UNIVERSITY OF CALIFORNIA
SANTA CRUZ

GOING LIVE IN MICRO-ARCHITECTURE SIMULATION

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Sina Hassani

December 2015

The thesis of
Sina Hassani is approved:

Professor Jose Renau, Chair

Professor Matthew Guthaus

Professor Jishen Zhao

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

Abstract

Going Live in Micro-Architecture Simulation

by

Sina Hassani

Computer architects rely on simulators in order to explore their design space and evaluate innovations. However, the state-of-art cycle-accurate simulators are several orders of magnitude slower than the hardware they simulate. Long simulation times can result in a great decrease in productivity. We have developed LiveSim, a novel microarchitectural simulation methodology that provides simulation results within seconds, making it suitable for interactive use. LiveSim is able to report simulation results on the fly by incorporating a web interface. It is a scalable framework which efficiently takes advantage of resources to provide fast simulation results in an order of seconds.

LiveSim works by creating in-memory checkpoints of application state, and then executing randomly selected samples from these checkpoints in parallel to produce simulation results. The initial results, which we call LiveSample, are reported less than one second after starting the simulation. As more samples are simulated the results become more accurate and are updated in real-time. Once enough random samples are gathered, LiveSim provides confidence intervals for the reported values and continues simulation until it reaches the target confidence level. We call simulation towards a target interval LiveCI.

We evaluated LiveSim using SPEC CPU 2006 benchmarks and found that within 5 seconds after starting simulation, LiveSample results reached an average error of 3.51% com-

pared to full simulation, and the LiveCI results were available within 41 seconds on average.

Acknowledgments

Professor Jose Renau

For so many things I have learned from him, his guidance and funding.

Professor Matthew Guthaus

For his continued feedback and valuable thesis direction.

Professor Jishen Zhao

For her continued feedback and valuable thesis direction.

Gabriel Southern

For his great help in developing, implementing and publishing LiveSim.

Ethan Papp

For his great contributions in LiveOS which is the base infrastructure used to host LiveSim.

Chapter 1

Introduction

Computer architects are constrained by the fact that system design is a slow, expensive, and time-consuming process. To ameliorate this architects use a variety of techniques to prototype ideas and perform design space exploration. One of the most important techniques is architectural simulation where a software model of the simulated system is developed to evaluate its performance using realistic benchmarks. Unfortunately, software simulation is many orders of magnitude slower than the real systems being designed. This limits the length of the benchmarks that can be executed, and also forces architects to wait for long periods (from minutes to days) until new simulation results are ready.

Many techniques have been developed to reduce simulation time including: benchmarks workload reduction [19], specialized hardware [30], phase-based sampling [27], and statistical sampling [32]. Of these techniques, the sampling based approaches typically provide the best trade-off between simulation fidelity, speed, and flexibility.

The state of the art simulation techniques have reduced simulation time from weeks

to days or hours, but in many ways microarchitectural simulation still uses the methodology of the era of punched cards and batch queues. An architect typically first develops and configures the simulation parameters. Then the simulation is submitted to a batch queue and runs for hours or days while the architect works on something else. After the simulation finishes execution the architect must recall what she was working on, interpret the results and then repeat the cycle with new experiments.

This methodology contrasts with the rapid development techniques popular in many types of software engineering. We expect the productivity of computer architects to improve with an interactive development environment. To support this development model we propose LiveSim, a simulation framework that provides simulation results in near real-time. In this work we define our near real-time goal as within 5 seconds and we use the terms live and interactive[1] interchangeably. LiveSim provides initial results as soon as the first sample finishes simulation, and it provides a confidence interval to bound the expected error after a minimum number of samples have executed (which is typically within 5 seconds on our system). If necessary LiveSim continues simulating more samples until reaching a user defined threshold for the confidence interval. We call the results that are updated in real-time LiveSample, and the result that is within the desired confidence interval LiveCI.

LiveSim works by first running a setup phase that executes the applications in emulation only mode and creates in-memory checkpoints with architectural state. This step is completely independent of simulated microarchitecture. Next the microarchitecture simulator is loaded as a dynamic library, which allows it to be easily modified without rerunning the costly

[1] The simulator is used interactively, not the benchmark that is simulated

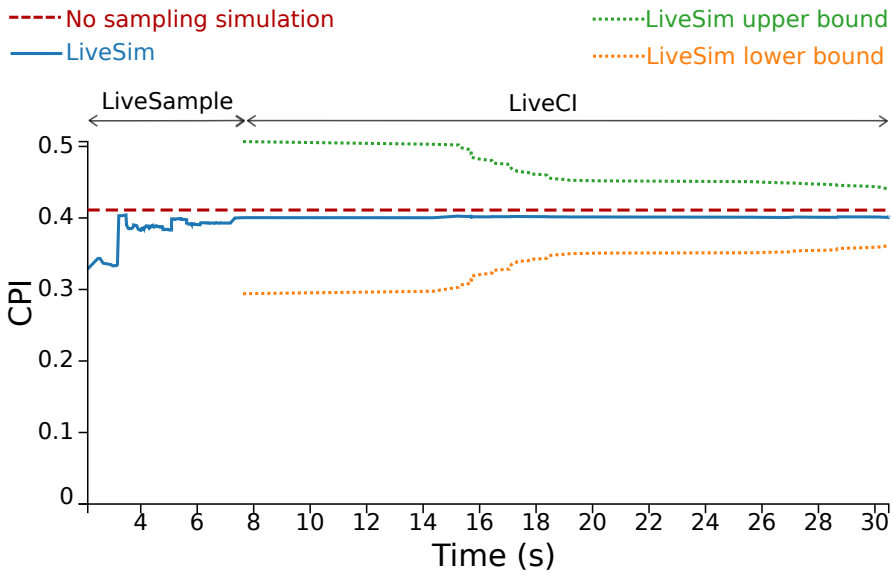

Figure 1.1: LiveSim timeline showing how the user is presented with LiveSample results from the beginning, how accurate the results get within a few seconds, and how the simulation continues running until the confidence interval reaches the threshold for the LiveCI results.

setup phase. Then a calibration phase runs which executes a sample from each checkpoint using the current simulator configuration. The samples are used to characterize the checkpoint and allow for clustering. Although the measured performance depends somewhat on the simulated microarchitecture, in practice the clustering tends to be associated with program phases, and as a result the calibration phase tends not to need to be repeated even if the simulated microarchitecture has radical changes. At this point LiveSim is ready for interactive use allowing the user to experiment with changes to the simulated microarchitecture. After making a change to the simulator the user requests new simulation results. LiveSim randomly selects the minimum number of checkpoints and begins simulating samples from the selected checkpoints in parallel and reporting the LiveSample results to the user. After meeting the cutoff for the LiveCI results the simulation halts and reports the final results.

Figure 1.1 illustrates an example of how LiveSim works by showing the simulation

result of running Namd benchmark from SPEC2006 for approximately 10 seconds in a Haswell CPU and simulating a similar CPU with LiveSim. Unlike traditional simulators, LiveSim starts to produce results as each simulation sample finishes (LiveSample). As the evaluation will show, after 5 seconds it is able to provide results that are within 4% of the correct result. The correct result is a full simulation of the 10 seconds without sampling shown as No Sampling Simulation in the figure. Once enough samples are gathered, it is possible to start reporting the confidence interval for the simulation. As more samples are added, the confidence interval decreases and stops the simulation when enough samples are processed. While LiveSample provides accurate results in a very short time, it cannot guarantee them. Live Confidence Interval (LiveCI) bounds the error according to the user requested acceptable error.

In this work we make the following novel contributions:

- Introduce LiveSim, a new architectural simulation methodology that enables interactive microarchitecture design space exploration.
- Demonstrate that LiveSim is able to provide very accurate LiveSample simulation results within 5 seconds. These results are independent of the length of the simulated benchmark and simulating more instructions does not increase the time it takes for LiveSim to provide the LiveSample results.
- Demonstrate that LiveSim is able to produce LiveCI results that bound the simulation error within 10% within 41 seconds on average.

The rest of this thesis is organized as follows: Chapter 2 provides some background about existing techniques to speed up simulation; Chapter 3 explains how LiveSim works;

Chapter 7 details the setup of our evaluation framework; Chapter 8 describes our results; Chapter 9 provides a comparison with related work; and Chapter 10 concludes.

Chapter 2

Background

2.1 Sampling in Micro-Architecture Simulation

Most architectural simulators are implemented as discrete event simulators where the simulator models the changes to microarchitectural state that occur each clock cycle while the simulated processor executes an instruction. Simulating a single instruction can require the host to execute thousands of instructions to update all of the simulated microarchitectural state for an advanced out-of-order processor, and even fast simulators are thousands of times slower than native execution.

There are a variety of ways to cope with the slow simulation speed. One is to simulate benchmarks that execute very small numbers of instructions; however, it is difficult to ensure that these results are comparable to those obtained with standard benchmark inputs [16, 19]. Another approach is to accelerate the timing simulation using FPGAs [6, 7, 30], but this requires custom hardware, increases simulator development complexity, and is not widely used in prac-

tice. The most popular approach is to use sampling to reduce the number of instructions that need to be simulated, and this is the technique we use for LiveSim.

Many simulators have a variety of levels of simulation detail ranging from the most detailed mode which models all microarchitectural details, to modes that only simulate structures with long lived state (such as caches or branch predictor), to emulation-only mode, or even modes that run parts of the simulated benchmark directly on the host system [33]. As the level of detail decreases, the speed of the simulation increases. Most sampling techniques take advantage of this difference in simulation speed by executing the majority of instructions in a faster simulation mode, and extrapolating statistics collected from a small percentage of instructions executed using full detailed simulation mode. The two main sampling techniques are profile based sampling and statistical sampling.

Profile based sampling attempts to identify a few regions of a benchmark that are representative of the behavior of the full benchmark execution. Sherwood et al. [27] developed SimPoint, which works by profiling a benchmark and collecting information about the distribution of basic blocks executed during benchmark execution. This information is used to find phases in program execution and then select representative samples for each phase. The statistics that are that are collected from these samples can be used to extrapolate results that tend to be very close to those from full benchmark execution. The effectiveness of the original SimPoint proposal was purely heuristic based, but Perelman et al. [24] extended SimPoint to provide statistical confidence measures.

Wunderlich, et al. [32] developed the SMARTS framework, which applies statistical sampling theory to computer architecture simulation. The main drawback with SMARTS is that

it requires continuous updates to the simulated cache and branch predictor microarchitectural state between sampling units. The simulation mode that updates this state is called functional warming, and while it is faster than detailed simulation, it is still much slower than native execution. Although SMARTS is the de facto reference for applying statistical sampling to microarchitecture simulation, earlier work from Conte et al. [8,9] also explored using statistical sampling with microarchitecture simulation.

Since functional warming of the cache dominates simulation time there have been a variety of proposals to reduce the amount of warmup that is needed and to speed up the emulation mode. One technique is to save some of the simulation state in a checkpoint and then load this checkpoint during future simulation runs [31]. Another technique is to forgo detailed cache modeling during the functional warmup phase and simply record the sequence of memory operations. This information can then be used to quickly rebuild the cache state prior to detailed simulation of a sampling unit [1].

LiveSim builds on existing work that uses sampling to accelerate microarchitecture simulation, but rather than simply trying to make the simulation faster, LiveSim is designed to be suitable for interactive use. LiveSim uses statistical sampling, in-memory checkpoints, checkpoint clustering, parallel checkpoint execution, and a fast cache warmup technique in order to support interactive simulation.

2.2 Statistical Sampling Theory

Statistical sampling can be applied to any situation when we are trying to make estimation without evaluating an entire population for reasons like cost, time, etc. The same case applies in micro-architecture simulation: it is very time-taking to fully simulate and evaluate applications. In the previous chapter, we discussed how this time-taking process can possibly affect productivity and the benefits of making it live.

In statistical sampling, we rely on a number of randomly drawn elements from the original population. The drawn elements form our sample distribution. The sample distribution parameters (m for mean, s for standard deviation, ...) are referred to as point estimates. If our sample distribution is large and representative enough, we can generalize the point estimates to represent the parameters of the original population.

How to make sure representative-ness of samples depends on the characteristics of the population and the way we take samples from it. A general rule of thumb is that taking one sample should not have any effect on taking another (no correlation between samples). In addition, samples should be taken completely randomly and be equal in size/weight. In the case of micro-architecture simulation, samples are chunks of the application code (e.g. 100K instructions) randomly picked from the whole application code (e.g. 50B instructions). To ensure representative-ness, we have to make sure our sampling mechanism gives any possible sample the same chance of being selected (randomness) and each sampling trial must be independent. More specific to micro-architecture simulation, each sample must have the same behavior as it had in the original population. In other words, the way that samples are executed and evaluated

should ensure that produced sample results are the same as when that chunk of code is evaluated in the whole application execution.

How many samples to take is a question of how precise we would like the point estimates to be and how much error we can tolerate. A good way of measuring the possible error amount is calculating confidence interval around the point estimates. In micro-architecture simulation we are interested in average results. Therefore in this context, we focus on calculating mean and confidence interval around the mean.

Confidence interval around mean is defined as: In a large number of sampling trials $(1-\alpha)$ proportion of the mean results fall into a $\pm\epsilon$ interval of the original population mean. This interval is defined as confidence interval and α is the probability of having a sample mean outside of it. In theory, confidence interval can be calculated using the following equation:

$$CI = \mu \pm z\frac{\sigma}{\sqrt{n}}$$

Where z is the $100[1-(\alpha/2)]$ percentile of the standard normal distribution, μ is the population average, σ is the standard deviation of the population and n is the number of samples in each trial.

In one sampling trial, we have a $(1-\alpha)$ probability of being in the calculated $\pm\epsilon$ of the true population mean. This is equivalent of saying that we have $(1-\alpha)$ probability that the actual population mean lies within a $\pm\epsilon$ interval around the sample (estimated) mean. Using this theory with user defined α, we are able to present the confidence interval that the actual result lies at any point.

The theory of confidence interval supports any population when having large enough number of samples (e.g. 30) where the sampling distribution (the distribution of sample means in a large number of trials) is normal. Having such population, it is safe to use the sampling population parameters (point estimates) instead of the original population parameters that we do not have access to. As a result, we are able to calculate the confidence interval as follows:

$$CI = mean \pm z\frac{S}{\sqrt{n}}$$

Knowing our target confidence level ($(1 - \alpha)$) and confidence interval (ϵ), we can start with a minimum number of samples (e.g. 30) that allows calculating confidence interval. Then we can add more samples and re-calculate confidence interval until we reach the desired level.

