



11/20/2018

PHASE 5 REPORT

Gem5 Simulation with
Parsec- Results and Analysis

Submitted by: Maliha Arif
PID: 4506817
Submitted to: Dr. Dan C. Marinescu
Submitted on: 20th November 2018

INTRODUCTION:

The aim of this class project was to analyze the performance of Parsec-3.0 benchmark [1] on the gem5 simulator [2]. The project was divided into 5 phases. Phase 1 was about installing the gem5 simulator, the report detailed all the steps required to download and install it successfully. Phase 2 and 3 were merged and involved the installation of the Parsec benchmark and then running Parsec with the gem5 simulator. Phase 4 was about the design of the experiment, i.e. how different parameters will affect the CPU performance on running the benchmark assigned to different groups on gem5.

Below is a summary of points from all of these phases. Section 1 gives a brief information on the installation of gem5 and Parsec. Section 2 details the design of the experiment, simulation objectives and research hypothesis. Section 3 discusses the experimental results, and in-depth analysis of experiments together with statistical processing.

Section 1: OVERVIEW OF THE PROJECT - SUMMARY OF POINTS IN PHASE 1-3

Phase 1 is mostly about gem5 installation. It includes instructions on how to install GEM5 simulator on Linux machine, Operating System Ubuntu 16.01. Apart from the instructions, we discussed test cases that are successfully run with screenshots together with a discussion on challenges that were faced during the installation of Gem5.

Architecture/Operating System Used: Linux – Ubuntu 16.01

1.1. WHAT IS GEM5?

Gem5 is a modular discrete event driven computer system simulator platform. It is written primarily in C++ and python. It can simulate a complete system with devices and an operating system in full system mode (FS mode), or user space only programs where system services are provided directly by the simulator in syscall emulation mode (SE mode). There are varying levels of support for executing Alpha, ARM, MIPS, Power, SPARC, and 64 bit x86 binaries on CPU models including two simple single CPI models, an out of order model, and an in order pipelined model. A memory system can be flexibly built out of caches and crossbars. Recently the Ruby simulator has been integrated with gem5 to provide even more flexible memory system modeling.

1.2. INSTRUCTIONS FOR INSTALLING GEM 5

- 1) First, we need to clone the GitHub repository by running the following command on a Linux terminal

```
> git clone https://gem5.googlesource.com/public/gem5
```

- 2) After this to ensure, everything is updated, run the following command on the same terminal

```
> git pull
```

- 3) Before we build the binary file, we have to install the scons library using this command

```
> apt-get install scons
```

- 4) Once done with the above, we need to build the binary.

- 5) We built the **gem5.opt** binary and ARM architecture using the following command

```
> scons build/ARM/gem5.opt
```

1.3. TESTING THE BUILD

There are 2 modes of Gem5,

- 1) SE mode
- 2) Full system mode

Testing the built in SE (Syscall emulation mode):

We ran a basic hello world program to test the built using the following command.

```
➤ build/ARM/gem5.opt configs/example/se.py -c tests/test-progs/hello/bin/arm/linux/hello
```

1.4. INSTALLING FULL SYSTEM VERSION

- 1) Create a directory > full_system_files
- 2) Download 2014 tar file of the FS version from this link

<http://www.gem5.org/dist/current/arm/>

The file I downloaded was arm_system_2014_08.tar.bz

- 3) Then enter this folder using > cd full_system_files
- 4) Create an environment variable M5_path=/home/mkhan/.... >> ~/.bashrc

This command adds the environment variables

- 5) > source ~/.bashrc (this command refreshes all variables)

These steps install the FS version. To test the build, run the following command

build/ARM/gem5.opt -d/tmp/output configs/example/fs.py

This command runs the simulation. Now in FS mode, we can connect to the machine by opening another command line window and typing the following

```
>telnet localhost 3456
>login:root
```

CHECKPOINT – Checkpoints are essentially snapshots of a simulation.

>m5 checkpoint

To resume or restore simulation, following command is used.

```
build/ALPHA/gem5.debug configs/example/fs.py --checkpoint-restore=N
```

Following the above instructions, we were able to install gem5 simulator. Phase 2 and Phase 3 of the project were about installing the Parsec benchmark [2] and then running Parsec on gem5 simulator.

SUMMARIES OF PHASE 2 AND PHASE 3

1.5. PARSEC BENCHMARK INSTALLATION

The Princeton Application Repository for Shared-Memory Computers (PARSEC) [1] is a benchmark suite composed of multithreaded programs. It consists of 13 workloads, 10 applications and 3 kernels common to desktop and server programs. It has been developed at Princeton as a PhD dissertation of a researcher in 2008.

The current version of the suite contains the following 13 programs from many different areas such as computer vision, video encoding, financial analytics, animation physics and image processing:

The 13 workloads are as follows:

- blackscholes - Option pricing with Black-Scholes Partial Differential Equation (PDE)
- bodytrack - Body tracking of a person
- canneal - Simulated cache-aware annealing to optimize routing cost of a chip design
- dedup - Next-generation compression with data deduplication
- facesim - Simulates the motions of a human face
- ferret - Content similarity search server
- fluidanimate - Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method
- freqmine - Frequent itemset mining
- raytrace - Real-time raytracing
- streamcluster - Online clustering of an input stream
- swaptions - Pricing of a portfolio of swaptions
- vips - Image processing
- x264 - H.264 video encoding

Our group 5 basically has been assigned benchmark 5- facesim to simulate and work with in phase 3-simulation with gem5. But before we go to that phase, let's first quickly discuss how the benchmark was installed.

