UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EFFECTIVE PERFORMANCE ANALYSIS OF MODERN CPUS

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

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Gabriel Southern

December 2016

The Dissertation of Gabriel Southern
is approved:

Professor Jose Renau, Chair

Professor Cormac Flanagan

Professor Matthew Guthaus

Professor Jishen Zhao

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Effective Performance Analysis of Modern CPUs

by

Gabriel Southern

Performance analysis is a critical aspect of CPU design, but it has become more difficult during the past decade as physical constraints limit improvements in single-threaded performance. This dissertation analyzes three interrelated problems associated with effective performance analysis. First, high-level microarchitecture simulation is orders of magnitude slower than native execution. I propose a novel statistical sampling technique called LiveSim that dramatically reduces simulation time. Second, multithreaded benchmarks may use input sets that produce misleading results. I demonstrate, for the first time, the true scalability of the PARSEC benchmark suite using real multiprocessor systems, and show how to accurately evaluate the performance of simulated multiprocessor systems. Third, modern software is often written in dynamic languages, and the interaction of a JIT compiler and different CPU microarchitectures can be difficult to analyze using simulation. I modified the V8 JavaScript engine and ran benchmarks using real systems to attain statistically sound novel insights that would be difficult to attain using simulation alone. These three solutions demonstrate ways to perform effective and statistically valid microarchitectural performance analysis.

Acknowledgments

The pursuit of a Ph.D. is a lengthy process that provides many opportunities for professional and personal growth and I benefited greatly from my experience as a Ph.D. student. I have been fortunate to interact with and learn from numerous people while working on my dissertation and I cannot possibly thank them all, but there are a few I want to mention by name.

First and foremost I want to thank my adviser, Professor Jose Renau, for his support and mentorship over the years. Jose possess a wealth of knowledge about computer architecture and I learned an incredible amount from him about the functionality and design of microprocessors. He allowed me to explore many topics and guided me to discover my interest in performance analysis of practical systems. Jose also directed and approved the research papers published in peer-reviewed venues.

I am fortunate to have three other active researchers—Professor Cormac Flanagan, Professor Matthew Guthaus, and Professor Jishen Zhao—who agreed to serve on my dissertation committee and provided valuable feedback. I thank each of them both for their assistance with my dissertation and for the interactions I had with them in classes and seminars during my time as a Ph.D. student.

A Ph.D. dissertation focuses on the research contribution of its author, but most research in computer architecture is a collaborative effort. I was fortunate to participate in several joint research projects that resulted in peer reviewed publications.

Sina Hassani and I worked together on developing a statistical sampling technique for fast simulation, and along with our adviser Jose Renau, we published our results at the IEEE International Symposium on High Performance Computer Architecture (HPCA) in March 2016. Sina used part of the HPCA paper as the basis of his M.S. thesis, and I have included part of it in Chapter 2 of this dissertation.

I also worked closely with Dr. Madan Das to apply static analysis techniques to reduce instrumentation overhead of dynamic analysis of multithreaded applications. We published our work in the ACM Transactions on Architecture and Code Optimizations (TACO), and Madan used much of our TACO paper as part of his own dissertation. I also worked closely with Dr. Ehsan Ardestani and Elnaz Ebrahimi in applying statistical sampling techniques to thermal simulation. We published our work at the International Symposium on Low Power Electronics and Design and in the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Going through the process of having peer reviewed research papers published was invaluable for my growth as a researcher, and I thank my co-authors for the opportunity to collaborate in their research efforts.

I am grateful for my interaction with members of the Micro Architecture Santa Cruz (MASC) lab over the years including: Ehsan, Elnaz, David, Rafael, Sina, Madan, Javi, Sangeetha, Alamelu, Rigo, Daphne, Ian, Blake, Ethan, Jason, Meeta, Akash, Ramesh, Tom, and Michael.

I appreciate the support I received from my parents over the years. My father's experience as a newspaper editor, and willingness to help with proofreading, was especially helpful. Last, but not least, I appreciate my friend Sophie's support, companionship, and patience during the years I spent working on my dissertation.

Chapter 1

Introduction

Performance analysis has long been an important part of the development of CPUs. One of the earliest ways of measuring CPU performance was to simply count the number of instructions it could execute per unit time. This led to a popular performance metric of millions of instructions per second (MIPS). However, researchers discovered that this simple measurement was insufficient to give an accurate characterization of actual processor performance [37]. MIPS was insufficient because it failed to measure the complexity associated with running actual applications. It also failed to capture the relationship between instruction count, clock frequency, and time to execute an instruction.

$$\frac{time}{program} = \frac{instructions}{program} \times \frac{cycle}{instruction} \times \frac{time}{cycle} \tag{1.1}$$

The Iron Law—shown in Equation 1.1—did capture that relationship, and for many years it served as the most important way to evaluate proposals to improve processor performance. Computer architects would carefully balance cycle time and the number of instructions executed per cycle (IPC) when evaluating proposed changes to architecture and microarchitecture. For many years CPU performance for single threaded applications increased at a rate similar to the rate of increase of transistors in a CPU. Much of the performance improvement was driven by the ability of computer architects to extract instruction level parallelism (ILP) from a serial instruction sequence.

However, by 2005 the physical constraints of device level physics caused clock frequency growth to stall, and increases in IPC also slowed. CPU architects determined that using all the transistors on a die for a single monolithic CPU was no longer feasible or effective.

Architects responded to constraints on ILP by integrating multiple processing elements on the same processor die. Modern CPUs are developed as a system-on-chip (SOC) with multiple CPU cores as well as GPUs and other hardware accelerators depending on the system requirements. As a result, modern CPU design requires close collaboration between software and hardware developers to maximize performance. One way to improve performance of applications running on multicore processors is by running multithreaded applications.

Today, general purpose CPUs still are designed to attain the highest possible performance within the power, thermal, and cost constraints associated with their intended use case. Designing a CPU is typically a multi-year effort, and the applications that the CPUs are designed to run might not even exist during early stages of the CPU development. However, early design decisions are often difficult or impossible to change later on. CPU architects typically develop models early in the design process to estimate expected performance of the system under development and to tune design parameters. The complex interaction between hardware and software makes this a challenging process for a variety of reasons, including:

- Applications that the CPU is designed to run may not be developed until after the CPU is designed and manufactured.
- The interaction between CPU design parameters and performance is often too complex to model analytically.
- Simulation models of CPU microarchitecture require approximation of elements of system behavior in order to make development of the simulator feasible.
- Software simulation of CPUs is orders of magnitude slower than native execution on a real system.
- Behavior of the simulated CPU may differ from the actual CPU in ways that have a significant impact on the true performance.

While modern CPUs are designed as an SOC that can execute many independent threads in parallel, single threaded performance still remains critically important. Parallel programs are more difficult to develop than single threaded applications, some algorithms are not well suited to parallelism, and existing programs need to run on new CPUs. And even in parallel application, performance of sequential sections of code can be critically important for overall application performance.

In this dissertation I analyze three related aspects of CPU performance evaluation: microarchitectural simulation time, multithreaded benchmark fidelity, and the overhead of software runtime system for dynamic languages. The analysis of simulation time considers a problem that faces architects early in the design cycle, and it proposes a novel way to speed up simulation and improve designer productivity by using statistical sampling. The other analysis of multithreaded benchmarks and dynamic languages was done using real systems to allow for measurements that considered benchmark performance without the possibility of modeling error.

Chapter 2 first surveys existing research to reduce simulation time by using statistical sampling with microarchitectural simulation. Then it presents a novel statistical sampling technique called LiveSim that reduces high-level microarchitectural simulation time from hours to seconds. The main contributions of this chapter are:

- A new architectural simulation methodology called LiveSim that enables interactive microarchitectural design space exploration.
- A demonstration of how LiveSim provides accurate results within 5 seconds.
- A demonstration of how LiveSim is able to bound simulation error within 41 seconds on average.

Chapter 3 presents an analysis of the scalability of a popular multithreaded benchmark suite named PARSEC. It carefully considers how much speedup the components of the benchmark suite attain when increasing the number of CPU cores available in a system under test. PARSEC provides features to separate the parallel portion of each benchmark from the sequential regions, which is called the region of interest (ROI). In this chapter I also analyze the impact of measuring only the ROI. The main contributions of this chapter are:

- A systematic analysis of the runtime scalability of each of the PARSEC input sets.

- A systematic analysis of the runtime scalability of the full benchmark execution compared to the ROI only.

Chapter 4 presents an analysis of the performance overhead associated with deoptimization checks used by the V8 JavaScript runtime. Its analysis is performed using four different systems with different microarchitectures. It considers how a possible optimization to reduce the overhead of dynamic languages can have a different impact depending on the CPU microarchitecture. This is increasingly relevant as modern SOCs integrate different cores in the same heterogeneous multiprocessor. The main contributions of the chapter are:

- Characterization of the performance overhead of deoptimization checks for the V8 JavaScript engine.

- Showing that performance overhead does not always correlate with instruction count, and that the specific overhead can vary dramatically depending on the CPU core microarchitecture.

- Categorizing the types and frequency of deoptimization checks and their execution frequency when executing optimized code.

Chapter 5 concludes by noting how the end of exponential increases in clock frequency has changed the types of performance analysis that computer architects need to perform. It ties together elements from the previous three chapters to illustrate the need for architects to carefully consider which parts of the software stack to model, and how to effectively use performance models early in the design cycle. It also considers how emerging trends such as heterogeneous compute, reconfigurable compute, and persistent memory will provide new challenges and opportunities for effective performance analysis.

Chapter 2

LiveSim: Enabling Interactive Microarchitectural Simulation

The first problem with effective CPU performance analysis that this dissertation considers is that of slow simulation time. Software simulation models are widely used for design space exploration and development of CPUs. Architects want to determine what the expected performance of a proposed design is before investing the immense amount of time and effort needed to design and fabricate the CPU in silicon. In this chapter I survey some of the techniques used to reduce simulation time, with a focus on applying statistical sampling to microarchitectural simulation. This chapter is based largely on the paper *LiveSim: Going Live with Microarchitecture Simulation*, which was published in the proceedings of the IEEE International Symposium on High Performance Computer Architecture in March 2016. I collaborated closely with Sina Hassani in developing and tuning the LiveSim prototype system, and in jointly writing the text of the published paper. The research was supervised and directed by our adviser

Jose Renau.

2.1 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 size reduction [47], specialized hardware [77], phase-based sampling [72], and statistical sampling [80]. 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 chapter 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 (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 first runs 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 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 the calibration phase tends not to need to be repeated even if the simulated microarchitecture has radical changes. At

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

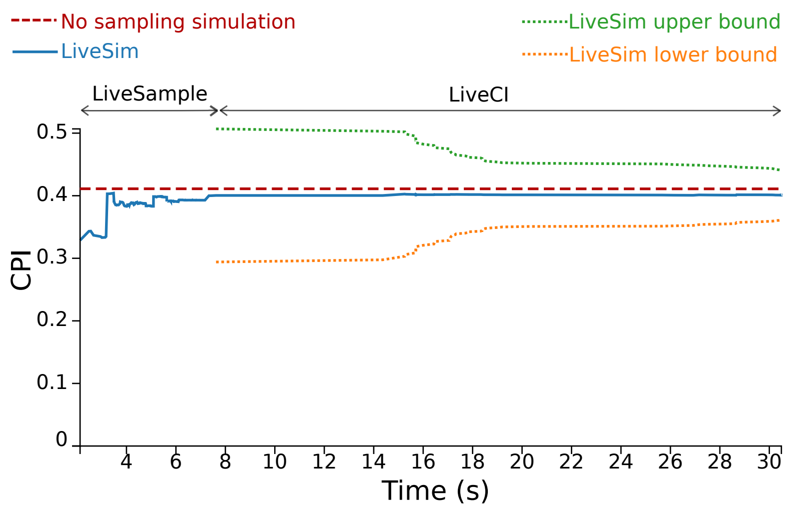

Figure 2.1: LiveSim timeline showing how the user is presented with LiveSample results from the beginning, how accurate the results get within a few seconds, and how the simulation continues running until the confidence interval reaches the threshold for the LiveCI results.

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 2.1 illustrates 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 within 4% of the correct result. The correct result is a full simulation of the 10 seconds without sampling shown as No Sampling Simulation. 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.

This chapter describes the following novel contributions:

- Introduces LiveSim, a new architectural simulation methodology that enables interactive microarchitecture design space exploration.
- Demonstrates 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.
- Demonstrates that LiveSim is able to produce LiveCI results that bound the simulation error within 10% within 41 seconds on average.

The rest of this chapter is organized as follows: Section 2.2 provides background about existing techniques to speed up simulation; Section 2.3 explains how LiveSim works; Section 2.4 details the setup of our evaluation framework; Section 2.5 describes our results; Section 2.6 provides a comparison with related work; and Section 2.7 summarizes the contributions of this chapter.

2.2 Background

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 [45, 47]. Another approach is to accelerate the timing simulation using FPGAs [77, 24, 23], but this requires custom hardware, increases simulator development complexity, and is not widely used in practice. 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 to modes that run parts of the simulated benchmark directly on the host system [81]. 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. [72] 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 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. [62] extended SimPoint to provide statistical confidence measures.

Wunderlich, et al. [80] 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. [27, 26] 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 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 [79, 78]. Another technique is to forego 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 [7].

LiveSim builds on existing work that uses sampling to accelerate microarchitecture simulation, but rather than simply trying to make the simulation faster, it 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.

2.3 LiveSim Methodology

The previous section provided 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 a number of checkpoints in parallel. As soon as simulation results are ready they are reported visually to the user.

- **LiveCI**: After a 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.

2.3.1 Sampling Setup

The field of inferential statistics provides well known techniques for inferring statistics about a population given a sample of that population. SMARTS [80] 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

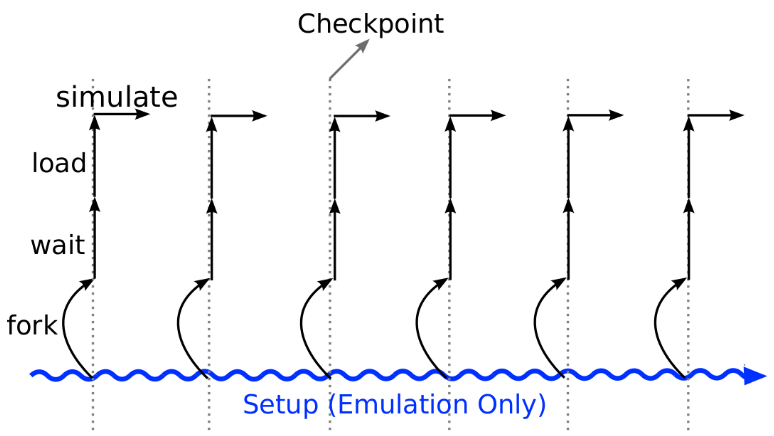

Figure 2.2: During setup the benchmark is emulated and checkpoints are created by periodically forking new processes. Each checkpoint process enters a waiting mode. Once LiveSim starts, each checkpoint loads the simulation dynamic library and configurations; then it starts simulation.

during the setup phase.

During the setup phase LiveSim runs a fast emulation-only process that periodically 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 2.2 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 tens of megabytes which makes its memory consumption worthy of study. In Section 2.5.4 we evaluate checkpoint size and how to determine the optimal number of them.

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 50 million instructions on average, and research indicates larger caches may require even more warmup [59]. 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. [7], 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 provide an ordering of all locations that could possibly be in the cache. Maintaining this state has a low overhead and does not slow down the LiveSim setup process much. It will automatically be made available to forked checkpoints 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. Once all LiveCache memory operations are executed, the real simulation starts. This is similar to the technique proposed by Barr et al. [7] with some slight changes to simplify the integration with the LiveSim simulation infrastructure.

2.3.2 Calibration

The central limit theorem is the underlying foundation for the statistics theory that 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 sampled population is not highly skewed. But if the population is highly skewed, more samples are required.

Figure 2.3 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 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 [48] and use this information to group the checkpoints into clusters. When results 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. [62] do to statistically bound the error for results obtained using SimPoint.

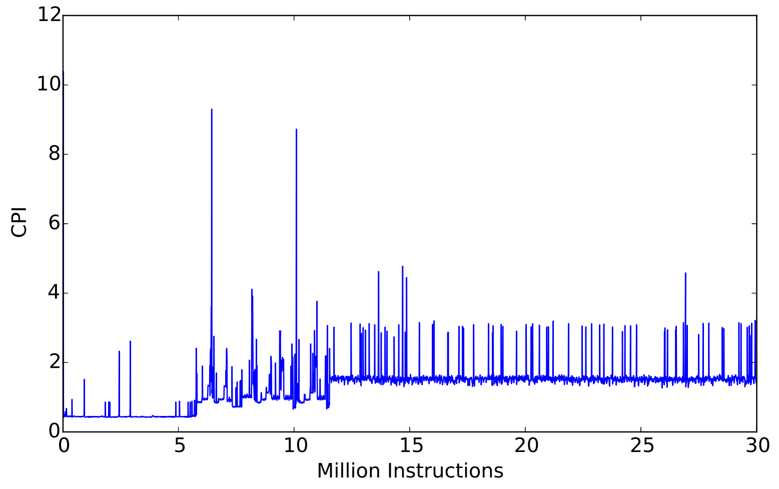

Figure 2.3: The CPI trace of Astar benchmark in SPEC 2006. The infrequent but extremely large spikes will have a considerable effect on the average CPI. Missing these spikes in random sampling will result in an unreliable sample mean.

LiveSim groups the checkpoints based on the performance statistics obtained with the baseline simulator configuration. Since performance statistics correlate with code signatures, this grouping tends to cluster the checkpoints together such that even radical changes in the simulated microarchitecture still cause 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 the checkpoints are spawned during setup, LiveSim begins calibration. For each checkpoint, LiveSim loads the baseline simulator configuration and simulates a sample. After all the checkpoint samples are collected, LiveSim uses a clustering algorithm to group the checkpoints into clusters.

For clustering, LiveSim uses a 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 the 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 assigned to each cluster. For example, if there were 2 clusters, and the first cluster had 900 checkpoints while the second had 100 checkpoints, 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 for averaging results obtained from simulated samples and reporting LiveSample results.

2.3.3 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 interested 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 spawn at least twice as many checkpoints as cores in the host system 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 that 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. Each child process, which 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, collects statistics for its sample, and 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. In our experiments these

LiveSample results typically reached a steady state value within 5 seconds of starting the simulation.

2.3.4 LiveCI

Figure 2.1 shows an example of how LiveSim produces LiveSample and LiveCI results for a user. The LiveSample result is provided as soon as the first checkpoint is simulated, and it usually reaches a steady state value very quickly (typically within 5 seconds). However, the initial LiveCI results take slightly longer and the simulation continues running until the LiveCI result reaches the user's specified threshold.

When LiveSim selects checkpoints to use for LiveSample results it ensures that at least 1 sample is selected from each cluster. Afterwards it begins selecting checkpoints completely randomly. However, for calculating the confidence interval, the 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 that 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 heuristic generally accepted as the minimum cutoff value for using the central limit theorem to approximate the sample mean distribution as normally distributed; and a normal distribution is required for calculating 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 2.3). 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.