Chapter 3

LiveSim Methodology

The previous section provided some background about the sampling techniques that LiveSim leverages to enable interactive or Live simulation. In this section we explain in detail exactly how we implemented LiveSim. The basic outline for how LiveSim works is as follows:

- **Setup**: Run simulated benchmark in emulation only mode and periodically create in-memory checkpoints that contain architecturally visible state. Information about the sequence of memory accesses is also recorded during checkpoint creation and used later for cache warmup. The state contained in the checkpoints is independent of the simulated microarchitecture.

- **Calibration**: Execute a sample from each of the checkpoints and use the recorded statistics to group the checkpoints into clusters.

- **LiveSample**: When the user requests new simulation results LiveSim begins executing the minimum number of checkpoints in parallel. As soon as simulation results are ready

they are reported visually to the user.

- **LiveCI**: After the minimum number of checkpoints complete execution LiveSim monitors the calculated confidence interval. If it is not within the user specified limit then LiveSim randomly selects more checkpoints to run. Eventually the confidence interval reaches the selected limit and the simulation halts.

In LiveSim checkpoints contain all of the architecturally visible state necessary to start simulation from a specific point in the benchmark and can be reused multiple times with many different simulator configurations, whereas samples represent the result of simulating a specific microarchitecture for a specific checkpoint. Samples are created by first copying the checkpoint state, next loading the simulator library and configuration for the microarchitecture that is being simulated, next warming up the microarchitecture state, and finally collecting statistics for the sample.

In the rest of this section we explain in detail how each of the steps in LiveSim works.

3.1 Sampling Setup

The field of inferential statistics provides well known techniques for inferring statistics about a population given a sample of that population. SMARTS [32] demonstrated that systematic sampling can be used to approximate random sampling when used with microarchitectural simulation. LiveSim also uses systematic sampling to approximate random sampling during the setup phase.

During the setup phase LiveSim runs a fast emulation-only process that periodically

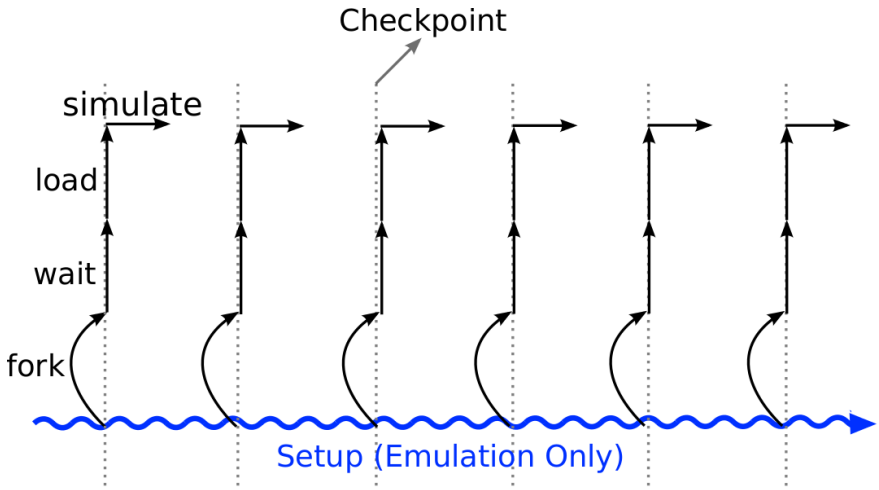

Figure 3.1: During setup the benchmark is emulated and checkpoints are created by periodically forking new processes. Each checkpoint process enters a waiting mode. Once LiveSim starts it loads the simulation dynamic library and configurations, then it starts simulation.

forks copies of itself to create an in-memory set of checkpoints that contain all the architecturally visible state necessary to continue benchmark execution. Figure 3.1 illustrates how these newly created checkpoints enter a wait mode listening for commands to start microarchitectural simulation. Since forking a process uses copy-on-write, the checkpoint creation step is relatively cheap during the setup phase. However, in our implementation each checkpoint uses up to 100 MB phase starts, and if LiveSim has to go to swap then it tends to run too slowly to meet our desired time targets. In Section 8.4 we evaluate how to determine the optimal number of checkpoints and checkpoint size.

In addition to the architecturally visible state, a checkpoint also needs a way to warm up the microarchitectural state before collecting statistics from a sample. In our experiments we found LiveSim was able to warm up most of the microarchitectural state, including an advanced O-GHEL branch predictor, with only 1 million instructions of warmup. However, effectively warming up the cache required more than 60 million instructions on average, and research

indicates larger caches may require even more warmup [23]. Executing this many instructions to warm up a sample would make the simulation too slow to meet the 5 second near real-time targets for LiveSim.

To solve the cache warmup problem we adapted the memory timestamp record (MTR) technique proposed by Barr et al. [1], and we call our adapted cache warmup technique LiveCache. LiveCache is implemented as a very large and highly associative cache that is larger than the largest cache that will ever be simulated. Each cache line in LiveCache has a counter field which stores a timestamp of the most recent access to this line. Each memory operation increases this timestamp and stores it in the counter field of the cache line it is accessing. These counters provides an ordering of all locations that could possibly be in the cache. Maintaining this state is has a low overhead and does not slowdown the LiveSim setup process very much, it will automatically be made available to the newly spawned checkpoint when they are forked, and it is independent of the microarchitecture that will be simulated later.

When LiveSim starts simulating a sample from a checkpoint it first loads the simulator's dynamic libraries and configuration information. Next LiveSim executes all memory operations saved in the checkpoint's LiveCache in least recently used order without advancing the clock or collecting any statistics. This warms up the sample's microarchitectural cache state. After all of the LiveCache memory operations are executed the real simulation starts. This is similar to the technique proposed by Barr et al. [1] with some slight changes to simplify the integration with the LiveSim simulation infrastructure.

3.2 Calibration

The central limit theorem is the underlying foundation for the statistics theory which allows us to approximate the distribution of sample averages as though it were normally distributed. A typical rule of thumb is that 30 samples are enough to apply the central limit theorem when the population that is sampled from is not highly skewed. But if the population is highly skewed then more samples are required.

Figure 3.2 shows a trace of CPI values for the Astar benchmark plotted over time for the first 30 billion instructions of the benchmark execution. This distribution is highly skewed for two reasons. First the minimum CPI in the simulated 4-wide system is 0.25, but the maximum CPI is effectively unbounded and we can see spikes as high as 10 for this benchmark. Second the spikes are relatively rare and most of the samples have a CPI much closer to 1. Simply using random sampling can require hundreds of samples for a distribution that is this highly skewed. Furthermore, while we can calculate the number of samples needed if we are given the population distribution, this information is not known a priori. Thus random sampling alone is unable to meet the execution time constraints of LiveSim.

However, LiveSim is able to take advantage of the correlation between code signatures and performance [20] and use this information to cluster the checkpoints. When the results for LiveSample are calculated, LiveSim ensures that each cluster has at least 1 sample that is selected, and it also weights the results from each cluster based on the cluster size. This technique has some similarities to what Perelman et al. [24] do to statistically bound the error for results obtained using SimPoint.

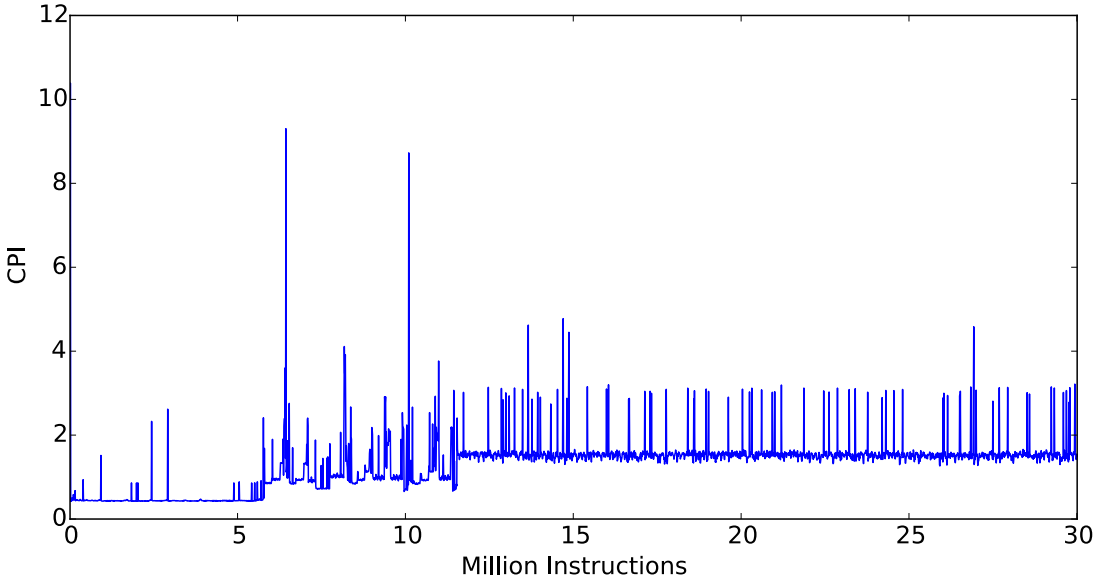

Figure 3.2: The CPI trace of Astar benchmark in SPEC 2006. The infrequent but extremely large spikes will have a considerable affect on the average CPI. Missing these spikes in random sampling will result in an unreliable sample mean

LiveSim groups the checkpoints based on the performance statistics that are obtained with the baseline simulator configuration. Since performance statistics correlate with code signatures this grouping tends to cluster the checkpoints together in a way where even radical changes in the simulated microarchitecture still causes the checkpoints to be clustered in a similar way. The clustering does not need to be exactly the same for different simulated microarchitectures, just close enough to avoid problems with extreme outliers that may otherwise skew the results.

After all of the checkpoints are spawned during setup LiveSim begins calibration. For each checkpoint LiveSim loads the baseline simulator configuration and simulates a sample from the checkpoint. After all of the checkpoint samples are collected LiveSim uses a clustering algorithm to group the checkpoints into clusters.

For clustering LiveSim uses K-means algorithm where K ranges from $K = 1$ to $K = numcheckpoints/2$ and LiveSim attempts to find the value of K that provides the optimal trade-off between variation of samples in clusters and the number of clusters (with the goal of minimizing both of these values). For each iteration of K LiveSim finds the best possible grouping of checkpoints to minimize the total variation of the metric of interest (typically CPI) across all K clusters. As LiveSim iterates through different values of K it keeps track of the value of K (and the associated configuration) seen thus far that minimizes total variation. When the algorithm finishes LiveSim uses the value of K that minimized total variation as the selected configuration for clustering checkpoints.

The final step in calibration is to assign weights to each cluster. This is done based on the number of checkpoints that are assigned to each cluster. For example if there were 2 cluster, and the first cluster had 900 checkpoints, while the second cluster had 100 checkpoints, then the first cluster would have a weight of 0.9 and the second cluster would have a weight of 0.1. These weights are used when averaging results obtained from simulating samples from these checkpoints and reporting LiveSample results.

3.3 LiveSample

Once the setup and calibration phases are complete LiveSim is ready for interactive use. The usage scenarios that we envision is that the setup and calibration phases can be completed when the architect is not actively using the simulator (similar to how simulation batch jobs are run today). The LiveSample and LiveCI results are what the architect would be inter-

ested in seeing while using LiveSim for Live simulation. An architect may make a configuration change and then request results from LiveSim.

At this point LiveSim randomly selects the first batch of checkpoints to simulate. The selection algorithm depends on the number of clusters and the computation resources of the system used for running the simulation and is shown in Algorithm 1. The reason that we used spawn at least twice as many checkpoints as cores is that it provided the highest sample execution throughput on our system. Too many running samples will overload the computation resources of the system, while too few limit opportunities for overlapping computation with I/O. This part of the initial checkpoint selection algorithm could be tuned differently for different systems, but it is important to ensure at least one checkpoint is executed from each cluster, regardless of the amount of samples that the system can execute in parallel.

for *all clusters* **do**
  randomly select 1 checkpoint from the chosen cluster;
end
while *num selected checkpoints ¡ num cores * 2* **do**
  randomly select 1 checkpoint from any cluster
end

Algorithm 1: Initial checkpoint selection algorithm

The selected checkpoints are contacted by the LiveSim controller process and each selected checkpoint forks another copy of itself to run the simulation, while the parent checkpoint process goes back to its waiting mode. The child processes that will execute a sample loads the simulator library, initializes the cache state using the LiveCache data, warms up the rest of the microarchitectural state using detailed warmup, and finally collects statistics for its sample and the reports them to the LiveSim controller process.

LiveSim begins reporting simulation statistics to the user as soon as the first checkpoint finishes execution. These statistics are calculated by computing the arithmetic mean of the sample values, after weighting each sample by its cluster weight. These results are what we call the LiveSample results and in our experiments they typically reached a steady state value within 5 seconds of starting the simulation.

3.4 LiveCI

Figure 1.1 shows an example of how LiveSim produces LiveSample and LiveCI results for a users. The LiveSample result is provided as soon as the first checkpoint is simulated, and it typically reaches a steady state value very quickly (at roughly 3 seconds in Figure 1.1). However, the initial LiveCI results take slightly longer before they are available and the simulation continues running until the LiveCI result reaches the users specified threshold.

When LiveSim selects checkpoints to take samples from for the LiveSample results it first ensures that at least 1 sample is selected from each cluster. Afterwards it begins selecting checkpoints completely randomly. However, for calculating the confidence interval these samples that were initially selected from clusters cannot be used. The reason is that the confidence interval calculation requires samples to be chosen randomly from the population, and the clustering violates this requirement.

Consequently when LiveSim selects checkpoints for the LiveCI results it starts by selecting 30 completely random checkpoints to take samples from. If any of the checkpoints happen to have already been simulated then the earlier sample results can be used directly.

Otherwise the LiveCI results have to wait until at least 30 completely random samples have been simulated. LiveSim requires a minimum of 30 samples before calculating the confidence interval because 30 is a generally accepted heuristic as the minimum cutoff value for applying the central limit theorem to assume that the sample mean distribution is normally distributed. And a normal distribution is required in order to calculate the confidence interval.

However, the minimum value of 30 is simply a heuristic, and in a highly skewed distribution it may not be enough. In some of our initial experiments we observed that this could result in the confidence interval being reported as more precise than it really was. This happened in cases with a population that was mostly homogeneous, but had a few large spikes (such as the Astar example shown in Figure 3.2). If the initial set of samples did not contain one of the spikes the samples variance could be very small which would lead to a very tight confidence interval being calculated for a sample mean that did not match the true sample mean of the population. On the other hand if the initial set of samples did contain a spike the confidence interval would be very large and outside of the user defined range. In this case the simulation would continue running and eventually enough samples would be collected so that an accurate sample mean and confidence interval was calculated.