1.6. PARSEC DOWNLOAD AND INSTALLATION

Use the following sequence of commands to download and extract the Parsec-3.0 benchmark suite on the terminal. In our case, we are using an Ubuntu desktop and hence these commands have to be entered in the terminal.

- **wget http://parsec.cs.princeton.edu/download/3.0/parsec-3.0-core.tar.gz**
- **tar -xzf parsec-3.0-core.tar.gz**

This basically downloads the tar file in your home directory. Once extracted, do the following.

- **cd parsec-3.0**
- **source env.sh**

This takes you to the parsec directory where you set environment variable. Here you can build different benchmarks using the following 2 commands. 'Parsecmgmt' command is specific to this benchmark suite and will run once you add the environment variable using the source env.sh command.

- **parsecmgmt -a build -p "benchmark"**
- **parsecmgmt -a run -p "benchmark"**

Testing with different benchmarks and then our benchmark '5' facesim gives us the following output at run time.

```
vision@maliha-precision-workstation-t3500:~/parsec-3.0$ parsecmgmt -a build -p facesim
[PARSEC] Packages to build: parsec.facesim

[PARSEC] [===== Building package parsec.facesim [1] =====]
[PARSEC] [----- Analyzing package parsec.facesim -----]
[PARSEC] Package parsec.facesim already exists, proceeding.
[PARSEC] BIBLIOGRAPHY
[PARSEC] [1] Bienia. Benchmarking Modern Multiprocessors. Ph.D. Thesis, 2011.
[PARSEC] Done.
vision@maliha-precision-workstation-t3500:~/parsec-3.0$ parsecmgmt -a run -p facesim
[PARSEC] Benchmarks to run: parsec.facesim

[PARSEC] [===== Running benchmark parsec.facesim [1] =====]
[PARSEC] Deleting old run directory.
[PARSEC] Setting up run directory.
[PARSEC] No archive for input 'test' available, skipping input setup.
[PARSEC] Running 'time /home/vision/parsec-3.0/pkgs/apps/facesim/inst/amd64-linux.gcc/bin/facesim -h'
[PARSEC] [----- Beginning of output -----]
PARSEC Benchmark Suite Version 3.0-beta-20150206
Usage: /home/vision/parsec-3.0/pkgs/apps/facesim/inst/amd64-linux.gcc/bin/facesim [-restart <int>] [-
-restart <no description available> (default 0)
-lastframe <no description available> (default 300)
-threads <no description available> (default 1)
-timing <no description available>

real    0m0.057s
user    0m0.000s
sys     0m0.000s
[PARSEC] [----- End of output -----]
[PARSEC] BIBLIOGRAPHY
[PARSEC] [1] Bienia. Benchmarking Modern Multiprocessors. Ph.D. Thesis, 2011.
[PARSEC] Done.
vision@maliha-precision-workstation-t3500:~/parsec-3.0$
```

Parsec benchmarks can also be run with some inputs, the command to be used for that is as follows:

➤ **parsecmgmt -a run -p facesim -i simsmall**

1.7. RUNNING PARSEC BENCHMARK WITH GEM5

To run Parsec with gem5, we need to download a Research Starter Kit. This kit includes patches, 2 of which we will use to enable Gem5 and Parsec to work together.

This repository can be downloaded in the home folder using the following command.

➤ **git clone https://github.com/arm-university/arm-gem5-rsk.git**

To simulate the benchmarks on Gem5, we need to compile Parsec benchmarks with ARM architecture.

There are 2 ways of doing this

- 1) **Cross-compiling on x86 machine**
- 2) **Compiling on QEMU**

We used the first technique, i.e. cross compile with x86 and it worked successfully. There were a series of steps common to both techniques. Summary is as follows:

We start off by applying a patch followed by changes in the config.guess and config.sub directories.

1.7. CROSS-COMPILING ON X86 MACHINE

After these 2 common steps, we compiled Parsec benchmark using x86. Following commands were used.

Download and extract the aarch-linux-gnu toolchain using

➤ **wget https://releases.linaro.org/components/toolchain/binaries/latest-5/aarch64-linux-gnu/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xz**
➤ **tar xvfJ gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xz**

Once it is extracted outside the main Parsec directory, we should do the following:

➤ Open “arm-gem5-rsk/parsec_patches/xcompile-patch.diff” with an editor and change the following lines

```
+ export CC_HOME="<gcc-linaro-directory>"
+ export BINUTIL_HOME="<gcc-linaro-directory>/aarch64-linux-gnu"
```

Now apply the patch from within the parsec directory by typing:

➤ `patch -p1 < ../arm-gem5-rsk/parsec_patches/xcompile-patch.diff`

Further compile parsec by typing the following:

```
➤ export PARSECPLAT="aarch64-linux"
➤ source ./env.sh
➤ parsecmgmt -a build -p facesim
```

We did the above and got no error, hence we were able to cross compile Facesim with x86 successfully. Since the gem5 FS mode doesn't support shared directories with the host, the compiled PARSEC benchmarks need to be copied to the ARM disk image. This required expanding the disk image, which we did by following a couple of steps as shared in detail in phase 3 report.

1.8. RUNNING GEM5 FS WITH PARSEC BENCHMARK

The command that worked correctly, i.e. allowed gem5 to simulate properly with Parsec benchmark is as under, other commands had some issue, reasons for which were discussed in the phase 3 report.

```
./build/ARM/gem5.opt -d fs_results/facesim configs/example/fs.py --disk-
```

```
image=/home/vision/full_system_images/disks/expanded-linaro-minimal-aarch64.img -
```

```
-machine-type=VExpress_EMM64
```

So, we use fs.py as our config file, disk image is the new expanded disk and since the disk is 64 bit, we have to specify the machine type in terminal so that Gem5 simulates correctly.

