ABSTRACT

Title of Dissertation: ANALYZING THE COMBINED EFFECTS OF MEASUREMENT ERROR AND PERTURBATION ERROR ON PERFORMANCE MEASUREMENT

Geoffrey M. Stoker, Doctor of Philosophy, 2014

Directed By: Professor Jeffrey K. Hollingsworth, Department of Computer Science

Dynamic performance analysis of executing programs commonly relies on statistical profiling techniques to provide performance measurement results. When a program execution is sampled we learn something about the examined program, but also change, to some extent, the program's interaction with the underlying system and thus its behavior. The amount we learn diminishes (statistically) with each sample taken, while the change we affect with the intrusive sampling risks growing larger. Effectively sampling programs is challenging largely because of the opposing effects of the decreasing sampling error and increasing perturbation error. Achieving the highest overall level of confidence in measurement results requires striking an appropriate balance between the tensions inherent in these two types of errors. Despite the popularity of statistical profiling, published material typically only explains in general qualitative terms the motivation of the systematic sampling rates used. Given the importance of sampling, we argue in favor of the general principle of

deliberate sample size selection and have developed and tested a technique for doing so. We present our idea of sample rate selection based on abstract and mathematical performance measurement models we developed that incorporate the effect of sampling on both measurement accuracy and perturbation effects. Our mathematical model predicts the sampling size at which the combination of the residual measurement error and the accumulating perturbation error is minimized. Our evaluation of the model with simulation, calibration programs, and selected programs from the SPEC CPU 2006 and SPEC OMP 2001 benchmark suites indicates that this idea has promise. Our results show that the predicted sample size is generally close to the best sampling rate and effectively avoids bad choices. Most importantly, adaptive sample rate selection is shown to perform better than a single selected rate in most cases.

ANALYZING THE COMBINED EFFECTS OF MEASUREMENT ERROR AND PERTURBATION ERROR ON PERFORMANCE MEASUREMENT

By

Geoffrey M. Stoker

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Advisory Committee:
Professor Jeffrey K. Hollingsworth, Chair/Advisor
Professor M. Cole Miller, Dean's Representative
Professor Ashok K. Agrawala
Professor Peter J. Keleher
Professor Alan L. Sussman

Dedication

For Akemi.

Acknowledgements

My PhD adventure has largely mirrored Winston Churchill's description of his book writing experience – from toy, through amusement, mistress, master, tyrant, and finally, to slain monster. I will be forever grateful to those who accompanied me for any length of time on this journey. The good that has come out of this process is due in the largest part to them.

To my advisor, Professor Jeffrey K. Hollingsworth, thank you for the years of patiently-provided, wise advice and thoughtful guidance, without which this adventure would not have been possible.

To the members of my dissertation committee, thank you for your willingness to help develop me as a researcher and improve my scholarship.

To my fellow research group members, AT, Mike, Nick, Ray, and Tugrul, thank you so much for the encouragement to persevere that you provided through personal example and regular interaction.

To my leaders at EECS, Colonel Ed Sobiesk and Colonel Gene Ressler, thank you for pushing me, encouraging me, and taking a chance on me. Teaching Cadets has been the highlight of my career thus far.

To the many, many members of the graduate community at UMD and the faculty at USMA who made the journey worthwhile, thanks for being such wonderful friends and colleagues. Thanks (in no particular order), Paul, Ed, Bill, Paulo, Rob, Todd, Devon, Tim, Mike, Mike, Kyle, Glenn, Mike, Ray, John, Patrick, Aaron, and many, many others I'm sorry to have to leave out.

To Kai-Lynn, Mirana, Geoffrey, Julia, and Sarah, thank you for providing me a large part of what makes life worth living – yourselves.

Finally, to my wife Akemi, you are truly a gift from the Lord. Without you by my side, this adventure would count for nothing. Thank you for enduring the thousands of absent weekend and late night hours along with my emotional ups and downs over the past six years. I love you and eagerly look forward to whatever life next holds for us.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Understanding exactly what a piece of software is doing during execution and how well it is performing have been of interest since programs were first written and run on the earliest computers over half a century ago. As software and the machines on which it runs have become more complex, interest in evaluating performance has only intensified. To aid in understanding software performance two general methods of dynamic analysis, termed profiling [36], have been developed. The first is counting events of interest, like the entry and/or exit points of a particular function, and is commonly called measurement-based profiling or just measured profiling [43]. This is typically accomplished by adding additional lines of code, before or during runtime, that will execute during the normal control flow of a program either just before or just after (or sometimes both) the code of interest being profiled. This research is not primarily concerned with measured profiling.

The second category of dynamic software performance analysis methods is program status sampling [36], also called sample-based profiling or statistical profiling [43]. This technique often uses some type of operating system or underlying hardware functionality to interrupt the program being profiled and then note some aspect of its current execution before allowing the program to resume running. For instance, we may halt a program, record the current program counter (PC) or the name of the function at the top of the call stack, and then resume execution of the program. With the collected samples, experimenters produce a measured value (statistic) from which inferences about the executing program's population parameters

can be made. The research in this dissertation is framed entirely within the context of this kind of dynamic software performance analysis.

Effectively sampling executing programs for the purpose of dynamic performance analysis via statistical profiling is a challenging problem. This is due in large part to the opposing effects of measurement error and perturbation error [17, 41]. Very frequent sampling of a program execution can provide statistically precise measured values, but it also risks perturbing the measured program so that it behaves very unlike the original object of the analysis (the un-sampled program) and results in the production of inaccurate measured values. Sparse sampling, on the other hand, perturbs the executing program to a lesser degree, but also provides less statistically precise measurement results that may be less accurate and therefore less useful than required for meaningful analysis. Achieving the highest overall level of confidence in measurement results requires understanding the effects of and striking a balance between the tensions inherent in these two types of errors.

We can't determine (predict) application performance from first principles, so we use performance analysis to try and understand program behavior [55]. Choosing how and when to capture performance data is a key part of the performance analysis process and while the process is part science and part engineering, there is still a portion that is largely an art [12, 34]. When creating an execution profile via sampling, how is a sampling period chosen? What is an appropriate sampling period? It is well understood that increased sampling decreases measurement error and makes estimates of population parameters more precise and accurate. It is generally accepted that the act of sampling alters program execution, thereby perturbing

performance. So, general qualitative guidance is to sample enough to get precise measurements, but not so much that perturbation causes the measurements to be inaccurate and reflect something unlike the object of interest (the un-sampled program). How do we balance measurement error and perturbation error? Can we model the interaction of measurement and perturbation error? Is there an optimal sampling period? If so, can we find it? Can we predict it?

We hypothesize that along the continuum of possible performance measurement results with varying sample-dependent combinations of measurement error and perturbation error there exists a "sweet spot" where the combined effects of these two error types is minimized and the performance measurement is the smallest distance from the true value being sought, and thus the most accurate. The primary aims of the research reported here are the investigation of the existence of the hypothesized "sweet spot," the development of a general model for the analysis of the interaction between measurement error and perturbation error during statistical profiling, and the pursuit of a method for identifying a sample size (and determining an appropriate sampling period) that minimizes the combined effects of those errors on a particular performance analysis experiment.

1.1 Motivation

Our research is motivated by several ideas. First and most simply, as Linus Pauling and others have said, "Science is the search for truth." Researchers generally seek the most feasibly correct results from a given experiment out of a desire to get a measurement as close to truth as possible. We believe that calculating statistics from performance measurements taken with a sample size that minimizes the combined

effects of measurement and perturbation error will be more accurate than when any other sample size is used. Second, in many cases there are practical limits to the goodness that additional sampling can provide. As a somewhat naive, but straight-forward example, consider a situation where you are sampling to calculate the amount of time function *foo* takes to execute in a particular program. If *foo* is responsible for 25% of the execution time and each sample adds 20 microseconds to the overall run time, the improvement in the confidence interval around the run time value calculated for *foo* will be less than the cost of an additional sample after about 55,000 samples. Another aspect of practical limits comes into play when a very large number of samples is collected. With computers, it's quite easy to collect huge numbers of samples and possible to generate confidence intervals that are so narrow that very slight differences in performance measurements could lead investigators to conclude a statistically significant difference exists between two experiments, when the difference is related to some non-deterministic event. As well, it has been found that simply increasing the sample size does not guarantee more accurate performance measurement results [62].

