



PHASE 4

The Design of the Simulation Experiments



Submitted by: Maliha Arif

PID: 4506817

Submitted to: Dr. Dan C. Marinescu

Submitted on: 25th October 2018

ADVANCED COMPUTER ARCHITECTURE

Phase 4: The design of the simulation experiments

1. INTRODUCTION:

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.[1]

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.

1.1. OUTLINE:

This report gives an outline on the simulation experiment in the following ways.

- 1) Introduction on why designing CMPs is important and what parameters are generally modified.
- 2) What is Parsec and how it is used for designing CMPs(overview)?
- 3) Example of modifying one parameter (e.g. thread) and observing performance change
- 4) Parsec Scaling Properties and Bottlenecks
- 5) Brief introduction of our benchmark (Facesim) and whether it is I/O intensive, memory intensive, CPU intensive and what parameters are relevant for our experiment.
- 6) Which CPU parameter and memory parameter we will use in our experiments and why?
- 7) In-depth explanation of 'Branch Predictors' and their ranges we will be using in our experiments.
- 8) Discussion of the number of runs and confidence interval.

1.1. USING PARSEC FOR DESIGNING CMP's (CHIP MULTIPROCESSORS)

The PARSEC benchmark suite is designed to provide parallel programs for the study of CMP's. It was introduced in 2008 and has been widely used for computer architecture research since then. Developed with the needs of researchers in mind, it has features that make it easier to use with architectural simulators. Each benchmark has multiple input sets, including three that are intended to run with simulators; *simsmall*, *simmedium*, *simlarge*, and one that is intended to be representative of a real application called *native*. This allows users to simulate a smaller workload but obtain results representative of a real workload, this is something which will help us in our experiments as each simulation takes at least 3 hours to run. Each benchmark also defines a Region of Interest (ROI) indicating which part of the benchmark executes in parallel. By simulating only the ROI, PARSEC users can reduce simulation time. The ROI is also important for ensuring that results obtained using simulation inputs are representative of real program behavior [2].

Though we know this, our experiments will be largely based on the *simsmall* input which consists of 1 frame.

Choices in input set size and whether to model the whole program, or only the ROI, can lead to different interpretations when analyzing benchmark results. Below figure shows an example of Blackscholes benchmark when run with a small input set and when run with the full native input set. The parameter being measured is speed up and the parameter being modified is # of threads. The results show that using the whole native input set and small input set changes speed up with different magnitude but most importantly increasing the threads, increase the overall performance. This is what we will be observing when modifying the CPU and Memory model for our benchmark as discussed ahead, the performance metric that change.

Example:

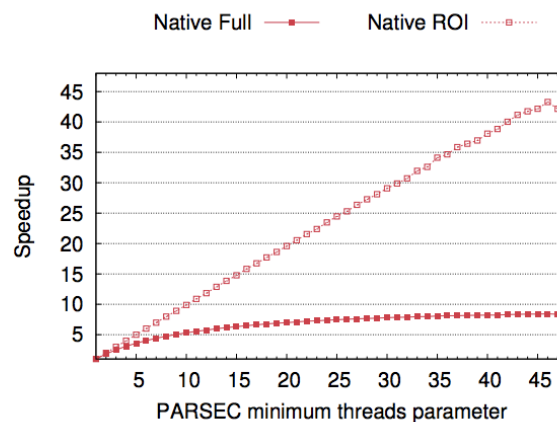


Fig. 1. Speedup for the full execution of Blackscholes is much less than ROI, when using the native input set and running on a 48-core system.

1.2. 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 this report and later 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 [3] 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. Half the suite fails to scale linearly with increasing number of threads, and some applications saturate performance at few threads on all platforms tested. Exploiting thread level parallelism delivers greater payoffs than exploiting instruction level parallelism. To reduce power and improve performance, we recommend increasing the number of arithmetic units per core, increasing support for TLP, and reducing support for ILP.

Our experiments and chosen parameters will confirm the above notion.

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 design our experiment accordingly.

1.3. 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.