After this, we should open another command terminal and type

➤ **telnet localhost 3456**

This opens a session and if simulation starts fine, we will see this. It took around 20,30 mins for the simulator to start. When the initialization process is complete, the client terminal will be able to execute commands.

➤ **root@genericarmv8:**

Here we need to enter the parsec-3.0 bin folder so that we can run the benchmark

➤ **cd parsec-3.0/bin**

➤ **./parsecmgmt -a build -p facesim**

➤ **./parsecmgmt -a run -p facesim**

In the above manner, we were able to run our benchmark 'facesim' on gem5 simulator. Now we shall discuss the design of our experiments and what parameters etc. we are tweaking and modifying to study the effects on Parsec.

SECTION 2: EXPERIMENTAL DESIGN AND SETUP – SUMMARY OF PHASE 4

2.1. MAIN IDEA

When designing a multicore processor, there are many architectural decisions to make at various levels of the design. These decisions include number of cores; last level cache (LLC) capacity, L1 cache, L2 cache, cache block size, line size, associativity, replacement policy, and distribution; cache hierarchy, coherency, and interconnects; lower level cache capacity, line size, associativity, and replacement policy; TLB size and associativity; branch predictors and training; ROB (reorder buffer size), Size of MSHR (miss status and handling register), multiply and divide latency and throughput; FPU latency and throughput; and many more decisions depending on the specific optimizations performed in the processor pipeline such as superscalar, out-of-order execution, in-order execution and register renaming etc.[3]

Once these parameters are chosen and modified, we can evaluate how each affect different types of benchmarks and subsequently how an architecture should be designed.

2.2. PARSEC SCALING PROPERTIES AND BOTTLENECKS

We know that PARSEC is a reference application suite used in industry and academia to assess new Chip Multiprocessor (CMP) designs, however no investigation to date has profiled PARSEC on real hardware to better understand scaling properties and bottlenecks. This understanding is crucial in guiding future CMP designs for these kinds of emerging workloads.

In phase 5 of the project, we aim to use different performance counters such as ROB, Branch Prediction, No. of ALUs, and MSHR together with memory models, cache size, cache capacity to see how Parsec benchmark performs under different constraints.

Research described in [4] affirms that the benchmarks in Parsec are largely compute-bound, and thus limited by number of cores, micro-architectural resources, and cache-to-cache transfers, rather than by off-chip memory or system bus bandwidth.

The benchmark our group [Group 5] has been assigned is 'facesim' which is an animation benchmark and we will be evaluating its performance using different CPU and memory models. Before we go into the details of the parameter I will be working on, let's look at the nature of the benchmark briefly so we can then discuss how the experiment was designed.

2.3. FACESIM - BENCHMARK

Facesim is an Intel RMS application originally developed by the University of Stanford [4]. Its goal is to obtain visually realistic animations by simulating the physics involved in the motion of a 3D model. Facesim is an important application of Parsec. The future of video games and movies needs a realistic way of animating faces as it is the place where the users focus more when watching a movie or video games. Character faces get a lot of attention and it is important to give them a convincing appearance.

The model has a total of 30 thousand surface triangles dedicated to the cranium and jaw. The full head has about 850.000 tetrahedra but only the face flesh movements are actually being simulated.

The inputs for facesim are defined as follows:

- test: Print out help message.
- simdev: 80,598 particles, 372,126 tetrahedra, 1 frame
- simsmall: Same as simdev
- simmedium: Same as simdev
- simlarge: Same as simdev
- native: Same as simdev, but with 100 frames



Figure 3.1: 3D face model - Image obtained from [3]

2.4. NATURE OF THE BENCHMARK/ PARAMETERS OF CPU TO STUDY:

Hence, we observe that Facesim is not IO intensive or network intensive rather it is **CPU intensive** as it requires a lot of processing to simulate a human face at run time.

These are the CPU parameters that we will be modifying

- 1) ROB
- 2) MSHR
- 3) Branch Prediction
- 4) ALU

I will be modifying **Branch Prediction parameter** and observing different performance metrics as discussed below.

2.5. WHAT IS BRANCH PREDICTION?

A very important microarchitecture technique used since the very early processors has been branch prediction. When a branch is fetched, the next instructions to be executed depend on the branch outcome, which is not available until the branch is executed some cycles later. Stalling the fetch until the outcome of the branch is known would cause a significant penalty in pipelined processors since branches are very common in many programs (in the order of one out of ten instructions for non-numerical codes). Predicting the outcome of the branch and speculatively executing the instructions of the predicted path can alleviate this penalty

provided that the predictor is highly accurate. This technique is known as branch prediction and was used as early as the late 1950s in the IBM Stretch computer. Branch predictors based on a table of two-bit saturating counters were introduced by James Smith in the late 1970s and have been used in practically all microprocessors since then. Since the penalty of a branch misprediction grows linearly with the pipeline depth, deeper pipelines prompted research on more sophisticated predictors such as the two-level branch predictor proposed by Yeh and Patt in the early 1990s, variants of which have been used by many modern microprocessors. These predictors are based on two-bit saturating counters, but the particular counter used in each case depends not only on the particular branch being predicted but also on the outcome of recent past branches.

2.6. PERFORMANCE METRICS

Branch prediction is essentially an optimization (minimization) problem where the emphasis is on to achieve lowest possible miss rate, low power consumption and low complexity with minimum resources hence miss predictions is one of the performance metric we will looking at when modifying branch predictors.