A third reason we are motivated in our research is that while the most possibly correct measurement results are valuable for an individual tool in the absolute sense, the implications when comparing different performance measurement tools highlights additional value. Consider two theoretical performance measurement tools. The best tool (BT) is known to generate the least overhead when sampling program executions. The second best tool (SBT) is a good tool, but because of implementation differences generates slightly more overhead than BT. For a particular program

execution both BT and SBT reduce the measurement error an equal amount in accordance with established statistical principles. However, because of the overhead (perturbation) differences, any result taken at the same sample point will have BT closer to truth than SBT. If, however, the sample point used by SBT more effectively minimizes the overall effect of measurement and perturbation error, SBT could produce more accurate results despite inflicting more perturbation per sample than BT. Fig. 1.1 is a theoretical example of this idea based in part on previously published results [65].

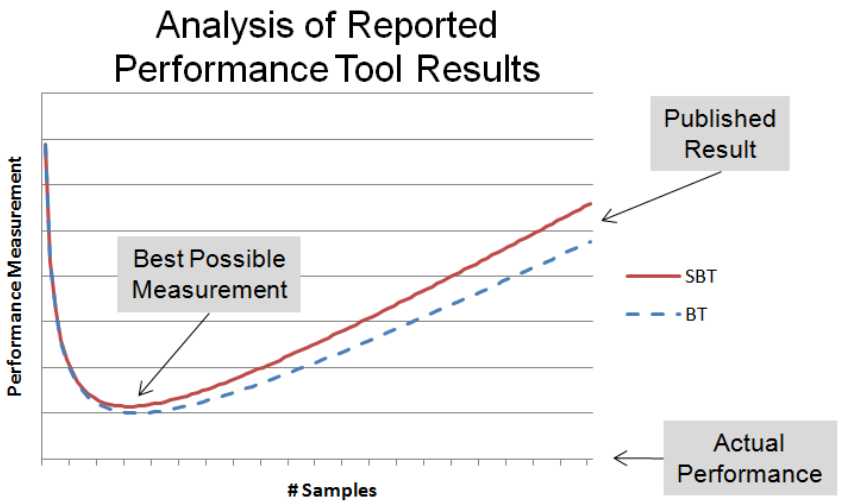

Figure 1.1: Theoretical outcome of two performance measurement tools

A fourth motivation stems from speculation that perturbation can be severe enough in some cases that results will lead to incorrect inference. This result has been published before [62] and presented in the conditions of one of our early experiments. As can be seen in Fig. 1.2, as sampling increased, the measurements of two particular functions, though they became more tightly grouped, were perturbed to the point that they changed positions relative to the order of most computationally expensive.

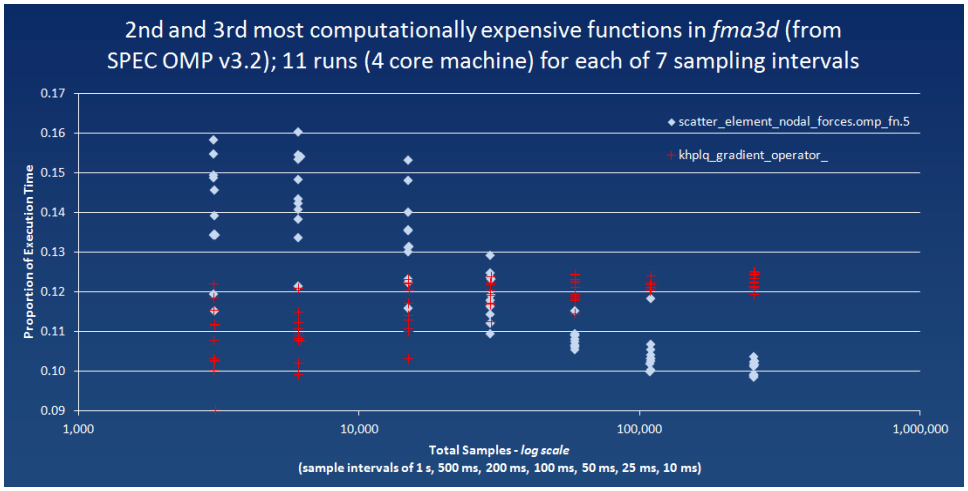

Figure 1.2: Example experiment results where measurements were perturbed by sampling

The results in Fig. 1.2 show the proportion of execution time calculated for two particular functions, "scatter" and "khplq," from 11 different executions at each of 7 different sampling intervals. When the executions are lightly sampled, the calculated values clearly indicate that scatter takes a greater proportion of execution time than khplq; however, when heavily sampled, the results get reversed and it appears that khplq takes the greater proportion of execution time. Despite the tighter grouping of the calculated values (higher precision), the values, especially for scatter, appear to be much less accurate with higher sampling. This is potentially a very insidious result as efforts to optimize a program based on incorrect ordering of functions could result in wasted programmer effort.

1.2 Contributions

With our research, we make the following contributions:

Abstract Model. We introduce an abstract model that depicts the interaction of measurement error and perturbation error and shows how their interplay affects the accuracy of measurement results generated by sample-based profiling.

Mathematical Model. We develop a simplified mathematical model of the affects of decreasing measurement error and increasing perturbation error on measurement results and use it to predict the measurement point – the "sweet spot" – at which overall error should be at a minimum.

Experimental Results. We conduct empirical studies and present results of a series of simulations, experiments with a calibration program, experiments with a subset of sequential programs from the SPEC CPU 2006 benchmark suite [29, 58], and experiments with a subset of shared memory parallel programs from the SPEC OMP 2001 benchmark suite [6, 59].

Quantitative Guidance. Departing from traditional qualitative guidance encouraging the use of "reasonable" or "appropriate" sampling intervals, we provide the first quantifiable guidance for choosing a sample period for experimenters conducting sample-based performance analysis.

1.3 Outline

The remainder of this thesis is divided into seven chapters. Chapter 2 discusses background and related work in the field. We discuss existing techniques for dealing with perturbation during performance analysis, outline some of the key aspects of the statistical foundations that underpin our research, and provide a survey of various profiling tools' process of sample period selection.

Chapter 3 presents the results of an empirical sampling study on both a series of simulations of program execution and a series of calibration programs with known population parameters.

Our abstract model and analytical model are introduced and described in detail in Chapter 4. We test the applicability of these models with simulation and derive a formula to use for finding an experiment sampling "sweet spot."

In Chapter 5 we present the results of intensive sampling experiments with a calibration program and selected benchmarks from the SPEC CPU 2006 benchmark suite. We compare the empirical results of our study against the prediction of our derived formula from Chapter 4.

Results of intensive sampling experiments using selected benchmarks from the SPEC OMP 2001 benchmark suite are described in Chapter 6. As in Chapter 5, we compare the empirical experimental results against the predictive power of our analytic function.

Future work suggested and motivated by our research is outlined in Chapter 7 and finally, Chapter 8 presents conclusions.

Chapter 2: Background and Related Work

Previous research related to our work can be broadly categorized in three ways: techniques for dealing with perturbation during performance analysis, statistical foundations of experiment design, and existing tools' handling of sample period selection.

2.1 Handling Perturbation

Many researchers have wrestled with the problem created when the pursuit of accurate and detailed performance measurements results in application perturbation effects [39, 40, 41]. We use the term perturbation throughout this dissertation, but note that the phenomena has also in the past been variously referred to as interference [12,16, 23], degradation [14, 26], artifact [52], side effects [63], probe effect [17], intrusion [4, 30], and invasion [48]. It is generally agreed that at this point in time perturbation caused by software performance tools cannot be eliminated, so the two general approaches to handling perturbation revolve around the strategies of compensation and minimization.

2.1.1 Perturbation Minimization

Dynamic statistical projection pursuit [68] is a performance analysis technique developed out of a motivation to minimize the perturbation effects of instrumentation in addition to a desire to decrease the amount of performance data generated and reduce the number of performance metrics managed by a performance analysis system. The primary idea is to regularly identify performance metrics of special interest and focus only on collecting data for them.

Dynamic instrumentation, an idea pioneered with the Paradyn parallel performance measurement tool [46], is another performance analysis technique designed to limit the effects of perturbation. Paradyn is intended primarily to be used with very long-running applications (hours or days) on large parallel machines. By delaying inserting instrumentation code until the moment it is needed and then removing it when it is no longer needed, Paradyn significantly reduces measurement overhead and minimizes the global perturbation effects of the instrumentation.

Though their techniques applied primarily to the realm of measured profiling, Kumar, et al, [38] reduced perturbation effects by optimizing the instrumentation used. They focused on reducing the number of instrumentation points, the number of times each point executed its instrumentation code, and by transforming, when possible, the instrumentation code into a more efficient form.

2.1.2 Perturbation Compensation

Using a timed Petri-net model for intrusively monitored software, Andersland, et al [2], and Gannon, et al [18, 19], recover true program traces from corrupted event traces post-mortem. This is one of the techniques developed to handle perturbation effects by compensating for them in order to recover the original execution times. They view profiled applications as discrete event dynamic systems that can adequately be modeled by timed Petri nets.