So the only problem with the 30 sample minimum heuristic was that sometimes the simulation could end earlier than it should have because LiveSim missed one of the infrequent spikes. To solve this problem we developed a heuristic where we inserted a synthetic sample in the sample set when calculating the confidence interval. The synthetic sample was created by computing the average of the samples collected thus far and multiplying this average by 10. Then the confidence interval was calculated using the true samples as well as the synthetic one.

This heuristic solved the problem of ending the simulation too early due to missing extreme outlier values and it works well in practice for the SPEC CPU 2006 benchmarks that we simulated. In the event that a users was simulating a workload with even more extreme variation in sample metric then a different heuristic might be required, but we expect that adding a synthetic point that is 10 times the sample average should work well enough for most workloads that computer architects simulate.

Figure 3.3 shows the high level sampling mechanism of LiveSim. As discussed before, LiveSim starts presenting the LiveSample results to the user from the beginning and as soon as enough samples are simulated, it starts reporting confidence intervals and continues simulation towards a higher accuracy.

3.5 Delta Sampling

A majority of micro-architecture simulations are performed to evaluate the effects of a change in the architecture. For example if a change in L2 associativity will result in a speedup. If the two simulations (before and after the change) use the same random samples, we are able to form a sample population of ratios which provides an accurate sample mean (the overall ratio) and a relatively smaller confidence interval around it. In other words, if we are trying to simulate a change, we might not be interested in having a precise estimate for the absolute metric results, rather we would like to ensure the estimated speedup/slowdown is bounded by a small error range. In such cases, delta simulation will provide higher accuracy while increasing the simulation speed.

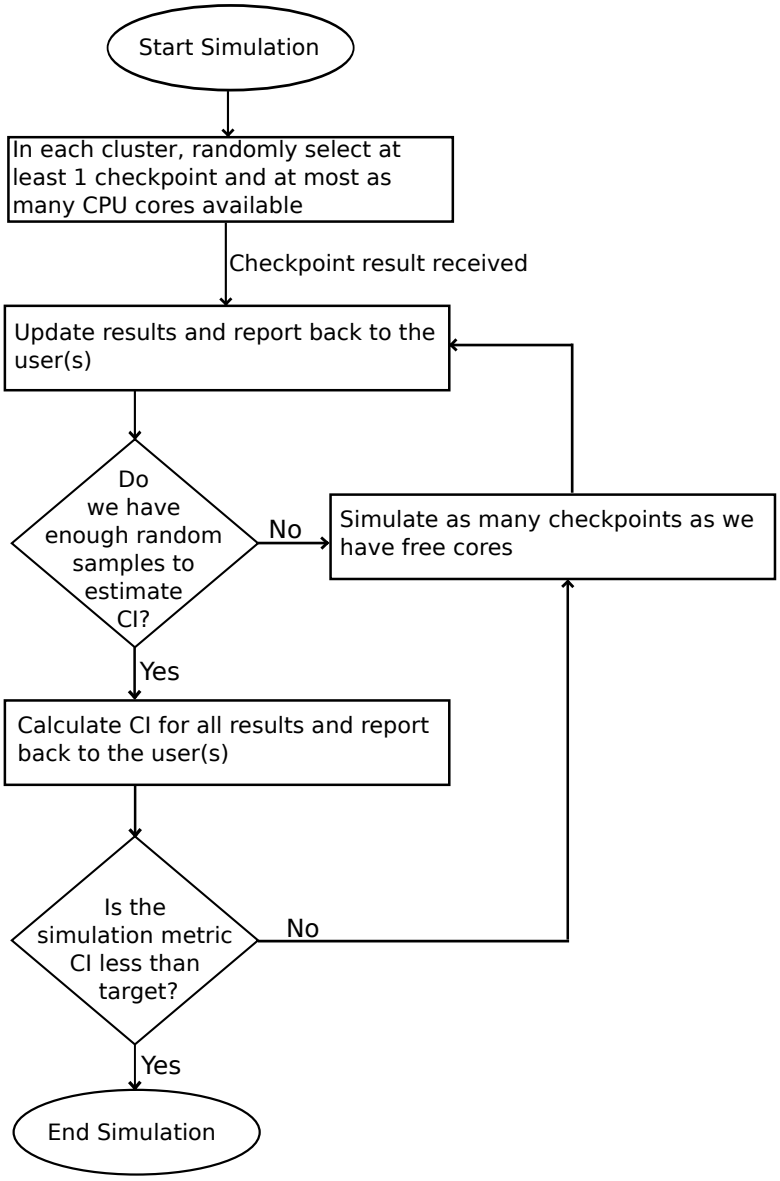

Figure 3.3: The high level sampling mechanism in LiveSim. LiveSim tries to use as much resources as possible to report accurate LiveSample and mathematically bounded LiveCI results in the shortest possible amount of time.

The algorithm explained in the previous section is used for any simulation. When a user decides to simulate a change in the architecture, he/she has to select a baseline. The detailed baseline results will be stored in memory upon such selection. In delta, since we plan

to simulate the same samples as the baseline, we use the same random sequence as used to simulate baseline. Since ratio results across samples form a distribution with relatively less standard deviation, probably a subset of the random samples in baseline will be enough to reach the user defined confidence level.

3.6 Parameter Sweep

Plotting effects of a parameter change over a wide domain will provide architects with a great perspective as well as finding out optimal points in the design. For example, trying to decide what ROB size is optimal for an out-of-order core, if we have a plot for changes in CPI over changes in ROB size from a minimum to maximum value, we can easily find the optimal point and evaluate the trade-off.

Having a fast delta simulation enables LiveSim to incorporate this feature. We call this feature parameter sweep as the user is able to select an architecture parameter (e.g. ROB Size) and simulate different possible values in a domain, plotting different metrics (e.g. CPI) over the changes. In order to do this, first the simulator does a baseline simulation for a selected (usually the median of the domain). For all other possible values in the selected domain, delta simulation will run (in or out of order depending on resources). Having the absolute results for the baseline and delta results for each point in the domain, absolute results for all points will be calculated and plotted.

3.7 Power Setup

Typical architecture simulators estimate the power consumption per structure based on the architecture configuration. This can be a slow process, mostly because of using CACTI. Obviously, this delay multiplies by the number of times a simulation is done which in the case of LiveSim, it is as many spawned checkpoints.

In order to address this issue, in every checkpoint, before CACTI begins to do power estimation, we check if the same estimation for the same SRAM configuration has been done before. If not, we allow CACTI to do the estimation and store the results indexed by a hash of the configuration struct. Later on, if any other checkpoint needs to do CACTI power estimation for the same SRAM, it can simply load the dumped results. If the CACTI code is changed, then all the previously stored dumps will be wiped.

Chapter 4

Other Explored Options

While creating LiveSim, we explored a few design options that showed good prospect at the time however proved to be insufficient in the end. These features are implemented in LiveSim but they are not currently active. However, it is still worthy to mention them as they bring us a better perspective for LiveSim design as well as making the audience aware of the options we tried and failed.

4.1 Range Detection Warmup

In the previous chapter we talked about how LiveSim uses LiveCache to eliminate the effect of cold state in architecture. We also discussed that still a small amount of warmup period is required for other elements. Before designing LiveCache, we spent a considerable effort on designing an adaptive and heuristic warmup methodology. The basic idea of this method is to use early results of checkpoints and try to find data trends which can be caused by a cold state. If such trends are found, the data points will be discarded until the trend disappears.

When a checkpoint starts simulation, we can form a time-line of the incremental simulation results. Going from a cold to warm state results in decreasing or in some cases increasing systematic trend in CPI. The reason is that over time, we get more data stored in memory hierarchy and less cold-misses. In addition, cold branch predictors produce more miss-predictions hence more cycles for the code to be executed. As the architecture modules are filled with more data, these misses decrease and CPI stabilizes as well. This trend continues until all modules are warmed up and the systematic trend disappears.

Although such trend is expected to be observed in any part of the application code we start simulation from, changes in the application phase will make it hard to distinguish. In other words, application phase and performance changes especially in small scales can be a great noise to the warm-up trend. In order to minimize this effect, LiveSim uses Theil-Sen noise-free trend estimation which produces an overall slope given a series:

$$slope = median_{(i<j-m)}(\frac{IPC_{av}(j) - IPC_{av}(i)}{j - i})$$

Using the above equation, we are able to evaluate different regions in checkpoint results and decide what regions have a warm-up trend and need to be discarded and what regions can be considered stable enough to form the sample results. To be safe, we assume only a series of samples with no significant increasing or decreasing trend as reliable. Figure 4.1 shows a very short CPI trace of one checkpoint and the selected reliable region.

Once trend detection selects the reliable region, we can use the simulation result of that region to average and find sample (checkpoint) results. Having trend detection results in

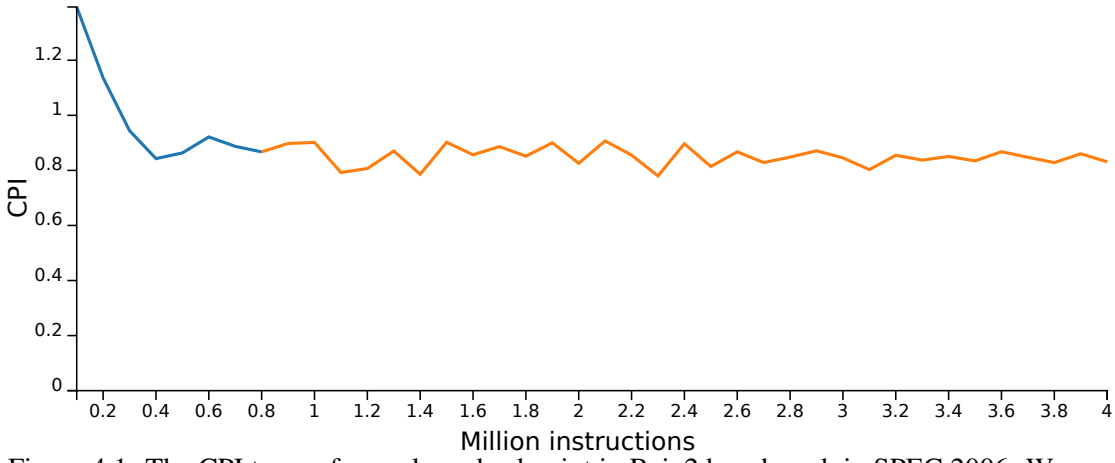

Figure 4.1: The CPI trace of a random checkpoint in Bzip2 benchmark in SPEC 2006. We can notice a systematic decreasing trend in CPI as different modules are getting warmed-up. The noise-free trend detection mechanism is able to detect the part that reflects unreliable results.

variable sample sizes in one simulation. However, each sample will be given equal weight in calculating the final results since they are randomly selected.

In spite of spending a great amount of time and energy on utilizing trend detection warmup, the results were not quite as ideal. One reason for such results was the larger higher level caches in some CPU configurations (e.g. L3 in Haswell architecture). Such caches require millions of instructions to get warmed-up and also the effect of cold miss in that level of cache is not so great on CPI. As a result, although cold misses on L3 may happen in the checkpoint, it does not appear in a relatively short trend of results.

Once we noticed sub-optimal results from range detection, we started utilizing LiveCache which eliminated the need for any warmup as far as memory hierarchy was concerned. We however still needed warmup for other modules (e.g. branch predictor) so we decided to use a combination of trend detection and LiveCache. This results and our later evaluations with static amount of warmup showed that the needed warmup after LiveCache was so little that the

trend detection overhead exceeded the processing time it could save. As a result, we decided to disable this function and instead do a static amount of warmup in combination with LiveCache for all checkpoints and applications.

4.2 Turbo-Charging

In LiveSim we are trying to be minimalistic on the number of samples taken and the size of samples to push the speed into its bounds. The minimum required number of samples is determined by the confidence interval around mean. The samples size by default is set in a way to the minimum that can maximize simulation performance (explained more in the following chapter). While having smaller sample size results in shorter simulation time for that sample, in some benchmarks it significantly increases variability across samples. Larger sample sizes capture larger portions of the application behavior which results in less variability among them.

Figure 4.2 show the CPI trace of the first 10 seconds native run in Perlbench. This benchmark has a great number of significant (high frequency high amplitude) changes in performance due to its nature. Small samples taken from this benchmark can randomly capture any part of the wave-like trend. As a result we will have samples varying from CPI 2 to 15 which result in a large confidence interval which requires us to simulate a great number of samples. However, if sample size is increased, a sample would not only include a short period in the wave-like trend, but probably even one whole cycle. Since each sample CPI is the average CPI over the sampling period, all most samples will have CPIs close to each other which results in a smaller confidence interval and could probably lead into earlier convergence.

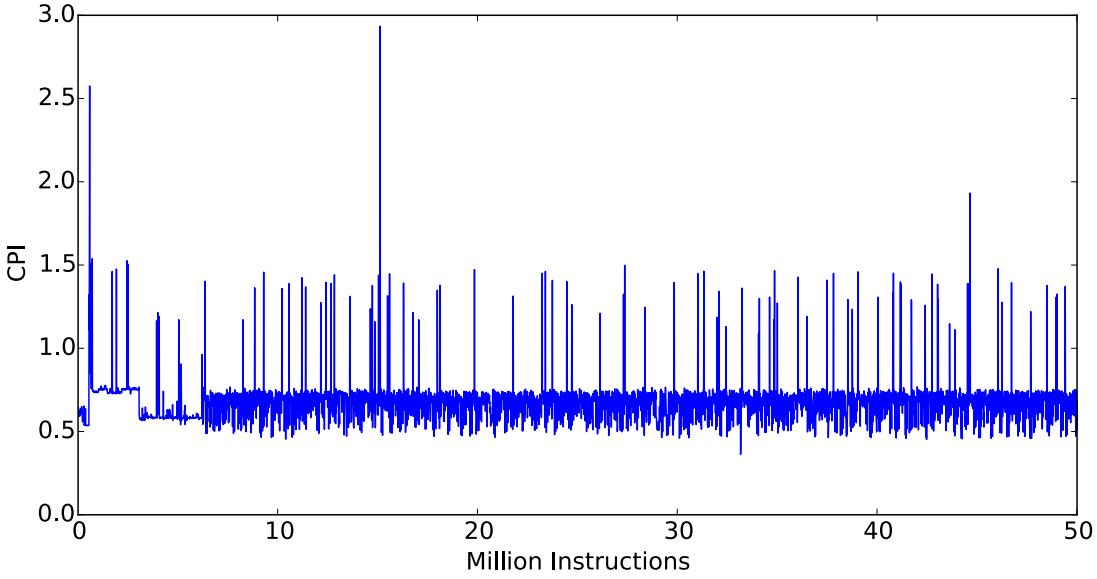

Figure 4.2: CPI trace of Perlbench in SPEC CPU 2006. Some applications have high frequency and high altitude changes in performance which makes them respond better to larger sample sizes.