On the other hand, complex branch predictions – either neural based or variants of two level branch prediction provide better prediction accuracy but consume more power and complexity increases exponentially. As a result, **IPC (Instructions per cycle), execution cycles, BTB miss rate, Misprediction rate**, are other parameters we will be observing when changing the branch prediction parameter.

2.7. TYPES OF BRANCH PREDICTIONS

So, there is static branch prediction and dynamic branch prediction. For dynamic branch prediction, there are 3 kinds of Branch Predictors available in GEM5 (stable). [5]

1. 2bit_local Predictor

2. Bi_mode Predictor: Uses a Branch History Table (BHT)

3. Tournament Predictor or Hybrid Predictor: Uses a combination of two or more (usually two) branch prediction mechanisms.

These files are found in the “\$gem5/src/cpu/pred/” folder and the python file to modify the size of predictor is named **BranchPredictor.py**.

The experiment can be designed in 2 ways, one by choosing different predictors and testing between them or by choosing one predictor and modifying its size.

We went for the latter option and picked the tournament or hybrid predictor which is the default one in gem5 and modified its parameters. (size mostly)

2.8. CPU PARAMETER We modified AND ITS RANGE

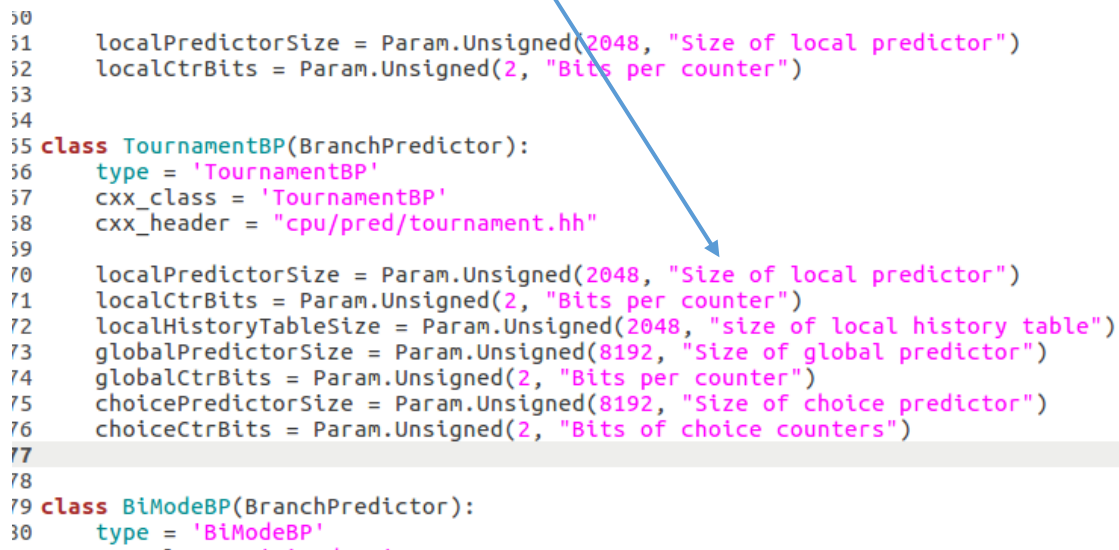
So, we have performed the experiments in the following manner. We are using the tournament predictor, and modifying its size in the “\$gem5/src/cpu/pred/BranchPredictor.py” file.

The values we have taken and worked on are as follows:

Predictor size (Bytes)
512
1024
2048
4096

Hence our range of values will be as given above. Here is the screenshot of where these vales will be changed.

Size of local Predictor will be modified. By default, it is 2048.



```

50
51     localPredictorSize = Param.Unsigned(2048, "Size of local predictor")
52     localCtrBits = Param.Unsigned(2, "Bits per counter")
53
54
55 class TournamentBP(BranchPredictor):
56     type = 'TournamentBP'
57     cxx_class = 'TournamentBP'
58     cxx_header = "cpu/pred/tournament.hh"
59
60     localPredictorSize = Param.Unsigned(2048, "Size of local predictor")
61     localCtrBits = Param.Unsigned(2, "Bits per counter")
62     localHistoryTableSize = Param.Unsigned(2048, "size of local history table")
63     globalPredictorSize = Param.Unsigned(8192, "Size of global predictor")
64     globalCtrBits = Param.Unsigned(2, "Bits per counter")
65     choicePredictorSize = Param.Unsigned(8192, "Size of choice predictor")
66     choiceCtrBits = Param.Unsigned(2, "Bits of choice counters")
67
68
69 class BiModeBP(BranchPredictor):
70     type = 'BiModeBP'
71     ...

```

2.9. PERFORMANCE PARAMETERS Studied

Below are the parameters that we studied for our experiment, we used information such as total number of BTB hits, references, total number of branches fetched to compute BTB miss rate, misprediction rate etc.

The **BTBMissPct** is defined as:

$$\text{BTBMissPct} = (1 - (\text{BTBHits}/\text{BTBLookups})) * 100$$

where: BTB Hits -> total number of BTB Hits
 BTBLookups -> total number of BTB References

Similarly, we also implemented the **BranchMispredPercent** as below:

$$\text{BranchMispredPercent} = (\text{numBranchMispred} / \text{numBranches}) * 100;$$

where: numBranchMispred -> total number of mispredicted Branches
 numBranches -> total number of branches fetched

With this, we also studied the following parameters when changing the predictor size. Graphs and results for each are ahead in section 3.