Malony, et al, developed perturbation models for sequential [42] and parallel [43] executing programs that capture and then remove the additional execution time attributable to perturbation effects from instrumentation. Using these models they can determine an approximate actual code execution time from a run conducted with

performance measurement. The process involves determining the measurement overhead and then removing it from the results of the profiling runs.

Sarukkai and Malony [57] considered methods for removing perturbation effects during performance analysis of highly parallel Single-Program, Multiple Data (SPMD) programs. In general terms, the process involved analyzing a trace file with a time ordered set of events, eliminating or reducing the perturbation effects, and generating a new trace of events with a time ordering that more closely reflects the actual execution.

Najafzadeh and Chaiken [49] developed a flow-graph based perturbation model designed to compensate for or minimize perturbation effects. They then used the model [50] to estimate performance from the instrumented execution output they collected during performance analysis.

Lehr [40] considered the problem of perturbations caused by software monitors in parallel programs and, coming to the conclusion that the effects could not be eliminated, focused instead on detecting, measuring, and compensating for them. With 99% confidence, he claimed that with the best case results the difference between compensated estimates of the average mean and the real average mean could be calculated within $1.2 \pm 0.9\%$. His is the rare case of research in this area that tried to quantify the topic.

Investigators continue to seek better ways to mitigate the effects of perturbation that occur during performance analysis. Beyond minimizing perturbation via thoughtful programming and clever instrumentation, we focus on minimizing perturbation by identifying when it affects measurement to a greater

degree than measurement is reducing statistical error. Compared to compensation, minimization seems like a simpler and safer idea. Because sampling has decreasing utility value, each sample has less value, in terms of reducing measurement error, than the sample before it. Compensating for perturbation seems to require a much greater understanding of the effects of perturbation since each sample reduces measurement error to a lesser degree, thus setting a higher requirement to get compensation right. So, though much effort and significant strides have been made in the area of perturbation compensation, we believe, as has been observed before [22], that "the best solution remains minimizing perturbation."

2.2 Statistical Science

2.2.1 Approximating Hypergeometric with Normal

To determine how best to characterize the distribution of results from program profiling experiments, we start with the observation that any given program execution can be considered a discrete population of elements from which samples are taken without replacement. Whether we consider the program's elements to be delineated by a chip cycle, a machine instruction, a software clock tick, or something else; they can be viewed as a large bin of items from which we extract samples that are not replaced. From this collection of samples, we will find that some occurred within the execution context of a given function *foo*, while the remainder occurred outside the execution context of *foo*. For the purposes of this section, samples taken of *foo* are a "success."

It follows from our understanding of a profiling experiment that it is hypergeometric and that the number of successes we find for each experiment is a

hypergeometric random variable. Thus, we would use the hypergeometric distribution to describe the expected outcome or probability of the results for our experiment. The hypergeometric distribution describes how many successes you could expect after *n* samples are taken, without replacement, from a population of size *N*. Success is defined as picking one of the *M* total items of interest from among the *N* total items in the population. The probability that exactly *m* successes are picked can be calculated as follows:

$$H(m; N, n, M) = \frac{\binom{M}{m}\binom{N-M}{n-m}}{\binom{N}{n}} \quad (2.1)$$

For small populations where *N* and *M* are known, the hypergeometric distribution is appropriate and useful to use. However, because a program execution represents a very large population of events and since *N* and *M* are generally unknown (likely unknowable for a program execution), the hypergeometric distribution is rather complicated and labor intensive to use at best and impossible to be used at worst. For these reasons the hypergeometric distribution is often approximated by distributions that are simpler to use. A binomial approximation is appropriate when the population is very large and the sample taken is relatively small.

$$B(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} \quad (2.2)$$

The difference between a binomial experiment (aka Bernoulli trial) and a hypergeometric experiment is whether sampling is done with or without replacement. Since samples are replaced with a binomial experiment, the probability of success remains the same for each trial. The hypergeometric distribution calculation takes into account the fact that sampling is done without replacement, thus the probability

of a particular sample being a success is dependent on the previous samples. However, with a very large population and a sample size that is small relative to the population size, it becomes apparent that removing the sample size will have very little effect on the population. So, the probability of choosing a particular sample a second time (assuming we sampled with replacement) would be negligibly small, thus making binomial and hypergeometric distributions extremely similar under these conditions [51]. Deciding when the sample size is small enough has to do with being satisfied with the calculated error. As a sample size increases, the error calculated in the standard way becomes less correct and should be augmented with the finite population correction (fpc) factor:

$$\sqrt{\frac{(N-n)}{(N-1)}} \tag{2.3}$$

The recommendations for a small enough sample size relative to the population size seem to land on the two suggestions $N > 10n$ (sample size is 10% or less of the population) and $N > 20n$ (sample size is 5% or less of the population). We did come across the suggestion that $N > 2n$ is even sufficient [16], but the prevalence of the other two standards caused us to not seriously consider it. There is a preference among the many sources we consulted that covered this topic for $N > 10n$, of which Brunk, et al, may have been the earliest [10]. Though we expect the sample size of a program execution to be much smaller than either of these two standards, we will justify our approximating the hypergeometric on the more common of the two standards, $N > 10n$.

A further approximation using the Normal (Gaussian) distribution is appropriate when some additional conditions hold. These conditions have to do with

the observation that any binomial distribution where the probability of success is .5 ($p = .5$) is symmetrical and thus even for small *n*, a binomial distribution with $\hat{p}$ close to .5 is appropriately approximated by the normal. The question then becomes how many more samples are required for binomial distributions to approximate the normal when $\hat{p}$ is closer to the edges, $\hat{p} = 0$ or $\hat{p} = 1$. One of the first efforts to quantify this, where $\hat{q}$ was defined as $(1 - \hat{p})$, resulted in $(n + 1)\hat{p}\hat{q} \geq 9$ [15]. Over time, the rule has been modified somewhat. Some indicate that both $n\hat{p} \geq 5$ and $n\hat{q} \geq 5$ should hold [27, 61]. Others, including Lilja [39], recommend $n\hat{p} \geq 10$. Once again, we expect to encounter no problem meeting any of these guidelines, but we identify $n\hat{p} \geq 10$ as the standard on which we base our appropriate use of the Normal distribution.

2.2.2 Use in Performance Analysis

Many investigators advocate for additional statistical rigor and better experimental design in computer system evaluation across different areas [8, 20, 21, 34, 35 39, 47, 53, 69]. We surveyed much research related to this area, but will limit comments to a representative few. Georges', et al, concern flowed from the area of Java performance analysis and the disparate uses of statistical rigor among the various performance evaluation methodologies. Yi, et al, observed that statistically rigorous simulation methodologies are typically not used during computer architecture design. Jain and Lilja have provided books dedicated to advancing the process of computer performance analysis with specific emphasis on providing statistical tools and concepts that will permit more thorough experiments and analyses. Patil, co-authoring with Lilja, provides the most recent comprehensive discussion of the

application of statistical theories to computer performance measurement. The primary points of all the proponents are to provide the highest level of confidence in experimental results and to ensure that the best possible conclusions are made from the data generated by computer system experiments.

Statistical science provides a standard error calculation for determining the confidence interval of a proportion measurement at a particular confidence level for a Normally distributed (Gaussian) random variable. This standard error formula can be algebraically manipulated to permit calculation of the number of samples required to produce a proportion measurement within a desired confidence interval and confidence level. Jain, equation (2.4), and Lilja, equation (2.5), each provide an exposition on the way this formula is arranged and used when calculating proportions. In each case the result, *n*, is the sample size required to calculate a confidence interval of $\pm r$ (the error) for the statistical measure $\hat{p}$ at the confidence level established by *z* (i.e. $z = 1.96$ for the 95% confidence level).

$$n = z^2 \frac{\hat{p}(1-\hat{p})}{r^2} \tag{2.4}$$

$$n = z^2 \frac{\hat{p}(1-\hat{p})}{(\hat{p}r)^2} \tag{2.5}$$

Note that Lilja includes $\hat{p}$ in the denominator which calculates the confidence interval with respect to the size of $\hat{p}$ rather than with respect to the experiment overall and computes a much larger sample requirement. This means that given a proportion, $\hat{p} = .20$, and desired error range, $r = .005$, we could expect a confidence interval of (.195, .205) and result of $n = 24{,}587$ with equation (2.4) and expect a confidence interval of (.199, .201) and result of $n = 614{,}656$ with equation (2.5) at the 95%

confidence level, $z = 1.96$. Though they differ somewhat in form, they otherwise encompass the same principle of targeting a confidence interval at a given confidence level in order to determine how many samples to take. It is through examination of these techniques that we received the motivation for our investigation into other methods for deliberate sample calculation in other contexts.