The 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 that 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 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 user was simulating a workload with even more extreme variation in sample metric, a different heuristic might be required, but we expect that adding a synthetic point 10 times the sample average should suffice for most workloads that computer architects simulate.

2.4 Measurement Setup

We evaluated LiveSim using 24 of the 29 SPEC CPU 2006 benchmarks that we were able to run with our simulator. Our simulation infrastructure used the MIPS64r6 ISA, which was missing support for some Fortran libraries, preventing us from running 5 of the CPU 2006 benchmarks (bwaves, gamess, cactusADM, leslie3d, wrf).

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 15 billion to 90 billion instructions depending on the benchmark.

We implemented LiveSim using a modified version of ESESC [5] as the timing simulator and a modified version of QEMU [9] 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 2.1 provides 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.

Our host system had 2 Intel Xeon CPU E5-2689 CPUs and 192 GB of main memory.

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 2.1: The three different architectures used to evaluate LiveSim

Each CPU had 8 cores with 2 SMT threads per core, giving a total of 32 OS visible logical processors. We executed the benchmarks one at a time on the host system, allowing the host to use all its resources to run a single benchmark. In all our evaluations, the simulated architecture is single-core and only one single threaded benchmark is running at a time.

2.5 Evaluation

Our evaluation of LiveSim is focused on four areas: speed, accuracy, warmup, and sample characterization. For speed and accuracy we compared LiveSim with no sampling simulation and with a sampling mode very similar to SMARTS [80].

2.5.1 Speed

Our primary goal for LiveSim was to enable interactive design space exploration using a microarchitectural simulator. To evaluate this we ran all 24 benchmarks for each of the 3

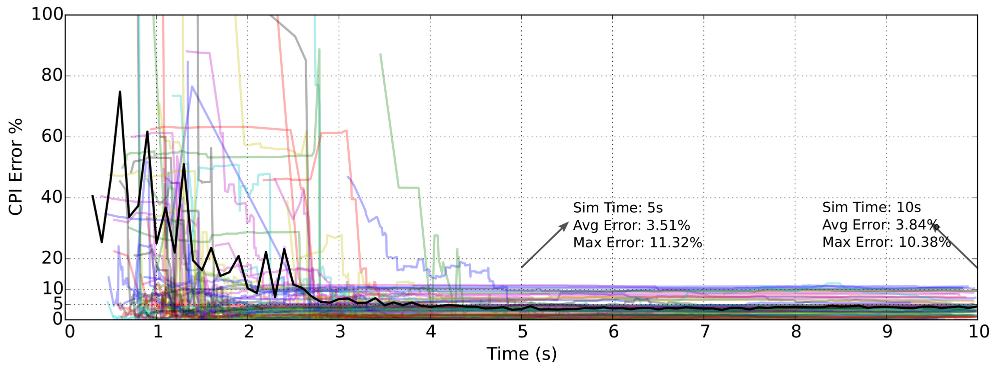

Figure 2.4: LiveSim CPI error for all 3 configurations and all 24 benchmarks (black line shows average error). LiveSim achieves an average of 3.51% CPI error within 5 seconds.

configurations using both sampling and no sampling, and we 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 2.4 shows a graphical representation of this data. The important thing to note is that although many of the initially reported values have a large CPI error, this quickly stabilizes and within 5 seconds the average error drops to 3.51%. Furthermore this error stays roughly the same over the next 5 seconds. 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 enables 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 2.4 indicate that the LiveSample results are usable within 5 seconds.

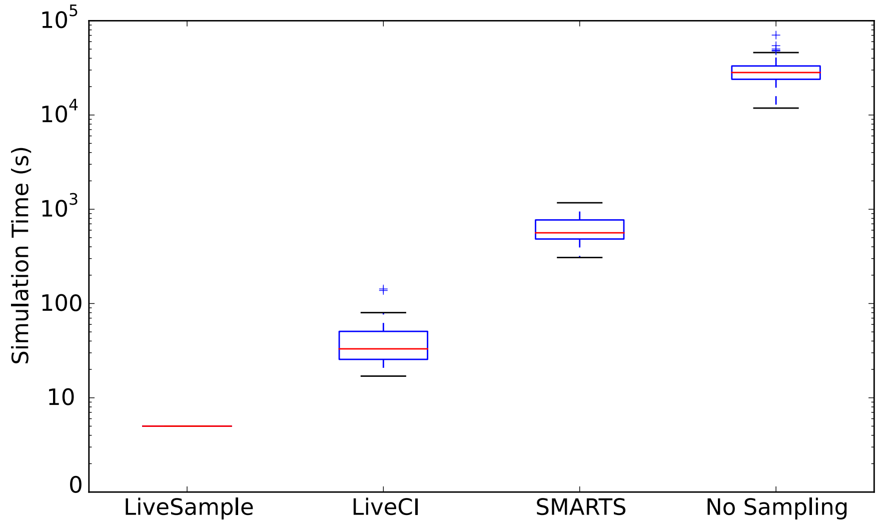

Figure 2.5: Average simulation time of all benchmarks and configurations. LiveSample results are ready within 5 seconds, LiveCI takes tens of seconds, SMARTS takes tens of minutes, and running without sampling takes many hours.

LiveCI results take longer because they require true random selection of a larger number of samples. Figure 2.5 shows a comparison of runtime for all of the different configurations for LiveSample, LiveCI, SMARTS, and running the simulation without sampling. It is important to note that the execution time of SMARTS and no sampling is 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. Hence, simulating a longer benchmark won't necessarily increase the simulation time for either LiveSample or LiveCI.

Table 2.2 provides additional insight about the speed 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 it is not determined algorithmically. But as illustrated earlier it is typically stable within 5 seconds. The

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	56.750
gromacs	2	246	30.327
h264ref	1	178	24.756
hmmer	1	177	21.247
lbm	3	239	32.679
libquantum	1	194	33.360
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	137.470
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	**5.4**	**315.9**	**54.8**

Table 2.2: Number of checkpoints needed for LiveSample, and LiveCI as well as LiveCI execution time for each benchmark

execution time of LiveCI is determined algorithmically and it varies quite a bit from one benchmark to another. However, for most benchmarks it finishes 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.

Setup and calibration only need to be performed once for each benchmark even when there are big changes in configurations (e.g. LP to HP) or even code changes in simulator. In our experiments the average setup time was 18 minutes and the average calibration time was 3 minutes. The only steps taken in every simulation are LiveCache reload which takes 0.6 seconds on average, branch predictor warmup which takes 1 second, LiveSample timing which takes 3.4 seconds and LiveCI which can add 16 to 132 additional seconds to the simulation time.

2.5.2 Accuracy

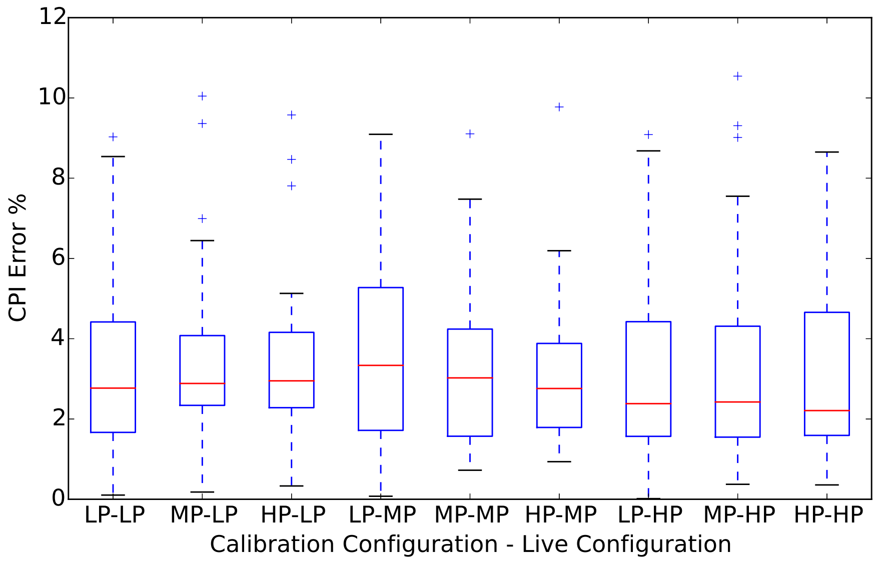

Figure 2.6: CPI error distribution across benchmarks in LiveSim. Each box label shows calibration and live simulation configurations respectively.

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. When evaluating accuracy there are two things to consider: how close is the point estimate to the true value, and how 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. In each experiment, we first ran calibration with one configuration (HP, MP, or LP) and then ran LiveCI with another configuration (we tested all possible combinations). Of the total 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 2.6 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.

Figures 2.7, 2.8, and 2.9 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.

2.5.3 Warmup

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

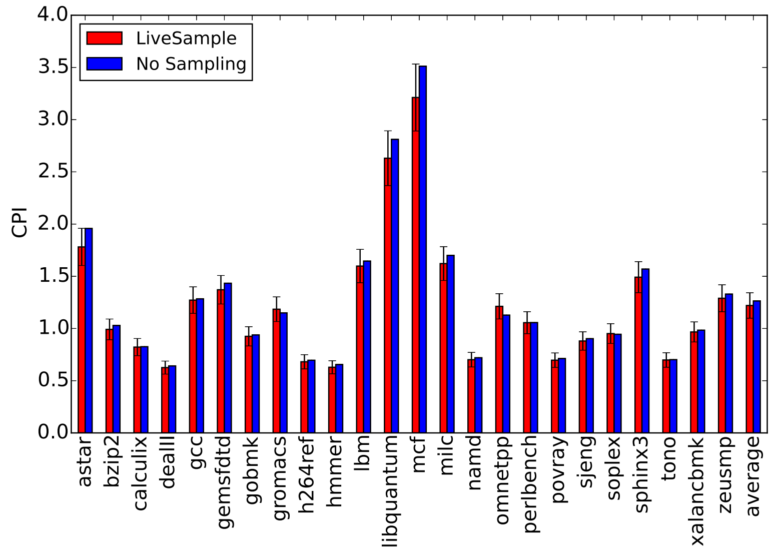

Figure 2.7: LiveCI results of SPEC benchmarks simulating the LP architecture compared to no-sampling simulation. The reported CPI results have 3.32% error and CI estimation is 100% accurate.

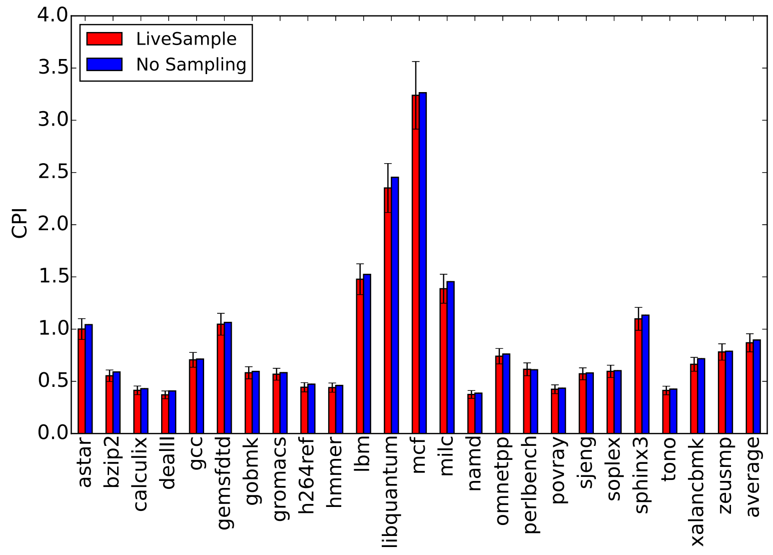

Figure 2.8: LiveCI results of SPEC benchmarks simulating the MP architecture compared to no-sampling simulation. The reported CPI results have 3.32% error and CI estimation is 100% accurate.

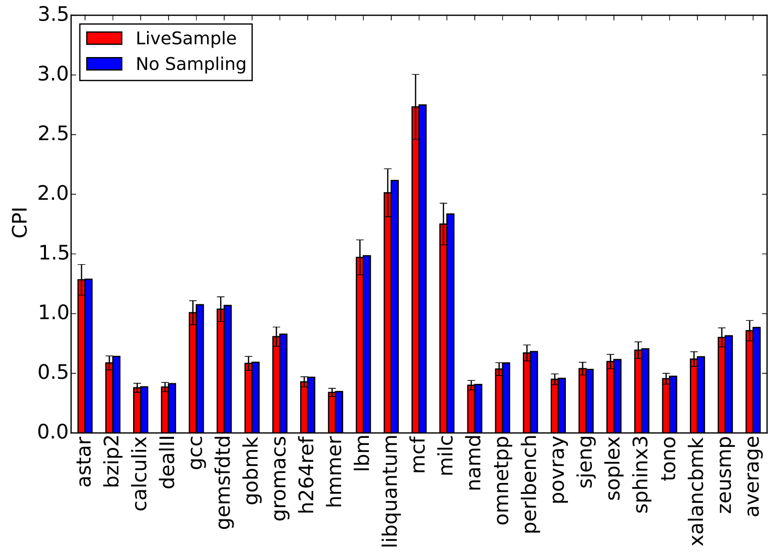

Figure 2.9: LiveCI results of SPEC benchmarks simulating the HP architecture compared to no-sampling simulation. The reported CPI results have 3.33% error in average and CI estimation is 100% accurate.

LiveCache eliminates the need for traditional cache warmup for millions of instructions. To see how it compares, we ran a set of experiments with warmup varying from 0 to 102.4 million instructions. In each experiment we measured and recorded the AMAT error for all benchmarks. Figure 2.10 shows the results comparing them to an experiment in which LiveCache was enabled and no other warmup was done. The evaluation demonstrates that LiveCache results in less AMAT error on average than 50 million instructions of warmup and less maximum AMAT error than 102 million instructions of warmup. The overhead associated with LiveCache per checkpoint 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 are other microar-

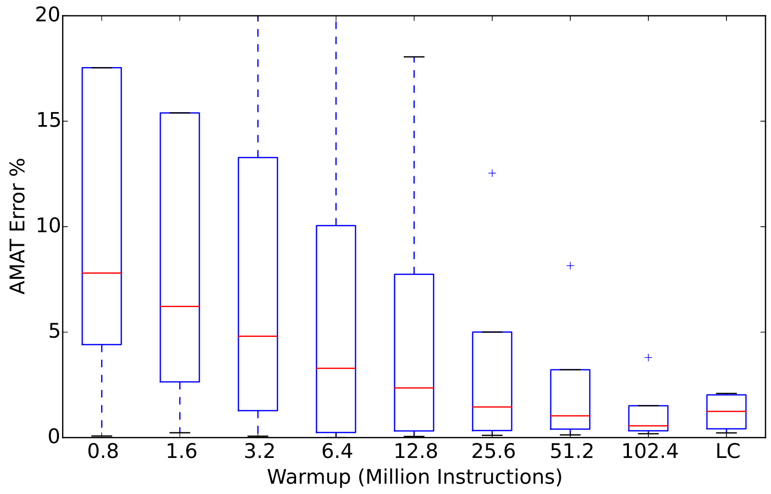

Figure 2.10: Comparison of AMAT error for LiveCache (LC) and traditional cache warmup.

chitectural components that need to be warmed up, especially the branch predictor. Figure 2.11 shows how branch prediction accuracy error rate decreases as the amount of warmup increases. It hits a plateau around 900K instructions, and we found that in practice 1 million instructions was a good value for warmup.

In our evaluations, we used ESESC [5], which has a typical timing simulation speed of 1 MIPS. Since ESESC is fast, it does not need additional functional warmup for the branch predictor. However, a slower simulator may need functional warmup for the branch predictor and a shorter detailed warmup phase.

2.5.4 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. We needed to ensure that enough checkpoints were created for any possible combination of configuration and benchmark that

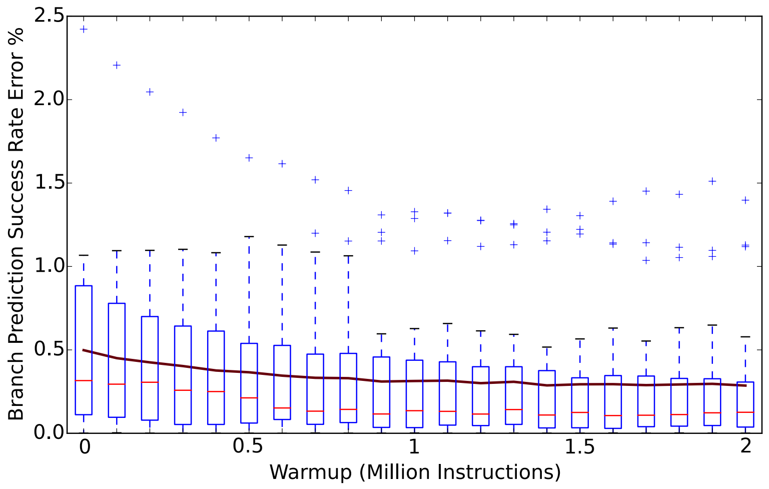

Figure 2.11: Average error of branch prediction statistics based on amount of detailed warmup at the start of a checkpoint.

might be simulated. However, we also needed to limit the number of checkpoints for two reasons: First, since each checkpoint needs to be run during the calibration phase, additional 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.) Second our current implementation potentially uses a large amount of memory for each checkpoint, and if LiveSim uses swap, performance drops dramatically. We think these problems are more significant in simulator development than if LiveSim were used in practice. We also believe that this is mostly an implementation issue rather than something intrinsic to the LiveSim system, and that memory use per checkpoint could be reduced if more time were spent optimizing this bottleneck.

LiveSim uses copy-on-write when creating checkpoints so the memory utilization depends on how many pages the application writes to. We have measured the memory usage

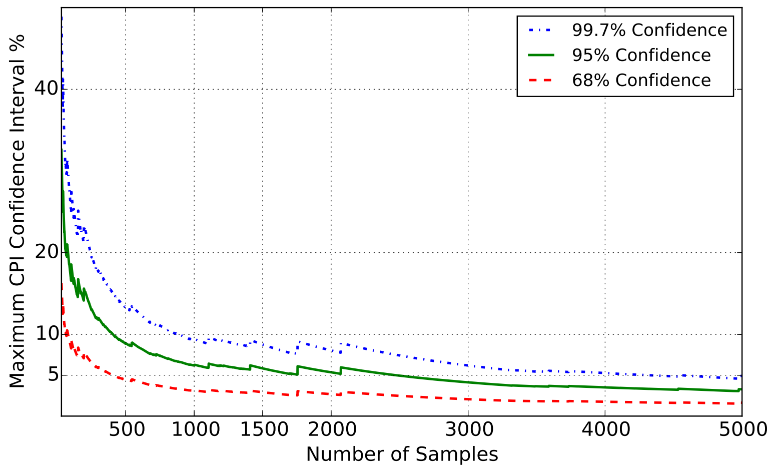

Figure 2.12: Maximum number of samples needed for a given confidence interval and confidence level in LiveSim, estimated using Monte Carlo simulation.

per checkpoint for SPEC benchmarks, and on average, a new checkpoint adds 45 MB memory occupancy. The maximum checkpoint size belongs to MCF, which is 165 MB.