- 1) Miss predictions
- 2) IPC (Instructions per cycle)
- 3) Execution cycles

2.10. MEMORY MODEL PARAMETER – we modified and studied

The parameter I modified is

- **L1 Cache Block Size**

Range as follows:

Block size /KB
64
128
256
512

Since this is the cache size we are talking about, following were our performance metrics as cache block size greatly affects cache miss rate and total performance.

1. **Data L1 cache miss rate**
2. **L2 Bus Traffic (transactions per processor)**
3. **Total Performance (execution cycles), and IPC**

To note, we kept the number of processors fixed at 4 and L1 capacity fixed at 32 KB while performing our experiments to fully observe the effects of L1 Cache block size.

2.11. STATISTICAL PROCESSING:

It is important to be confident about our results and ensure the values we are getting are correct and not because of any ambiguity or error. Hence creating confidence intervals is very important and repeating the experiments is the best option for certainty.

Note that since the simulation takes a lot of time, we might not be able to repeat the experiments a large number of times but we intend to repeat the experiment 5 times for different size of CPU parameter and memory parameter as our sample data.

Though we intended to perform the experiment 5 times with each parameter, due to lack of time. I was able to run 1 CPU parameter 4 times whereas the rest I ran **3 times each**.

Confidence interval:

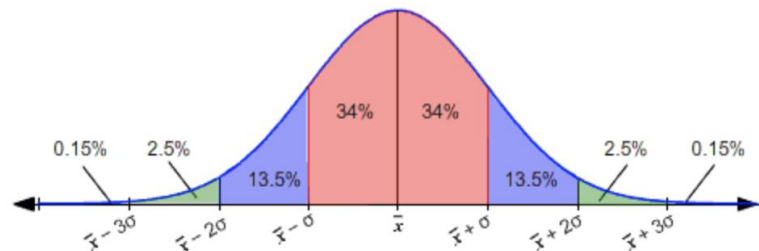
A **confidence interval** gives an estimated range of values which is likely to include an unknown population parameter, the estimated range being calculated from a given set of sample data.

- If we perform the experiment for our CPU parameter 3 times, we will then take out its average, the average will of the performance metric we are observing example miss prediction.

- If the measurements follow a normal distribution, then the sample mean will have the

distribution $N(\mu, \frac{\sigma}{\sqrt{n}})$. Where μ (mew) is our mean and sigma is standard deviation.

We use 95% confidence in our experiments. This technique allowed us to be more certain of our experiments and if we were calculating everything correctly.



SECTION 3: ANALYSIS AND RESULTS

This section discusses the results we got following our experiments. It first shows tables which contain the result for different parameters. The tables include results for all 3 experiments we conducted per parameter value – both for memory model and CPU model.

Simulation accuracy is proved by taking our confidence interval for each set of experiments. We use the R compiler [6] to calculate mean and confidence interval for each set.

This is followed by graphs for the different metrics involved. We then discuss what each graph means, and share our conclusions and analysis.

3.1. SIMULATION RESULTS

[Memory Model]

Parameter modified: Cache Block size

Metrics Observed:

- 1) IPC
- 2) Instruction Execution Rate
- 3) Row buffer Hit Rate
- 4) Row buffer Miss rate
- 5) L1 Data Cache demand misses & hits
- 6) L1 Data Demand accesses
- 7) L1 Instruction demand misses & hits
- 8) L1 Instruction demand accesses

The command we use to obtain these results is as follows:

```
./build/ARM/gem5.opt -d fs_results/memory_models/facesim1 configs/example/fs.py --disk-image=/home/vision/full_system_images/disks/expanded-linaro-minimal-aarch64.img --machine-type=VExpress_EMM64 --cpu-type=DerivO3CPU --caches --ruby --l2cache --l1d_size=128kB --l1i_size=128kB --l2_size=1MB --cacheline_size=64
```

and this is for cache block size 256 KB and above

```
./build/ARM/gem5.opt -d fs_results/memory_models/facesim3 configs/example/fs.py --disk-image=/home/vision/full_system_images/disks/expanded-linaro-minimal-aarch64.img --machine-type=VExpress_EMM64 --cpu-type=DerivO3CPU --caches --ruby --l2cache --l1d_size=128kB --l1i_size=128kB --l2_size=1MB --l1d_assoc=2 --l1i_assoc=2 --cacheline_size=256 --mem-type=HBM_1000_4H_1x128
```

Experiment Number	Cache Block Size (KBytes)	IPC	Instruction Execution rate	Row buffer Hit rate & Miss rate for Reads	Row buffer Hit rate & Miss rate for Writes
1	64	0.97118	1.339563	Hit=73.72 Miss=26.28	Hit=44.37 Miss=55.63
2		0.96145	1.334521	Hit=73.81 Miss=26.19	Hit=43.37 Miss=54.63
3		0.97214	1.325417	Hit=72.56 Miss=27.44	Hit=44.21 Miss=55.79
1	128	0.906396	1.348081	Hit=81.40 Miss=18.6	Hit=61.64 Miss=38.36
2		0.905381	1.349081	Hit=81.20 Miss=18.80	Hit=60.64 Miss=39.36
3		0.916832	1.346751	Hit=82.31 Miss=27.69	Hit=61.63 Miss=38.35
1	256	0.917172	1.375923	Hit=84.70 Miss=15.3	Hit=73.72 Miss=26.26
2		0.912542	1.378294	Hit=83.60 Miss=16.4	Hit=73.62 Miss=26.36
3		0.916739	1.376451	Hit=84.90 Miss=15.1	Hit=72.44 Miss=28.56
1	512	0.9035721	1.438271	Hit=88.30 Miss=11.70	Hit=82.34 Miss=18.66
2		0.9057922	1.423780	Hit=87.91 Miss=12.09	Hit=82.45 Miss=18.55
3		0.8947392	1.419929	Hit=88.63 Miss=11.37	Hit=81.38 Miss=18.62

