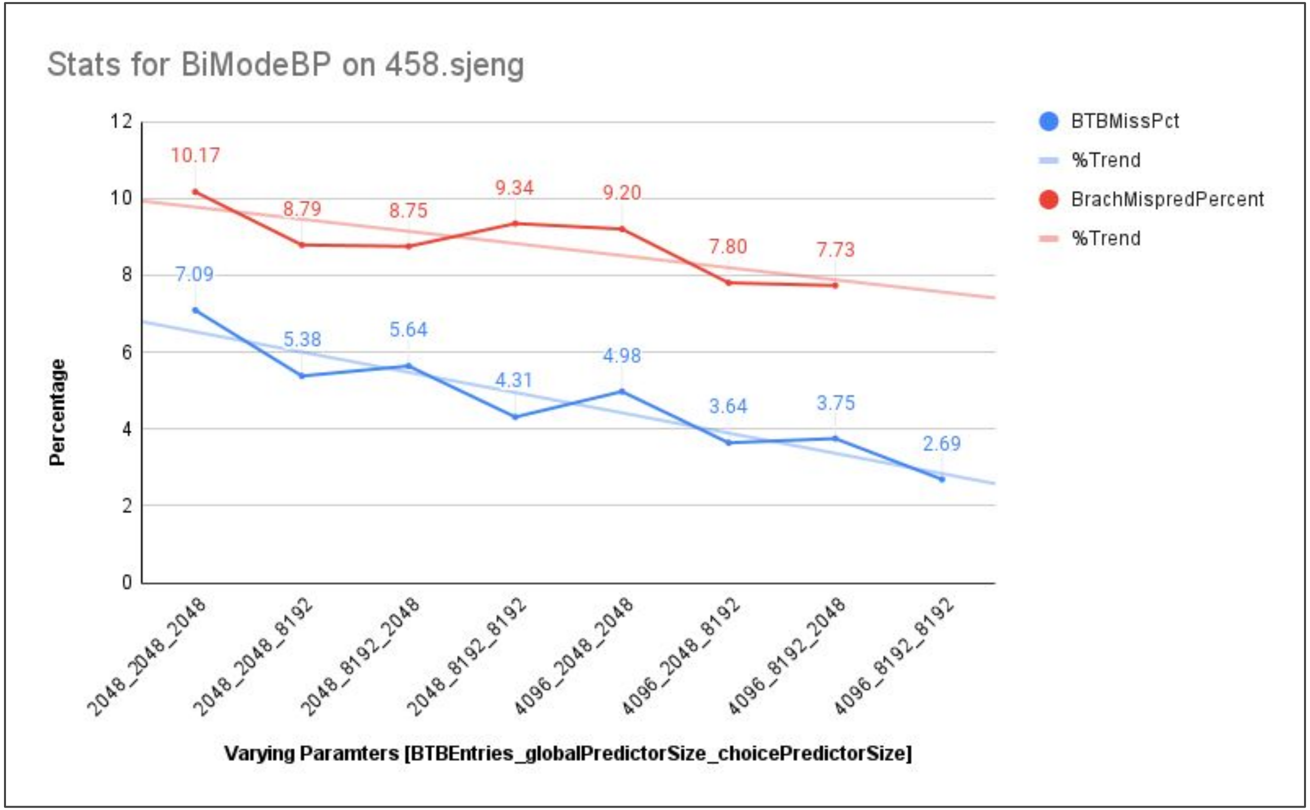
- Rewriting output location in shell scripts, rebuilding scons, running shell script

```
#call function to rewrite the shell scripts
rewriteshell(outloc2)

#call scons for every change
subprocess.call(['scons', sconspath], cwd="/home/011/a/ax/axb200166/Desktop/Akash/CompArch/gem5")

#subprocess call to run shell script
subprocess.call(['sh', runG52], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
print("Completed iteration of " + tempfolder2)
```