1.3.1. Facesim simulation setting

Facesim simulation does an animation of a human face. That face is made of a 3D tetrahedra mesh. For each tetrahedron there are associated forces in terms of stiffness, strains and stress of materials. Before starting any computation, there are some parameters to be set for the simulation. Those parameters form a set of constraints which limit the number of computations spent in each frame of the simulation. The constraints also define constants related with physic characteristics of the model. Physics are computed via the PhysBAM library. That library implements all the functionality related with physics simulation. Facesim loads the 3D model of a human face and sets the needed simulation parameters specified in code by the programmer.

1.3.2. 3D model anatomy

The 3D model is a mesh of tetrahedra and their nodes are what is being animated. The mesh represents the flesh of the head. In each frame, the position of the nodes has to be computed by solving a system of partial differential equations. The forces affecting tetrahedra determinate the final positions of the nodes found by solving the system. Forces affecting each node are a mix of activated muscle forces, elastic response of the flesh and passive forces present in muscle regions. Muscle definition is embedded into the 3D face model. For each muscle that overlaps a tetrahedron, the fraction of overlap is stored in the tetrahedron. For each tetrahedron, vectors with the same direction as the muscles overlapping it along the fraction of overlapping is stored.

One vector per muscle. That fraction scales both the active and passive forces derived from the activation of a muscle overlapped with the tetrahedron. All of this data is already included in the anatomical model and Facesim only loads it.



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

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

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.

1.4 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.

1.5. 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) and execution cycles** are other parameters we will be observing when changing the branch prediction parameter.

1.6. TYPES OF BRANCH PREDICTIONS

So, there is static branch prediction and dynamic branch prediction.

Static Branch Prediction:

Static Branch Prediction predicts always the same direction for the same branch during the whole program execution. It comprises hardware-fixed prediction and compiler-directed prediction. Simple hardware-fixed direction mechanisms can be:

- Predict always not taken
- Predict always taken
- Backward branch predict taken, forward branch predict not taken

Dynamic Branch Prediction:

In dynamic Branch Prediction, the hardware influences the prediction while execution proceeds. Prediction is decided on the computation history of the program. During the start-up phase of the program execution, where a static branch prediction might be effective, the history information is gathered and dynamic branch prediction gets effective. In general, dynamic branch prediction gives better results than static branch prediction, but at the cost of increased hardware complexity

Dynamic branch prediction schemes are different from static mechanisms because they utilize hardware-based mechanisms that use the run-time behavior of branches to make more accurate predictions than possible using static prediction. Hence, we will be working with Dynamic Branch predictors in our experiments.

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 will go for the latter option and pick the **tournament or hybrid predictor** which is the default one in gem5 and modify its parameters. (size mostly)

1.7. CPU PARAMETER TO MODIFY AND ITS RANGE

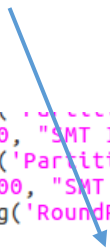
In this experiment, we will use GEM5 to investigate how architecture affects performance. We will use the default cache configuration--l1size 32kB --cache block size 64and modify the base processor configuration.

The architecture is defined in~/gem5-stable/src/cpu/o3/O3CPU.py.

When we build an architecture using scons, it reads this file.

Here we will keep the Tournament branch predictor as follows:

O3CPU.py



```
smtIQPolicy = Param.String('Partitioned', "SMT IQ Sharing Policy")
smtIQThreshold = Param.Int(100, "SMT IQ Threshold Sharing Parameter")
smtROBPolicy = Param.String('Partitioned', "SMT ROB Sharing Policy")
smtROBThreshold = Param.Int(100, "SMT ROB Threshold Sharing Parameter")
smtCommitPolicy = Param.String('RoundRobin', "SMT Commit Policy")

branchPred = Param.BranchPredictor(TournamentBP(numThreads =
                                                Parent.numThreads),
                                   "Branch Predictor")
needsTSO = Param.Bool(buildEnv['TARGET_ISA'] == 'x86',
                     "Enable TSO Memory model")

def addCheckerCpu(self):
    if buildEnv['TARGET_ISA'] in ['arm']:
        from ArmTLB import ArmTLB
```