2.3 Tools

Many sample-based profiling tools have been created to assist with analyzing the performance of executing programs. They all tend to work in a similar manner. The execution of a program is interrupted, some aspect of the state of the program is sampled and noted, often the program counter (PC) or call-stack, and then execution is allowed to continue.

2.3.1 Progtime

The program status sampling (statistical profiling) tool used by Knuth in the 1971 empirical study of FORTRAN programs was an extension of a routine called PROGTIME originally developed by T.Y. Johnson and R.H. Johnson for use on the IBM System/360 [36]. PROGTIME would, while running, spawn the program of interest as a subtask, then sample that program's status word at regular intervals. At runtime completion PROGTIME would produce a histogram of the execution frequency for all program instructions.

2.3.2 Prof

One of the oldest and most common statistical profilers is the Unix tool *prof* [66]. By compiling a program with a specific flag, typically −p, a monitor function is

added to the executable code of that program. When the executable is run, profile data is generated and then output to a file (*mon.out* by default) when the program terminates normally. The profile data consists of the number of times each function in the program is entered and a statistically-based break-down of the processing time used by the program. The statistical profile of execution time is captured by interrupting the program and sampling the PC at regular intervals governed by the software clock, commonly every 10 milliseconds. Executing *prof* results in the creation of a profiling report that displays the data in tabular form.

2.3.3 Gprof

The profiling tool *gprof* [24, 25] was created as an extension to *prof*. In addition to counting each time a function is entered during execution, it included functionality to capture the return address of the call, the name of the caller function, and how many times that particular caller-callee arc was traversed. With this additional information a call graph of the profiled program emerged which permitted execution times attributed to function calls to be further distributed among whatever subsequent function calls might have been invoked. Though only one-level incoming call graph arcs were recorded and not complete call graphs, it allowed for a good approximation of the distribution of execution time with greater context. The profile report was extended to display the time spent executing a function as well as the time spent executing other functions on its behalf. As well, each function provides statistics about the functions that called it. The other functionality remains similar to *prof* including the PC sampling rate of 100 samples per second.

2.3.4 ATOM

Analysis Tools with OM or ATOM [60] introduced the concept of an instrumentation framework that could be used to build customized program analysis tools. The ATOM framework, in part, enables a program to analyze itself and has been used to create all kinds of performance analysis tools, including profilers.

2.3.5 XProfiler

XProfiler [32] is part of the International Business Machine (IBM) High Performance Computing Toolkit (HPCT) capable of profiling both serial and parallel applications on the IBM Unix variant Advanced Interactive eXecutive (AIX). Given a program compiled with an IBM XL compiler and the proper flags, information similar to that provided by the gprof tool process is dumped to a profile file which XProfiler reads and presents via a graphical interface rather than in a text file. The sampling rate is also 100 samples per second.

2.3.6 HPCToolkit

HPCToolkit [1, 56] is a collection of performance analysis tools designed to support dynamic performance analysis. The specific component that collects profile data via sampling is *hpcrun*. In addition to time intervals, *hpcrun* will sample based on events tracked by hardware counters and accessed via the Performance Application Programming Interface (PAPI) [9], like a specified number of cycles or specified number of L2 data cache misses. The time interval between samples can be set by the user and is recommended to be something that will generate between hundreds and thousands of samples per second. They note that Linux ITIMER

interrupts cannot occur with greater frequency than a jiffy which is likely 1, 4, or 10 milliseconds making thousands of samples per second unattainable. Note that jiffy in this Linux context is the smallest unit of time of the software clock. It is determined by the value of the kernel constant HZ which can be set to 100, 150, or 1000.

2.3.7 VTune

The VTune Performance Analyzer [33] is a commercial performance analysis tool developed by Intel which also includes time-based profiling as well as event-based profiling. It allows the user to specify the timing interval, but recommends 1000 samples per second and will default to that number via automatic calculations if not specifically overridden by the user.

2.3.8 STAT

A tool specifically designed to collect, analyze, and visualize the stack trace profiles of very large parallel and distributed applications is the Stack Trace Analysis Tool (STAT) [5]. The tool conceptually contains three parts, the front-end, tool daemons, and stack trace analysis routines. The front-end establishes the tool's components; the daemons collect, process, and transmit the stack samples; and the analysis routines analyze the data. Sampling rates are established by specifying a sample count and an interval to wait between samples.

2.3.9 DCPI

The Hewlett-Packard (HP) Digital Continuous Profiling Infrastructure (DCPI) [3, 31] appears to no longer be supported as it was specifically designed to interface with the now defunct Digital Equipment Corporation (DEC) Alpha microprocessor.

It's included here because it is one of the best examples of the class of statistical profilers that is continuously running and capable of providing data not only at the application level, but also for the entire software system. As DCPI was integrated into the operating system and made use of hardware timing, high-rate sampling was possible. Samples were collected at a fixed rate of 5200 per second.

2.3.10 SimPoint

Casas, et al, [13], have done work that is in a substantially different area and travels a much different solution path, but in spirit is more closely aligned with ours than the others. They investigate the manner in which portions of instruction streams are selected as appropriate benchmarks for evaluating the performance of existing computer architectures and assisting with the design of future architectures. They observe that in previous efforts to reduce the number of instructions executed during evaluation of an architecture, and thereby trim the time spent, data generated, and memory required during analysis, investigators choose a sample length in an arbitrary manner and then leave it fixed. Casas, et al, run an entire benchmark application, take periodic hardware counter samples, conduct spectral analysis on the samples to determine the existence of periodic phases, and then use this result to select the portion of the instruction stream that best represents the overall instruction stream of the benchmark application. In this way, they are able to determine the optimal sampling length of the whole instruction stream and provide subsets that improve the accuracy of the results of architecture analysis. They take an existing tool called *SimPoint* [28, 67], which uses a fixed and arbitrary length for the sampling interval, and modify its use with their technique to demonstrate that more deliberate and

careful selection of the instruction stream sample provides better overall analysis results.

Chapter 3: Sampling Empirical Study

In order to gain a better understanding of the behavior of programs when being sampled we conducted an empirical study. We started with a series of simulations to both validate what statistical analysis tells us we can expect and also to determine how closely the normal approximation fits the expected hypergeometric distribution that results from sampling a program execution. We then conducted a series of experiments with a calibration program designed to mirror as closely as possible the context of the simulations.

3.1 Sampling Simulation

For the set of simulation experiments we constructed a C program that manipulated a 1,000,000 member integer array used to represent the execution flow of a program that runs for 300 seconds with 1,000,000 function calls. With each experiment, we focused on two functions, *foo* and *bar*, that had execution percentages within 1% of each other. We assigned different integer values to the array to represent *foo*, *bar*, and all other functions in exactly the quantities we wanted for a particular experiment. The array was then shuffled with the Fisher-Yates shuffle algorithm making use of the C standard library function *rand*. And, we took the first *x* members of the shuffled array as our simple random sample of size *x*. For each *foo* and *bar* execution percentage of interest we ran 10,000 trials and created histograms of the results. So, for example, in our first experiment we setup *foo* to have an execution percentage of 49% and assigned the value 1 to 490,000 integers in the array. *Bar* had an execution percentage of 48%, so we assigned the value 2 to

480,000 integers. The remaining 30,000 integers were assigned the value 0. After the shuffle, we selected the first 40,000 integers in the array as our simple random sample. We used this array technique in our simulation rather than directly employing the *rand()* function to generate samples as we might in other statistical analysis simulations in order to guarantee that the execution percentages were precisely what we wanted and to provide a more intuitive equivalence between the statistical simulation and an actual statistical analysis of an executing program.

3.1.1 Fixed Sample Size

For the first set of simulation experiments we took 40,000 samples of each of the 10,000 trials done with 6 different sets of values for *foo* and *bar*. The pairs of execution percentages we used are in the table below.

	Execution percentage pairings for six different experiments					
foo	.49	.40	.30	.20	.10	.02
bar	.48	.39	.29	.19	.09	.01

Table 3.1: Simulation experiment execution percentages for *foo* and *bar*.

We created histograms for each of the six different pairings and recorded the number of times that the statistics generated from the 40,000 samples resulted in *foo*'s calculated proportion – the inferred execution percentage – incorrectly indicating it was less than *bar*'s. These histograms follow below. Note that when reading the graphs, the bar above a given number on the x axis indicates the number of trials that had calculated statistics less than or equal to that number. For example, as shown in Fig. 3.1, the bar above .48 indicates that a little more than 3,500 of the 10,000 trials run resulted in a calculated statistic for *bar's* execution percentage (let's call it $\hat{p}_{bar}$) to be $.4775 < \hat{p}_{bar} <= .48$.