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 thousands of potential samples. 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. Figure 2.12 shows a plot of the results for the three most common confidence levels. The plot indicates that for our target of 10% confidence interval at a 95% confidence level we need roughly 500 checkpoints. Since this is simply a heuristic, to be safe we recommend doubling the number shown here when picking how many checkpoints to actually use, because if LiveSim does not have enough checkpoints it may be unable to meet the

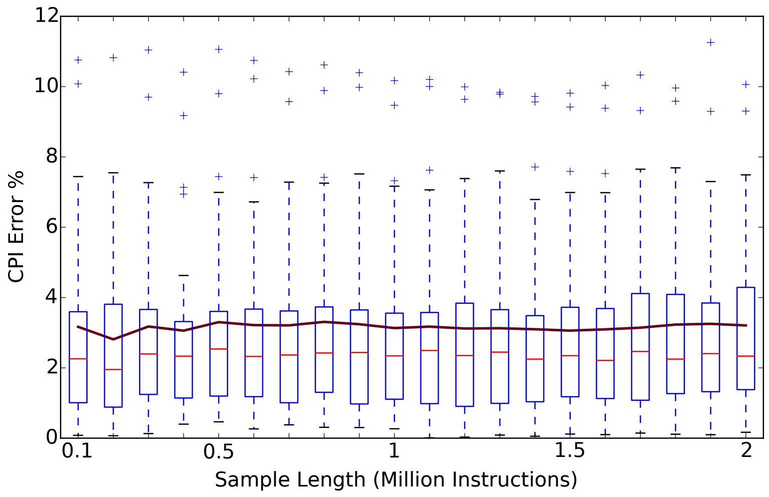

Figure 2.13: The effect of sample size on CPI error. Each box shows the error rate distribution for SPEC benchmarks and the line shows the average error across benchmarks.

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 accuracy but decrease simulation speed. We found a 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 smaller than 100,000 instructions because of communication overhead between the simulation server and the worker nodes. We experimented with sample sizes ranging from 100,000 instructions to 20 times that amount. Figure 2.13 shows that the average error does not decrease with larger sample sizes.

Although Figure 2.13 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 2.14 shows the effect of sample size on simulation time (LiveCI). This figure

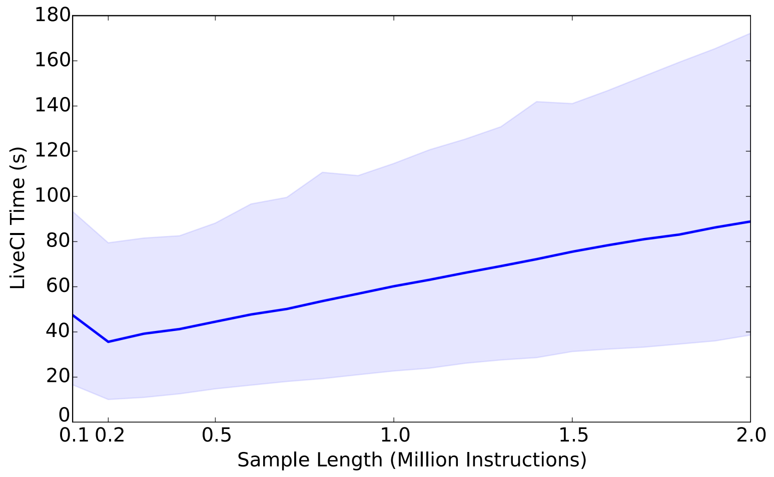

Figure 2.14: The effect of sample size on LiveCI time. The line shows the average LiveCI time across benchmarks and the area is where the actual distribution lies.

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

2.6 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 [72, 62] and statistical sampling [80, 27, 26] for microarchitecture simulation in Section 2.2. 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. [70, 71]. 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 native 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 [76, 56].

SMARTS is effective at minimizing the number of instructions that need detailed simulation; however, its conservative always-on warmup of caches and branch predictor 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. [7] 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 [41, 42] 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. [34] proposed a similar technique that further reduced the amount of warmup required. Luo [51] 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. [59] shows that some workloads may require up to 100 million instructions of warmup for caches larger than 64 MB. They propose a technique that uses native execution to capture a sample of memory accesses and use this technique 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. 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 [28, 4], multithreaded simulation [5, 19, 20], and simulation of soft-errors in caches [74]. 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 being simulated [36, 18, 40]. While these techniques work well for accelerating simulation they fall short of our goal of supporting simulation speeds suitable for interactive use.

2.7 Summary

This chapter presents 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 allows architects to quickly change parameters and get immediate feedback using 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 demonstrate how sampling can be used to support interactive microarchitectural simulation.

The prototype demonstrates the feasibility of LiveSim and obtains accurate results within 5 seconds and bounds the possible error within 41 seconds on average for the benchmarks we evaluated. It is available as an open source project at https://github.com/masc-ucsc/liveos. Although the LiveSim methodology was evaluated using this prototype, the concepts are general and can be adopted for use with other simulators.

Chapter 3

Analysis of PARSEC Scalability

The previous chapter examined the problem of slow simulation speed and demonstrated how to use statistical sampling to dramatically reduce simulation time for single threaded applications. Statistical sampling has also been used to reduce simulation time for multithreaded applications [5, 19]. However, despite this promising research, it is still relatively difficult to use statistical sampling correctly when simulating multithreaded applications.

Multithreaded applications are also difficult for computer architects to analyze during early design space exploration because there are fewer multithreaded applications in widespread use, making it difficult to design benchmark suites. Despite this challenge there have been efforts to create multithreaded benchmarks, and one of the most widely used multithreaded benchmark suites is PARSEC. I used PARSEC extensively while studying ways to improve parallel programming. During this research I discovered that some of the features PARSEC included to simplify simulation of multithreaded applications could potentially result in misleading conclusions.

In this chapter I present a detailed analysis of the scalability of the PARSEC benchmarks—that is, how much faster the benchmarks complete execution when adding additional parallel computing resources. The research presented in this chapter was also published in the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) in April 2016.

3.1 Introduction

The PARSEC benchmark suite is "designed to provide parallel programs for the study [of] CMPs" [14]. It was introduced in 2008 and has been widely used for computer architecture research since then. Developed with the needs of researchers in mind, it has features that make it easier to use with architectural simulators. Each benchmark has multiple input sets, including three that are intended to run with simulators (*simsmall, simmedium, simlarge*), and one that is intended to be representative of a real application (*native*).[1] This allows users to simulate a smaller workload but obtain results representative of a real workload. Each benchmark also defines a *Region of Interest* (ROI) indicating which part of the benchmark executes in parallel. By simulating only the ROI, PARSEC users can reduce simulation time. The ROI is also important for ensuring that results obtained using simulation inputs are representative of real program behavior [12].

Choices in input set size and whether to model the whole program, or only the ROI, can lead to different interpretations when analyzing benchmark results. For instance, Figure 3.1

[1]Two additional input sets (*test* and *simdev*) are included for simulator testing and are not appropriate for scientific studies.

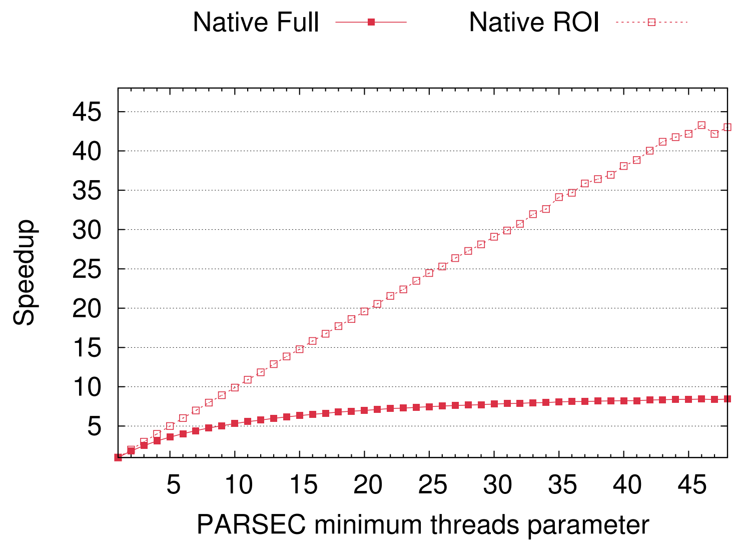

Figure 3.1: Speedup for the full execution of Blackscholes is much less than ROI, when using the native input set and running on a 48-core system.

shows the scalability of ROI only and full benchmark execution for blackscholes running on a 48-core system and using the native input set. The ROI achieves a maximum speedup of 43 times, while the maximum speedup of the full benchmark is less than 9 times.

Prior research characterized the behavior of the PARSEC benchmark suite [8, 10, 13, 12, 14, 15, 16, 21, 65] but none of them compare the runtime scalability of the native input sets with that of the simulation input sets, nor do they compare the runtime scalability of the ROI to that of the full program. This chapter provides this missing characterization and highlights the importance of PARSEC users reporting data about what input set was used in their own papers.

In this chapter I measure the runtime scalability of the four main PARSEC input sets, and I compare the scalability of the ROI with that of the whole program. I do so by running the benchmarks on real multicore systems, varying the number of threads, and measuring the *runtime* of the ROI and the full program for the four different input sets. My contributions are as follows:

- Systematic analysis of the runtime scalability of all PARSEC input sets. I found that in many cases different inputs to the same benchmark have very different scalability characteristics.

- Systematic analysis of the runtime scalability of the ROI compared with full-program execution for all input sets. I identify 6 benchmarks where the scalability of ROI and full program execution has significantly different behavior, while for the other 7 benchmarks ROI and full program execution have very similar behavior.

The rest of this chapter is organized as follows: Section 3.2 provides background about PARSEC; Section 3.3 describes the experiment setup; Section 3.4 provides detailed results; Section 3.5 surveys related work; and Section 3.6 concludes.

3.2 Background

PARSEC was developed between 2005 and 2009 as part of a collaboration between Princeton and Intel [12]. The developers' goal was to create a benchmark suite of emerging parallel workloads that would help architects and researchers design emerging multicore and multiprocessor systems. After its initial release in 2008, PARSEC quickly became popular among computer architecture researchers and has since been widely used in published research.

Six different input sets are defined for each benchmark: *test, simdev, simsmall, simmedium, simlarge, native*. Test and simdev should only be used to test that the benchmark can run. Native is intended to approximate realistic input indicative of how the benchmark application would be used in practice. The remaining three simulation inputs were created by scaling down the native input sets in a way that maintained a representative mix of instructions. The inputs were selected so that serial execution of the native input sets on a real machine should complete in 15 minutes or less, while the simlarge, simmedium, and simsmall inputs should complete execution within 15 seconds, 4 seconds, and 1 second respectively.

The inputs set scaling process skews the amount of time spent in serial phases compared to parallel phases. As a result PARSEC defines an ROI for each benchmark that marks the parallel phase of the benchmark. Although the PARSEC documentation stresses the importance of only using results from the ROI, my results show that this is only important for 6 of the 13 benchmarks.

PARSEC supports three different threading models: pthreads, OpenMP, and Intel Thread Building Blocks (TBB). PARSEC also allows users to specify the *minimum* number of threads run with the benchmark by setting a parameter (*n*) when the benchmark is started. Table 3.1 shows the correspondence between the user-specified number of threads and how many threads the benchmark actually spawns.

In most cases there is a single main thread which spawns *n* worker threads, but some of the benchmarks use a pipelined parallelization model and spawn multiple threads for each one the user specifies. In addition x264 spawns twice as many threads as there are frames in its input (native has 512 frames), but it uses the parameter *n* to limit how many threads run in parallel. Most of the benchmarks allow *n* to range from 1 up to at least 128; however, there are a few restrictions:

Table 3.1: Benchmarks and threads in PARSEC

Benchmark	**Threads**
blackscholes	$1+n$
bodytrack	$2+n$
canneal	$1+n$
dedup	$3+3n$
facesim	$1+n$
ferret	$3+4n$
fluidanimate	$1+n$
freqmine	n
raytrace	$1+n$
streamcluster	$1+2n$
swaptions	$1+n$
vips	$3+n$
x264	$1+2\times frames$

- Facesim is limited to the values 1, 2, 3, 4, 6, 8, 16, 32, 64, 128.
- Swaptions is limited to the number of entries in its input set (16 for simsmall, 32 for simmedium, 64 for simlarge, and 128 for native).
- Fluidanimate requires the number of threads to be a power of 2.
- x264 is limited by the number of frames in its input set.[2] I restricted n to 1–8 for simsmall and 1–32 for simmedium in my experiments.

In this chapter I analyze the 13 benchmarks and input sets first released with PARSEC 2.0.[3] I use the native, simlarge, simmedium, and simsmall input sets, and I use pthreads for all benchmarks except freqmine (which requires OpenMP). I varied the number of threads using the minimum threads parameter n and I report n as the parameter of interest in my results instead

[2] This limitation is not reported when the benchmark is launched, but in my experiments I observed the output was not correct for n greater than 9 for simsmall and n greater than 33 for simmedium.

[3] I used PARSEC 3.0 downloaded from the PARSEC website, but there are minimal changes between 2.0, 2.1 and 3.0 for the benchmarks I analyzed.

of reporting how many threads were actually spawned.

3.3 Experiment Setup

When PARSEC was developed, multicore systems only had a few cores, and most of the initial characterization of PARSEC was done using simulators. In the intervening years the number of cores available in mainstream server systems has increased significantly. I wanted to understand how benchmark performance using the simulation inputs compared to the native inputs when running on real systems, which have overheads and bottlenecks that are not always accounted for when using cycle accurate simulation.

I analyzed the scalability of PARSEC by running the benchmarks on three different real multicore/multiprocessor systems and measuring the runtime. The systems each had different microarchitectures and were developed by two different processor vendors.

The precise scalability results are specific to the systems that I used for my evaluation. However, I expect that the relative scalability trends I identified will apply to most systems because often the underlying cause is differences in workload distribution between the various different input sets or between the ROI and full benchmark execution. The configurations of the systems I used are as follows:

- A single CPU system with 4 cores, 2 threads per core, for a total of 8 logical processors, along with 16 GB of RAM.
- A dual socket system with 8 cores per socket, 2 threads per core, for a total of 32 logical processors, along with 64 GB of RAM.

- A quad socket system with 12 cores per socket, 1 thread per core, for a total of 48 logical processors, along with 64 GB of RAM.

The detailed system specifications are shown in Table 3.2. In the rest of this chapter I refer to the 8-logical processor system as *M8*, the 32-logical processor system as *M32*, and the 48-logical processor system as *M48*.

Table 3.2: Specifications of systems used for experiments

System	**Configuration**
M8	1 x Intel Xeon E3-1275 v3 (4 core, 2-way SMT) 32 KB L1, 256 KB L2, 8 MB L3 cache 16 GB DRAM
M32	2 x Intel Xeon E5-2689 (8 core, 2-way SMT) 32 KB L1, 256 KB L2, 20 MB L3 cache 64 GB DRAM
M48	4 x AMD Opteron 6172 (12 core) 64 KB L1, 512 KB L2, 5 MB L3 cache 64 GB DRAM

All of the systems used the x86_64 version of Arch Linux with version 3.18.6-1 of the Linux kernel. All benchmarks were compiled with version 4.9.2 of gcc/g++. I disabled ASLR but did not do any other special tuning. The OS and hardware were allowed to schedule threads and control CPU frequency using default scheduling algorithms. I used PARSEC hooks to identify the ROI, and for each configuration I recorded the runtime of the ROI and full benchmark execution.

I repeated each experiment at least 10 times and calculated the mean and the confidence interval at a 95% confidence level. For configurations where the initial confidence interval after 10 runs was not within 5% of the mean I repeated the experiment until the confidence interval was within 5% of the mean. The speedup results I present are computed by dividing the mean execution time of a system, input set, and ROI or full configuration with a single thread by the mean execution time of the same configuration with multiple threads.

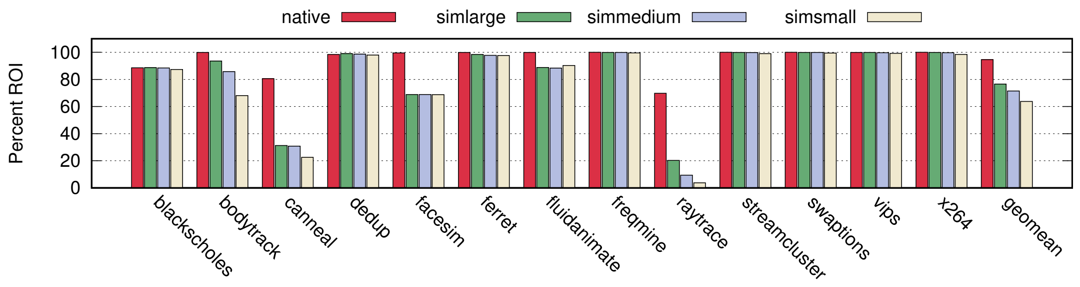

Figure 3.2: Percentage of full execution time that is in the ROI measured by running benchmarks on M8 with a single thread. Seven of the benchmarks spend nearly 100% of their execution time in the ROI, while the other six spend significantly less for at least some input sets.

3.4 Results

The two main questions that I sought to answer were: how does the scalability of the ROI compare to the scalability of full benchmark execution? And how does the scalability of each of the simulation inputs compare to the native input set? In this section I present the results of my analysis. First in Section 3.4.1 I analyzed the theoretical maximum speedup of the full input sets and identified where it is limited in comparison to the ROI. Next I analyzed the actual speedup results measured using real systems. Section 3.4.2 presents results for the maximum speedup for each of the input sets; Section 3.4.3 compares the average speedup of the ROI to full, and of the native input set to each of the simulation inputs. Section 3.4.4 presents a graphical view of scalability trends for each of the benchmarks along with some insights about the reasons different input sets have differing scalability characteristics.

3.4.1 ROI Percentage

The ROI is the only part of the PARSEC benchmarks that executes in parallel, and thus the only part where parallel execution can speed up the benchmark. Figure 3.2 shows the percentage of time that each benchmark spent executing the ROI with the number of threads $n = 1$. Seven of the benchmarks spent over 98% of their execution time in the ROI for all input sets, and so the ROI percentage is unlikely to limit potential speedup from parallel execution.

The other six benchmarks spent less than 90% of their execution time in the ROI for at least some of their input sets, and the maximum theoretical speedup is less than 10 times. Table 3.3 lists the percentage of time each of these benchmarks spends executing in the ROI along with the maximum theoretical speedup. The maximum speedup is calculated using Amdhal's law and not listed for input sets where it is not a bottleneck.

Table 3.3: ROI percentage and maximum theoretical speedup

Benchmark	Native		Simlarge		Simmedium		Simsmall	
blackscholes	89	9	89	9	88	9	87	8
bodytrack	100	–	94	16	86	7	68	3
canneal	81	5	31	1	31	1	23	1
facesim	100	–	69	3	–			
fluidanimate	100	–	89	9	88	9	90	10
raytrace	70	3	20	1	9	1	4	1

- **Blackscholes**: The time outside of the ROI is spent initializing the input array and writing out the results. This amount of work scales linearly with the input set size; consequently a larger input set does not improve the scalability. I confirmed this experimentally by creating an input set 10 times larger than the native input set included with PARSEC and measured the same ROI percentage for this larger input set.