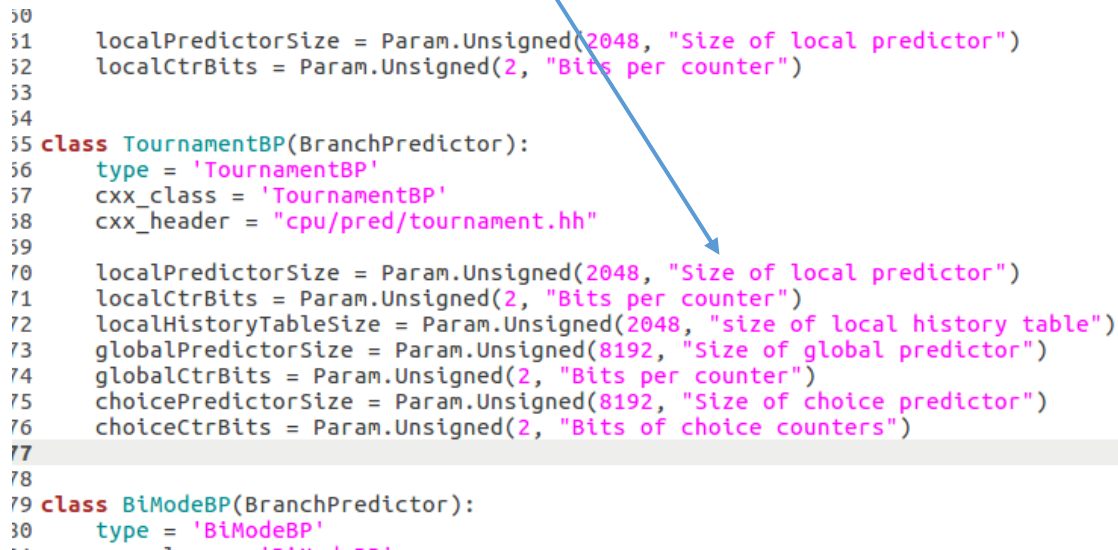
While keeping the tournament branch Predictor, we will modify the **Predictor size**, this variable is present in another file.

“\$gem5/src/cpu/pred/BranchPredictor.py”

Predictor size (Bytes)
512
1024
2048
4096

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

```

1.8. PERFORMANCE PARAMETERS TO STUDY

Below is the “stats.txt” file that shows what parameters we need to observe when we change our input. Screenshot is on the next page.

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

Here is the screenshot and the performance metric we will observing.

of Branch mispredictions
percent of Branch Mispredict
BTB Miss percentage