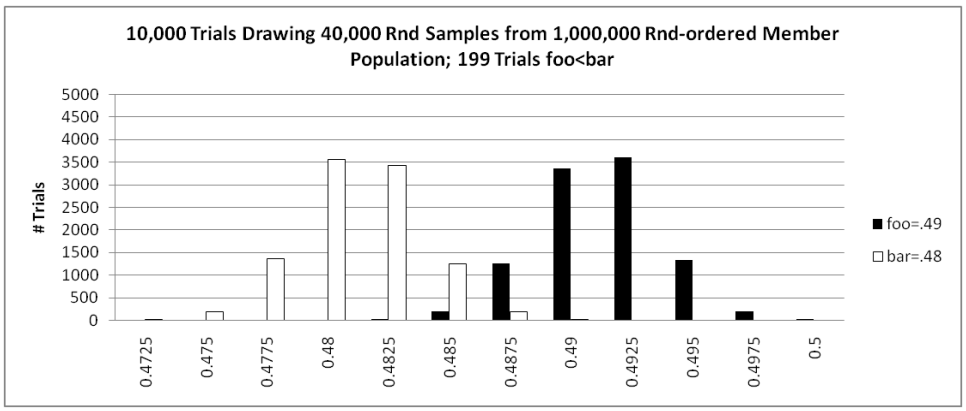

Figure 3.1: Simulation results of foo=.49; bar=.48; 40,000 samples

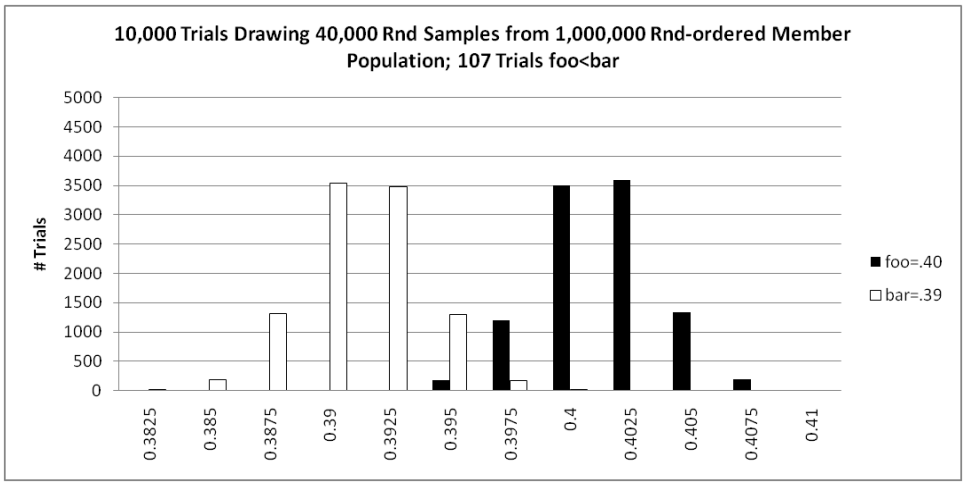

Figure 3.2: Simulation results of foo=.40; bar=.39; 40,000 samples

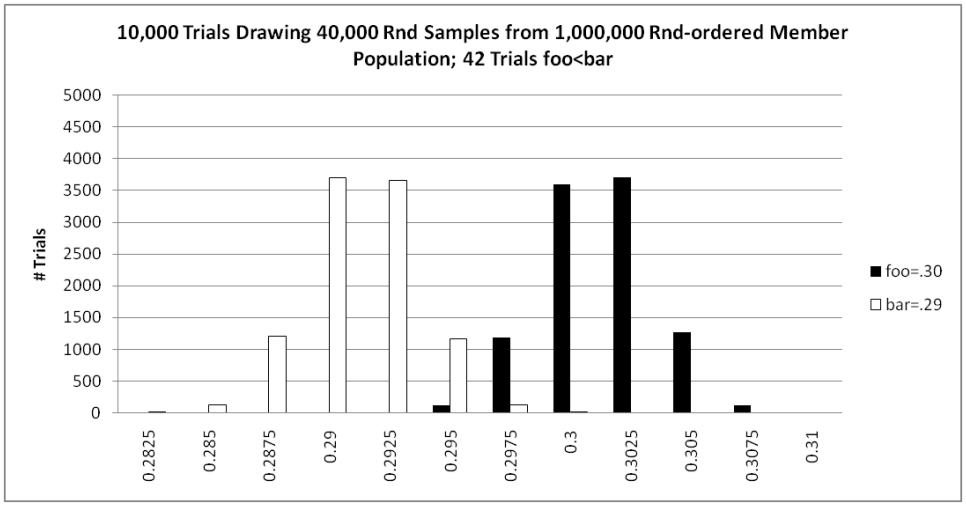

Figure 3.3: Simulation results of foo=.30; bar=.29; 40,000 samples

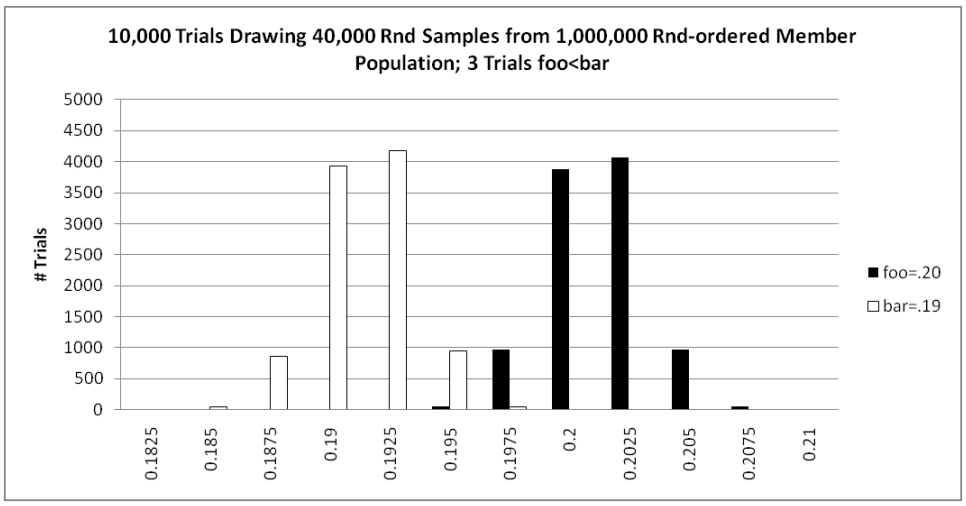

Figure 3.4: Simulation results of foo=.20; bar=.19; 40,000 samples

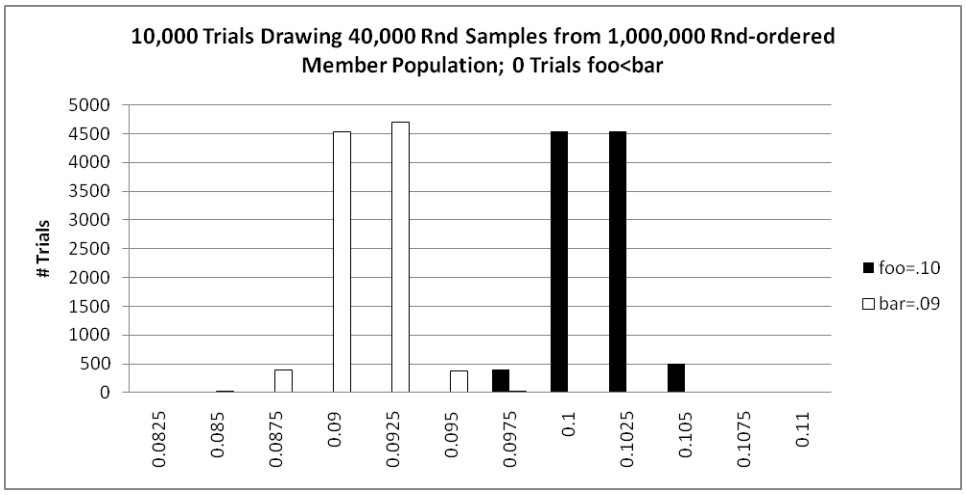

Figure 3.5: Simulation results of foo=.10; bar=.09; 40,000 samples

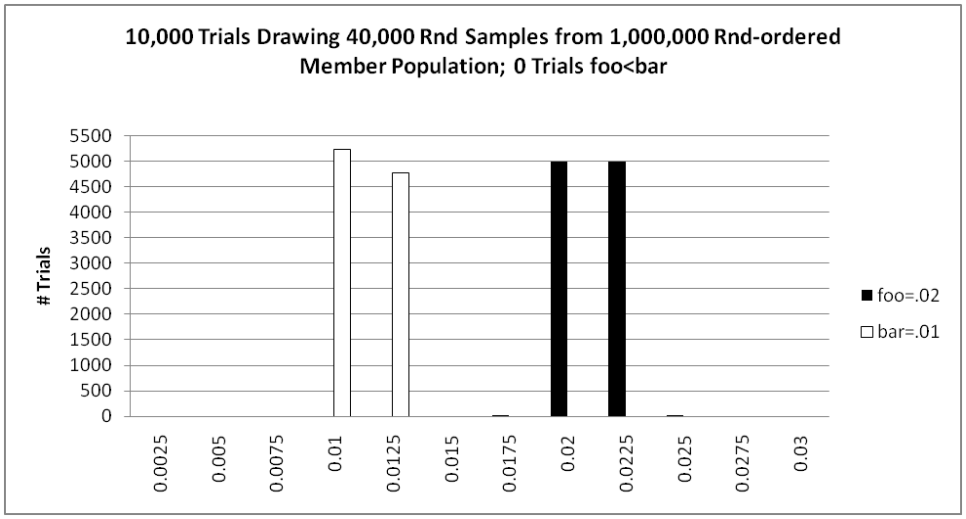

Figure 3.6: Simulation results of foo=.02; bar=.01; 40,000 samples

The first observation of note from this set of experiments is that the quality of the results differs markedly based on the percentage execution time of the functions being analyzed. In the first simulation with *foo* = .49 and *bar* = .48, the distribution of the statistics calculated from the samples is noticeably flatter and includes 199 trials where the execution percentage statistics calculated for *foo* and *bar* incorrectly indicate that the actual execution percentage of *foo* is less than that of *bar*. In the sixth simulation context where *foo* = .02 and *bar* = .01, the distribution is quite peaked and there are no trials incorrectly indicating that the actual percentage execution percentage of *foo* is less than that of *bar*.

The second observation of note is that the distribution does seem to generally conform to that of a normal/Gaussian distribution and reinforces our understanding that a normal approximation to our hypergeometric distribution would be appropriate for our use. To further validate this idea, we compared the first and sixth simulation results to calculated predictions of both hypergeometric and normal distributions with the same parameters. These results are depicted in the following two graphs.

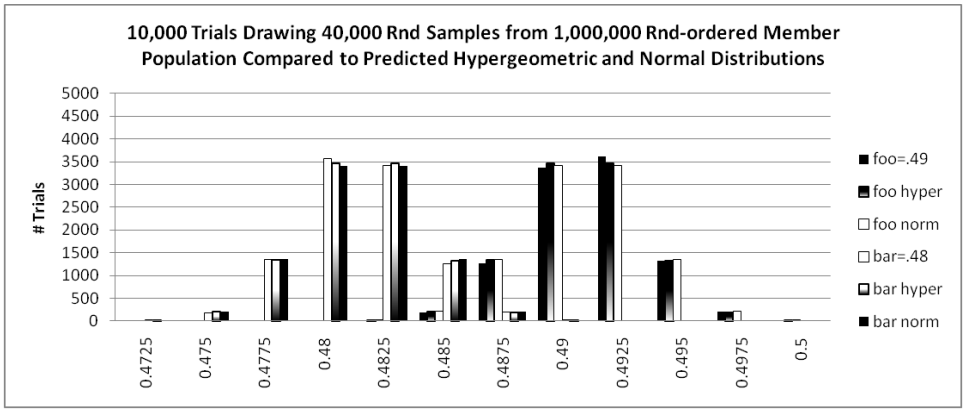

Figure 3.7: Comparing simulation, hypergeometric, and normal when foo=.49; bar=.48

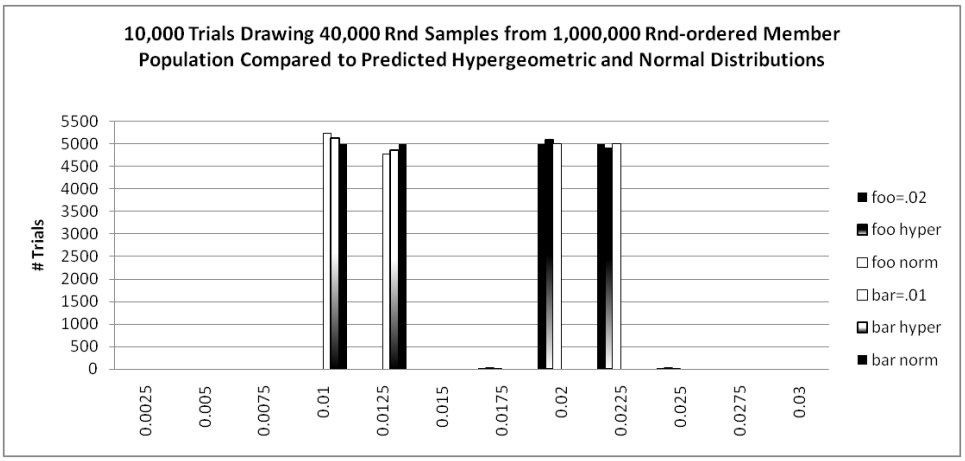

Figure 3.8: Comparing simulation, hypergeometric, and normal when foo=.02; bar=.01