- **Bodytrack**: This uses helper threads to load image data for the next frame concurrently with the threads processing the current frame. However, the thread pool must be created and the first image loaded before any parallel computation can occur. This initialization is done before the ROI starts. A negligible percentage of total execution time is consumed for the native input set, but a significant fraction of time is consumed for the two smallest input sets.

- **Canneal**: The majority of the time outside of the ROI is spent initializing the netlist. This initialization time is proportional to the size of the netlist. However, there are two ways to increase the work done by the benchmark. Either the netlist size can be increased, or the number of temperature steps can be increased. I tested using 10 times as many temperature steps for the native input set with the same netlist, which increased the ROI percentage to 98%.

- **Facesim**:[4] There is a fixed amount of work done outside the ROI based on the facial features that will be animated. The work done to animate each frame is in the ROI, and this work scales with the number of frames. The native input set processes 100 frames, while the simulation inputs process only a single frame.

- **Fluidanimate**: The benchmark simulates fluid dynamics for use in animation sequences. The work outside of the ROI is mostly involved with partitioning how a single animation frame is processed. The work in the ROI scales with the number of frames in the workload, and adding frames adds more work in the ROI. The native input set has 500 frames,

[4]Facesim only has one simulation input set.

while the simulation inputs only have 5 frames.

- **Raytrace**: It is possible to increase the amount of work in the ROI by rendering more frames. The native input set renders 200 frames; when I increased this to 2,000 frames the ROI increased to 95%.

3.4.2 Maximum Speedup

The fraction of time that a benchmark spends executing parallel code is not the only limiting factor on its scalability. In many cases other factors, such as inter-thread communication and imbalances in workload distribution, limit scalability more than the fraction of code that can be executed in parallel.

Figures 3.3 and 3.4 show the maximum speedup for each input set on the M48 and M32 systems respectively. Since M48 has 48 cores the maximum expected speedup is 48 times (assuming no superlinear effects). The M32 system has 16 cores, and each core can execute 2 threads simultaneously, so the maximum linear speedup is 32 times. However, since simultaneous multithreading shares core resources it is unlikely that benchmarks will have linear speedups for all 32 threads.

Blackscholes executing the ROI of the native input comes closest to achieving the maximum linear speedup. But most of the benchmarks and input set combinations have a much lower maximum speedup. As expected, the six benchmarks I identified with low ROI percentage showed a big difference between the speedup of full and the ROI. Thus results from ROI and full may not be comparable in these cases and it is important for users to properly specify which region of the benchmark they measured.

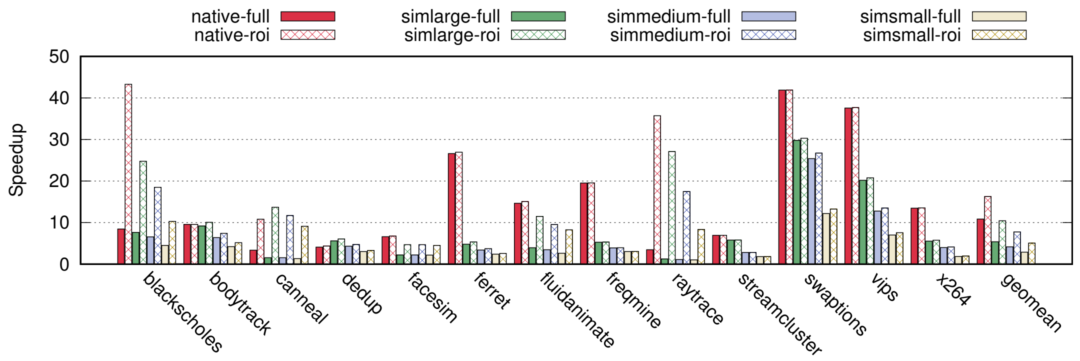

Figure 3.3: Maximum speedup measured on M48 for each benchmark, region, and input set combination.

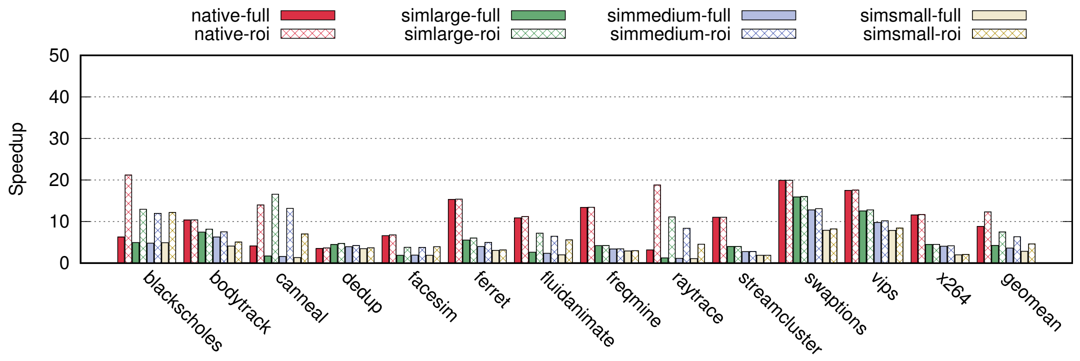

Figure 3.4: Maximum speedup measured on M32 for each benchmark, region, and input set combination. Scale is same as Figure 3.3.

The divergence between the speedup of ROI for each of the native inputs and the speedup of ROI for each of the simulation inputs is potentially more problematic because results obtained using simulation inputs may not be representative of actual application behavior. But in many instances simulators are used because real hardware is not available.

3.4.3 Quantifying Similarity

The maximum speedup results presented in the previous section demonstrate that there are bottlenecks which limit the scalability of the PARSEC benchmarks, and that these bottlenecks affect different benchmark and input set combinations in different ways. But comparing maximum speedup only shows the scalability difference for a single data point. I also quantified the average scalability difference between ROI and full, and between the native input set and each of the simulation inputs.

Table 3.4: Speedup % difference: ROI and full

Benchmark	Native	Simlarge	Simmedium	Simsmall
blackscholes	174	129	108	93
bodytrack	0	6	11	15
canneal	151	528	463	368
dedup	2	4	4	7
facesim	1	55	–	
ferret	1	9	14	6
fluidanimate	1	59	65	77
freqmine	0	0	1	2
raytrace	394	887	735	414
streamcluster	0	0	0	1
swaptions	0	1	2	2
vips	0	2	4	5
x264	0	2	4	3
geomean	1	9	11	14

ROI and Full: Table 3.4 shows the percentage difference between the average speedup of the ROI and full benchmark execution averaged over all data points. As an example of how this was calculated, consider blackscholes: When running with the full native input set on the M32 system with 8 threads results it has a 4.3 times speedup compared to a single thread, while the speedup of 16 threads is 5.6 times. For the same configuration the speedup of the ROI is 7.7

Table 3.5: Speedup % difference: Native and Simulation

Benchmark	Simlarge		Simmedium		Simsmall	
	full	roi	full	roi	full	roi
blackscholes	10	28	20	54	41	104
bodytrack	11	9	39	24	96	69
canneal	104	15	109	10	142	40
dedup	30	31	15	16	22	24
facesim	89	21	–			
ferret	218	192	337	289	505	471
fluidanimate	99	21	115	24	152	30
freqmine	150	149	230	228	347	337
raytrace	134	28	162	62	176	169
streamcluster	231	231	961	962	3449	3482
swaptions	16	15	33	31	39	36
vips	31	29	75	69	197	181
x264	117	114	140	131	127	120
geomean	61	49	96	79	155	139

times for 8 threads and 15.0 times for 16 threads. The average speedup of these two points is 5.0 times for full and 11.4 times for ROI. The difference of these averages is 6.4, and this difference is 128% of the average of full. The reason for presenting the difference as a percentage of full is that the same absolute speedup difference represents less relative error if the total speedup for the full execution is larger. This explanatory example uses only two data points, but the results in Table 3.4 were calculated by taking the average of all of the data points for each configuration that was compared.

For six of the seven benchmarks that I identified as spending nearly all their time in the ROI (dedup, freqmine, streamcluster, vips, x264) the average speedup difference is usually less than the 5% margin of error I set when computing average execution time.

Although ferret spends 98% of its execution time in the ROI for the simulation inputs, it also has low scalability. For instance on the M32 system the maximum speedup for the simmedium input set is 4 times for full execution and 5 times for ROI. As a result ferret stands out as having a significant percentage difference between ROI and full execution despite spending nearly all of its execution time in the ROI when running with a single thread.

The remaining six benchmarks are ones I identified as spending a significant percentage of benchmark execution time in the single threaded region. Of these, bodytrack has a relatively small percentage difference between ROI and full, but this is because its overall maximum speedup is less than 10 times.

The other five benchmarks (blackscholes, canneal, facesim, fluidanimate, raytrace) have very large relative differences between the speedup of ROI and full for at least some of the input sets. Consequently it is not a good idea to compare full and ROI results when using these benchmarks. This is particularly noteworthy for blackscholes, canneal, and raytrace, where even the native input sets have large speedup differences between ROI and full.

Native and Simulation: Table 3.5 shows the average speedup difference between the ROI and full for each of the input sets. These results were computed using a methodology similar to the ones used to compare the percentage difference in ROI and full speedup. In this case the speedup for each simulation input was compared to that of the native input. The percentage difference is calculated by dividing the average speedup difference of the native input by the average speedup difference of the simulation input.

The first thing that stands out is that none of simulation inputs are particularly similar to the native input set. The best case is for bodytrack executing the ROI for the simlarge input set at 9% speedup difference compared to the native input set. As expected the larger simulation input sets are generally more similar to the native input set. However, even for the ROI of simlarge the geometric mean for all the benchmarks of the speedup difference compared to the native input set is nearly 50%.

The most noteworthy outlier is streamcluster where in the worst case for simsmall the speedup differs by over 3,000%. The reason for this is that streamcluster is able to maintain at least slight speedup when executing the native input set. But it suffers extreme slowdown for small simulation inputs. For instance when running on M48 with 48 threads using the simsmall input set, streamcluster is 100 times slower than when running the same input set with a single thread.

3.4.4 Measured Scalability

The previous section quantifies the similarity of the different input sets. In this section I present a visualization of the scalability data. I varied the number of threads from 1 to the number of logical processors in the system and measured the runtime of full execution and ROI for all benchmarks using all input sets. For each data point I plotted the speedup of the multithreaded benchmark execution compared to executing with a single thread. Although I plotted 8 data sets for each benchmark, in some cases fewer points are visible because the ROI and full results overlap completely. I split the results into four figures, which are interspersed along with benchmark analysis. Figure 3.5 shows the scalability of blackscholes, bodytrack, and canneal; Figure 3.6 shows dedup, facesim, and ferret; Figure 3.7 shows fluidanimate, freqmine, and raytrace; Figure 3.8 shows streamcluster, swaptions, and vips; and Figure 3.9 shows the scalability of x264.

- **Blackscholes** has the best scalability of any benchmark for the ROI with the native input

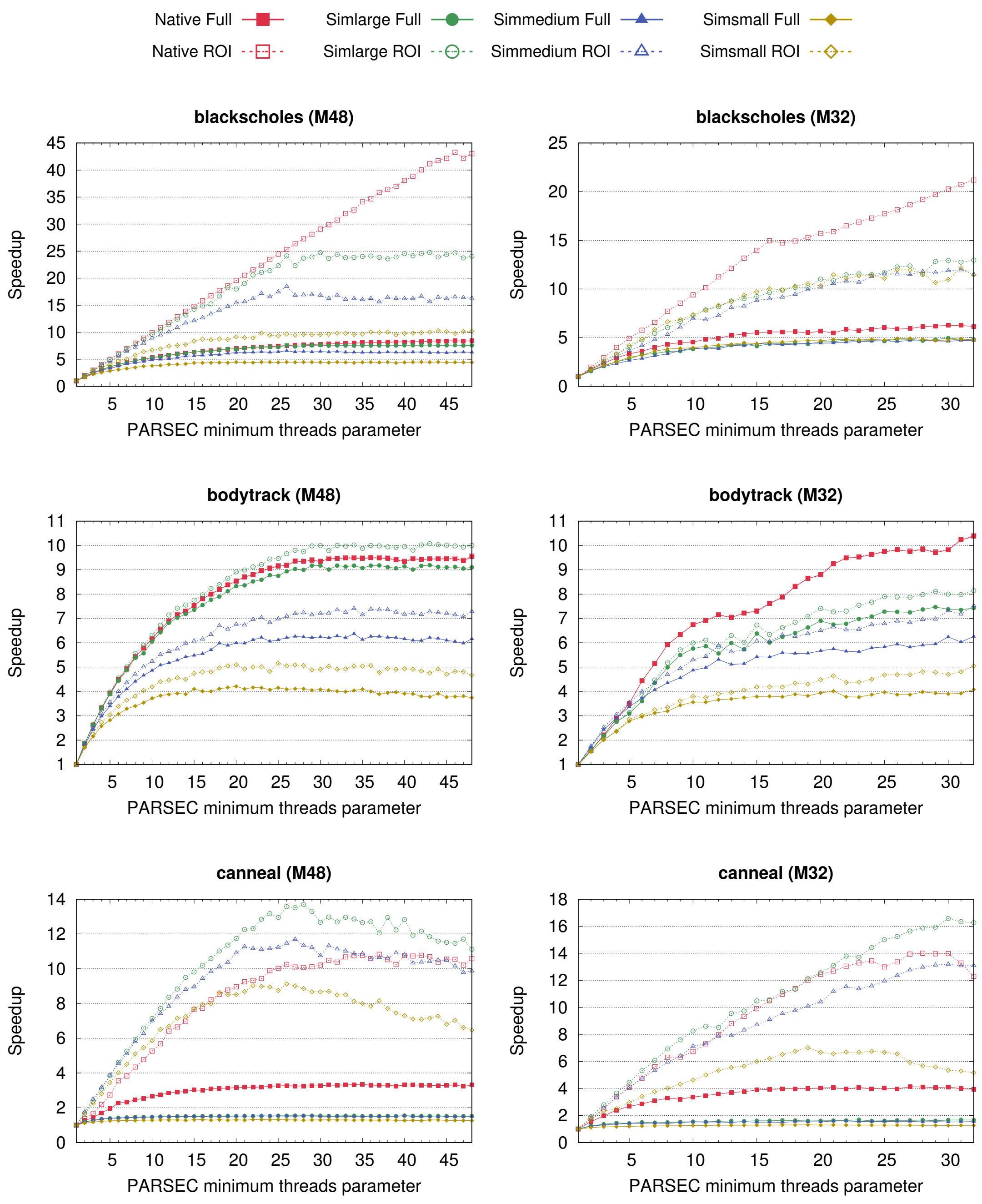

Figure 3.5: Scalability of blackscholes, bodytrack, and canneal on the 48-core M48 system and 32-thread M32 system. Solid points indicate results for full input, and hollow points are for ROI only.

set. The maximum speedup of the full execution is limited to less than 9 times because of the serial portion of the benchmark. It is also noteworthy that even for the ROI, the simulation inputs do not scale as well as the native input set for large numbers of threads. On M48 the execution time of the ROI for the simsmall input when running with 48 threads is 14.4 ms. I tested a modified version where the worker threads spawn and return immediately without doing any work, and the ROI time dropped to 4 ms. Thus it does not appear that the thread creation overhead prevents further scaling of the benchmark.

- **Bodytrack** has relatively good similarity between ROI and full for native and simlarge. For simmedium and simsmall, speedup of full and ROI drops noticeably, particularly on M48 with many threads.

- **Canneal** has limited scalability for full for all of the input sets with a maximum speedup of less than 5 times for native input. Even for ROI only, the total scalability is limited and on M48 the speedup of ROI drops after roughly 30 threads. Canneal uses atomic operations to synchronize data between threads; as a result, adding more threads increases the chance of conflicts between threads. There is a tradeoff between the size of the netlist and the number of temperature steps. I think this is why the simlarge input has higher scalability than native. When I tested with an input with more time steps than native I found that the scalability of full improved, but the scalability of ROI dropped.

- **Dedup** has an erratic speedup pattern due to a work distribution imbalance between threads. Dedup creates queues for partitioning work between threads, and the number of queues created is $n\ threads/4 + n\ threads \bmod 4$. Afterwards each queue is assigned 4

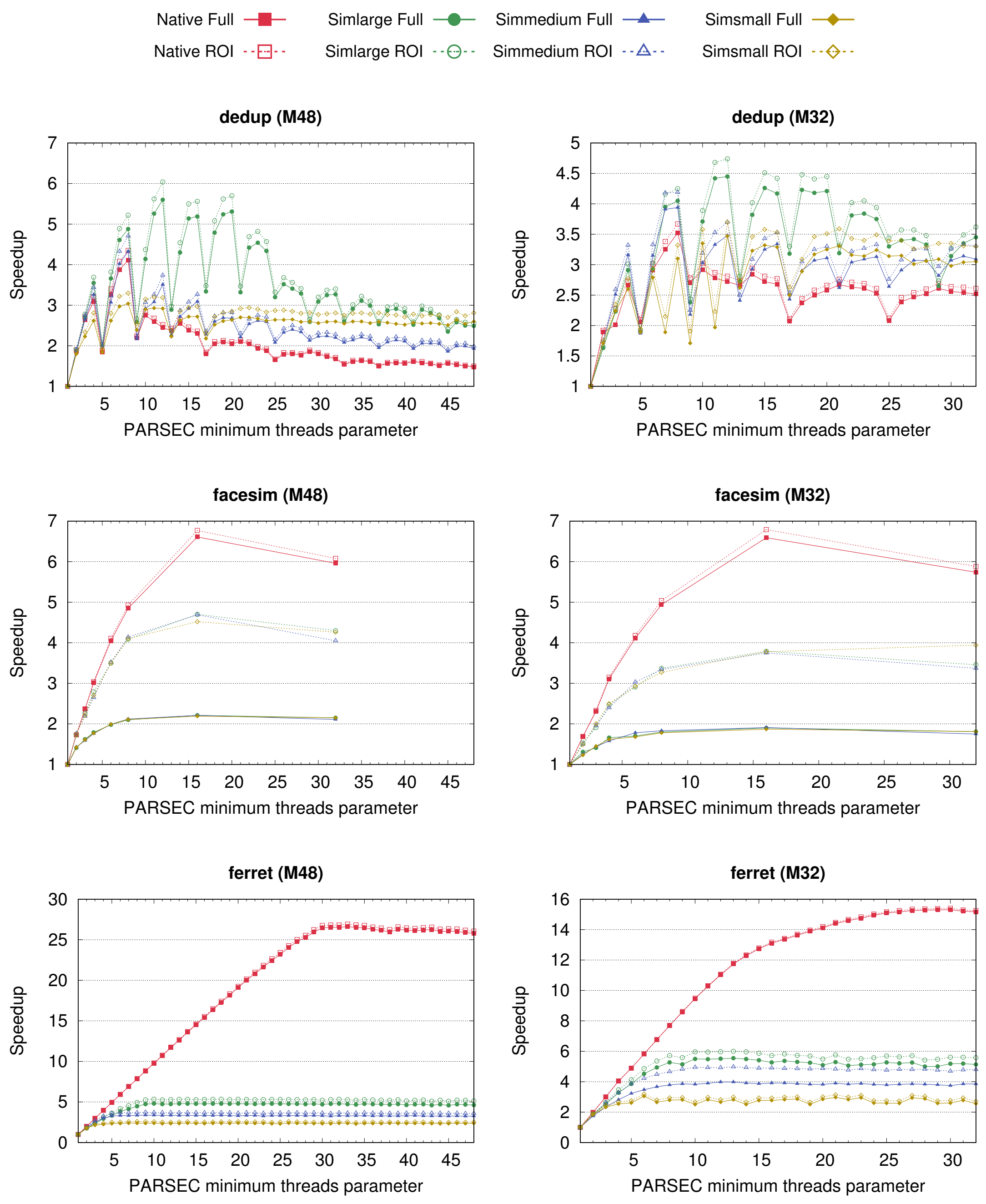

Figure 3.6: Scalability of dedup, facesim, and ferret on the 48-core M48 system and 32-thread M32 system. Solid points indicate results for full input, and hollow points are for ROI only.

worker threads, except the last queue, which has *n threads* mod 4 threads. This workload imbalance causes the scalability to be best when the number of threads is a multiple of 4. It is also noteworthy that the simlarge input set has a maximum speedup with 12 user threads while the other inputs sets have a maximum speedup with 8 user threads, and the maximum speedup of simlarge input is much higher than that of native for both ROI and full. I suspect this is caused by differences in the input set data. Dedup performs deduplication, and simlarge achieves 2.38X compression factor, while native, simmedium, and simsmall achieve 1.05X, 1.06X, and 1.09X compression respectively.