Although turbo-charging is a good approach for simulating some applications, it does not scale out well for all SPEC benchmarks as not too many of them have phase changes in short time ranges. In fact, further evaluations in this work show that it is always good to increase the number of samples to be able to produce more accurate results as opposed to increasing sample size. While increasing sample size is harm-less and can potentially lead to better results, it is time and resource consuming. In the evaluation chapter we talk more about how to optimize the used sample size.

Chapter 5

Implementing LiveSim

In the previous chapter we learned that LiveSim is highly scalable and parallel by its nature. The more computing resources available, the more checkpoints can run in parallel which results in less simulation time. Implementing a scalable solution always introduces new challenges which we would like to discuss in this chapter. Moreover, the live nature of LiveSim requires certain platform setup which is interesting to look at.

LiveSim is implemented based on LiveOS which is a web based collaborative development and research framework. LiveOS includes many features or what we would like to call applications each bringing certain functionality to the platform. Examples are a collaborative code editor, LaTeX, Markdown compiler, code search, chat, file manager and Linux terminal. The front-end of LiveSim is another app in LiveOS which makes it easy to access and use along with the other tools.

LiveSim uses ESESC [18] as the simulation engine. ESESC is a fast micro-architecture simulator that can model a micro-architecture at the software level and produce performance,

power and thermal results running different benchmarks. ESESC has a few built-in sampling methods which are often recommended to use for speed purposes. However, since LiveSim has its own sampling mechanism, it uses only the timing (full simulation and gathering results) mode in ESESC.

5.1 Code Base

LiveSim is coded partly in C++ and partly in JavaScript. It uses object oriented design and is optimized for high throughput. We can divide LiveSim to the following sections:

- **Simulator Side:** This part of code is the new libraries and some patches for ESESC in order to integrate it with the rest of LiveSim as well as implementing some features like LiveCache. This code is in C++.

- **Back-end:** This part of code implements the actual LiveSim sampling method. It is a NodeJS server (coded in JavaScript) which gets service from the simulator side code and sends its results to the front-end.

- **Front-end:** This part of code is responsible for getting and presenting the results as well as setting configurations, etc. It is mostly coded in JavaScript and uses Epoch and D3 libraries for data visualization.

- **Transporter:** Transporter is a cross-platform application layer library for communication between different parts of the code base. It is developed in C++ and JavaScript as a part of the LiveSim project.

5.2 Architecture

Before we begin to explain how LiveSim works, let us define some widely used terms in this context:

- **Web Server** is a NodeJS process in charge of receiving simulation commands from the user, communicating with controllers and simulation nodes and publishing results back to the user. It is also in charge of some other logistics including settings, saving, etc.
- **Compute Machine** is a machine that runs the simulation jobs. It is good to have multiple compute machines with high number of cores and high enough memory. Storage is not a big constraint but it is best if it low latency (not NFS mounted).
- **Controller** is a NodeJS daemon residing in each compute machine for launching the simulation nodes, making proper introductions and terminations.
- **Simulation Server**s are created in the web server per application to do sampling and generate reliable results. Simulation server is the sampling core of LiveSim.
- **Simulation Node** is an instance of the micro-architecture simulator launching a specific benchmark/application.
- **Checkpoints** are processes in compute machines that hold equally distributed points in terms of instruction count in an application code. During setup, simulation node starts emulating the benchmark and checkpoint processes are forked at their specified locations in the application code.

- **Sample** is a small but representative part of the application code by randomly selecting one checkpoint and simulating it. In this context we refer to checkpoint as the process and the sample as the result we get from a checkpoint.

- **Simulation metric** is the main metric we are interested in simulating. All sampling decisions are based on this metric (chosen by the user) while other metric results will also be calculated. In this context, we assume simulation metric to be CPI.

Figure 5.1 shows the topology of LiveSim. Clients connect to the web-server who talks to the simulation nodes. In the simulation server we connect to the simulation nodes which are responsible for simulating one application and they can be on any compute machine. Each simulation node has a great number of checkpoints which are processes that each can start simulation on a specific part of the application. Checkpoints might be located in any host machine and are all tied to their simulation nodes through a connection serviced by the Transporter library.

The first process to launch is the web server. Once launched, each compute machine controller introduces itself to the web server. Once the machines are registered, the web server has enough information about the resources available.

The next step is setup which happens as soon as the first simulation request is received from any client (the LiveSim app). In setup, simulation server decides how to distribute the benchmarks across compute machines based on their resources and sends setup commands to the controller daemons. Controller daemons start the simulation nodes that introduces themselves to the web server. Upon this introduction, web server creates simulation server instances

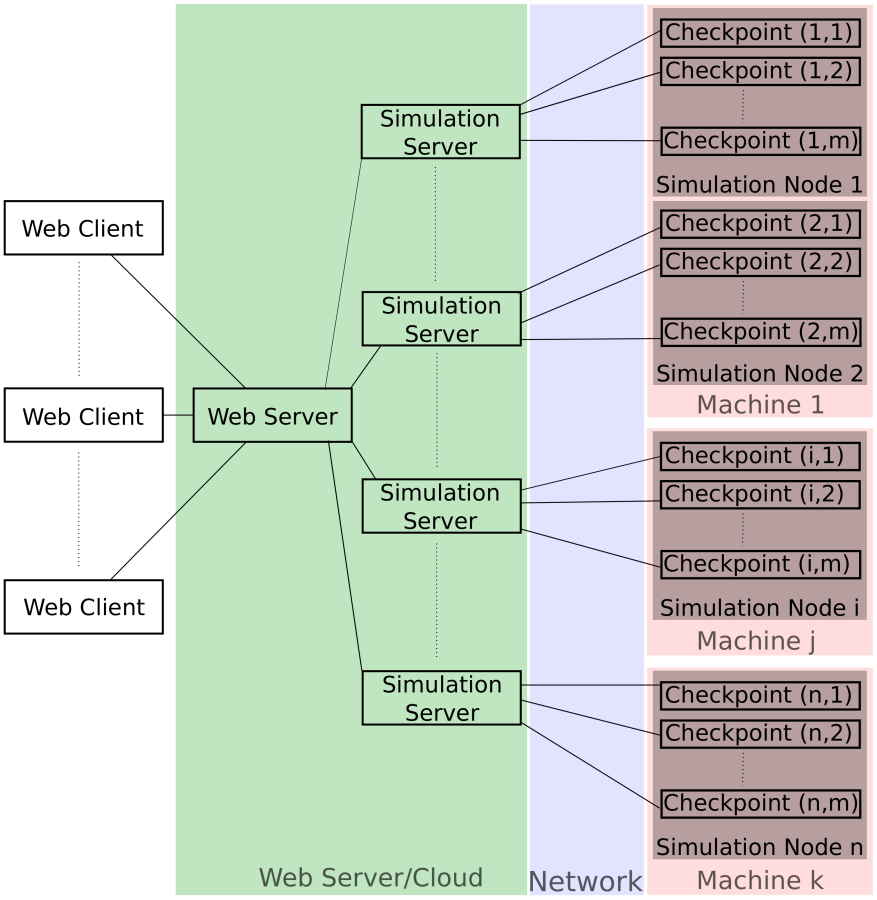

Figure 5.1: LiveSim setup. The web server talks to web clients and simulation servers. Each simulation server talks to its simulation node and spawned checkpoints.

one per application. Each simulation server asks its corresponding simulation node to prepare its checkpoints and register them back. Once all checkpoints are registered, setup is done and we are ready to start the next step which is calibration.

In calibration, all checkpoints for all applications are asked to run simulation and send their results back so that we are able to do clustering. Once calibration is done, the actual simulation can happen in which only a subset of checkpoints are selected and simulated. In the previous chapter we discussed how this selection mechanism works in theory and how calibration and normal simulation are the same except for the number of checkpoints they simulate.

It is worthy to mention that all the steps so far need to happen only once in lifetime for each application. It is only the live simulation that will be repeated upon any code or configuration change.

When each checkpoint is asked to be simulated by its corresponding simulation server, it forks a new copy of itself, loads the simulation libraries and starts the work. Upon simulating a specific number of instructions (smaller or equal to the sample size), it sends the results back to the server and continues until it reaches the sample size. The reason for these incremental results is live updates in the client side.

The client (LiveSim app) includes a plotting tool which can be adjusted to present different types of live updating plots for selected metrics and benchmarks. Within certain intervals, the client asks the web server to send updated results and updates itself. D3 library enables updating graphical elements without need to reload the object which enables us to have live updating plots in the client side.

In the front-end, user has the ability to change simulator source code and configurations and trigger a new simulation. In such condition, the source code will be re-compiled in the machine hosting the web server. Then the compiled simulation dynamic library along with a copy of all modified configuration files will be transferred to the compute machines replacing the existing files. Finally, checkpoints on compute machines will start simulation using the new library and configuration files.

One challenge in LiveSim is data transfer among simulation servers and checkpoints. Each incremental result consists of a large set of simulation statistics that need to get transferred accurately and instantly to the server. The text output of a single result dump occupies around

40KB. Considering the number of checkpoints running at the same time (hundreds) and sending this amount of data multiple times per second, we understand that network can easily become a bottle-neck for LiveSim performance.

Transporter is designed to address the performance issue in LiveSim as well as making socket coding easier and cleaner. This library can be used independently for any program-to-program communication over the network or even the same machine. In the next chapter, we will discuss Transporter in more details.

Chapter 6

Transporter

Transporter is created to provide a fast, reliable, secure and convenient mean of communication and data transfer in distributed systems. It is built based on TCP connections and in a sense can be considered an application layer protocol which has an interface to connect to TCP.

We can divide the functionality of Transporter to three layers as shown in Figure 6.1. The data is encoded depending on its structure and sent to the next layer to be encapsulated as a message. Then the message is encrypted with AES-128 and sent to the TCP connection to be streamed to the receiver. The receiver buffers the stream and once it has received the whole encrypted message, it starts the process of decryption, decapsulation and decoding.

In the next few sections, we will overview the steps that the library takes to send and receive data. First, we will discuss connection establishment. Then we will explain encoding methodology in details. After discussing encapsulation and the necessary overhead data, we will overview the encryption method.

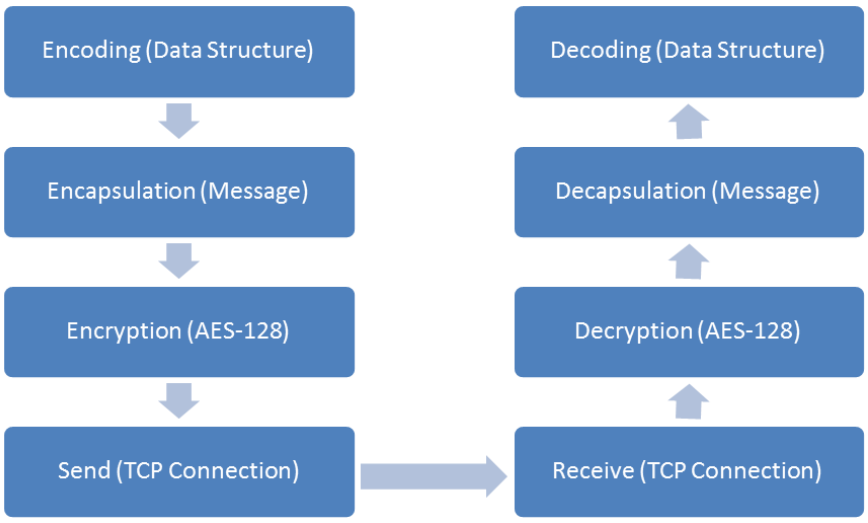

Figure 6.1: Layer-based structure of Transporter. Each layer is responsible for specific functions and is interchangeable without affecting its neighbors.

6.1 Connection Establishment

In Transporter connections as well as TCP and SocketIO connections, there is always a server and one or more clients. Each server will listen to a specific port on a host and other instances can try connecting to it having the host address and port number. The following example shows the JavaScript code for creating the server and C++ code for connecting to it. Using TCP sockets directly will require the developer to take many more additional steps to set up a reliable connection which Transporter eliminates.

```
1 //NodeJS server code
2 var io = new transporter_server(1111, function (socket) {
3   //on each socket connection this code is executed
4 });
5
6 //C++ client code
```

```
7 Transporter * t = new Transporter("mada1.cse.ucsc.edu", 1111);
```

Looking inside transporter connect function, once the client sends a connection request to the server, after receiving server acknowledgment, it will send its passkey encoded with a default AES key. The default AES key is static and both parties have it in the beginning. The server evaluates the passkey and in case of success, it will send the success message to the client. In case of failure, server shuts down the connection and sends a fail message to the client. Client can retry to connect with the correct passkey.

6.2 Encoding

In distributed applications, data is usually not in simple formats. Most times, developers need to transfer complex data structures across nodes and platforms. While TCP provides us with a byte stream connection, we will need to have a generalizable method of encoding data structures to an array of bytes.

The first method implemented for encoding is JSON. JSON encodes JavaScript objects to strings and is able to parse them back to objects. While JavaScript has JSON encoding built-in, C++ does not provide these methods hence we created a JSON encoding/parsing library to be used in Transporter. This feature supports recursion meaning that if we have a structure than includes other structures as fields, we can still easily encode and decode JSON as long as all structs have their fields introduced to a JSON object.

JSON has two problems: 1. For complex and large data, parsing JSON data will become a time-taking task and results in great delay overhead. 2. We send the structure along

with the data every time which can result in data overhead. In many applications we are sending data of the same structure over and over which makes one think, if we can send the structure once and send the data in a compact format every time, we will save some overhead.

For this reason, we created the LightSON encoding format. In LightSON, we use schema to introduce fields and their types. The fields can be complex data structures themselves. Each schema is generated and sent to the other party assigned with a unique schema ID. At any point that one party sends data encoded in LightSON, it has to send the schema ID along with it so that the receiver knows how to decode it.

Since we have a known schema for each structure, the actual data is a serialized byte stream of all the fields in the order that schema defines. Knowing the schema, the decoder can accumulate the correct bytes and convert them to the correct types.

LightSON was implemented and evaluated in two versions in this project: string-based (v1) and Binary-based (v2). The string-based LightSON is more compact when working with small amounts of data. Version 2 however is more compact when working with large amounts of data and is easier to process. The parsing and encoding mechanisms for each are different. In version 1, we used std::string functions and for version 2, we used Bytes arrays in C++ and Buffer in NodeJS.