The graphs appear to confirm that the normal approximation of our hypergeometric distribution generating experiments is reasonable. The approximation is slightly better when the execution proportion of interest is closer to .50, but is not that much worse with lower execution percentages.

3.1.2 Calculated Sample Size

For the second set of simulation experiments, rather than take the same fixed number of samples like the 40,000 we took for all previous simulations, we calculated how many samples were required for each of the 6 different sets of values we were using for *foo* and *bar* to achieve a similar statistical error result. Proceeding from our understanding that the normal approximation is a reasonable and appropriate approximation, we know that a confidence interval for a normally distributed random variable is calculated as follows:

$$\left(\hat{p} - z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \hat{p} + z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) \qquad (3.1)$$

In Eqn. 3.1, $\hat{p}$ is the execution proportion, *z* is the standard score (number of standard deviations) required for whatever confidence level is used, and *n* is the number of samples taken.

We wanted to determine the number of samples required to have high confidence that our experiments would result in a proper ordering of the execution percentage for *foo* and *bar*. So, for a given confidence level (we chose 95%) we wanted to know how many samples were needed for the lower end of *foo*'s confidence interval to be equal to the upper end of *bar*'s confidence interval. Letting *p* represent the actual proportion, or percentage execution, for *foo* and *q* represent the same for *bar*, we start from the following equation and then algebraically solve for *n*, the number of samples.

$$p - z\sqrt{\frac{p(1-p)}{n}} = q + z\sqrt{\frac{q(1-q)}{n}} \qquad (3.1)$$

$$n = \frac{z^2p - z^2p^2 + z^2q - z^2q^2 + 2z^2\sqrt{p(1-p)}\sqrt{q(1-q)}}{(p-q)^2} \qquad (3.2)$$

Using this equation to determine our target sample size for each of the 6 different simulation contexts we again ran 10,000 trials with *foo* and *bar* set to 6 different sets of values and taking 6 different numbers of samples per the table below. The graphs that follow provide histograms of the results.