Table.1

[CPU Model]

Parameter modified: Local Predictor Size – Tournament Predictor

Metrics Observed:

- 1) Total Number of BTB Hits
- 2) Total Number of BTB lookups
- 3) BTB Miss Percent
- 4) Indirect Predictor Miss Rate
- 5) IPC
- 6) Instruction Execution Rate

Command Used for execution:

```
./build/ARM/gem5.opt -d fs_results/facesim4 configs/example/fs.py --disk-  
image=/home/vision/full_system_images/disks/expanded-linaro-minimal-aarch64.img --  
machine-type=VExpress_EMM64 --cpu-type=DerivO3CPU --caches --ruby
```

An important thing to note here is that, we are changing CPU parameters in the python files. These 2 in our case.

```
gem5/src/cpu/pred/BranchPredictor.py  
gem5-stable/src/cpu/o3/O3CPU.py
```

Hence every time we make a change, we need rebuild the binary 'gem5.opt' using the scons command as follows.

```
>cd gem5  
>scons build/ARM/gem5.opt
```

Below is a table that shows the metrics and their values when CPU model- Branch Prediction was modified.

Experiment #	Predictor Size (Bytes)	BTB Miss Percent	Number of Conditional Branches Incorrect	Mispredicted Conditional Branches-%	Indirect Predictor Miss Rate-%	IPC
1	512	37.630643	67730090	9.31	56.59	0.963604
2		37.857284	56682759	9.42	55.89	0.972845
3		37.3746	35769256	9.35	56.24	0.96874
1	1024	37.8628	88329773	8.65	51.32	0.96724
2		36.8374	35978269	8.63	51.42	0.97257
3		37.82794	79578256	8.78	50.13	0.957247
1	2048	36.983	53340566	8.37	45.67	0.967635
2		37.52	98725965	8.23	42.4	0.991210
3		37.8	87263502	8.36	43.5	0.960701
4		37.89	23895025		43.7	0.967511
1	4096	36.82462	74834916	7.45	41.89	0.962747
2		37.93623	26526505	7.42	41.63	0.972468
3		36.8744	49852864	7.64	41.85	0.924672

Table.2

3.2. SIMULATION ACCURACY

We used R compiler to calculate confidence interval for each set of experiments. The method to compute has been discussed in detail in section 2 above. In general, a **confidence interval** gives an estimated range of values which is likely to include an unknown population parameter, the estimated range being calculated from a given set of sample data. We use 95 % confidence in our calculations. Resulting values are shared below on the next page in table 3 and table 4. We obtain 1 mean for each set of metric. We use these values to plot our graphs in section 3.3.

Code to compute confidence intervals for different samples of our data is as follows:

Input:

```
list<-c(1.339563,1.334521,1.325417)
t.test(list)
```

Output:

```
data: list
t = 322.07, df = 2, p-value = 9.64e-06
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 1.315357 1.350977
sample estimates:
mean of x
 1.333167
```

[Memory Model]

	Cache Block Size (KBytes)	IPC	Instruction Execution rate	Row buffer Hit rate & Miss rate for Reads	Row buffer Hit rate & Miss rate for Writes
Mean	64	0.9682567	1.333167	Hit=73.36333 Miss=26.63667	Hit=43.98333 Miss=55.35
Mean	128	0.90638123	1.3491812	Hit=81.70672 Miss=18.80248	Hit=61.64562 Miss=38.97234
Mean	256	0.91254253	1.3782944	Hit=84.907281 Miss=15.17428	Hit=73.62 Miss=26.36
Mean	512	0.90579222	1.4137802	Hit=88.374644 Miss=11.27628	Hit=82.378234 Miss=18.68173

Table. 3**[CPU Model]**

	Predictor Size (Bytes)	BTB Miss Percent	Number of Conditional Branches Incorrect	Mispredicted Conditional Branches-%	Indirect Predictor Miss Rate-%	IPC
Mean	512	37.68735	53972425	9.36	56.43	0.964604
Mean	1024	37.86643	67962099	8.64	51.72	0.96734
Mean	2048	37.883	63341566	8.34	44.67	0.957635
Mean	4096	36.5325	54834916	7.55	41.79	0.972747