```
./stats.txt:system.cpu.branchPred.lookups      1317    # Number of BP lookups
./stats.txt:system.cpu.branchPred.condPredicted 1317    # Number of conditional
branches predicted
./stats.txt:system.cpu.branchPred.condIncorrect 621     # Number of conditional
branches incorrect
./stats.txt:system.cpu.branchPred.BTBLookups    203     # Number of BTB lookups
./stats.txt:system.cpu.branchPred.BTBHits       202     # Number of BTB hits
./stats.txt:system.cpu.branchPred.BTBCorrect    0       # Number of correct BTB
predictions (this stat may not work properly.
./stats.txt:system.cpu.branchPred.BTBHitPct     99.507389 # BTB Hit Percentage
./stats.txt:system.cpu.branchPred.BTBMissPct    0.492611 # BTB Miss Percentage
./stats.txt:system.cpu.branchPred.usedRAS       8       # Number of times the
RAS was used to get a target.
./stats.txt:system.cpu.branchPred.RASInCorrect  0       # Number of incorrect
RAS predictions.
./stats.txt:system.cpu.Branches                 1317    # Number of branches
fetched
./stats.txt:system.cpu.predictedBranches        210     # Number of branches
predicted as taken
./stats.txt:system.cpu.BranchMispred            621     # Number of branch
mispredictions
./stats.txt:system.cpu.BranchMispredPercent     47.152620 # Percent of Branch
Mispredict
```

The highlighted parameters are not there by default and you need to add them to the source files and recompile to generate the new “stats.txt” file with the parameters.

As discussed in section 1.5, we will be observing these 3 performance metrics and plotting graphs against them.

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

So far, we have talked about the CPU model, I will be studying. The memory model that I will be modifying is L1 cache block size.

1.9. MEMORY MODEL PARAMETER TO MODIFY AND STUDY

The parameter I am going to study is

- **L1 Cache Block Size**

Range would be as follows:

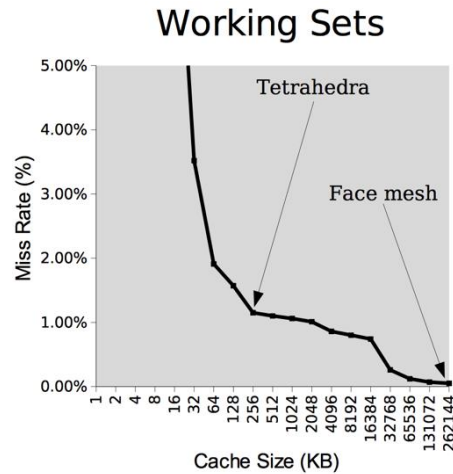
Block size /KB
32
64
128
256
512

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

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

To note, we will keep 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.

As per the official site of Parsec benchmarks, facesim benchmark should show this trend, where increasing the cache size reduces the miss rate as follows. We will comment on this once we perform the experiments.



1.10. 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.

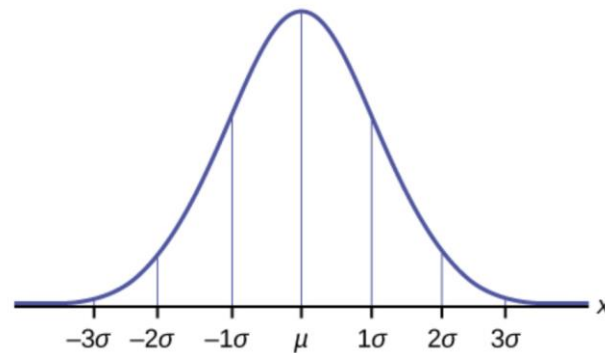
Note that since the simulation takes a lot of time, we might not be able to repeat the experiments a large number of times but we intend to repeat the **experiment 5 times** for different size of CPU parameter and memory parameter as our sample data.

The method to be employed is briefly explained as follows:

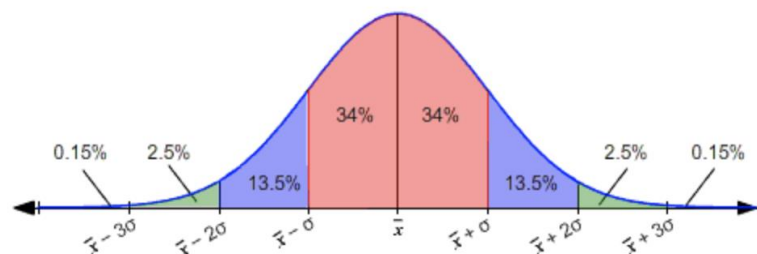
Confidence interval:

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 5 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.
- Though we are taking finite number of samples (5 in our case), if we take infinite number of samples, we will obtain a bell shaped curve as follows



If we know this, the mean (e.g 64 put of 100 samples, let's say – our case will be different in actual) (and we know the standard deviation) we are able to say that ~64% of the samples will fall in the red area or, more than 95% of the samples will fall outside the green area in this plot:



Hence, having 95% confidence in our experimented values. This technique allows us to be more certain of our experiments and if we are calculating everything correctly. We will employ this method while experimenting in phase 5.

References:

- [1] <https://arxiv.org/pdf/1610.02094.pdf>
- [2] https://users.soe.ucsc.edu/~gsouther/papers/wddd2015_deconstructing_parsec_scalability.pdf
- [3] <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5306793&tag=1>
- [4] https://upcommons.upc.edu/bitstream/handle/2099.1/24827/103589.pdf?fbclid=IwAR3ICrg0jU1rCnrk_uiOrl_kc_5MZD81ldzyeoWpY1CRnBDDyUVl0Qyfc0Y
- [5] http://www.utdallas.edu/~gxm112130/EE6304FA17/project2.pdf?fbclid=IwAR3ICrg0jU1rCnrk_uiOrl_kc_5MZD81ldzyeoWpY1CRnBDDyUVl0Qyfc0Y