	Execution percentage pairings for six different experiments					
Foo	.49	.40	.30	.20	.10	.02
Bar	.48	.39	.29	.19	.09	.01
# samples	38,375	36,718	31,956	24,115	13,201	2,204

Table 3.2: Execution percentage pairings

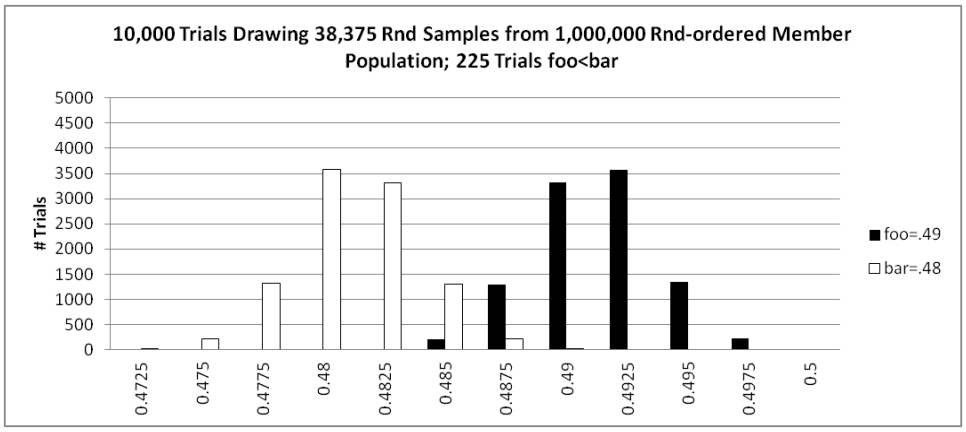

Figure 3.9: Simulation results of foo=.49; bar=.48; 38,375 samples

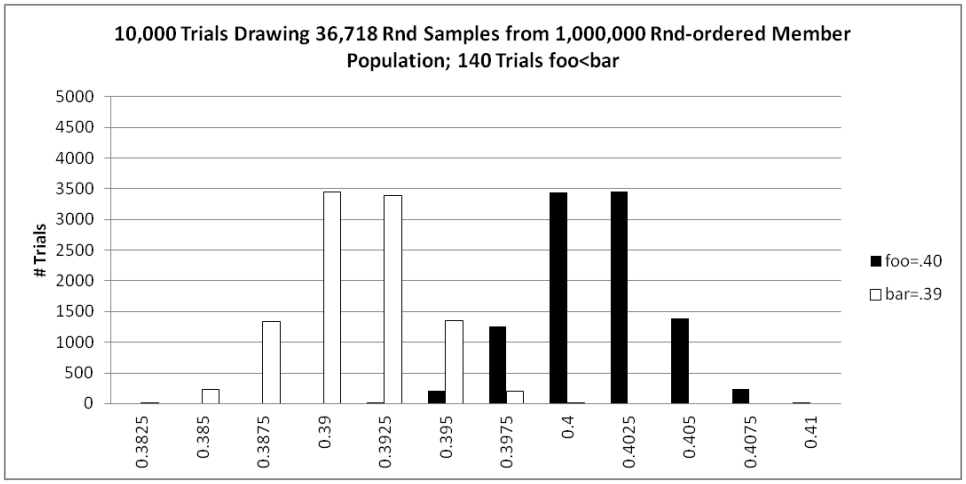

Figure 3.10: Simulation results of foo=.40; bar=.39; 36,718 samples

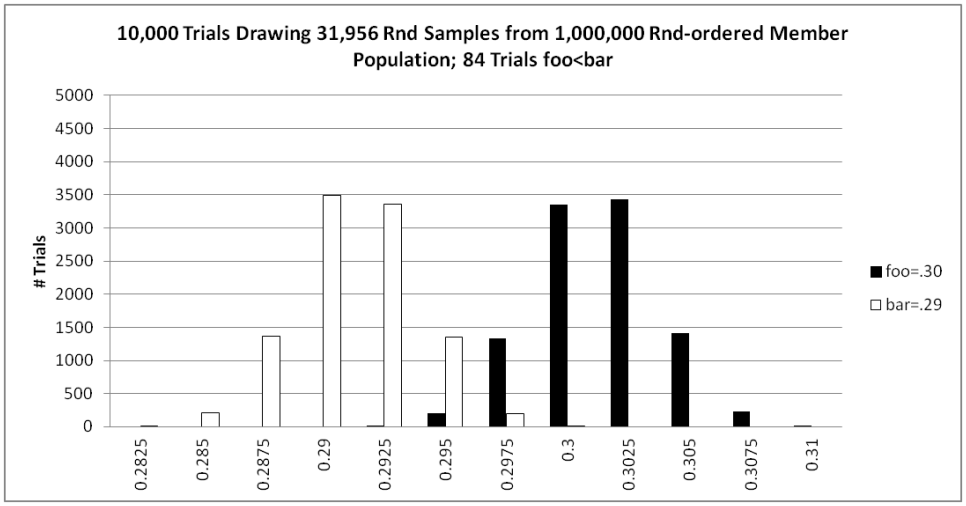

Figure 3.11: Simulation results of foo=.30; bar=.29; 31,956 samples

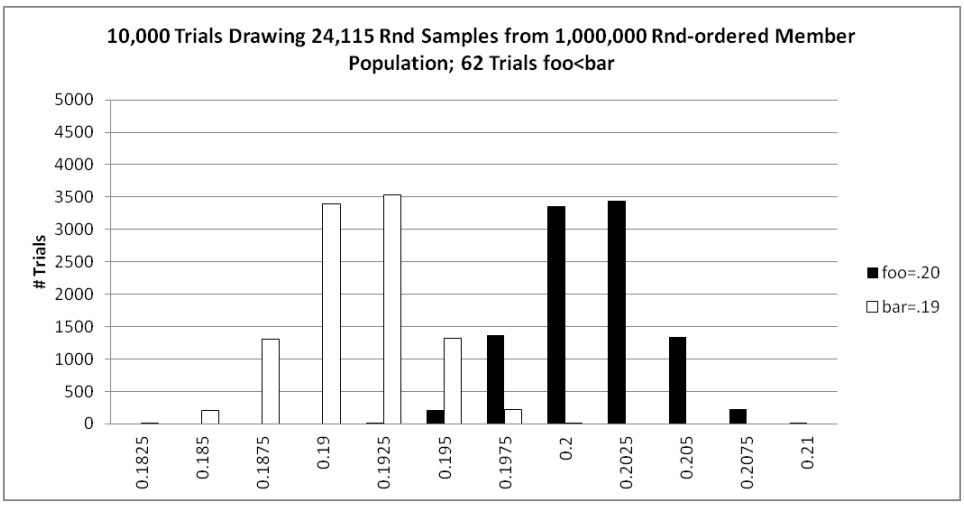

Figure 3.12: Simulation results of foo=.20; bar=.19; 24,115 samples

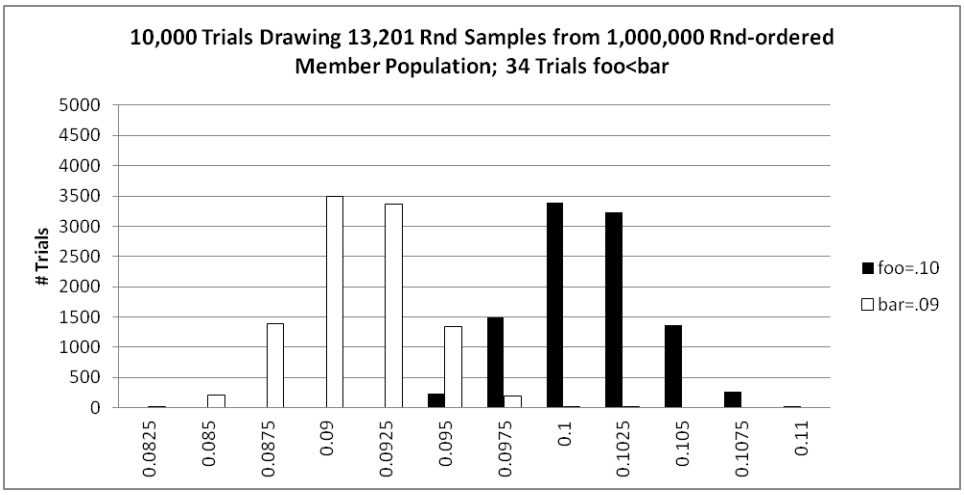

Figure 3.13: Simulation results of foo=.10; bar=.09; 13,201 samples

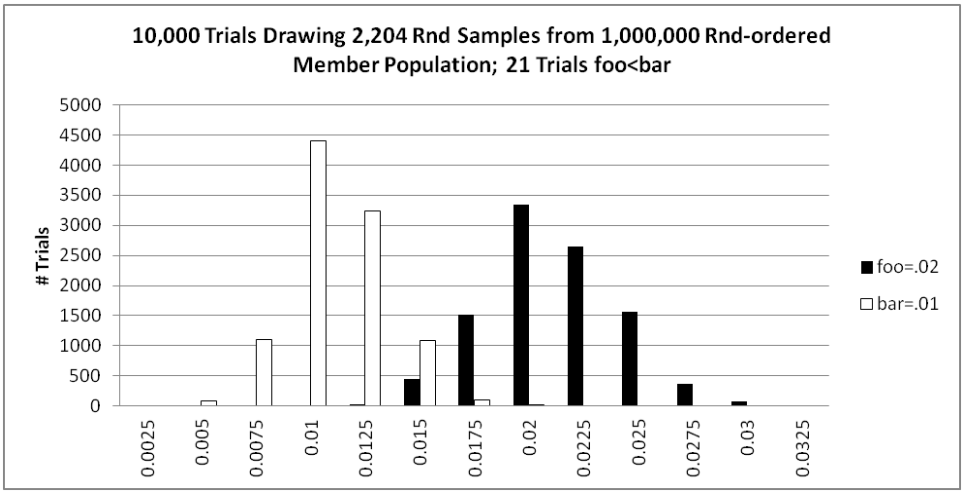

Figure 3.14: Simulation results of foo=.02; bar=.01; 2,204 samples

From this second set of simulation experiments we first note that once again, the quality of the results differs based on the percentage execution time of the functions being analyzed. With *foo* = .49 and *bar* = .48 there are 225 trials in which the statistics calculated for *foo* and *bar* incorrectly indicate that the actual execution percentage of *foo* is less than that of *bar*. Though the shape of the distributions change little among the six sets of simulations, as the values of *foo* and *bar* are changed there are fewer statistics calculated that incorrectly order *foo*'s and *bar*'s relative execution percentage. By the sixth simulation context in this set where *foo* = .02 and *bar* = .01, there are only 21 trials that produce statistics indicating an incorrect relative ordering of *foo* and *bar*. This result is rather surprising. In addition to taking over 90% fewer samples during the sixth simulation context, compared with the first, there were also 90% fewer trials where the statistics incorrectly indicated the percentage of execution for *foo* < *bar*.