Here is an example of C++ struct converted into all three formats:

```
1 //C++ struct
2 struct inner_test {
3   int a;
4   int b;
```

```
5 }
6 struct test {
7   int a;
8   double b;
9   string c;
10  inner_test d;
11 }
12
13 //JSON data
14 {"a": 10,
15  "b": 15,
16  "c": "foo",
17  "inner_test": {"a":20,"b":c}
18 }
19
20 //LightSONv1 schema
21 0a,1b,5c,{0a,0b}d
22
23 //LightSONv1 encoded data
24 10,15,"foo",20,30
25
26 //LightSONv2 schema
27 {"a":4,"b":8,"c":9,"inner_test":{"a":4,"b":4}}
28
29 //LightSONv2 encoded data (printed in ASCII)
30 1015foo2030
```

In the schema, the first byte, ensure the type of data and the rest is the field name in string format. In case of inner structures, the schema of that structured will be printed within curly brackets as the type. Encoded data is always depth-first serialized values of the struct and the inner structs.

/sectionEncapsulation TCP provides us with a stream of bytes but we want to send data and signals as objects. In other words, it will be very convenient for programmers to have a way of interrupting the code upon receipt of specific signals that carry data. As a standardized format, we define each message as an entity that carries a title and a data object. This way, we can easily emit and listen to signal-data pairs as follows:

```
1  //Emitting in C++
2  transporter->send_data(
3    "hello", data_obj->get_data(), schema_id);
4
5  //Listening in JavaScript
6  socket.on("hello", function (obj) {
7    //obj carries the data
8  });
9
10 //Emitting in JavaScript
11 socket.send_data("hello", data, schema_id);
12
13 //Listening in C++
14 data_obj->set_data(
15     transporter->receive_data("hello"));
```

Please note that in this example we are sending a C++ struct and receiving a JavaScript object.

Looking inside, each message has the following elements as bytes in order:

- message type (1 byte)
- message title (20 bytes)
- schema id (20 bytes)
- data length (4 bytes)
- data (variable length)

This is the most complete list of things a message can include. Data messages use all the fields but system and control messages may not use one or two fields. The decapsulation mechanism in the receiver will extract and trim each field deciding to take the necessary actions (e.g. parsing LightSON) based on message type. Figure 6.2 shows the encapsulated data frame format.

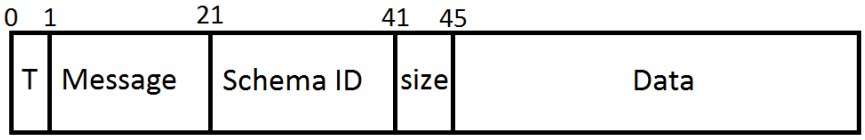

Figure 6.2: Encapsulation adds 45 bytes of overhead to the data. However, the 20 bytes for indicating message cannot be really considered an overhead. Another 20 bytes that show the schema ID are meant for avoiding a greater overhead: data structure

6.3 Encryption

Security is an important concern as soon as we begin to transfer data across hosts in a network. There are two concerns to be addressed: 1. Authentication: who to allow to connect and do data transfer. 2. Encryption: The transferred data should not be readable by other nodes in the network.

Authentication will be done using static passkeys that are stored in the server and the client. In the beginning of each connection, the passkey is encoded and submitted to the server. Server evaluates the passkey and decides if the client should be allowed to transfer data or not.

For encryption, we use AES-128 with dynamic keys. In the beginning, both client and server start using a static default key to connect. Once connection is set up, the client sends a key renewal request to the server and server generates a new random key sending it back to the client. Both parties will start using the new key right away. Each new key is encoded with its predecessor.

Each node has a timer for key usage, after a specific number of transmissions or a certain time has passed, the client will request a key renewal from the server and follows the same renewal procedure.

While doing the new key negotiation, there might be some data stored in the buffer or in the middle of transfer that is encrypted with the old key. This requires us to always be able to decode data that is encoded with the previous key. As a result both server and client always remember the previous key. In addition, along with each key, there comes a single bit flag in order to differentiate old and new keys. On each key renewal he flag is toggled in each

node. Therefore, if the node flag is equal to the data flag, it should use the new key to decode it, otherwise, it should use the old key.

TCP does not include a concept of object. It simply provides us with a stream of bytes that will be transferred in order. At any point, we can read and write to TCP buffers. In order to be able to figure out the start and end of each encrypted data chunk, we need to send the size and the flag prior to sending the data itself. Figure 6.3 shows the frame format that is written to TCP buffers.

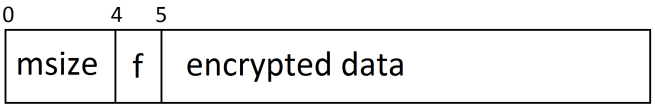

Figure 6.3: Encryption adds from 5 to 20 bytes of overhead. 5 bytes are consumed to indicate the data size to be decoded and what key to be used. An additional overhead might be caused since AES-128 encrypted data is blocked into 16 byte chunks.

Chapter 7

Measurement Setup

We evaluated LiveSim using the 24 of the 30 SPEC CPU 2006 benchmarks that we were able to run with our simulator (listed in Table 7.1). Our simulation infrastructure uses the MIPS64r6 ISA and it was missing support for some Fortran libraries which prevented us from running 6 of the CPU 2006 benchmarks.

We ran all of the benchmarks on an x86 system with Haswell microarchitecture for 10 seconds using the first reference input set and used performance counters to determine how many instructions the benchmark executed during this time. We then rounded this up to the nearest 5 billion instructions and used this as the number of instructions to simulate during our evaluation. This ranged from 30 billion to 90 billion instructions depending on the benchmark.

SPEC CPU 2006	perlbench, bzip2, gcc, mcf, milc, zeusmp, gromacs, namd, gobmk, dealII, soplex, povray, calculix, hmmer, sjeng, gemsfdtd, libquantum, h264ref, tono, lbm, omnetpp, astar, sphinx3, xalancbmk

Table 7.1: List of benchmarks used for evaluations in this paper.

We implemented LiveSim using a modified version of ESESC [18] as the timing

simulator and a modified version of QEMU [2] as the emulation engine.

We compared 3 different simulated microarchitectures: a high performance (HP) configuration that was modeled on Intel's Haswell microarchitecture, a medium performance (MP) configuration modeled on the ARM A72 microarchitecture, and a low performance (LP) configuration modeled after MIPS Apache microarchitecture. Table 7.2 provides more details about the simulated microarchitecture configurations. The confidence level for all evaluations was set to 95% and the target confidence interval was 10% of the reported values.

Parameter	**HP**	**MP**	**LP**
Freq	3.5 GHz	2.5 GHz	1.7 GHz
I$	32KB 8w 2cyc	32KB 2w 2cyc	32KB 4w 2cyc
D$	32KB 8w 4cyc	32KB 4w 4cyc	32KB 4w 4cyc
L2	256KB 8w 11cyc	2 MB 16w 16cyc	1MB 8w 26cyc
L3 (shared)	8MB 16w 20cyc	N/A	2MB 32w 14cyc
Mem.	110 cyc	100 cyc	60 cyc
BPred.	ogehl 10*2K	Hybrid 16K	2level 2K
Issue	4	3	2
ROB	192	128	64
IWin.	60	72	64
LSQ	72/42	32/32	24/24
Reg(I/F)	168/168	128/128	40/40

Table 7.2: The three different architectures used to evaluate LiveSim

Our host system had 2 Intel Xeon CPU E5-2689 CPUs and 192 GB of main memory. Each CPU had 8 cores with 2 SMT threads per core, giving a total of 32 OS visible logical processors. When running the benchmarks we executed them one at a time on the host system, which allowed the host to use all its resources for running a single benchmark.

Chapter 8

Evaluating LiveSim

Our evaluation of LiveSim focused on four different areas: speed, accuracy, warmup, and sample size and number characterization. For speed and accuracy we compared LiveSim with no sampling simulation and with a sampling mode that was very similar to SMARTS [32].

8.1 Speed

Our primary goal for LiveSim is to enable interactive design space exploration using a microarchitectural simulator. To evaluate this we ran all 24 benchmarks for each of the 3 different configurations using both no sampling and recorded a time varying trace of the LiveSim results. We calculated the time varying CPI error percentage for each benchmark by comparing the benchmark CPI reported by LiveSim with the CPI simulated without sampling. Figure 8.1 shows a graphical representation of this data. The import thing to note is that although the many of the initially reported values have a large CPI error, this quickly stabilizes and within 5 seconds the average error has dropped to 3.51%. Furthermore this error stays roughly the same

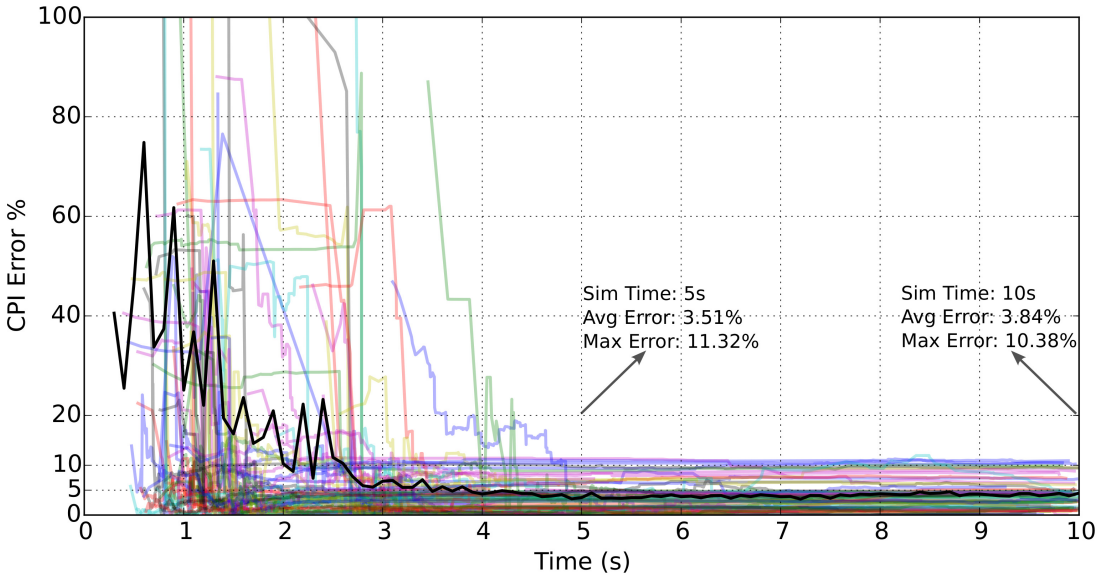

Figure 8.1: LiveSim CPI error for all 3 configurations and all 24 benchmarks (black line shows average error). LiveSim achieves an average of 3.51% CPI error within 5 seconds.

over the next 5 seconds. As a result we believe that the LiveSample results reported after 5 seconds of simulation make LiveSim suitable for interactive microarchitectural simulation. This is even more impressive considering that the portion of the benchmark simulated is equivalent to 10 seconds of execution on a high performance system with a Haswell microarchitecture. This means that LiveSim enable simulation that is even faster than native execution.

With LiveSim we define LiveSample as the initial results that are produced using a weighted average of samples from the checkpoint clusters created during the calibration step. The results in Figure 8.1 indicate that the LiveSample results are usable within 5 seconds. However, the LiveCI results take longer because they require true random selection of a larger number of samples. Figure 8.2 shows a comparison of average runtime for all of the different configurations for LiveSample, LiveCI, SMARTS, and running the simulation without sampling. One thing that is important to note is that the execution time of SMARTS and no sampling is

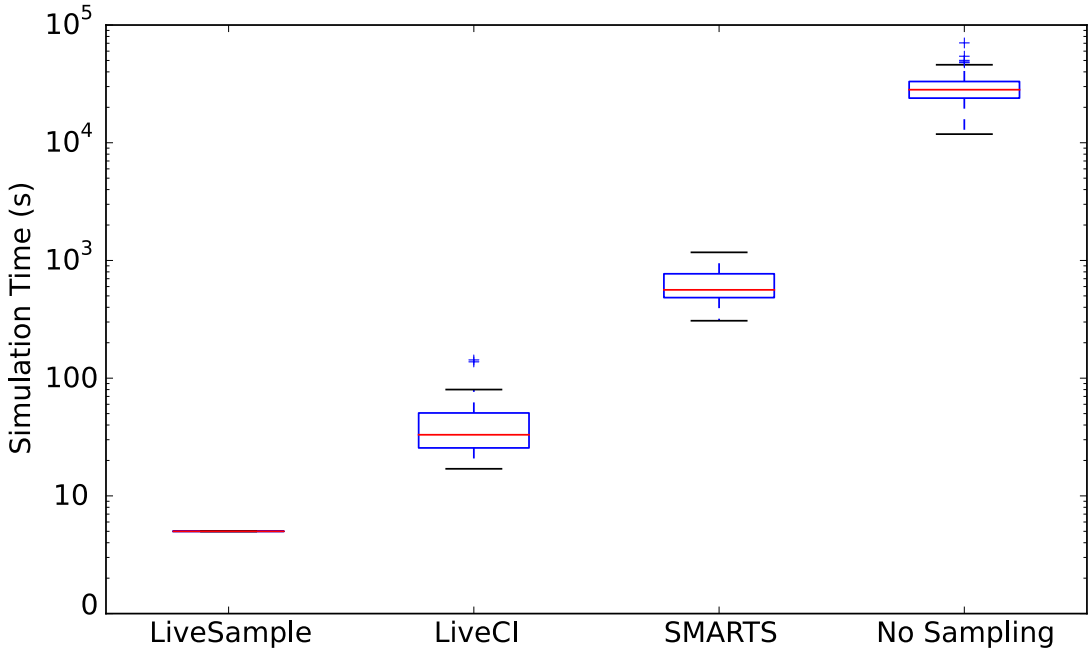

Figure 8.2: Average simulation time of all benchmarks and configurations. LiveSample results are ready within 5 seconds, LiveCI takes tens of seconds, SMARTS takes tens of minutes, and running without sampling takes many hours.

proportional to the length of the benchmark. While the execution time for LiveSample is nearly constant and for LiveCI it is proportional to the variability of the samples. Simulating a longer benchmark won't necessarily increase the simulation time for either LiveSample or LiveCI.