- **Facesim** has much lower scalability for the simulation input than for the native input set for both full and ROI. The lower scalability of full is explained by the fraction of the benchmark that is in the ROI. Since simulation and native inputs both process the same frame, I expect the differences between the scalability of the two inputs are related to additional initialization overhead that is included in the ROI time but not amortized over multiple frames.

- **Ferret** also has much lower scalability for the simulation inputs than for native. It uses pipelined parallelization, but the first and last stages in the pipeline only spawn a single thread. I experimented with removing these stages from the ROI, but that did not improve the ROI scalability for the simulation inputs. Bienia [12] notes that the simulation inputs for ferret have less opportunity for parallel execution, while Pusukuri et al. [65] found that the speedup for the native input set was limited by lock contention. Although I do not have a definitive explanation for the scalability difference, I do note that the simulation inputs reach their maximum speedup much earlier than when using the native input, and this maximum speedup is much lower than when using native inputs.

- **Fluidanimate's** low scalability for full execution of the simulation inputs is explained by the lower fraction of the benchmark in the ROI. However, even for the ROI the scalability of the simulation inputs is lower than that of the native input set. I suspect this is due to the thread communication overhead that is proportional to the number of particles in the input set, and so the overhead is more for the smaller input sets. It is also noteworthy that the speedup of simsmall on M48 *drops* when increasing from 16 to 32 threads.

- **Freqmine** is another benchmark with much lower scalability for the simulation inputs than the native input set. Both the native and simulation inputs have a high percentage of their work included in the ROI. However, freqmine uses OpenMP and I suspect that the smaller simulation inputs have their scalability constrained by the serial portions of the workload and that the parallel loops are too small to provide much speedup.

- **Raytrace** has extremely limited scalability for the full execution because of the low fraction of the benchmark in the ROI. Increasing the number of frames from the 200 used by the native input set to 2,000 increases the maximum speedup from 3.5X to 16X. Cebrián et al. [21, 22] argue that the version of raytrace in PARSEC should be replaced by one with SIMD instructions. It is also noteworthy that the ROI for the simulation inputs does not scale nearly as well as the native input set, particularly for simmedium and simsmall.

- **Streamcluster** uses barrier based synchronization, and the scalability for simmedium and

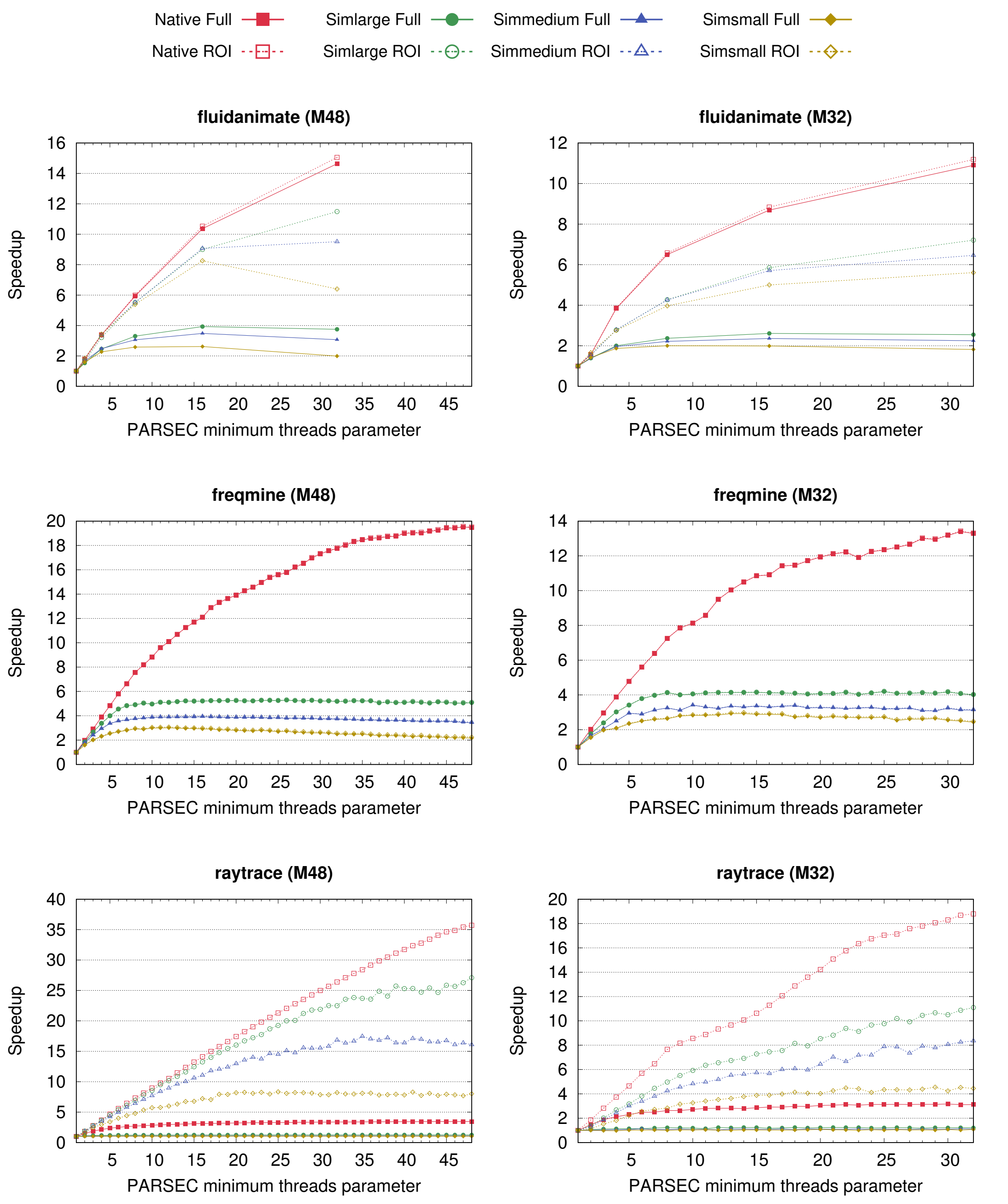

Figure 3.7: Scalability of fluidanimate, freqmine, and raytrace on the 48-core M48 system and 32-thread M32 system. Solid points indicate results for full input, and hollow points are for ROI only.

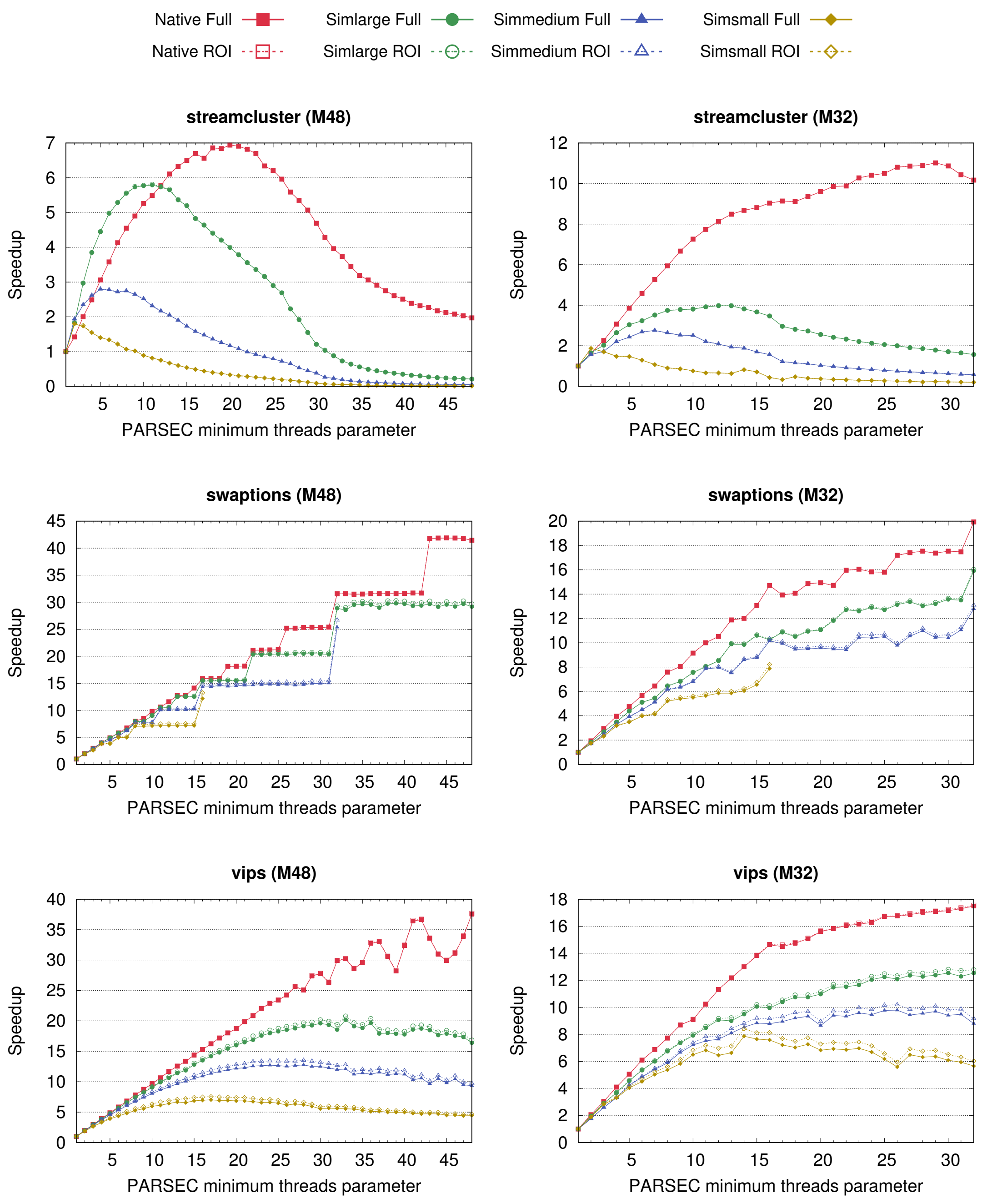

Figure 3.8: Scalability of streamcluster, swaptions, and vips on the 48-core M48 system and 32-thread M32 system. Solid points indicate results for full input, and hollow points are for ROI only.

simsmall is much worse than for native. On M48, running simsmall with 48 threads is 68 times *slower* than running with 1 thread. Of even greater concern, the maximum speedup for simsmall on M48 occurs when the PARSEC minimum thread parameter is 2, and afterwards the performance worsens as more threads are added. Roth et al. [69] observed similar behavior and attributed it to inefficiency in the barrier synchronization.

- **Swaptions** has very similar scalability for the ROI and the full program. The simulation inputs also match native scalability at some points but diverge at others. The application itself has a stairstep type of scalability caused by an imbalance in workload distribution between the threads. Earlier papers [65, 69] also identified this problem and my results support their observations.

- **Vips** also uses a pipelined parallel programming model with two threads for performing I/O and then *n* threads for processing the data, and I suspect this is what causes worse scalability for smaller input sets. Bienia's dissertation [12] notes that the size of the output buffers can limit parallelism, and that this problem may be corrected in future versions of PARSEC. Although most of my tests used the benchmark code from PARSEC 3.0, for vips I reverted to PARSEC 2.1 because the source code for PARSEC 3.0 was missing ROI annotations. After noting Bienia's comment I tested using the vips code in PARSEC 3.0, but observed the same scalability as PARSEC 2.1.

- **x264** also has much lower scalability for the simulation inputs than for the native input set. The x264 application compresses an input video stream, and the simulation inputs

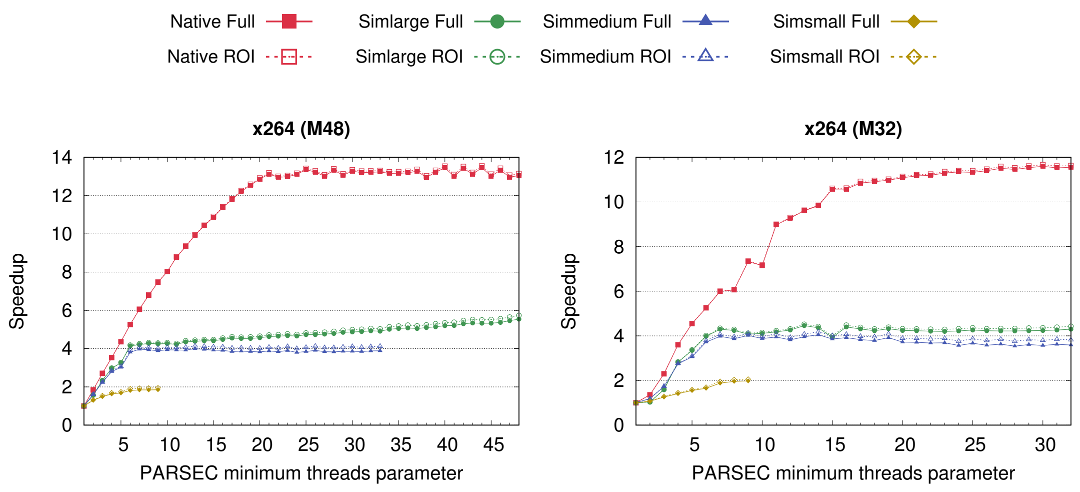

Figure 3.9: Scalability of x264 on the 48-core M48 system and 32-thread M32 system. Solid points indicate results for full input, and hollow points are for ROI only.

have fewer and smaller frames than the native input. The smaller inputs have more dependencies between frames, and this limits the overall potential for achieving parallel speedup.

3.5 Related Work

Christian Bienia's 2011 dissertation [12] is the most comprehensive study of the PARSEC benchmarks and extends material published earlier [13, 14, 15, 16]. The characterization of PARSEC in Bienia's work relies on simulation and is intended to be machine independent. In contrast, my characterization is done using real machines and I focus on runtime as the performance metric of interest.

Pusukuri et al. [65] developed *Thread Reinforcer* to pick an optimal number of threads for a parallel application. They evaluated their proposal using 8 of the 13 PARSEC benchmarks running on a 24-core system. They used the native input sets for their evaluation, and like me, they found that the maximum speedup of the full execution of blackscholes and canneal was limited due to the fraction of serial code. Part of the motivation for my study was noting differences in Bienia and Pusukuri et al.'s characterization of PARSEC's scalability.

Several other papers have also studied the problem of thread scheduling and included characterization of some of the PARSEC benchmarks as part of their evaluation [49, 53, 54, 55, 58, 66, 73].

There are also several papers that characterized the performance of PARSEC. Like me, Bhadauria et al. [10] studied the scalability of PARSEC workloads using real machines, but they did not compare different input sets, or ROI and full program execution. Barrow-Williams et al. [8] analyzed communication patterns between threads in PARSEC and SPLASH using Simics. Bhattacharjee and Martonosi [11] analyzed TLB behavior of PARSEC benchmarks using a combination of native execution and simulation. Ferdman et al. [39] analyzed the single threaded performance of a variety of benchmark suites including PARSEC. Cebrián et al. [21, 22] proposed extending PARSEC with better support for SIMD hardware. Bryan et al. [17] examined how synchronization overhead and other system-level effects limited the potential scalability of PARSEC 1.0 benchmarks.

Several papers have also analyzed the scalability of some of the PARSEC benchmarks while developing techniques to find performance bottlenecks in parallel applications [29, 33, 35, 38, 43, 69].

3.6 Summary

Benchmarks are a critical part of the quantitative approach to computer architecture research. But the complex interaction of the many layers of the computing stack, coupled with the slow speed of architectural simulators, forces architects to make approximations when simulating benchmark execution. PARSEC provides reduced size input sets and demarcates an ROI as ways to reduce simulation time while still approximating the behavior of the actual workload. However, initial characterization of the PARSEC input set scalability was performed using simulation, and it may have overlooked bottlenecks that exist in real systems.

My characterization of the scalability of PARSEC using real multiprocessor systems shows two important ways in which results can vary depending on benchmark parameter selection. First, the choice between measuring ROI and full only has a significant impact on the scalability of seven of the benchmarks. Second, I showed that the scalability of the four different input sets that PARSEC provides differs dramatically when the benchmarks are executed on a real system. I recommend that users of PARSEC report the parameters selected for their experiments (both input set size and whether measuring ROI or full).

Chapter 4

Overhead of Deoptimization Checks in the V8 JavaScript Engine

In the first chapter of this dissertation I examined the problem of slow simulation speed, proposed a way to accelerate simulation using statistical sampling, and evaluated it using the single-threaded SPEC CPU 2006 benchmark suite. Next I analyzed the multithreaded PARSEC benchmark suite and examined whether simulation results obtained using PARSEC are representative of what would be attained in an actual system. Both the SPEC CPU 2006 benchmarks and PARSEC have relatively little interaction with the operating system or other runtime system software, and these benchmarks are often simulated using systems that exclude kernel code from simulation time. However, there is a wide diversity of workloads running on modern systems, and many CPU bound workloads are written in languages other than statically compiled C++, C, or Fortan, which are used for PARSEC and SPEC CPU 2006.

In this chapter I analyze overhead associated with executing JavaScript, a dynamically typed language that is typically executed using a multi-tier JIT compiler. I focused my analysis on the V8 JavaScript engine and performed experiments using real systems in order to measure actual benchmark behavior when using a complex software stack. The results demonstrate the complexity of evaluating trade-offs between CPU microarchitecture and JIT compiled code, something that is difficult for simulators to evaluate that need to be tuned to ensure they are providing realistic results. The results of this study were published in the IEEE International Symposium on Workload Characterization (IISCW) in September 2016.