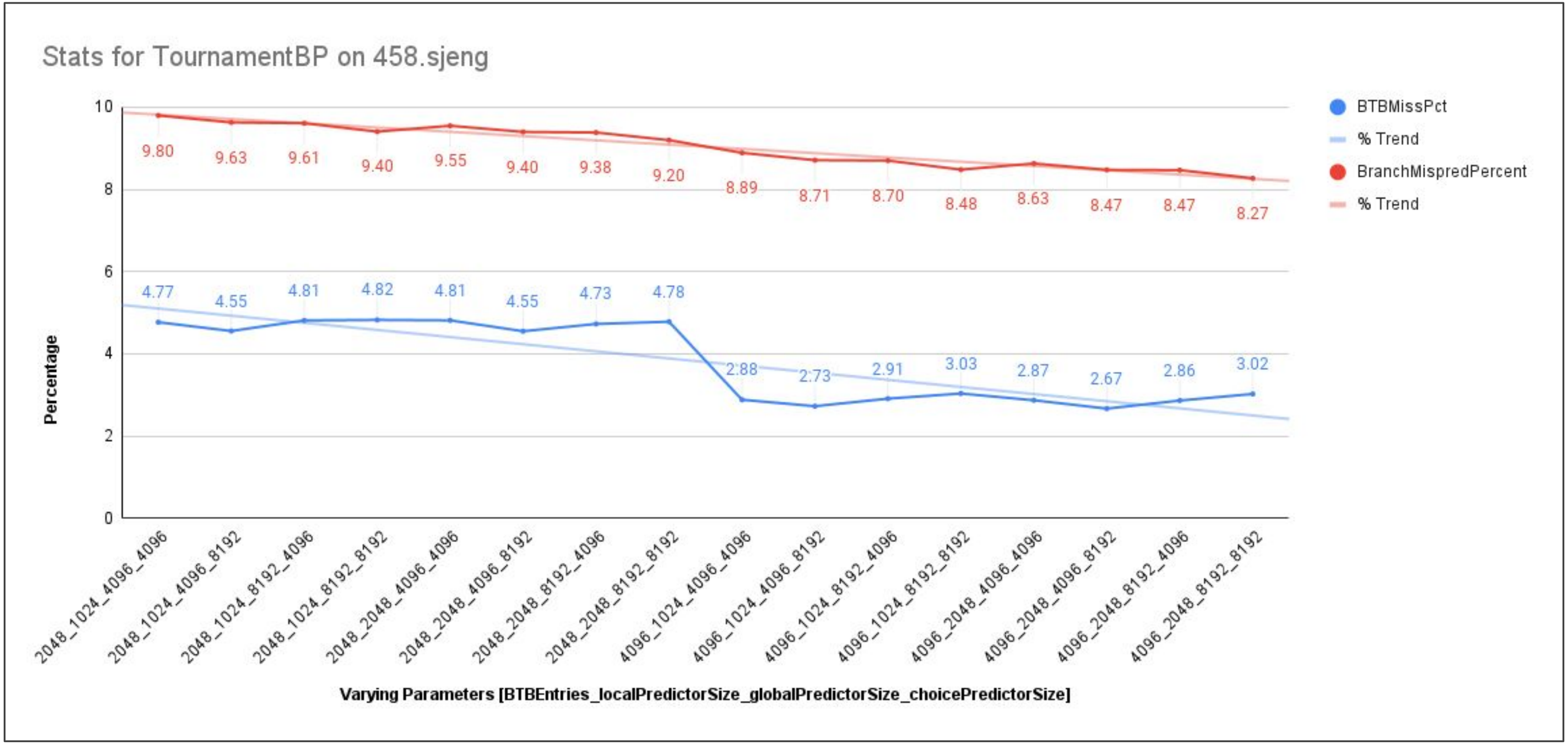
Part4: Results and Observations



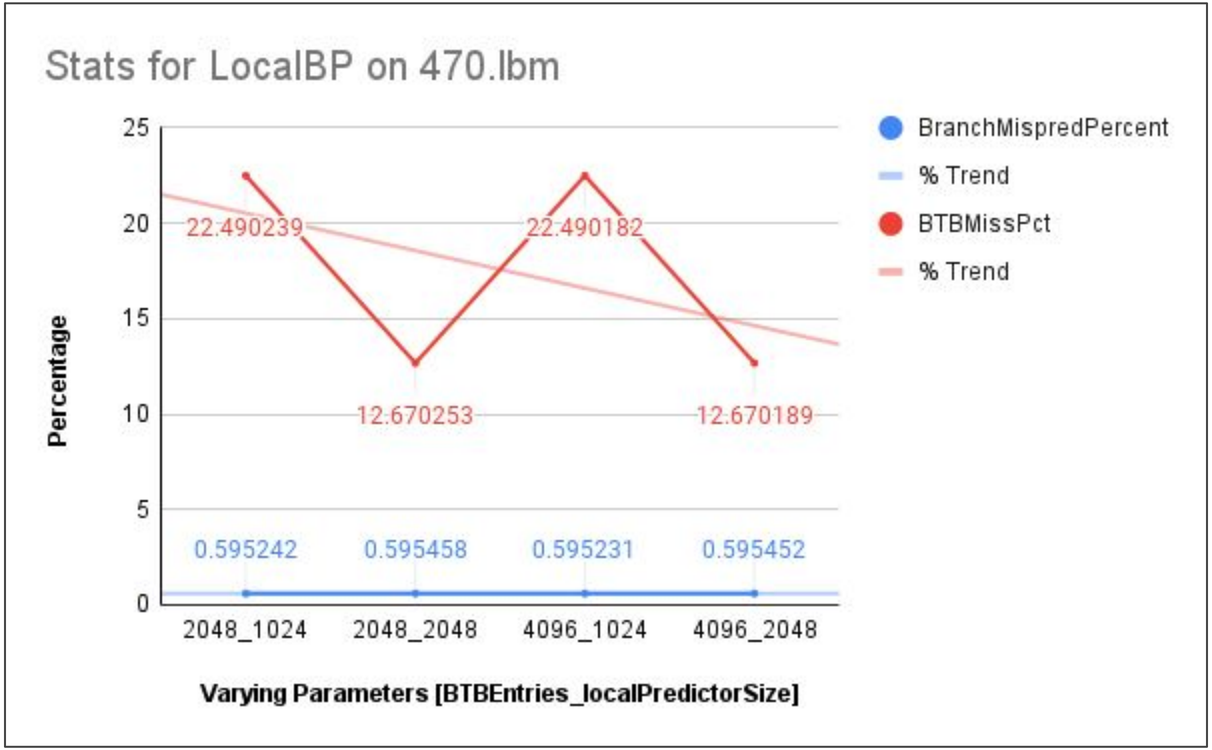
Part4: Results and Observations



Part4: Results and Observations

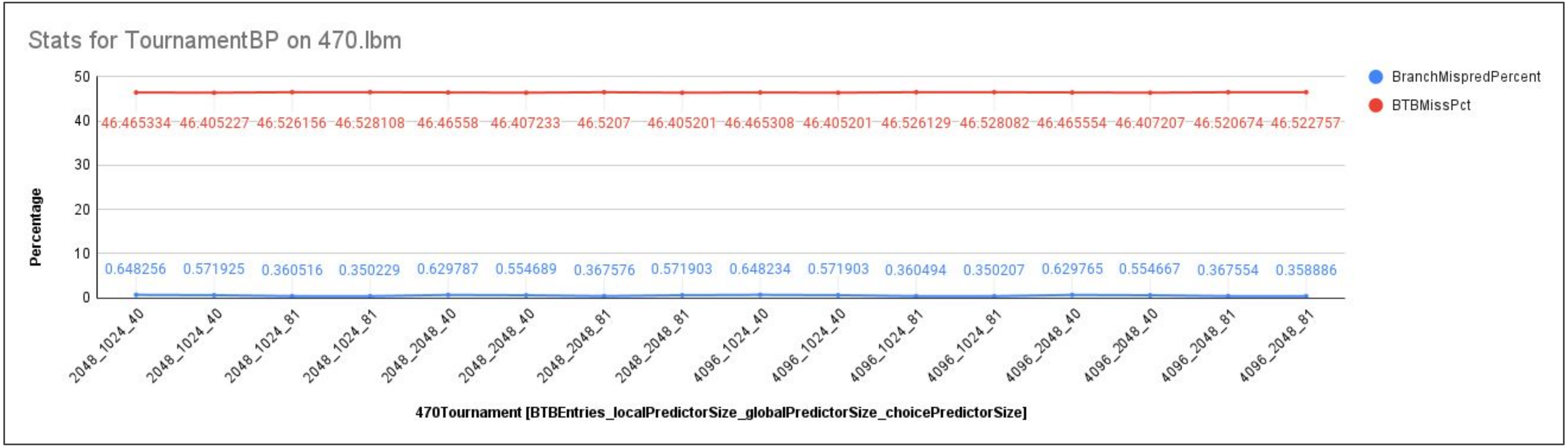


Part4: Results and Observations



Part4: Results and Observations

Part4: Results and Observations



Part 4: Results and Observations

Our inferences:

- For LocalBP on 458.sjeng with increase in BTBEntries the BTBMissPct and BranchMispredPercent reduces with a linear trend
- For BiModeBP on 458.sjeng with increase in globalPredictorSize the BTBMissPct and BranchMispredPercent reduces with a linear trend
- For TournamentBP on 458.sjeng with increase in BTBEntries from 2048 to 4096 there is a significant reduction in BTBMissPct by around 2%.
- For TournamentBP on 458.sjeng with changing globalPredictorSize does not visibly contribute to the change in trend of BranchMispredPercent

Part 4: Results and Observations

Our inferences:

- For LocalBP on 470.lbm increase in localPredictorSize causes a significant reduction in BTBMissPct by around 12%, whereas BranchMispredPercent remains the same overall
- For BiModeBP on 470.lbm increasing the globalPredictorSize reduces BranchMispredPercent as well as BTBMissPct
- All parametric variations For TournamentBP on 470.lbm does not change the trend in BTBMissPct or BranchMispredPercent

Part 4: Results and Observations

Final Conclusions:

- Of all the Branch predictors the Tournament Branch Predictor has the least BranchMispredPercent compared to the other predictors run on the same benchmark
- LocalBP uses lesser memory but the tradeoff is higher mispredictions and misses
- BiModeBP seems heavily dependent on globalPredictorSize for performance
- Going by the percent trends TournamentBP seems most optimized and stable as the performance does not change significantly with parameters