Table 8.1 provides additional insight about the execution time of LiveSample and LiveCI and how it varies per benchmark using the HP configuration. The execution time of LiveCI is proportional to the number of samples that need to be simulated, and this is typically proportional to the variability of the benchmarks. For LiveSample we don't show a specific time since this is not determined algorithmically. But as was illustrated earlier it is typically stable within 5 seconds. The execution time of LiveCI is determined algorithmically and it varies quite a bit from one benchmark to another. However, for most benchmarks it finishes

Benchmark	LiveSample	LiveCI	LiveCI Time (s)
astar	10	433	67.163
bzip2	10	496	64.021
calculix	6	398	48.227
dealII	1	178	17.937
gcc	14	490	80.032
gemsfdtd	6	328	45.724
gobmk	4	360	56.75
gromacs	2	246	30.327
h264ref	1	178	24.756
hmmer	1	177	21.247
lbm	3	239	32.679
libquantum	1	194	33.36
mcf	13	321	64.381
milc	13	500	74.741
namd	1	180	22.842
omnetpp	4	185	33.932
perlbench	6	183	33.197
povray	1	177	21.083
sjeng	2	899	137.47
soplex	9	427	61.676
sphinx3	8	213	27.493
tono	4	190	21.105
xalancbmk	1	177	31.461
zeusmp	8	412	54.872
Average	**5.37**	**412**	**54.78**

Table 8.1: Number of checkpoints needed for LiveSample and LiveCI for each benchmark for the HP configuration as well as the execution time for LiveCI.

within a minute or less. The MP and LP configurations typically finish running more quickly than the HP configuration, which is why the overall average execution time for LiveCI of all benchmarks and all configurations is 41 seconds.

8.2 Accuracy

In the previous section we demonstrated that LiveSim is fast enough to be used for interactive simulator use. However, fast results are only useful if they are reasonably accurate.

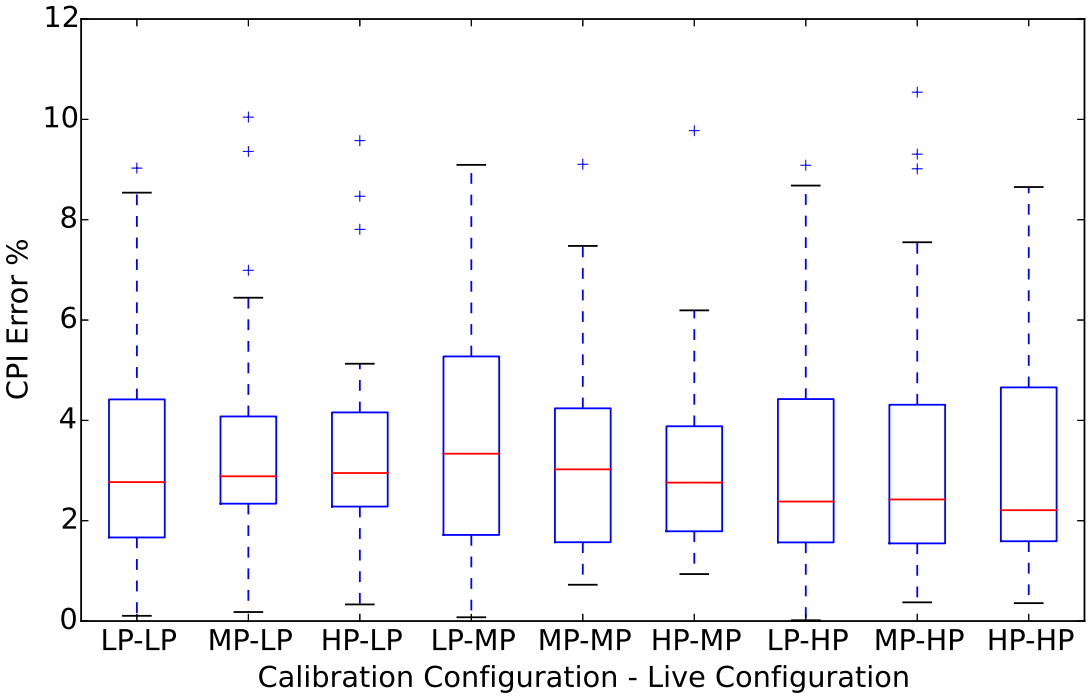

Figure 8.3: CPI error distribution across benchmarks in LiveSim. Each box label shows calibration and live simulation configurations respectively.

When evaluating accuracy there are two things to consider: how close is the point estimate to the true value, and how often often is the true value within the confidence interval. For our evaluation we selected a confidence interval of 10% and a confidence level of 95%. This means that we expect at most 5% of the simulation results to vary by more than 10% from the true value.

To evaluate this we ran 9 different experiments for each of the 24 benchmarks where we first ran calibration with one configuration (HP, MP, or LP) and then ran LiveCI with another configuration (we tested all possible combination, so the calibration and LiveCI configuration were the same in some cases). Of the 216 experiments there were 9 instances where the true CPI value was outside of the range reported by LiveCI. This is 4.16% of the time, which is within

the expected range for a 95% confidence level. Figure 8.3 shows the distribution of CPI error for this set of experiments. Although the target confidence interval was set to 10% in most cases the actual CPI error was much less, and the overall average error was 3.39%. These results also support our contention that the calibration step is microarchitecture independent. The overall error is roughly the same regardless of whether calibration is done with the same configuration as LiveCI or if the configuration used for LiveCI is very different from that used for calibration.

Finally for evaluating accuracy we included Figures 8.4,8.5 and 8.6 which compare the CPI from LiveCI and full simulation when using the same configuration for calibration and LiveCI. In this case all of the benchmarks are within the configured confidence interval.

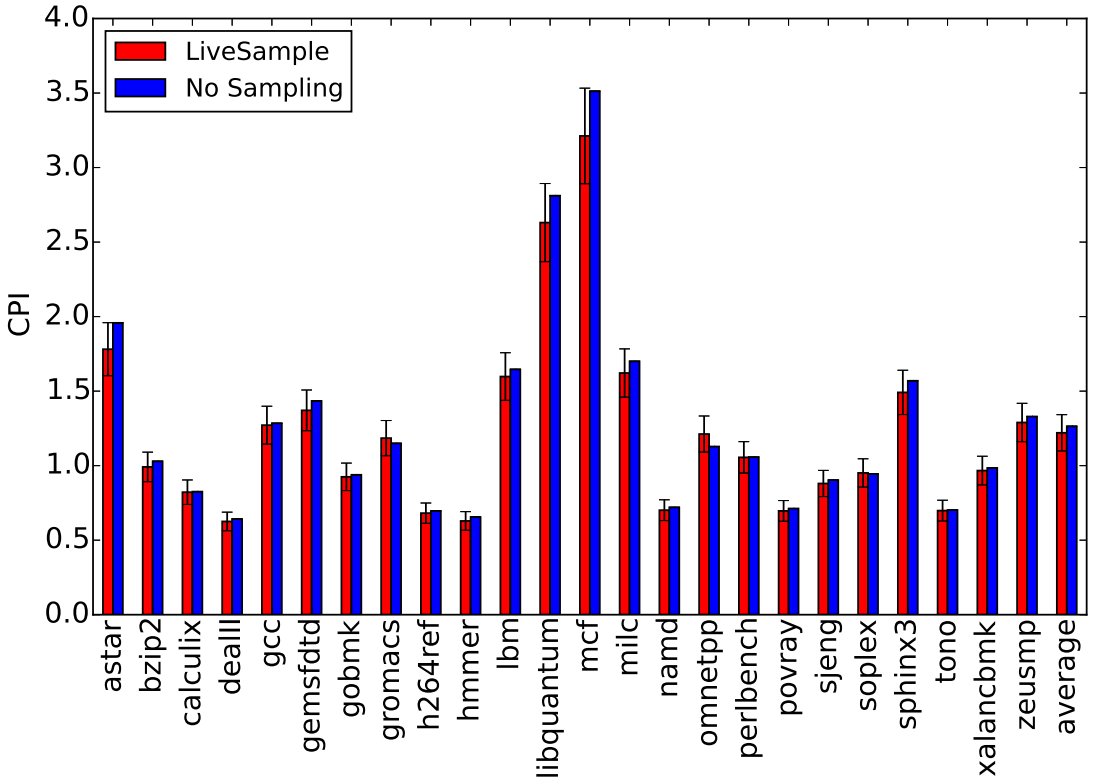

Figure 8.4: LiveCI results of SPEC benchmarks simulating the LP architecture compared to no-sampling simulation. The reported CPI results have 3.32% error and CI estimation is 100% accurate.

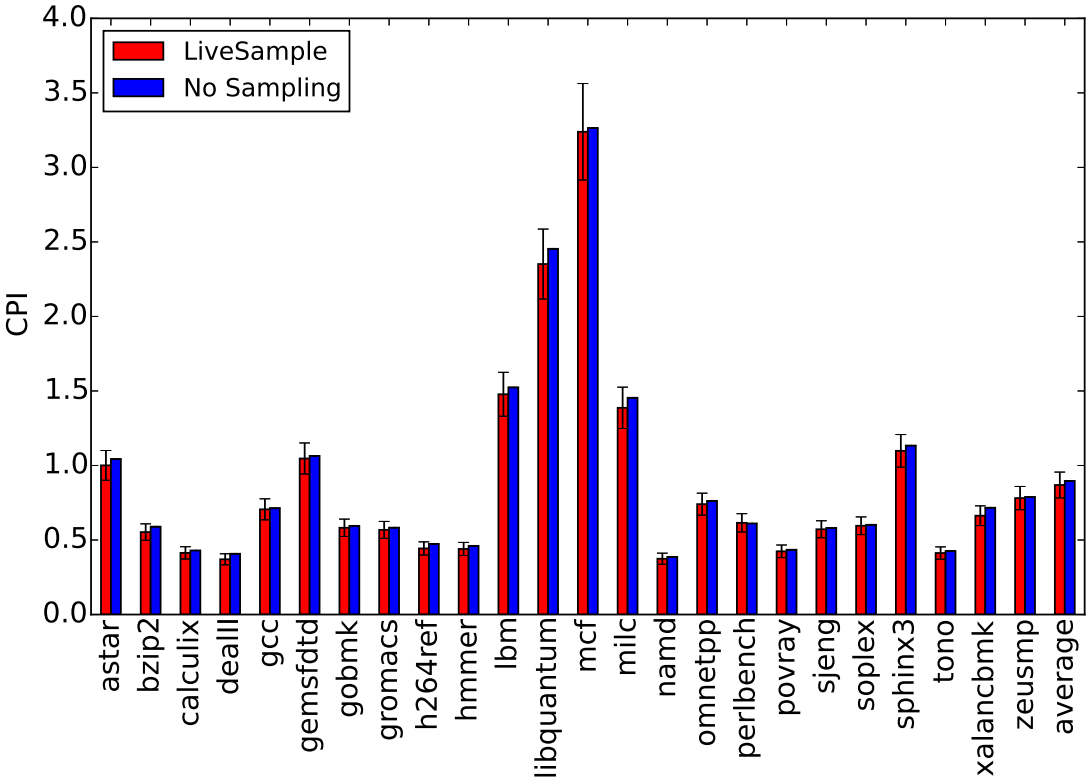

Figure 8.5: LiveCI results of SPEC benchmarks simulating the MP architecture compared to no-sampling simulation. The reported CPI results have 3.32% error and CI estimation is 100% accurate.

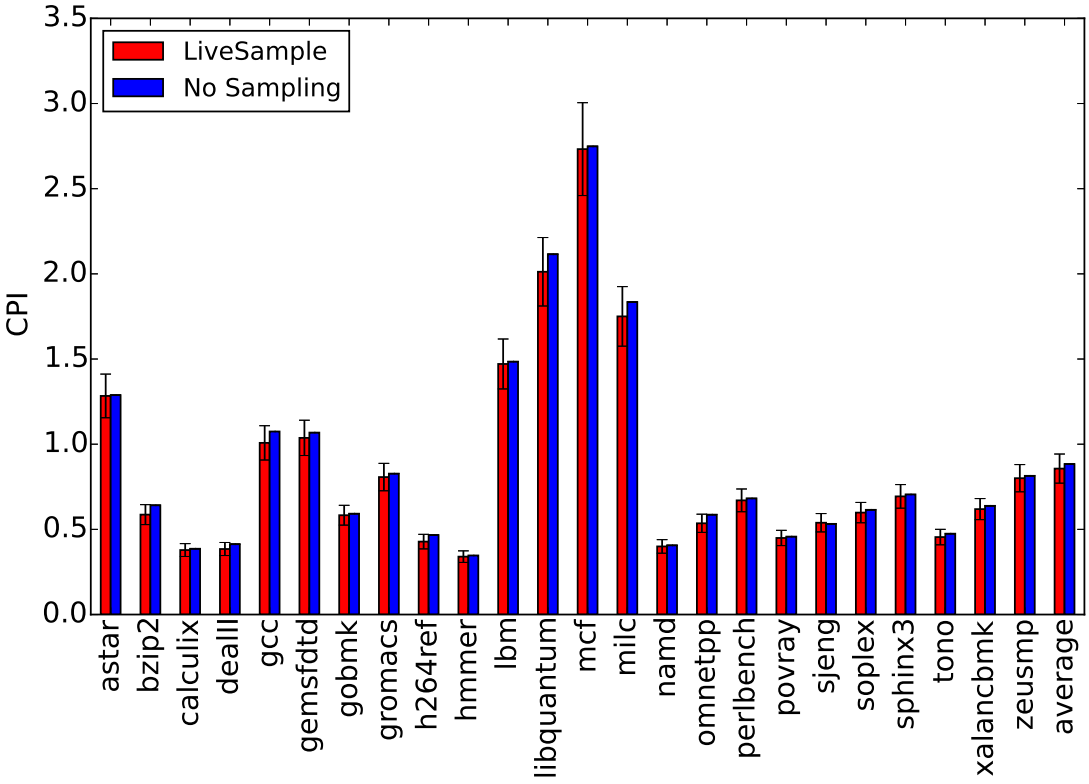

Figure 8.6: LiveCI results of SPEC benchmarks simulating the HP architecture compared to no-sampling simulation. The reported CPI results have 3.33% error and CI estimation is 100% accurate.

8.3 Warmup

Earlier we described how microarchitectural state warmup is a critical part of sampling. In this section we evaluate the effectiveness of LiveCache as well as the number of instructions for detailed warmup for other parts of microarchitectural state. All of these experiments were performed using the HP configuration since this has the largest and most complex microarchitectural structure that will need the most warmup.