Table.4

3.3. GRAPHS – CONCLUSIONS DRAWN

➤ CPU Model - Results

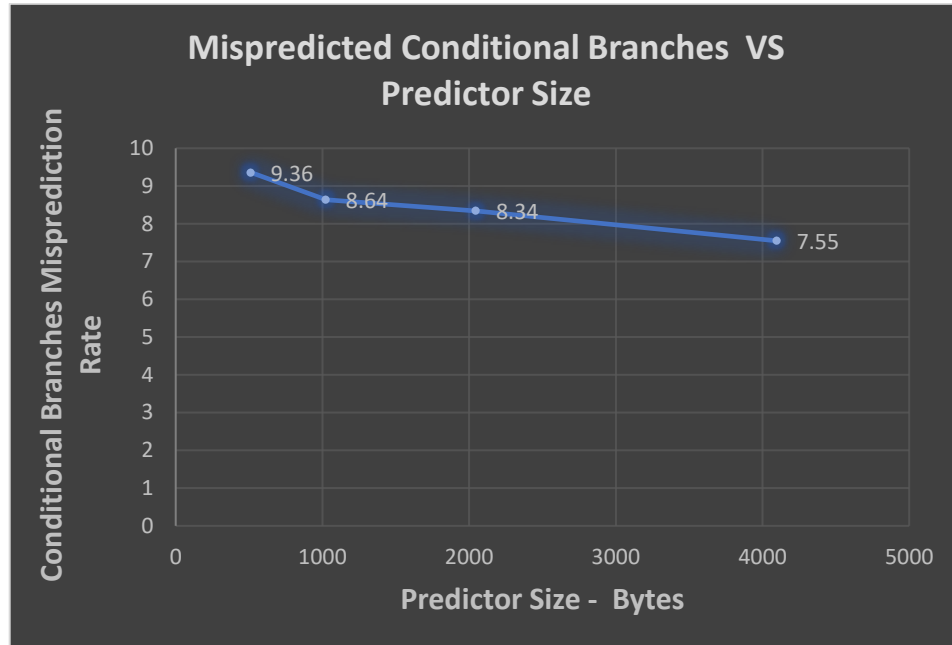


Fig.2

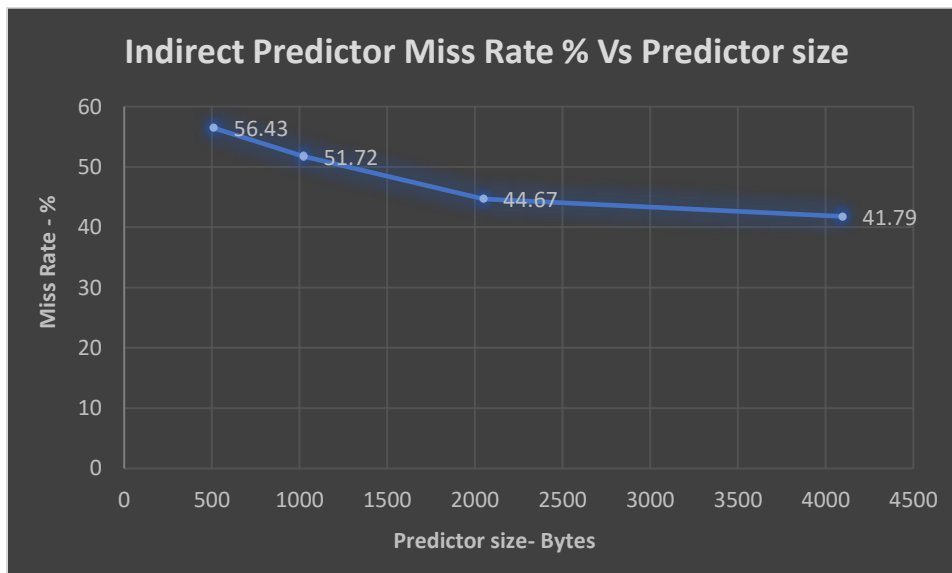


Fig.3

CPU model

- From the readings and graph shared above, we observe that IPC and Execution cycle increase but then saturate after a while as predictor size increases.
- On the contrary Indirect predictor miss rate decreases with increased predictor size, this is good as miss rate should increase with increase in predictor size, we will discuss this further in the analysis section 3.4.
- BTB miss percent remains the same as we are not altering BTB size
- Conditional branches misprediction rate also decreases as tournament predictor's size increases.

Memory Model

- From the graphs shared below for memory model, we observe that the miss rate decreases with increase in block size, Fig.5
- IPC saturates so does Execution cycle with increase in block size. Fig.4