4.1 Introduction

Dynamic languages such as JavaScript, Python, and Ruby have become extremely popular in recent years. Applications written in dynamic languages have historically executed much more slowly than those written in statically typed languages, in part because variable-type information is only known at runtime, and not at compile time. The performance of JavaScript runtime and compilers has improved dramatically, and JavaScript is now widely used in client and server applications requiring high performance.

JavaScript implementations attain high performance by using multi-tiered just-in-time (JIT) compilation combined with type speculation [30, 52, 63, 64]. The initial tier quickly parses JavaScript source code and executes it with an interpreter or a fast compiler that does not perform any optimizations. As this code runs it stores the types of variables used by a function in inline caches using techniques first developed for Smalltalk [31] and Self [44]. After type information has been collected, frequently executed code is recompiled with an optimizing compiler, which assumes that variable types will not change even though this is not guaranteed by the language semantics. In order to guarantee that these assumptions are not violated, the compiler inserts checks to ensure that the assumptions about types still hold. If the condition expected by the checks fails to hold then the function stops running the optimized code and switches back to the unoptimized version. The process is called deoptimization[1], and the checks are called deoptimization (or deopt) checks.

The cost of each conditional deoptimization check is relatively small, typically from 1 to 3 instructions. An example of a common check is the comparison of an object's pointer to its type with a pointer to the expected type. The comparison is followed by a conditional branch that branches to deoptimization code in case the pointers do not match. Although the cost of each check is small, the checks can occur very frequently for applications running mostly in optimized mode. Figure 4.1 shows the frequency of deoptimization checks for the Chrome V8 JavaScript engine when running the Octane benchmark suite [1]. Over half of the benchmarks execute at least 5 checks per 100 instructions, and many of the checks are comprised of more than 1 instruction.

In this chapter I analyze the type and frequency of deoptimization checks used in the Chrome V8 JavaScript engine, as well as the performance overhead associated with these checks. Although my analysis focuses on Google's Chrome V8 JavaScript engine, all major JavaScript engines including those from Apple, Mozilla, and Microsoft have deoptimization checks, and deoptimization checks are present in JITs for Python, Ruby, and other dynamic languages. I believe that the insights gained from studying deoptimization check overhead for

[1]Also sometimes called "bailout."

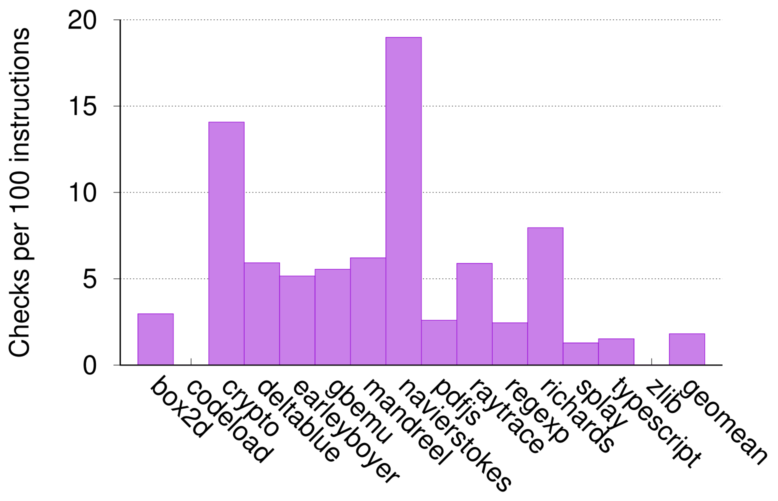

Figure 4.1: Frequency of deoptimization checks for the Octane benchmarks when executed with the V8 JavaScript engine. Eight of the 15 benchmarks have more than 5 checks per 100 instructions, and many of the checks are comprised of multiple instructions.

a specific JIT implementation can be useful in other contexts since speculative optimization is a widely used technique in JITs for dynamic languages.

I identified four main types of deoptimization checks that are used in Chrome V8 and characterized the frequency of checks when running the Octane Benchmark suite. I also analyzed the performance overhead of the conditional branches used for the checks by running the benchmarks on four different real systems with checks enabled and disabled. I found that modern wide issue out-of-order processors are usually able to absorb the overhead of executing the extra check instructions with minimal performance overhead. However, narrower issue systems suffer a larger slowdown that corresponds with the instruction count overhead.

This chapter's contributions are:

- Characterizing the performance overhead of deoptimization checks for the Chrome V8

JavaScript engine on four different machines.

- Showing that performance overhead does not always correlate with instruction count overhead, but rather depends on a combination of the workload and the system micoarchitecture.
- Categorizing the types of deoptimization checks and characterizing their execution frequency when running optimized code.

The rest of this chapter is organized as follows: Section 4.2 provides background information about JITs for dynamic languages, the V8 JavaScript engine, and the Octane benchmark suite that I used for my experiments; Section 4.3 shows the types of frequency of deoptimization checks that I observed; Section 4.4 provides an analysis of performance overhead associated with deoptimization checks; Section 4.5 surveys related work; and Section 4.6 concludes.

4.2 Background

4.2.1 Dynamic Languages

Dynamically typed languages do not require developers to declare types of variables in the program source code; instead, types are determined at runtime based on the state of the running program. These languages are often used to improve programmer productivity [50, 61], even though they can have a significant performance overhead.

The simplest way of implementing a runtime system for a dynamic language is to use an interpreter that directly executes statements in the program source code. Each time an operation is performed, the interpreter checks the type of the variables used in the operation and selects the method that is appropriate based on the variable types.

To make this more concrete, consider the snippet of JavaScript code shown in Listing 4.1. In this example the function `twice` has different behavior depending on the type of its input parameter. When given the numeric argument 1 it adds 1 and 1 and returns the value 2. But when given the string argument “1” it concatenates “1” and “1” and returns the string “11”. When an interpreter executes the function `twice` it first determines the type of the input parameter `arg` and then it chooses the correct operation to perform based on this type.

Listing 4.1: JavaScript dynamic dispatch example

```
function twice(arg) {
  return arg + arg;
}
res = twice(1);     // res is number 2
res = twice("1");   // res is string '11'
```

In general interpreted languages suffer a large performance overhead compared to statically typed compiled languages. Part of the overhead is associated with the dispatch loop, which must continually check the type of each variable and then execute the correct functionality based on the variable type. The other part of the overhead is caused by the fact that the type checks prevent many common compiler optimizations.

Despite the fact that these languages allow variables to change their type at runtime, often the runtime behavior is very predictable. Smalltalk [31] and SELF [44] pioneered techniques using inline caches and dynamic compilation to improve performance, and these techniques are used in the most popular JavaScript systems today [30, 52, 63, 64].

To understand how this works, consider the example in Listing 4.2 and imagine that the array `input` contains many integers. During the first iteration of the `for` loop the runtime system looks up the appropriate operation (addition) for the `twice` function based on the fact that its input parameter is a numeric type. At the same time the runtime system saves this information in an in-line cache to simplify the method lookup process in case the `twice` function is called with an integer argument again.

During future iterations of the `for` loop the runtime system will be able to use information cached in the in-line cache when determining how to execute the `twice` function. However, even with this optimization there is a significant performance overhead compared to the code that a statically compiled language like C++ could generate given the knowledge that the parameter for the function is always an integer.

To further improve performance the runtime system can generate an optimized version of the `twice` function that assumes that the `arg` parameter is always an integer. This optimized code does not need to check the inline cache to determine what method to execute; instead, the compiler can perform integer addition using a single machine code instruction.

Listing 4.2: Inline cache example

```
function twice(arg) {
  return arg + arg;
}
var input  = [1,2,3,4, . . .];
var output = [];
for(var i of input) {
  output.push(twice(i));
}
```

However, this only works if the `twice` function is called with an integer argument.

And the JavaScript semantics do not guarantee this—instead they allow any type of argument. The runtime system solves this problem by inserting a deoptimization check before using the `arg` parameter. The deoptimization check is called at runtime and ensures that the `arg` parameter has the correct type for the optimized code. If `arg` parameter is not an integer then the check fails and triggers a deoptimization event. The deoptimization restores the state of the program to one where the unoptimized compiler can resume code execution and look up the correct method based on `arg`'s type.

In this chapter I focus on understanding the frequency and performance overhead of the deoptimization checks that are inserted in the optimized code by the compiler. Speculative compilation with deoptimization checks is used by many runtime systems for dynamic languages. I chose to focus on the V8 runtime system that runs JavaScript code in order to study a specific language and runtime system combination. In the next section I provide some background about V8 and why I chose it as the runtime system to use for my study.

4.2.2 V8 JavaScript Engine

The V8 JavaScript engine was created as part of Google's project to develop the Chrome web browser and was released in 2008. However, V8 itself was developed as a standalone open source project. It remains under active development today and is still used in the Chrome browser. It is also used by node.js, which is widely used to run server side JavaScript code. The initial version of V8 only had a single compiler, but in 2010 a new version with an optimizing compiler was released. The optimizing compiler made use of speculative type information stored in in-line caches and required deoptimization checks. The initial compiler is called “full-codegen” and the optimizing compiler is called “crankshaft.” In 2014 Google added a third compiler to the V8 engine named TurboFan. This compiler attempts to improve performance by applying the “sea of nodes” [25] techniques. However, currently all the Octane benchmarks except for Zlib spend the majority of their time executing code generated with the Crankshaft optimizing compiler.

My study was motivated by a desire to understand the performance overhead the frequently executed but rarely taken deoptimization checks impose on dynamic languages and to consider possible hardware extensions to reduce this overhead. As a result I wanted to study a runtime system that delivers high performance, since a low performance system may imply the need for hardware extensions, when in fact better software engineering could remove overhead without changing the CPU architecture. JavaScript is one of the most widely used dynamic languages, and developers have invested more effort in developing high performance runtime systems for JavaScript than any other dynamic language in use today. V8 and Firefox are available as open source projects that run on Linux, and compete for the best performance results on widely used JavaScript benchmarks. Either would have been fine to use for my analysis. I chose V8 in part because of its availability as a standalone project separate from the Chrome browser it is used with.

4.2.3 Octane Benchmarks

Another choice I had to consider was what benchmarks to use for my study. There are many benchmarks suites available and in my initial characterization work I gathered results from the JetStream, Massive, Kraken, and SunSpider suites, in addition to the Octane suite.

SunSpider is very old and has effectively been replaced by JetStream. My tests with SunSpider showed relatively few deoptimization checks, but that is largely because the benchmarks are so short that they spend little time executing optimized code. My results from Kraken, Massive, and Jetstream were in roughly the same range as results I saw from Octane.

The Octane benchmark suite was developed by Google largely to help facilitate the development of the V8 JavaScript engine. As a result it is somewhat of an optimization target. However, my goal was to evaluate overhead of deoptimization checks in cases where the software had been tuned to get the best possible performance. So I chose it as the benchmark suite to use when collecting detailed performance results across a range of different systems.

4.3 Checks Categorization

In V8, deoptimization occurs whenever the runtime system transitions from executing optimized code to unoptimized code. There are two primary types of deoptimization events: conditional deopts and unconditional deopts.

When an unconditional deoptimization point is inserted in the optimized code, the program will always be deoptimized if it reaches this point. For example, if V8 were to generate optimized code for a hot loop, it might insert a deoptimization point at the loop exit. This way the optimizing compiler could compile just the body of the hot loop, and when the loop reached its exit condition the runtime would switch to unoptimized code where it could continue code profiling the code to identify new hot regions. This also means that the unconditional deoptimization check does not incur any additional overhead when executing *optimized code*.

The check for the loop exit condition is needed regardless of whether the code that is executed outside of the loop is more optimized code, or if it is a deoptimization point. Deoptimization is a necessarily expensive operation because it causes the runtime to execute unoptimized code, which is much slower than optimized code.

In contrast, a conditional deoptimization point is inserted in optimized code when the compiler needs to ensure that some condition is true before continuing to execute the optimized code. For example, the compiler may have generated optimized code for a function assuming that the values the function is operating on are integers. However, the language semantics do not guarantee this, so the compiler must insert deoptimization checks, and if the function receives a value that is not an integer then it is deoptimized and V8 switches to using the baseline compiler. The conditional checks need to be very low overhead because they can occur frequently in the optimized code.

The V8 source code uses an enumeration type to define possible reasons for deoptimization. There are 64 possible reasons defined; however, these include reasons for both conditional and unconditional deoptimizations. I identified 24 of these deoptimization reasons that are used in conditional deoptimization when code is generated for the x64 target. I instrumented the code to count how frequently each of these deoptimization checks occur when running the Octane benchmarks. Of the 24 deoptimizations reasons that I instrumented, 7 of them occurred relatively frequently in at least some benchmarks while the remaining ones were executed a negligible number of times (less than once per thousand instructions). I grouped the frequently occurring checks into the following categories: type checks, small integer checks, bounds checks, overflow checks. Figure 4.2 shows the frequency of each of these types of

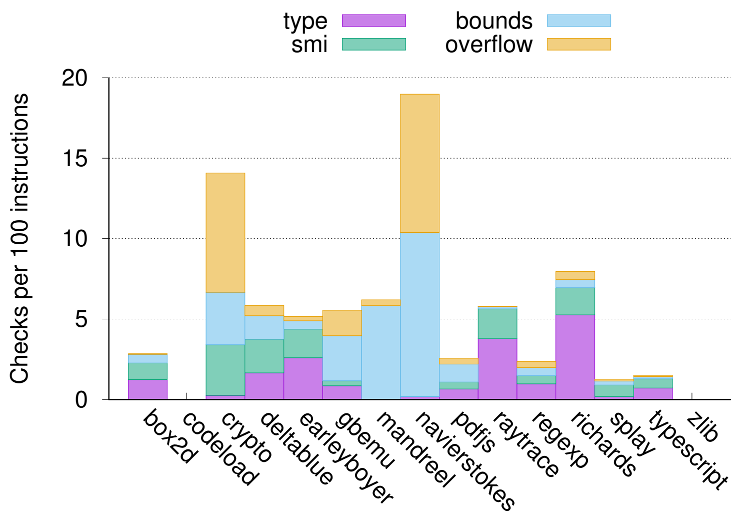

Figure 4.2: Frequency of four different categories of deoptimization checks for each benchmark. Zlib and Codeload have less than 1 check per thousand instructions in any category and so their results appear as 0 on the plot.

checks, and I provide more details about each type in the following sections.

4.3.1 Type Checks

JavaScript uses prototypes rather than classes for creating objects, and the language itself does not define types. However, V8 creates "hidden classes" to group together objects created using the same prototype chain. This allows V8 to effectively treat objects as having a type, which allows for optimizations in accessing an object's fields.

Type checks are necessary to ensure that a variable has the type that the optimizing compiler expects. I categorized the following deoptimization reasons together as type checks: `kAccessCheck`, `kProxy`, `kHole`, `kWrongInstanceType`, `kWrongMap`.

The most frequently executed of these checks is kWrongMap and it is perhaps the easiest to explain. In V8, each object has a pointer to its "hidden class," which effectively defines its type. The "maps check" checks what hidden class the object is pointing to and makes sure that it matches the hidden class type that the compiler expects. If it matches, then the check succeeds; if not, then it fails and the code is deoptimized due to "wrong map."

The remaining deoptimization reasons that I have categorized as type checks are associated with different aspects of JavaScript semantics and V8 optimizations, but they all follow a similar pattern where a pointer value is compared to an expected value.

4.3.2 Small Integer (Smi) Checks

V8 uses tagged pointers as an optimization for representing "small integers" and storing them on the heap. This technique takes advantage of the fact that pointers to heap objects are aligned to 4-byte boundaries when they are allocated. As a result the lowest bit in the pointer to an object is always 0. However, when storing pointers to objects V8 sets this bit to 1 (since the correct value is always 0 no information is lost). Signed integers that can be represented in 31 bits or fewer are stored directly in the pointer field, and the lowest bit is set to 0.

The deoptimization reasons associated with Smi checks are `kSmi` and `kNotASmi`. In both cases the checks compare the lowest bit of the object's pointer to determine if it is a 0 or not.

4.3.3 Bounds Checks

Bounds checks ensure that an access to an array is within the array bounds. They work by comparing the array length to the index of the array access. If the index is larger than bounds limit then the check triggers a deoptimization with the reasons `kOutOfBounds`.

4.3.4 Overflow Checks

The final category of checks is overflow checks and these are used in a variety of arithmetic operations. These checks differ from the previous ones because no additional comparison operation is generated for the x64 architecture. Instead the checks use the result of the overflow condition code that is set implicitly. In cases where deoptimization is triggered, the reason is `kOverflow`.

4.4 Performance Evaluation

In the previous section I characterized the types and frequency of conditional deoptimization checks. In this section I analyze the performance overhead associated with these conditional deoptimization checks by comparing the baseline performance metrics with metrics when the conditional deoptimization checks are skipped (i.e. not executed). In Section 4.4.1 I explain how I was able to remove the checks and still have correct benchmark execution. Section 4.4.2 describes my experimental setup for measuring performance metrics. Section 4.4.3 compares the relative instruction count and execution time between the baseline system and the system that skips deoptimization checks. Section 4.4.4 provides additional analysis looking at the IPC of these two configurations, and Section 4.4.5 examines the branch prediction accuracy.

4.4.1 Experiment Methodology

Although deoptimization checks occur relatively frequently, they are almost never taken in the Octane benchmarks. These checks evaluate a variety of conditions, but all of them include a conditional branch. In order to estimate the overhead of the conditional deoptimization checks I modified the V8 source code to remove the conditional branch when I could guarantee that it would never be taken. This allowed me to compare the performance of the baseline V8, which has deoptimization checks, with the performance when the checks are skipped. I selected which checks to remove based on the deoptimization reason. First I profiled the benchmarks and counted the number of times each type of deoptimization was triggered in each benchmark. Then I modified the code generation phase to skip insertion of the conditional branch for the deoptimization reasons that were never triggered by a specific benchmark.

By executing the same benchmark multiple times I determined exactly when the deoptimization events did and did not occur. When the profiling pass showed that a specific type of check never triggered a deoptimization event, then it was safe to remove that type of check. This technique could not be used to reduce the number of deoptimization checks for real applications because the checks that were removed may have been needed in the case of different inputs, but it was safe for benchmarks running with a known input set.

Unfortunately, if a specific deoptimization reason triggered even a single deoptimization event, the check was needed in order to ensure correct benchmark execution.[2] However, despite this restriction I was able to test with removing a majority of the checks for most of the

[2] I experimented with removing required checks, but this crashed the program. Results shown in this dissertation include all required checks.

benchmarks, and for some benchmarks only a negligible number of checks remained.

Figure 4.3 shows the number of checks that were removed and that remained for each of the benchmarks I evaluated. Codeload and zlib had a negligible number of checks to begin with, so only a negligible number could be removed. Typescript had a significant number of checks, but I was only able to remove a negligible number of them. As a result I saw no difference in performance between the baseline and when checks were removed, and so I do not include results from these benchmarks in the rest of my performance analysis. Pdfjs also had a significant number of checks, but I was only able to remove about 40% of the checks. I chose not to include results from this benchmark because I do not think I was able to properly characterize the performance overhead when less than half of the checks were removed.