To start with we varied the number of warmup instructions per checkpoint with and

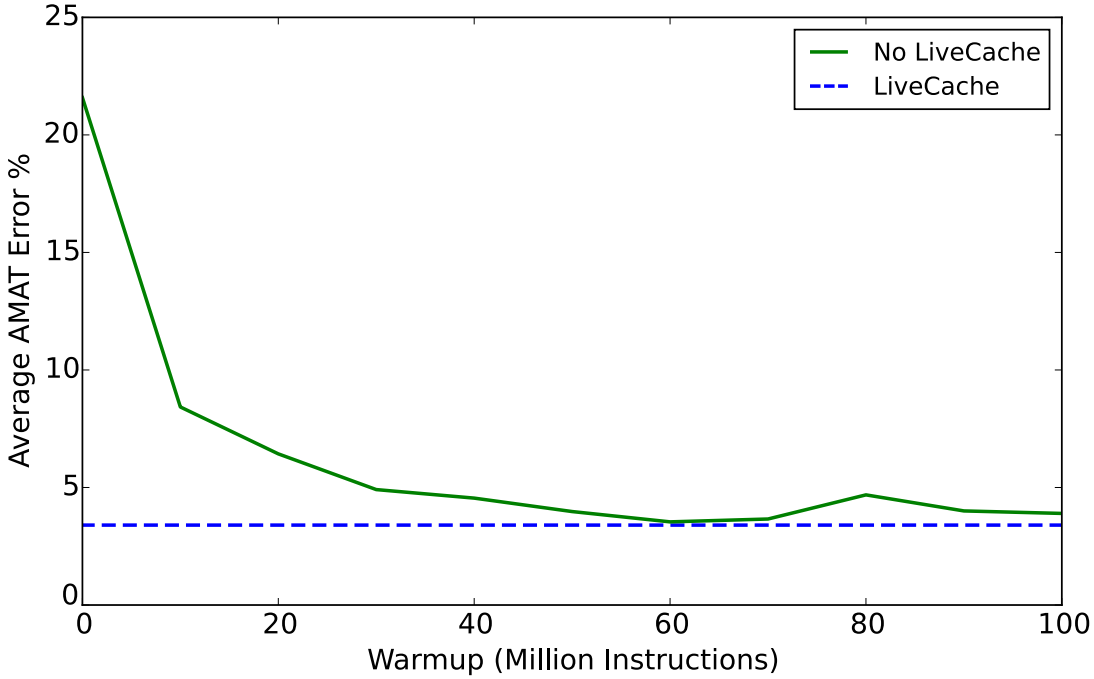

Figure 8.7: Comparison of average AMAT error for LiveCache and traditional cache warmup.

without LiveCache. Figure 8.7 shows the impact of LiveCache and warmup per checkpoint on AMAT error. When using LiveCache the error is nearly constant at less than 5% regardless of number of warmup instructions. But without LiveCache each checkpoint requires approximately 60 million warmup instructions before the AMAT error drops to the level of LiveCache attained with LiveCache. There is no further drop in the AMAT error because the remaining error is caused by sampling rather than inaccuracy in warmup. We also measured the overhead associated with LiveCache per checkpoint and it was less than 0.6 seconds on average. This is a relatively small overhead compared to the sample execution time, and is less significant because samples are executed in parallel.

Although LiveCache eliminates the need for cache warmup, there is other microarchitectural state that needs to be warmed up. The remaining structure that requires the most

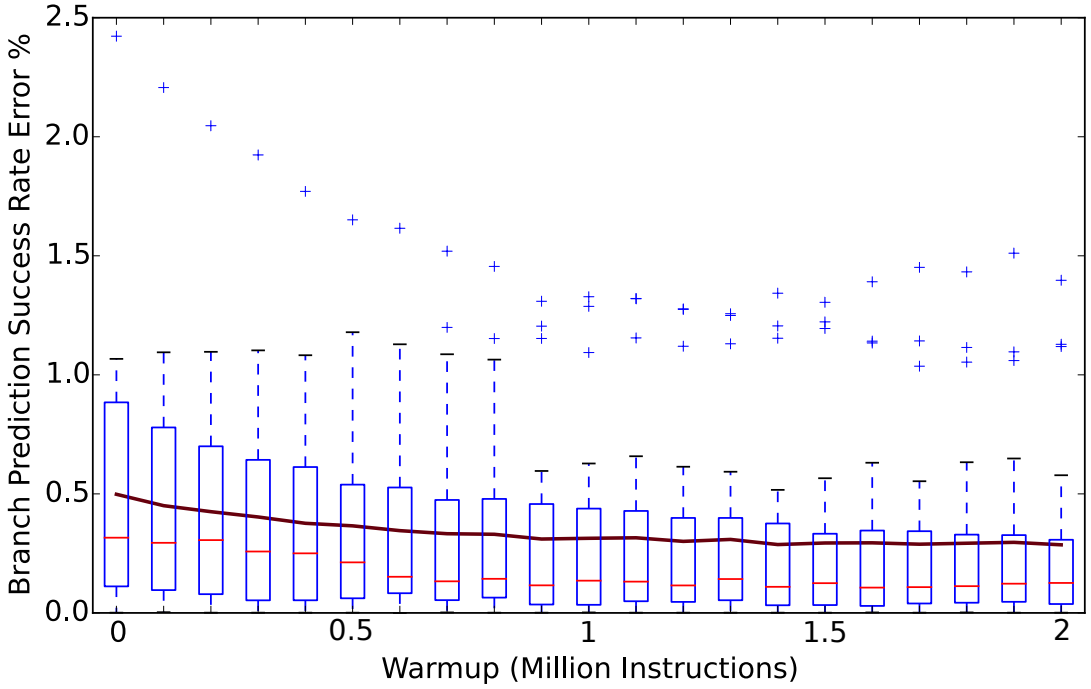

Figure 8.8: Average error of branch prediction statistics based on amount of detailed warmup at the start of a checkpoint.

warmup is the branch predictor. Figure 8.8 shows how the error rate decreases as the amount of warmup increases. It hits a pleteua around 900K instructions, and we found that in practice 1 million instructions was a good value for warmup.

8.4 Checkpoint Characterization

Although the number of samples that LiveSim uses is determined on-the-fly, the maximum value is limited by the number of available checkpoints. As a result we need to make sure to create enough checkpoints for any possible combination of configuration and benchmark that might be simulated. There are also two reasons why we want to limit the number of checkpoints, although we think these problems are more significant in simulator development

than they would be if LiveSim were used in practice. The first reason to limit the number of checkpoints is that each checkpoint needs to be run during the calibration phase, and so adding more checkpoints makes calibration take longer. However, if LiveSim were used in practice we expect that calibration would be done infrequently relative to how often the user collected LiveSim and LiveSample results, and longer calibration times would not be a problem. During the development and evaluation of LiveSim we experimented with many different parameters and frequently rerun calibration thus we were motivated to limit its execution time. The second reason is that our current implementation potentially uses a large amount of memory for each checkpoint, and if LiveSim has to use swap it has a dramatic performance drop. We believe that this is mostly an implementation issue rather than something that is intrinsic to the LiveSim system, and that memory use per checkpoint could be reduced if more time were spent optimizing this bottleneck.

We provisioned a system with 256 GB of DRAM when we were developing LiveSim, but one of our DIMMs failed when we were preparing to collect final experimental results, limiting us to 192 GB of main memory. This limited us to a maximum of 1000 checkpoints that we could create to guarantee that they were all in main memory. However, this was sufficient to guarantee a 10% confidence interval, but we would need more checkpoints to guarantee that we could meet a 5% confidence interval (even though even with a 10% confidence interval target most of the benchmarks we ran were within 5% CI).

To determine the maximum number of checkpoints that might be needed for various confidence interval and confidence level targets we ran the simulations without sampling and collected samples for every possible checkpoint candidate. This gave us a pool of tens of thou-

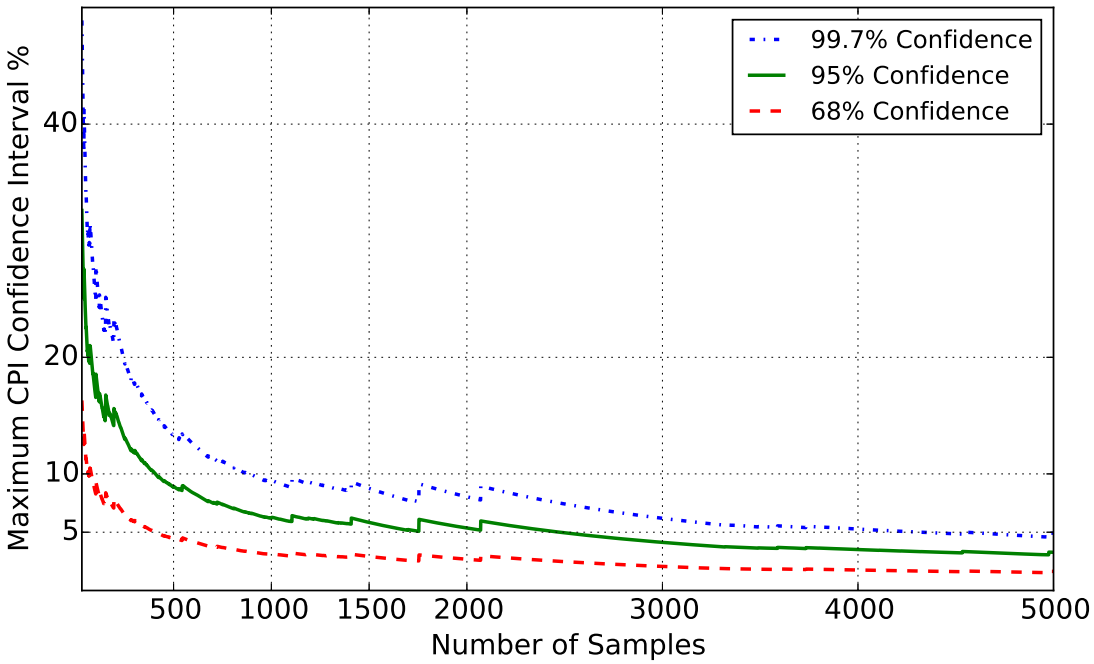

Figure 8.9: Maximum number of samples needed for a given confidence interval and confidence level in LiveSim, estimated using Monte Carlo simulation.

sands of potential samples to pick from. Next we ran a Monte Carlo simulation to randomly select from the pool of samples. For each benchmark and configuration pair, we calculated the confidence interval from the set of samples, and saved the maximum confidence interval calculated for that number of samples. We did this for the three most common confidence levels, and Figure 8.9 shows a plot of the results. The plot indicates that for our target of 10% confidence interval at a 95% confidence level we need roughly 500 checkpoints. However, this is simply a heuristic and we recommend doubling the number shown here when picking how many checkpoints to actually use to be safe, because if LiveSim does not have enough checkpoints it may be unable to meet the confidence interval target for LiveCI results.

We also evaluated how many instructions each sample should contain and the impact of sample size on simulation error and runtime. In general larger samples tend to improve accu-

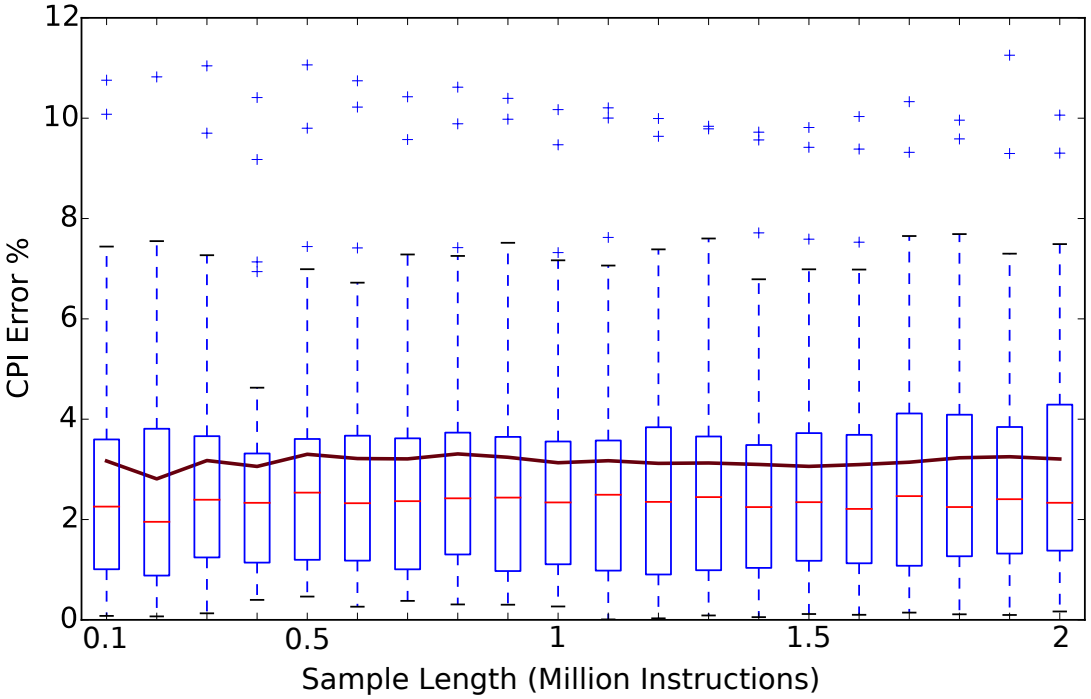

Figure 8.10: The effect of sample size on CPI error. Each box shows the error rate distribution for SPEC benchmarks and the line shows the average error across benchmarks.

racy, but decrease simulation speed. We determined that there is a sweet spot where increasing sample size does not improve accuracy but does decrease speed. Furthermore in our implementation we did not get any speedup for samples that were smaller than 100K instructions because of communication overhead between the simulation server and the client. We experimented with sample sizes ranging from 100K instructions to 20 times that amount. Figure 8.10 shows that the average error does not decrease with larger samples sizes.