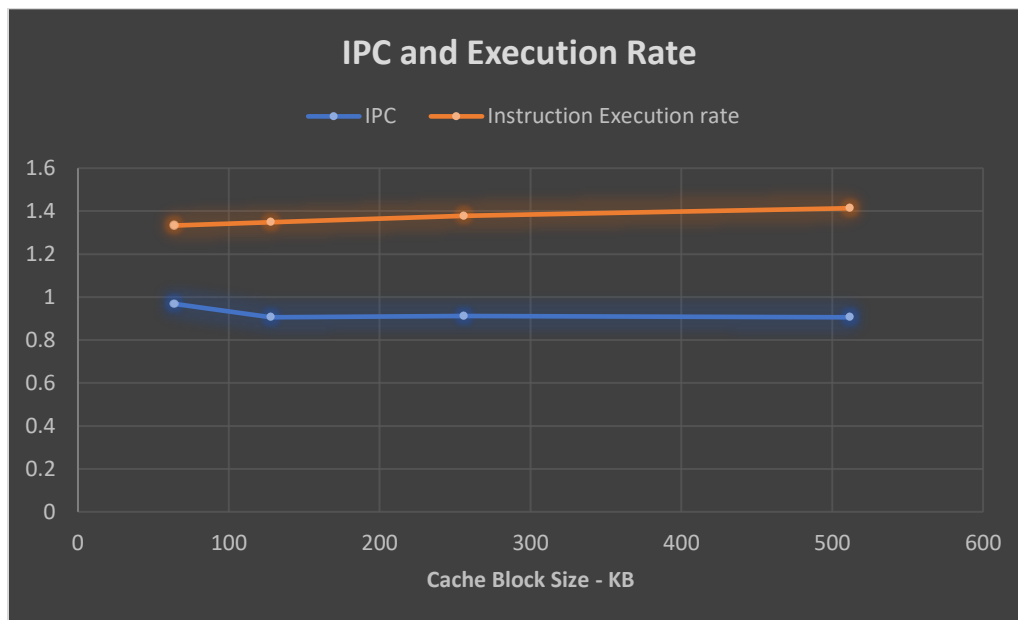


Fig.4

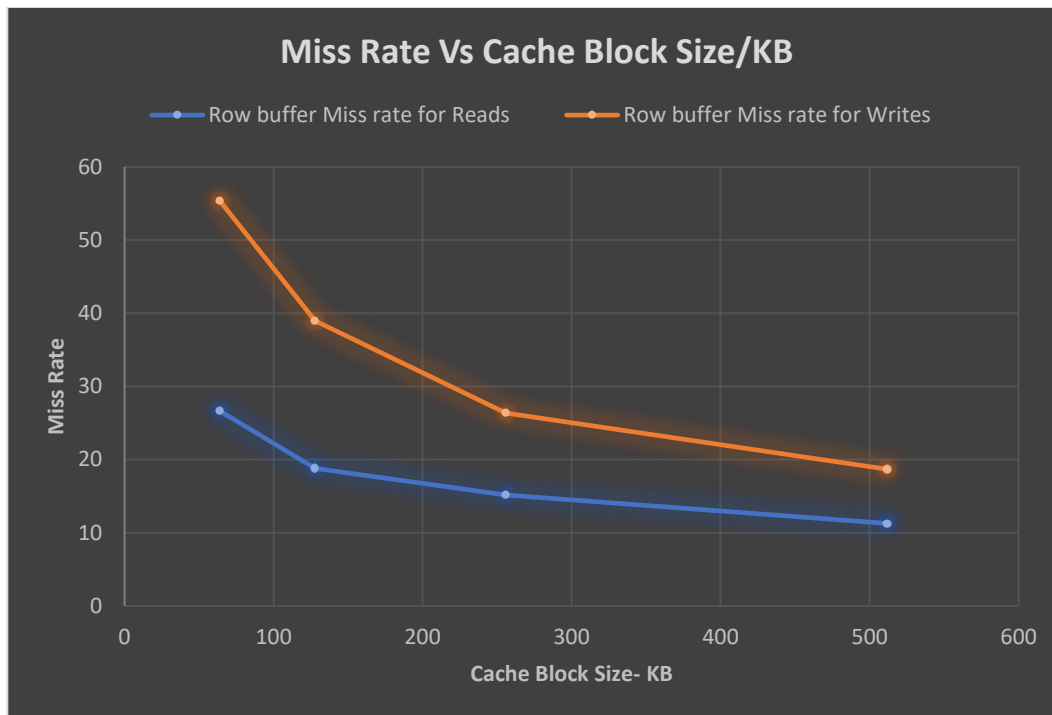


Fig.5

3.4. ANALYSIS- Conclusions

CPU Model:

We discussed in phase 4 about why we choose 'Branch Predictor' as our parameter. We realized that our benchmark 'facesim' was CPU intensive as it involved simulating the expressions on a face. It didn't require much data and showed how the muscles on the face moved. Hence, we picked a parameter such as branch predictor that can ensure pipelining and instruction execution simultaneously without having to wait for long.

We wanted to identify and exploit ILP – instructions that can potentially be executed at the same time and hence the concept of branches which form 15-20% of instructions is very important. We had decided to change the predictor size of the tournament predictor which was by default being used by gem5 in O3- Out of order full system (FS) mode.

Tournament predictors were developed as a motivation for correlating branch predictors as 2-bit local predictor failed on important branches; by adding global information, performance improved. Tournament predictors basically use two predictors, 1 based on global information and 1 based on local information, and combines this with a selector. This is done in hope to select right predictor for right branch (or right context of branch).

We modified the local predictor's size and since we were increasing its capacity, we expected that the tournament predictors 's prediction will improve as it uses information from both local and global predictors.

This meant the predictor's miss rate will go down.

Conclusion:

Our experiment proved this theory. When we increased the size of the local predictor from 512 bytes to 4096 bytes, we observed that the conditional misprediction as well as indirect misprediction rate went down and rightly so as now the predictor's capacity has been increased.

Hence it can be rightfully said that increasing local predictor size for dynamic branch prediction helps in decreasing miss rate and give better accuracy.

Another important parameter we observed was IPC – Instructions per cycle and execution rate, these 2 did improve slightly before reaching a saturation point since there is a limit to a systems maximum performance.

Memory Model:

The memory model and results we expected were easier to comprehend. We took cache block size as our parameter and observed cache miss rate, IPC and execution cycles from the stats.txt files we obtained as a result of our simulations. We expected the miss rate to go down as now the cache block size is greater.

We observed from the values we obtained that the miss rate did indeed decrease as we increased our cache block size as shown in fig.5. The reason is that the cache block size parameter changes the size of both L1 and L2 cache blocks. Since these are now bigger, they have the capacity to store more data hence number of hits increase. Cache hit data is found in cache which results in data transfer at maximum speed. Cache miss data is not found in cache as a result of which the processor loads data from Main memory and copies into cache. This results in extra delay, called miss penalty. And we observe that increasing cache block size avoids this miss penalty and decreases the miss rate.

IPC and execution rate are 2 other metrics we said we will monitor during our experiments. We observed that these 2 numbers increased as we increased the cache block size but they reached saturation soon enough as can be seen in fig.4. This is because maximum performance of the system considering the architectural limitations had been achieved.

REFERENCES:

- [1] <http://parsec.cs.princeton.edu/>
- [2] http://gem5.org/Main_Page
- [3] <https://arxiv.org/pdf/1610.02094.pdf>
- [4] <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5306793&tag=1>
- [5] http://www.utdallas.edu/~gxm112130/EE6304FA17/project2.pdf?fbclid=IwAR3ICrg0jU1rCnrk_uiOrl_kc_5MZD81ldzyeoWpY1CRnBDDyUVl0Qyfc0Y
- [6] https://rextester.com/l/r_online_compiler