In the rest of this section I refer to results from the unmodified V8 code as *baseline* and results when I skipped insertion of the conditional branches for deoptimization checks as *skip*.

4.4.2 Experiment Setup

To evaluate the performance overhead of deoptimization checks I ran my baseline configuration and my skip configuration on a variety of different x64 systems. After my initial characterization I selected four systems with very different microarchitectures to use for detailed analysis. The first two systems use Intel CPUs released in 2013 and fabricated in 22nm technology. One of them is a high performance system that uses the Haswell microarchitecture, and I refer to this as Intel Big Core (BC). The other has a low power CPU that uses the Silvermont microarchitecture, and I refer to it as Intel Little Core (LC). The other two systems used

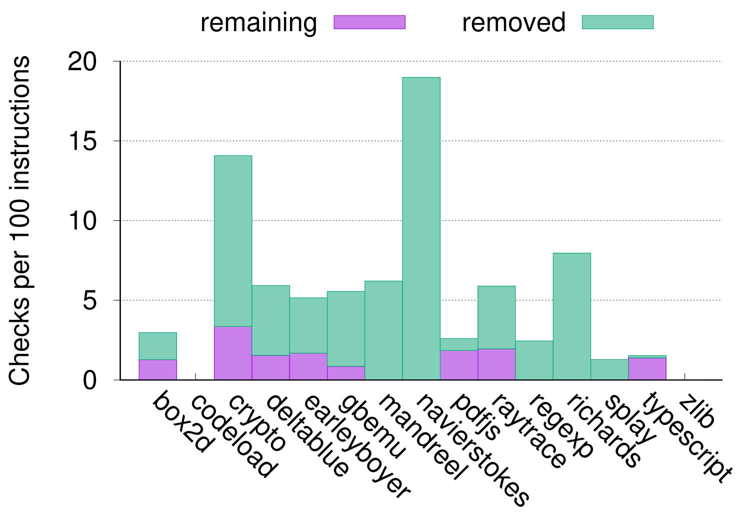

Figure 4.3: Frequency of deoptimization checks that I was able to remove in my skip configuration, and frequency of checks that remained for each benchmark. Codeload and Zlib had a negligible number of checks to begin with and I removed less than half of the checks for Typescript and Pdfjs. I excluded results from these benchmarks from the rest of my performance analysis. For the remaining benchmarks I was able to remove a significant fraction of the total number of checks.

AMD processors released in late 2010 and early 2011, and they are also split between big and little cores. The detailed specifications for all systems are shown in Table 4.1.

I ran all experiments using version 5.1.281.27 of the V8 JavaScript engine on systems running Arch Linux. The baseline results were collected using an unmodified version of V8, while the skip checks results were only modified to skip insertion of a conditional branch if I had determined that the deoptimization reason associated with the check was never triggered for that benchmark. I set the “doDeterministic” flag in the Octane benchmarks to true to ensure that each benchmark ran for a fixed number of iterations rather than a fixed amount of time.

When running the benchmarks I made sure that the test systems were not running any other compute intensive workloads. However, there was still some variation in execution time

Table 4.1: CPU Specifications

	Intel Big Core (BC)	Intel Little Core (LC)
Model	Xeon CPU E3-1275 v3	Celeron CPU N2820
Micro arch.	Haswell	Silvermont
Freq.	3.5 GHz	2.13 GHz
L1 i-cache	32 KB	32 KB
L1 d-cache	32 KB	24 KB
L2 cache	256 KB	1024 KB
L3 cache	8192 KB	–

	AMD Big Core (BC)	AMD Little Core (LC)
Model	Opteron Processor 6172	E-350
Micro arch.	Magny-Cours	Bobcat
Freq.	2.1 GHz	1.6 GHz
L1 i-cache	64 KB	32 KB
L1 d-cache	64 KB	32 KB
L2 cache	512 KB	512 KB
L3 cache	5118 KB	–

as a result of the complex interaction between hardware, OS, and the V8 runtime system that uses JIT compilation and garbage collection. To account for this variation I ran each benchmark configuration 500 times. I report results based on the mean of the 500 benchmark executions for each measured statistic, and I also calculated the confidence interval at a 95% confidence level. I report the confidence interval in my graphs, although in most cases its bound is small enough that it is not easily visible on the graph. I also show the geometric mean as a summary of all the benchmark results for each metric that I report, but I do not include a confidence interval when reporting this summarized result. I recorded the execution time of each experiment and also used the Linux perf tool to collect performance counter results for instructions, cycles, branches, and branch misses.

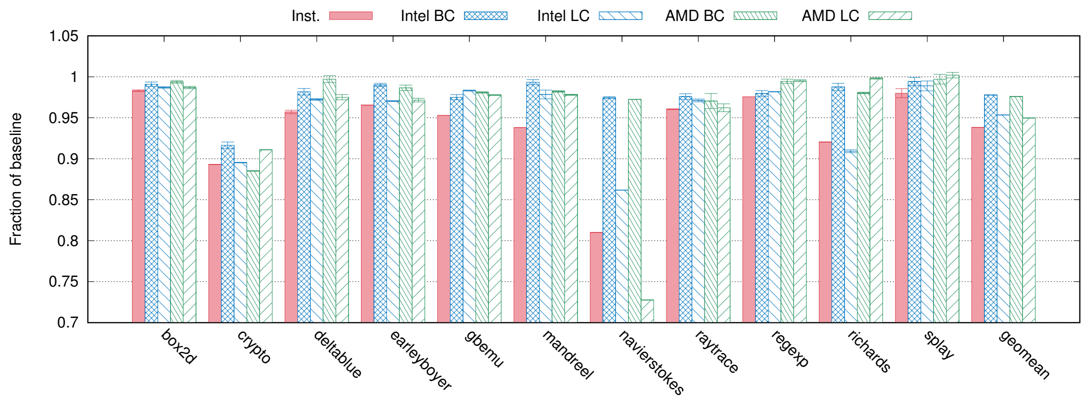

Figure 4.4: Relative instruction count and execution time of the configuration that skips insertion of conditional branches for deoptimization checks as a fraction to the unmodified V8 baseline results. The variation in instruction count was negligible between the four different test systems and so is shown only once and is the same for all of them. Execution time did vary significantly and the results show the relative execution time of the skip configuration on each system compared to the baseline execution time on the same system. In all cases lower is better since it indicates fewer instructions or faster execution time.

4.4.3 Relative Performance

Earlier I showed how many checks I skipped for each benchmark (see Figure 4.3). Here I show how the reduction in instructions executed due to skipping the checks impacts the execution time of the benchmark. Figure 4.4 compares the relative instruction count and execution time of the baseline and skip results. The relative instruction count was very similar across all four systems so I only show it a single time.[3] But the relative execution time had significant variation between the different systems. For example, removing 8 checks per 100 instructions in Richards resulted in an 8% reduction in instruction count for both Intel BC and Intel LC. But on the Intel BC this only resulted in a 1.2% decrease in execution time, while on

[3]For most benchmarks the instruction count was very stable between runs; only splay showed significant variation. Splay is designed to stress the garbage collection so this behavior is not surprising.

the Intel LC it resulted in a 9.1% decrease.

The general trend is that the systems with big cores tend to benefit less from removing checks than ones with little cores. On average across all of the benchmarks the instruction count was reduced by 7.2%. The Intel BC execution time was reduced by 2.2%, while the LC time dropped by 4.6%. Looking at the AMD CPUs, BC execution time dropped by 2.6% and the LC time dropped by 5.0%. However, for the AMD cores these results are skewed somewhat by Navierstokes, which has the largest reduction in execution time (nearly 30%) of any configuration tested.

Although the general trend is for the big cores to benefit less from removing checks, the extent varies significantly depending on the benchmark. For instance, Navierstokes is the benchmark where I removed the most checks, and the instruction count dropped by 18%. This resulted in only a 3.5% reduction in execution time on the Intel BC but a 14.8% reduction in execution time on the Intel LC. Crypto is another benchmark where I was able to remove a large fraction of checks, and the instruction count dropped by 11.7%. This corresponded to a 9.4% reduction in execution time for Intel BC and a 10.8% reduction for the Intel LC. To better understand this performance difference I looked at IPC and branch prediction accuracy.

4.4.4 IPC

Figure 4.5 shows the absolute IPC of the Intel BC and LC for each of the benchmarks for both the baseline and skip checks settings, and Figure 4.6 shows similar data for the AMD systems. In all cases I counted the total instructions and cycles that were actually executed by the benchmark for each of the different configurations and used this data to calculate IPC. This

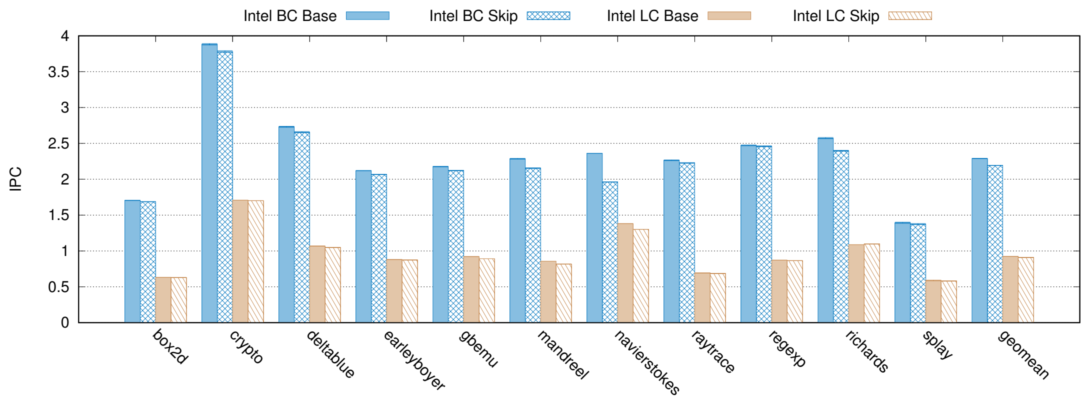

Figure 4.5: Absolute IPC of the Intel Big Core (BC) and Little Core (LC) systems for both the baseline and skip checks configuration. The IPC of the BC is much higher than the LC, and the BC's IPC tends to drop in the skip configuration compared to the baseline, while the LC's IPC stays constant or decreases only slightly for most benchmarks.

allowed me to compare the true instruction throughput for each configuration. But it also meant that higher IPC for the base configuration compared to the skip configuration didn't result in speedup, because the base configuration executed more instructions.

The maximum issue width of the Intel BC is 4 and for the Intel LC it is 2, so the large absolute difference between the Intel BC and LC makes sense. However, the Intel BC has a noticeable drop in IPC for many of the benchmarks while the Intel LC has a negligible drop in IPC in most cases. This explains why the Intel BC shows less performance improvement on average than the Intel LC when skip checks configuration removes conditional branches. The branches that are removed are typically easy to predict, and they do not cause many stalls on the wide issue Intel BC. However, on the narrow issue Intel LC even these easily predicted branches reduce instruction throughput. As a result the LC benefits much more from removing the checks than the BC does.

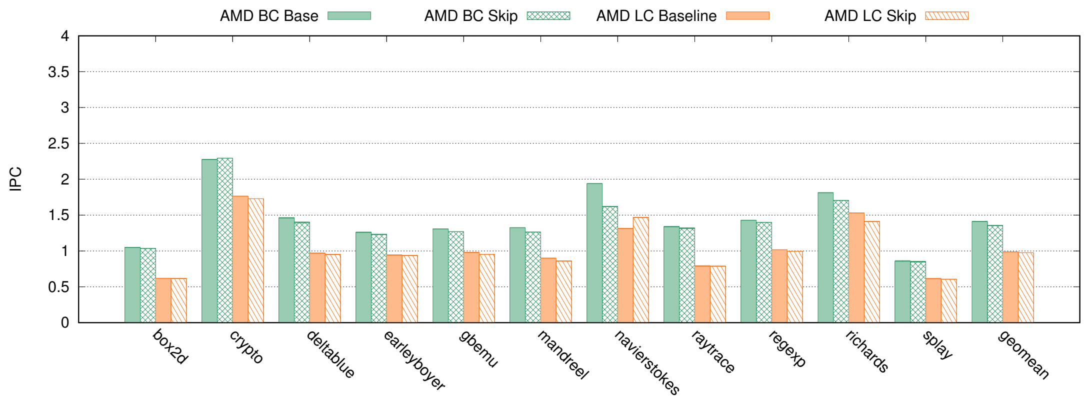

Figure 4.6: Absolute IPC of the AMD Big Core (BC) and Little Core (LC) system for both the baseline and skip checks configuration. There is less variation than with the Intel systems. Still, the overall average shows a larger IPC drop for the BC system than the LC, although part of this is due to the results from Navierstokes where the IPC of the skip configuration actually increases for the LC.

However, for a benchmark like Crypto the Intel BC is executing at near its peak IPC of 4. So in this case even the easily predicted branches associated with the deoptimization checks contribute to limiting execution throughput. Removing these branches provides comparatively larger benefits for this benchmark than one like Navierstokes where peak throughput is not constrained by issue-width.

For the AMD systems the trends are similar, but Navierstokes is an interesting outlier where the IPC improves for the AMD LC when going from the baseline to skip checks configuration. This is somewhat surprising since the branches removed by the skip checks are easily predicted. In fact when looking at the branch prediction data the number of misses is not significantly reduced. However, the number of branches as a percentage of total instructions drops from 27% of instructions to 8% of instructions. It's unclear whether there is a limit on how many branches the AMD LC system can retire per cycle, or if it is hitting some other bot-

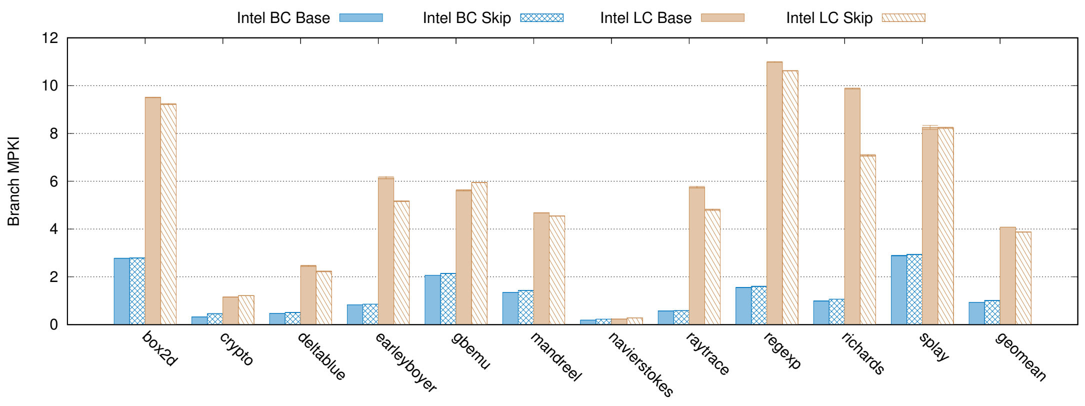

Figure 4.7: Branch misses per thousand instructions (MPKI) for the Intel Big Core (BC) and Little Core (LC) systems. For most benchmarks the BC has much better prediction accuracy than the LC. As a result, removing branches in the skip configuration does not reduce the absolute number of misses for the BC, and the MPKI increases slightly as misses stay constant and the number of instructions decreases. For the LC, removing the conditional branches helps the overall prediction accuracy despite the fact that the branches are typically very easy to predict.

tleneck that limits its throughput. The 27% of instructions as branches for the baseline is similar to many of the other benchmarks, but none of the other benchmarks with a high percentage of branches have as dramatic a drop in branch count to bring the total below 10%.

4.4.5 Branch Prediction

The final set of data that I show is branch misses per thousand instructions (MPKI). Figure 4.7 shows results for the Intel systems and Figure 4.8 shows results for the AMD systems.

In the case of the Intel BC the MPKI always increases when going from the baseline configuration to the skip configuration. In most cases this is because the total number of instructions is reduced, while the number of branch misses stays constant. However, Crypto was interesting because the absolute number of branch mispredictions increased when going from

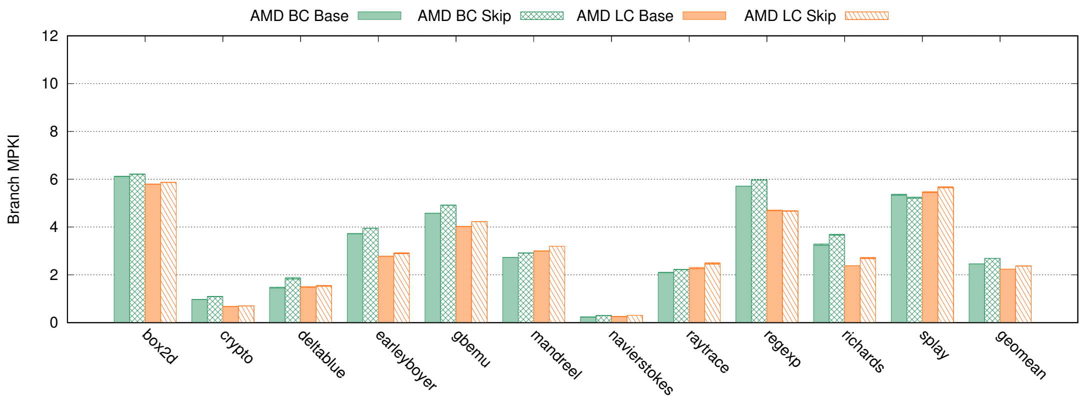

Figure 4.8: Branch misses per thousand instructions (MPKI) for the AMD Big Core (BC) and Little Core (LC) systems. The prediction accuracy for both cores is better than the Intel LC, but worse than the Intel BC. Both of the AMD cores have MPKI increase slightly when removing the easily predicted branches used by the deoptimization checks.

the baseline to the skip configuration. Removing the easily predicted branches in the baseline configuration apparently contributed to increasing the total number of mispredictions, not just the misprediction rate. Regardless, Crypto was still the benchmark that benefited the most from removing branches on the Intel BC.

The Intel LC shows opposite behavior to that of the BC. For the LC the MPKI is reduced from most benchmarks when switching from the baseline to skip configuration. Typically the branch prediction accuracy is still better for the baseline configuration because most of the branches that are removed are easily predicted, but even these easily predicted branches have some mispredictions, and so removing them reduces the overall number of mispredictions per instruction.

The results for the AMD systems were mixed, as there was not as great a difference between the BC and LC as there was for the Intel systems. The overall MPKI for both AMD systems is worse than the Intel BC, but better than the Intel LC. The difference between the AMD BC and LC is less pronounced than it is for the Intel systems.

4.5 Related Work

JavaScript performance has been widely studied and a variety of hardware and software optimizations have been proposed to improve the performance of optimized code or reduce the overhead associated with deoptimization checks.

The most closely related work is from Dot et al. [32]. Like me, they are interested in analyzing performance overheads associated with dynamic languages. They also proposed ISA extensions to reduce the overhead of deoptimization checks and evaluated the proposed extensions using a combination of data collected with Pin and simulation results from the Sniper simulator.