Although Figure 8.10 indicates that the minimum sample size is best in terms of speed and accuracy trade-off, this is not necessarily the case because smaller samples can have more variation, and more variation across samples increases the number of samples needed for LiveCI. Figure 8.11 shows the effect of sample size on simulation time (LiveCI). This figure

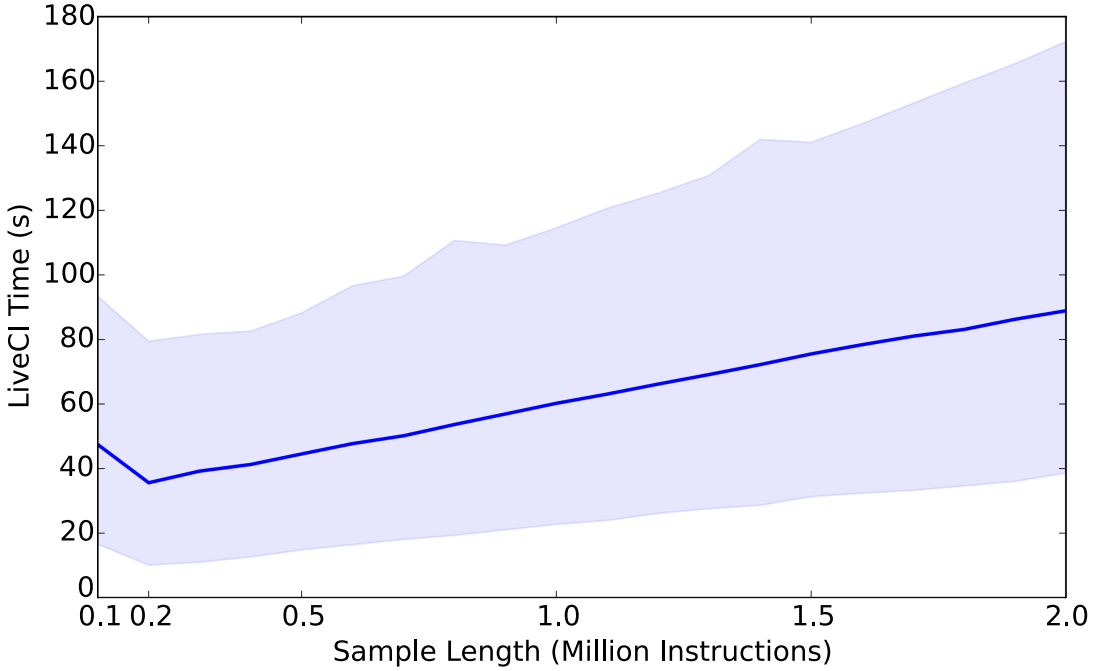

Figure 8.11: The effect of sample size on LiveCI time. The line shows the average LiveCI time across benchmarks and the area is where the actual distribution lies

.

shows that 200K instruction samples have the minimum execution time. Since the error rate is nearly the same for all samples sizes this is the sample size we used for LiveSim.

8.5 More Insight on Setup and Calibration

In the previous sections we discussed and evaluated LiveSample and LiveCI which can be quite often used to run many simulations very fast one after each other. However, initially there are two steps to be taken so that LiveSim becomes ready to use. We would like to emphasize that these steps are only to be taken once per benchmark and do not need to be repeated in any situation except the host machine is rebooted.

The first preparation step is setup in which checkpoints get created and LiveCache is

updated. LiveSim setup in our evaluations was done in average 18 minutes. Setup time is highly dependent on the benchmark, compute machine and the target simulation length. Once setup is finished, calibration needs to run once and only once which takes an average of 3 minutes in which all checkpoints are simulated.

8.6 Using LiveSim

Before we discussed how LiveSim is integrated with LiveOS to enable very efficient usage. In this section, we would like to depict how LiveSim can be used.

There are a few steps to be taken for installing LiveOS and LiveSim which are fully documented in their code repositories (will be released on github). Once set up, one can start running simulations by using the LiveSim app in LiveOS. The app itself allows the following:

- Starting and stopping simulation
- Choosing metric
- Choosing from different plot types
- Filtering benchmarks
- Saving and comparing results
- Changing different LiveSim settings

In order to make changes in the simulated architecture, one can modify ESESC code or configuration files and the re-run simulation. The files can also be hooked to LiveSim auto-

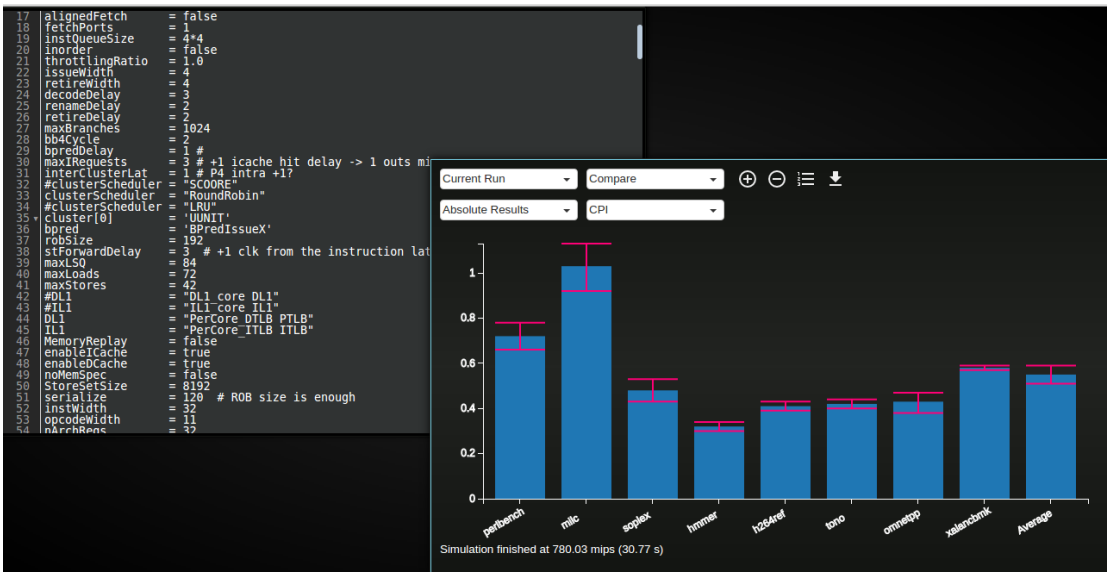

Figure 8.12: LiveSim app in the LiveOS environment. One can use the code editor and the options in the app to run live simulations and receive instant results on web.

run so that simulation results are updated as soon as the file is changes. Compilation and configuration errors will also appear in LiveSim app and the code editor. Figure 8.12 shows this flow.

Chapter 9

Related Work

Researchers have been working on ways to speed up simulation for decades and we surveyed some of the seminal work related to profile based sampling [24, 27] and statistical sampling [8, 9, 32] for microarchitecture simulation in Section 2. To the best of our knowledge no one has proposed simulation techniques that are suitable for interactive use (providing results in 5 seconds or less). LiveSim achieves fast simulation by combining three main techniques: random sampling of checkpoints, parallel simulation of checkpoints, and fast warmup of checkpoint state using LiveCache. There is a variety of related work in these various areas, but none of them attempt to achieve the goals of LiveSim.

The most closely related work to LiveSim is from Sandberg et al. [25, 26]. Like us, they use copy on write to fork multiple checkpoints and execute the checkpoints in parallel to speed up simulation. However, their proposal focuses on accelerating a single simulation run and only executes at 25% of native execution speed when simulating a system with an 8MB L2 cache. While this is an impressive result, our LiveSim system is able to execute at faster than na-

tive speed. After the initial setup step, LiveSim is able to provide simulation results in 5 seconds or less, even though we simulated 10 seconds of native execution. Sandberg et al. essentially use the SMARTS methodology, while using virtualization and parallel checkpoint execution to accelerate the function warming (which is the most time consuming part of SMARTS). In contrast we randomly select checkpoints in LiveSim and are able to report initial results within 5 seconds, and we choose how many total checkpoints to execute based on characteristics of the benchmark that we are simulating, whereas Sandberg et al. simulate all checkpoints as SMARTS would. Parallel execution of forked copies of an application has also been used by others to speed up analysis performed using dynamic binary instrumentation [22, 29].

SMARTS is effective at minimizing the number of instructions that need detailed simulation; however, its conservative always-on cache and branch predictor warmup makes warmup the simulation bottleneck (over 99% of simulation time). Many researchers have observed that always-on warmup of caches may be unnecessary and have looked for ways to accelerate warmup. For LiveSim we developed LiveCache by adapting a technique developed by Barr et al. [1] which keeps track of the sequence of memory operations during functional warmup and uses this information to rebuild the cache state before beginning detailed simulation of a sampling unit. We found that LiveCache technique works very well with LiveSim and helps us meet our goal of getting accurate simulation results in 5 seconds or less. However, there are a variety of other techniques that have been proposed for accelerating warmup.

Haskins and Skadron [14, 15] demonstrated that continuous cache and branch predictor warmup was unnecessary, and they proposed ways to determine when to begin warmup prior to simulating a sample. Eeckhout et al. [11] proposed a similar technique that further

reduced the amount of warmup required. Luo [21] proposed a method to monitor when a cache was warmed up and used that information to decide when to switch to full simulation. Recent work from Nikoleris et al. [23] shows that some workloads may require up-to 100 million instructions of cache warmup for caches larger than 64 MB. They propose a technique that uses native execution to capture a sample of memory accesses and uses this to reduce the amount of warmup for large caches. All of these techniques are effective for the types of simulation they evaluate, but they would not help with LiveSim because we still need to execute the application once during the setup phase, and so LiveCache is easily integrated with LiveSim's setup phase as a low overhead and relatively simple way to do cache warmup.

For LiveSim we have focused on developing a simulator that supports interactive use when evaluating new architecture proposals. Our work focuses on fast performance simulation for a single thread of execution because this is the baseline for microarchitecture simulation, and it must work correctly before considering more complex scenarios. Other researchers have looked for ways to speed up thermal simulation [10, 17], multithreaded simulation [3, 5, 18], and simulation of soft-errors in caches [28]. As future work we may extend LiveSim to support these additional simulation modes, but first we want to establish the usefulness of LiveSim using performance simulation only.

There are also proposals to accelerate simulation by varying the level of simulation detail depending on the region of code that is being simulated [4,12,13]. While these techniques work well for accelerating simulation they fall short of our goal of supporting simulation speeds that are suitable for interactive use.

Chapter 10

Conclusion and Future Work

We developed LiveSim, a novel simulation methodology that can be used for interactive microarchitectural design space exploration. Although analytical modeling can also be used for early design space exploration, eventually architects typically use simulation based methods to evaluate the usefulness of proposed ideas. LiveSim makes simulation fast enough for interactive use and allow an architect to quickly change parameters and get immediate feedback the impact on real benchmarks. LiveSim leverages many advances of the past two decades in applying statistical sampling to microarchitectural simulation. However, previous work on sampling has simply tried to make simulation faster. LiveSim is the first to demonstrates how sampling can be used to support interactive microarchitectural simulation.

As future work we hope to accelerate other parts of microarchitecture development including synthesis and simulator development. But microarchitectural simulation is one of the slowest parts of design space exploration for computer architects and we think the results in this paper are significant on their own.

Our LiveSim prototype demonstrates the feasibility of the LiveSim methodology and obtains accurate results within 5 seconds and bounds the possible error within 41 seconds on average for the benchmarks we evaluated. We evaluated the LiveSim methodology using a prototype we developed, but the concepts are general and can be adopted for use with other simulators. We also plan to release our implementation of LiveSim as an open source project to facilitate future research on live programming for microarchitecture development.

Bibliography

[1] K.C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 66–77, March 2005.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] T.E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. Barrierpoint: Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 2–12, March 2014.

[4] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[5] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 2–12. IEEE, 2013.

[6] D. Chiou, Dam Sunwoo, Joonsoo Kim, N.A. Patil, W. Reinhart, D.E. Johnson, J. Keefe, and H. Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, Dec 2007.

[7] Eric S. Chung, James C. Hoe, and Babak Falsafi. ProtoFlex: Co-simulation for Component-wise FPGA Emulator Development. In *Proceedings of the Workshop on Architecture Research Using FPGA Platforms*, 2006.

[8] T.M. Conte, M.A. Hirsch, and W.-M.W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, 47(6):714–720, Jun 1998.

[9] T.M. Conte, M.A. Hirsch, and K.N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *International Conference on Computer Design: VLSI in Computers and Processors*, pages 468–477, Oct 1996.

[10] Ayse K Coskun, Richard Strong, Dean M Tullsen, and Tajana Simunic Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37.

[11] Lieven Eeckhout, Yue Luo, Koen De Bosschere, and Lizy K John. Blrl: Accurate and efficient warmup for sampled processor simulation. *The Computer Journal*, 48(4):451–459, 2005.

[12] Magnus Ekman and Per Stenstrom. Enhancing multiprocessor architecture simulation

speed using matched-pair comparison. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 89–99. IEEE, 2005.

[13] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[14] Jr. Haskins, J.W. and K. Skadron. Minimal subset evaluation: rapid warm-up for simulated hardware state. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, 2001.

[15] John W Haskins Jr and Kevin Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, 2003.

[16] Wei Chung Hsu, Howard Chen, Pen Chung Yew, and H. Chen. On the predictability of program behavior using different input data sets. In *Workshop on Interaction between Compilers and Computer Architectures*, February 2002.

[17] E. K. Ardestani, E. Ebrahimi, G. Southern, and J. Renau. Thermal-aware Sampling in Architectural Simulation. ISLPED '12, July 2012.

[18] E. K. Ardestani and J. Renau. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. In *International Symposium on High Performance Computer Architecture*, HPCA'19, February 2013.

[19] A J KleinOsowski and David J. Lilja. Minnespec: A new spec benchmark workload for

simulation-based computer architecture research. *IEEE Comput. Archit. Lett.*, January 2002.

[20] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, March 2005.

[21] Yue Luo, L.K. John, and L. Eeckhout. Self-monitored adaptive cache warm-up for microprocessor simulation. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, Oct 2004.

[22] T. Moseley, A. Shye, V.J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, March 2007.

[23] N. Nikoleris, D. Eklov, and E. Hagersten. Extending statistical cache models to support detailed pipeline simulators. In *International Symposium on Performance Analysis of Systems and Software*, pages 86–95, March 2014.

[24] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, Washington, DC, USA, 2003.

[25] Andreas Sandberg. *Understanding Multicore Performance : Efficient Memory System Modeling and Simulation*. PhD thesis, Uppsala UniversityUppsala University, Division of Computer Systems, Computer Systems, 2014.

[26] Andreas Sandberg, Erik Hagersten, and David Black-Schaffer. Full speed ahead : Detailed architectural simulation at near-native speed. Technical Report 2014-005, Uppsala University, Computer Systems, 2014.

[27] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. ASPLOS X, pages 45–57, 2002.

[28] Jinho Suh, M. Annavaram, and M. Dubois. Phys: Profiled-hybrid sampling for soft error reliability benchmarking. In *International Conference on Dependable Systems and Networks*, DSN, pages 1–12, June 2013.

[29] Steven Wallace and Kim Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, 2007.

[30] J. Wawrzynek, D. Patterson, M. Oskin, Shih-Lien Lu, C. Kozyrakis, J.C. Hoe, D. Chiou, and K. Asanovic. Ramp: Research accelerator for multiple processors. *Micro, IEEE*, 27(2), March 2007.

[31] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. Turbosmarts: Accurate microarchitecture simulation sampling in minutes. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, 2005.

[32] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: Ac-

celerating microarchitecture simulation via rigorous statistical sampling. In *International Symposium on Computer Architecture*, ISCA '03, pages 84–97, 2003.

[33] M.T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, April 2007.