Like them I was interested in the potential for hardware optimization to reduce the overheads imposed by dynamic languages. But I chose a different technique to evaluate the potential benefit of possible hardware optimizations. My technique involved measuring performance using real systems while modifying the code generated by the V8 JIT compiler. I believe this allowed me to get better insights into the true performance impact of the conditional branches used by deoptimization checks. By using real systems I was able to run the benchmarks to completion multiple times and collect statistically significant data even in the presence of system noise associated with the complex interaction of a runtime system that includes a two-tiered JIT compiler and uses garbage collection for memory management, all running on a full Linux OS. I also evaluated four systems with very different microarchitectures rather than a single design point. My work is more of a limit study on the potential benefits of reducing the number of branches associated with deoptimization checks. Conversely, their work provides an evaluation of a specific technique that could be used to achieve this goal and is only simulated for one specific design point.

Anderson et al. [3] also observed that dynamic languages have a large number of runtime checks and proposed hardware extensions intended for mobile processors to decrease the overhead of checking that loads had the correct type. Their performance analysis focused on simpler in-order cores that were common in mobile processors at the time. In contrast, my work focuses on out-of-order designs that are more common today in both server systems and high end mobile CPUs.

Several researchers have studied JavaScript code in an effort to characterize it and understand the primary performance bottlenecks and their impact on the microarchitecture of contemporary CPUs.

Zhu et al. [82] studied the performance of event driven applications that run with node.js. Although node.js uses the V8 JavaScript engine to execute JavaScript code, their work focused on characteristics specific to the event driven programming model, rather than overheads in the V8 JavaScript engine associated with running optimized JavaScript code.

Ogasawara [60] also studied the performance characteristics of node.js workloads. He showed that most of the execution time is spent in C++ library calls, rather than running JavaScript code that can be optimized by the V8 runtime. This contrasts with my analysis since I focused on understanding overheads associated with the optimized code generated by the V8

JIT compiler.

Musleh and Pai [57] studied the impact of a variety of microarchitectural parameters on the V8 JavaScript engine, but they did not look specifically at the overhead of deoptimization checks, which is what I focused on.

Auler et al. [6] analyzed how possible JavaScript compiler optimizations can have different performance impacts depending on the system executing the code. They also proposed a methodology to automatically crowdsource the optimal configuration for a specific device.

Ahn et al. [2] proposed modifications to how V8 constructs types and demonstrated performance improvements on the JSBench benchmarks. Their work differs from mine because their proposed modifications were intended to increase the amount of time that V8 spent executing optimized code, whereas my work focused on deoptimization checks overhead that are only present when executing optimized code.

Kedlaya et al. [46] observed that when managed language runtimes (such as the JVM or CLR) are used to run dynamic languages, they typically do not include support for deoptimizations. This limits the quality of code that can be generated and inhibits common optimizations. They propose a way to do this and implemented it in MCJS, a JavaScript engine implemented on top of Microsoft's CLR runtime.

There has also been some work characterizing the behavior of JavaScript benchmarks in an effort to understand how they compare to real-world JavaScript code commonly seen on websites:

Richards et al. [68] performed a detailed analysis of how JavaScript is used in practice. They found many examples of widely used techniques that inhibit optimization. Their work is complementary to mine since it suggests the need for changes to increase the fraction of time that JavaScript engines can spend running optimized code. In contrast, I characterized overheads that only exist for optimized code.

Ratanworabhan et al. [67] performed a similar study and compared the characteristics of 11 commonly used websites with those of the SunSpider and V8 benchmarks (note that the V8 benchmark suite is a predecessor to the Octane benchmark suite, and is not the same as the V8 JavaScript engine).

Tiwari and Solihin [75] also studied benchmark characteristics, although in their case they compared the V8 benchmark suite with SunSpider. As I did, they used the V8 JavaScript engine and collected hardware performance counter results while running on real machines. Unlike my work, however, they did not look specifically at deoptimization checks; instead they focused on analyzing the similarity of benchmarks in the V8 and SunSpider suites.

4.6 Summary

Type speculation is a proven and effective technique for improving the performance of compute intensive code written in dynamic languages. But speculation can be incorrect, and the runtime system needs to detect when this happens and revert to running unoptimized code. The process of reverting from optimized code back to unoptimized code is called deoptimization, and the optimizing compiler inserts many deoptimization checks in the code that it generates. These checks consist of a comparison, or for overflow checks a condition flag that may be generated implicitly by a preceding arithmetic operation, and a conditional branch.

I evaluated the frequency of the deoptimization checks in code generated by the V8 JavaScript engine running the Octane benchmarks. I then evaluated the performance overhead associated with the conditional branch when running the complete benchmarks on four different real systems with varying microarchitectural parameters. This was done by skipping the branch insertion during the code generation phase in cases where I knew the benchmark would never trigger the branch.

The performance overhead of the deoptimization check's conditional branches varied significantly depending on the system parameters. On average I eliminated 6.2% of instruction by skipping the branches. This resulted in only a 2.2% performance improvement on my system with a high performance Intel CPU, but on an Intel CPU targeted towards low power operation performance improved by 4.6%.

The reason for this difference is that the high performance system had a wider issue width and a more accurate branch predictor that was capable of predicting more branches per cycle. The branches associated with deoptimization checks are highly biased towards not taken, and as a result they are easily predicted.

Previous work has suggested ways to reduce the overhead of deoptimization checks by eliminating the conditional branch and using special instructions that perform checks implicitly and trigger an exception in the rare case where the check fails. However, my analysis shows that even when checks are frequent, their performance overhead may be limited on wide issue CPUs. The easily predicted branches may not be a bottleneck for system throughput.

Wider issue CPU cores inherently consume more power, and current trends are for CPUs to have both high performance "big" cores along with simpler power optimized "little" cores. My analysis indicates that little cores may benefit more than big cores from optimization to reduce the number of branches for deoptimization checks. My analysis only considered the impact of removing conditional branches from the code generation phase. It may be possible to achieve additional benefits by making changes earlier in the code generation pipeline, but this would depend on the specific optimizations proposed and their interaction with the language runtime system.

My study characterizes the overhead of conditional deoptimization checks in a state-of-the-art dynamic language runtime system running on modern out-of-order processors with full OS and language runtime interaction. This complements previous work that has studied other aspects of these workloads using various architectural simulators.

Chapter 5

Conclusion and Future Work

Performance analysis has long been a critical part of the design of microprocessors. As the number of transistors fabricated on a die has increased from thousands to billions, microprocessor design has become much more complex—and so has performance analysis.

Today, architects typically develop a detailed software simulation model of a proposed CPU before implementing the actual hardware. The software model is intended to allow for faster and easier design space exploration that allows an architect to evaluate the impact on performance of potential design choices. However, simulation models have several potential problems. The models may execute too slowly to support rapid iterative design space exploration, or they may be designed for workloads that are not representative of how a real system would be used in practice. This dissertation analyzed three interrelated aspects of these problems by: first, proposing ways to speed up simulation; second, using real systems to study the behavior of benchmarks that were tuned to work with simulation environments; and finally by using a variety of real systems to analyze the performance overhead of an optimizing JIT compiler used in conjunction with a dynamic language. Although each of these problems is different, they share a common theme of seeking to improve the fidelity and understanding of the complex interaction between modern microprocessors and the software stack used by modern applications.

Chapter 2 demonstrated how statistical sampling techniques can be used to reduce simulation time to the point where the simulator can be used interactively. The simulation techniques presented draw heavily on prior research used to apply statistical sampling to microarchitectural simulation. The end result is a novel system that achieves fast and accurate simulation results for single threaded applications. The LiveSim prototype system focuses on single threaded applications as a baseline demonstration of how to enable interactive design space exploration of microarchitectural simulation.

I previously assisted with developing techniques to speed up thermal simulation using statistical sampling [4]. It may be possible to apply these techniques to work with LiveSim or to apply techniques used for statistical sampling of multiprocessor simulation [5]. Integrating thermal and multiprocessor simulation with LiveSim is a possible area of future work.

Chapter 3 analyzed the scalability of the widely used PARSEC benchmark suite of multithreaded applications. Multiprocessor systems have become prevalent as thermal and power constraints have limited architects' ability to increase the performance of a single processor core. However, multiprocessor systems require new workloads to take advantage of the resources provided by multiple CPU cores. An important characteristic of multithreaded applications is the ability to scale performance to reduce program execution time when additional parallel processing resources are used. There are many factors that make this a challenging problem. The analysis of PARSEC in Chapter 3 shows that the scalability of many PARSEC benchmarks is quite limited, and that it is often related to the input set size. Consequently the simulation inputs included with PARSEC may provide misleading results. This is a particular problem when evaluating research papers that do not provide details about the PARSEC parameters used in the paper's experiments. Chapter 3 suggests the need for careful choices and analysis of benchmark applications used in simulation environments.

Chapter 4 analyzed the overhead of deoptimization checks in optimized code produced by the V8 JIT compiler. The deoptimization checks are needed because of speculative optimizations performed when the JIT compiler speculates about the expected type and behavior of objects in JavaScript. As a dynamic language, JavaScript does not assign variable types at compile time; instead it infers them at runtime by looking at how a variable is used. However, since the language spec does not guarantee these values, they can change at runtime, and V8 has to provide a way to ensure correct operation when this happens. The behavior of V8 is relatively complex and may be difficult to fully analyze using simulation. Instead this chapter shows how modifications to the JIT itself can be used to estimate the performance overhead incurred by the checks. This performance estimate also provides an upper bound on possible benefits of hardware extensions that could eliminate the checks.

This dissertation tackles the problem of effective performance evaluation by showing how to accelerate simulations and demonstrating pitfalls of establishing a simulation setup that does not match the behavior of a real system. While I have contributed to advancing the state of the art of microprocessor performance analysis, there are several possibilities for future work. Two that I think are most promising are finding ways to speed up performance validation, and improving performance analysis of accelerators.

High-level microarchitectural simulation is used in part for design space exploration, but it is also used for performance validation of the selected hardware design. Typically this is done by correlating the performance of the high-level simulator and the hardware RTL simulation. Cases where these do not match indicate a bug somewhere, ether in the hardware RTL, the high level simulator, or both. Unfortunately, current tools do not provide an easy way to determine the root cause of performance differences; instead a manual process is required. It may be possible to extend sampling techniques to improve this process by identifying critical samples and using checkpoints to limit the simulation time needed to execute the critical samples after making a change.

The other area of future work is motivated by the fact that specialized hardware is becoming increasingly prevalent. However, programming models to use this hardware are still limited, and understanding what the performance will be early in the design cycle is challenging. As accelerators become increasingly important it may be possible to apply lessons learned through PARSEC analysis and the analysis of deoptimization overhead to improve benchmarking and analysis techniques used when developing custom accelerators.

Bibliography

[1] Octane benchmarks. URL: `https://developers.google.com/octane/`.

[2] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. Improving javascript performance by deconstructing the type system. In *Programming Language Design and Implementation*, PLDI '14, June 2014.

[3] Owen Anderson, Emily Fortuna, Luis Ceze, and Susan Eggers. Checked load: Architectural support for javascript type-checking on mobile processors. In *International Symposium on High Performance Computer Architecture*, HPCA '11, February 2011.

[4] Ehsan K. Ardestani, Elnaz Ebrahimi, Gabriel Southern, and Jose Renau. Thermal-aware sampling in architectural simulation. In *International Symposium on Low Power Electronics and Design*, ISLPED '12, July 2012.

[5] Ehsan K. Ardestani and Jose Renau. ESESC: A fast multicore simulator using time-based sampling. In *International Symposium on High Performance Computer Architecture*, HPCA '13, February 2013.

[6] Rafael Auler, Edson Borin, Peli de Halleux, Michał Moskal, and Nikolai Tillmann. Ad-

dressing javascript jit engines performance quirks: A crowdsourced adaptive compiler. In *Compiler Construction*, CC 2014, April 2014.

[7] Kenneth C. Barr, Heidi Pan, Michael Zhang, and Krste Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS '05, March 2005.

[8] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterization of SPLASH-2 and PARSEC. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, October 2009.

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, ATEC '05, April 2005.

[10] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, October 2009.

[11] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Parallel Architectures and Compilation Techniques*, PACT 2009, October 2009.

[12] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[13] Christian Bienia, Sanjeev Kumar, and Kai Li. PARSEC vs. SPLASH-2: A quantitative

comparison of two multithreaded benchmark suites on chip-multiprocessors. In *International Symposium on Workload Characterization*, September 2008.

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Parallel Architectures and Compilation Techniques*, PACT '08, October 2008.

[15] Christian Bienia and Kai Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Workshop on Modeling, Benchmarking and Simulation*, MoBS 2009, June 2009.

[16] Christian Bienia and Kai Li. Fidelity and scaling of the PARSEC benchmark inputs. In *International Symposium on Workload Characterization*, IISWC '10, December 2010.

[17] Paul Bryan, Jesse Beu, Thomas Conte, Paolo Faraboschi, and Daniel Ortega. Our many-core benchmarks do not use that many cores. In *Workshop on Duplicating, Deconstructing, and Debunking*, WDDD 2009, June 2009.

[18] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, SC '11, November 2011.

[19] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS '13, April 2013.

[20] Trevor E. Carlson, Wim Heirman, Kenzo Van, and Craeynest Lieven Eeckhout. Barrierpoint: Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS '14, March 2014.

[21] Juan Cebrian, Magnus Jahre, and Lasse Natvig. Optimized hardware for suboptimal software: The case for SIMD-aware benchmarks. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS '14, March 2014.

[22] Juan Cebrian, Magnus Jahre, and Lasse Natvig. Parvec: vectorizing the PARSEC benchmark suite. *Computing*, 97(11), 2015.

[23] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators. In *International Symposium on Microarchitecture*, MICRO 40, December 2007.

[24] 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*, WARFP, February 2006.

[25] Cliff Click and Keith D Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):181–196, 1995.

[26] Thomas M. Conte, Mary Ann Hirsch, and Wen-mei W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, 47(6), June 1998.

[27] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *International Conference on Computer Design: VLSI in Computers and Processors*, ICCD, October 1996.

[28] 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 *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, SIGMETRICS '09, June 2009.

[29] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, October 2015.

[30] John-David Dalton, Gaurav Seth, and Louis Lafreniere. Announcing key advances to javascript performance in windows 10 technical preview, October 2014. URL: `http://blogs.msdn.com/b/ie/archive/2014/10/09/announcing-key-advances-to-javascript-performance-in-windows-10-technical-preview.aspx`.

[31] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Principles of Programming Languages*, POPL '84, January 1984.

[32] Gem Dot, Alejandro Martinez, and Antonio Gonzalez. Analysis and optimization of engines for dynamically typed languages. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, SBAC-PAD '15, Oct 2015.

[33] Sourav Dutta, Sheheeda Manakkadu, and Dimitri Kagaris. Classifying performance bot-

tlenecks in multi-threaded applications. In *Embedded Multicore/Manycore SoCs*, MCSoc 2014, September 2014.

[34] 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), 2005.

[35] D. Eklov, N. Nikoleris, and E. Hagersten. A software based profiling method for obtaining speedup stacks on commodity multi-cores. In *ISPASS 2014*, March 2014.

[36] Magnus Ekman and Per Stenstrom. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *International Symposium on Performance Analysis of Systems Software*, ISPASS, March 2005.

[37] Joel S. Emer and Douglas W. Clark. A characterization of processor performance in the vax-11/780. In *International Symposium on Computer Architecture*, ISCA '84, 1984.

[38] S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *ISPASS 2012*, April 2012.

[39] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *ASPLOS 2012*, March 2012.

[40] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *International Symposium on High Performance Computer Architecture (HPCA)*, January 2010.

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

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

[43] W. Heirman, T.E. Carlson, Shuai Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *IISWC 2011*, November 2011.

[44] Urs Hölzle. *Adaptive optimization for SELF: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, 1995.

[45] 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 (INTERACT)*, February 2002.

[46] Madhukar N. Kedlaya, Behnam Robatmili, C&#289;lin Caşcaval, and Ben Hardekopf. Deoptimization for dynamic language jits on typed, stack-based virtual machines. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, 2014.

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

simulation-based computer architecture research. *IEEE Computer Architecture Letters*, January 2002.

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

[49] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *ISCA 2010*, June 2010.

[50] R.P. Loui. In praise of scripting: Real programming pragmatism. *Computer*, 41(7):22–26, July 2008.

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

[52] Kevin Millikin and Florian Schneider. A new crankshaft for v8, Dec 2010. URL: `http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html`.

[53] Ryan Moore and Bruce Childers. Inflation and deflation of self-adaptive applications. In *SEAMS 2011*, May 2011.

[54] Ryan Moore and Bruce Childers. Using utility prediction models to dynamically choose program thread counts. In *ISPASS 2012*, April 2012.

[55] Ryan Moore and Bruce Childers. Program affinity performance models for performance and utilization. In *DATE 2014*, March 2014.

[56] T. Moseley, A. Shye, V.J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *International Symposium on Code Generation and Optimization (CG))*, March 2007.

[57] Malek Musleh and Vijay Pai. Architectural characterization of client-side javascript workloads and analysis of software optimizations. Technical report, Purdue University, 2015.

[58] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Călin Caşcaval. Load balancing using work-stealing for pipeline parallelism in emerging applications. In *ICS 2009*, June 2009.

[59] 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 (ISPASS)*, March 2014.

[60] T. Ogasawara. Workload characterization of server-side javascript. In *IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2014.

[61] J.K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3):23–30, Mar 1998.

[62] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2003.

[63] Nicolas Pierron. Ionmonkey: Optimizing away, Jul 2014. URL: https://blog.mozilla.org/javascript/2014/07/15/ionmonkey-optimizing-away/.

[64] Filip Pizlo. Introducing the webkit ftl jit, May 2014. URL: https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/.

[65] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *IISWC 2011*, November 2011.

[66] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread tranquilizer: Dynamically reducing performance variation. *TACO*, January 2012.

[67] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, 2010.

[68] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, 2010.

[69] Mark Roth, Micah J. Best, Craig Mustard, and Alexandra Fedorova. Deconstructing the overhead in parallel applications. In *IISWC 2012*, 2012.

[70] A. Sandberg, N. Nikoleris, T.E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. Full speed ahead: Detailed architectural simulation at near-native speed. In *IEEE International Symposium on Workload Characterization (IISWC)*, October 2015.

[71] Andreas Sandberg. *Understanding Multicore Performance : Efficient Memory System Modeling and Simulation*. PhD thesis, 2014.

[72] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[73] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *PLDI*, June 2014.

[74] 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)*, June 2013.

[75] D. Tiwari and Y. Solihin. Architectural characterization and similarity analysis of sunspider and google's v8 javascript benchmarks. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2012.

[76] Steven Wallace and Kim Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *International Symposium on Code Generation and Optimization (CGO)*, March 2007.

[77] 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. *IEEE Micro*, 27(2), March 2007.

[78] T.F. Wenisch, R.E. Wunderlich, B. Falsafi, and J.C. Hoe. Simulation sampling with live-points. In *International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2006.

[79] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. Turbosmarts: Accurate microarchitecture simulation sampling in minutes. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2005.

[80] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *International Symposium on Computer Architecture (ISCA)*, June 2003.

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

[82] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural implications of event-driven server-side web applications. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.