The second thing to note from this second set of simulations is that the distributions for the first five experiments conform quite well with that of a normal distribution as well as with each other, while the sixth doesn't quite provide that same visual affirmation. We provide the following two graphs to once again compare the first and sixth simulation contexts to the expected distribution calculated for both a hypergeometric and a normal distribution.

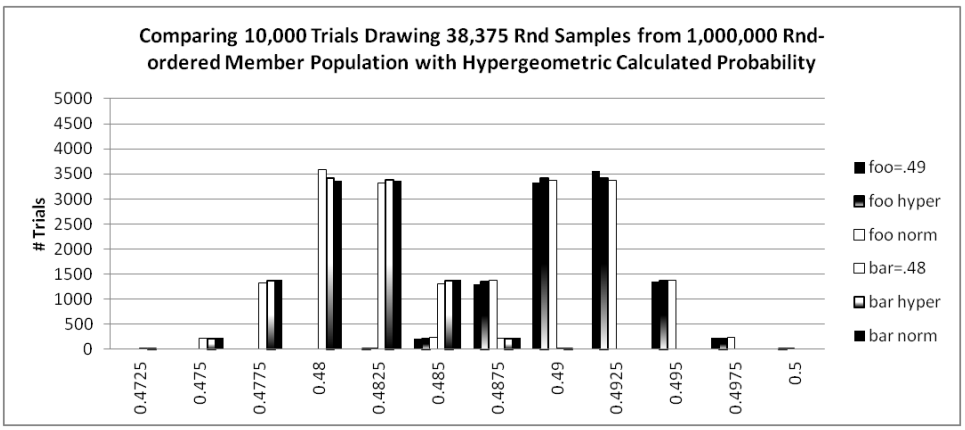

Figure 3.15: Comparing simulation, hypergeometric, and normal when foo=.49; bar=.48

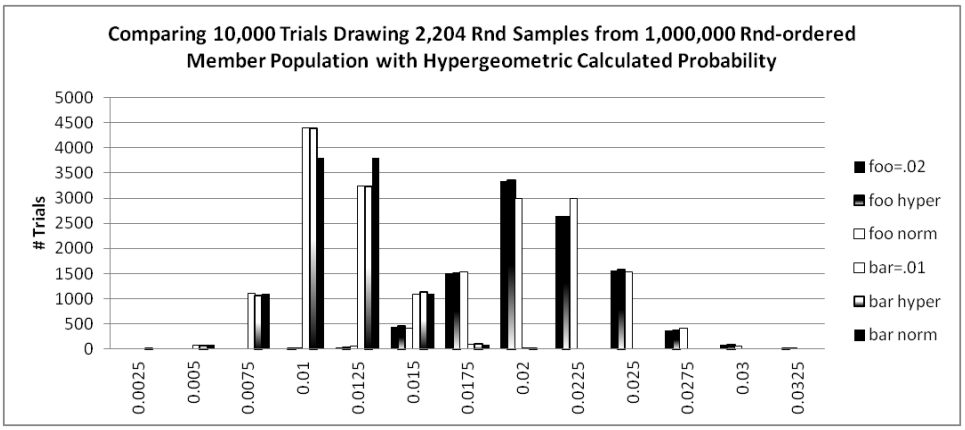

Figure 3.16: Comparing simulation, hypergeometric, and normal when foo=.02; bar=.01

The first graph once again appears to confirm that the normal approximation of our hypergeometric distribution generating experiments is reasonable, while the second graph clearly depicts a difference. The first interval less than (to the left) the actual execution percentage of both *foo* and *bar* is not proportionate to the first interval greater than the actual execution percentages. The difference is more marked with *bar* than with *foo*. Beyond the first intervals, the distributions do appear equivalent. It appears that the normal approximation of the hypergeometric distribution begins to break down when the execution percentage of a function of interest is small (near 0%). [Note: we expect we would find the same distribution divergence with large execution percentages (near 100%) due to the symmetric nature

of the distribution.] Fortunately, the divergence is such that results are statistically more accurate, so the approximation suits our purposes.

3.2 Sampling Calibration Program

We next conducted a series of experiments meant to closely mirror the conditions of the simulations. We created a calibration program that would execute 1,000,000 function calls in about 300 seconds that could be configured to distribute the function calls in such a way that we could control the percent of execution time for which each function was responsible. We used the same 1,000,000 integer array, shuffled in the same manner, to call functions in the same order as was done in the simulations. The functions called were composed of identical loops executing simple mathematical operations designed to take 300 microseconds each to execute and thereby provide a program that runs for a total of 300 seconds. This calibration program was designed with minimal memory requirements and a single function call depth, to help minimize the overhead and perturbation effects of sampling. Due to the imprecise nature of this technique the program ends up running approximately 300 seconds per execution iteration.

We used the hpcrun component of HPCToolkit [56] to sample our executing calibration program. Because we used timer intervals and the Linux ITIMER_PROF, interrupts cannot occur with greater frequency than a jiffy (1 millisecond in our case), the number of samples we could take was not as flexible and precise as with our simulations. Note that jiffy in this Linux context is the smallest unit of time of the software clock. It is determined by the value of the kernel constant HZ which can be set to 100, 150, or 1000. Because of the far greater total execution time required to

run experiments with the calibration program, as compared to the simulations, we conducted 100 trials per experiment configuration.

3.2.1 Fixed Sample Size

For the length of our program we had the option of taking about 42,800 samples (7 millisecond intervals) or about 37,500 samples (8 millisecond intervals). We chose 37,500 because it was slightly closer to the 40,000 we used in our simulation.

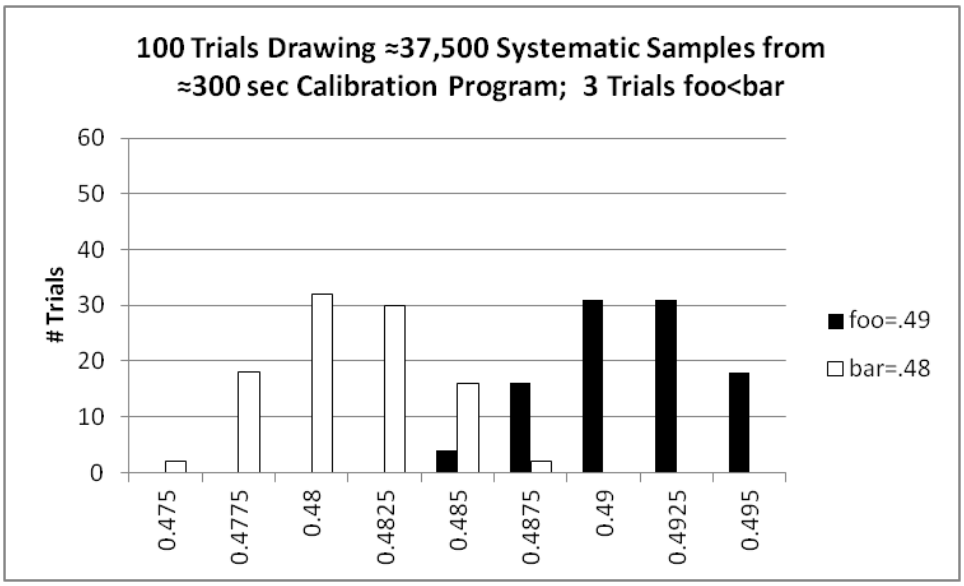

Figure 3.17: Calibration results of foo=.49; bar=.48; ≈37,500 samples

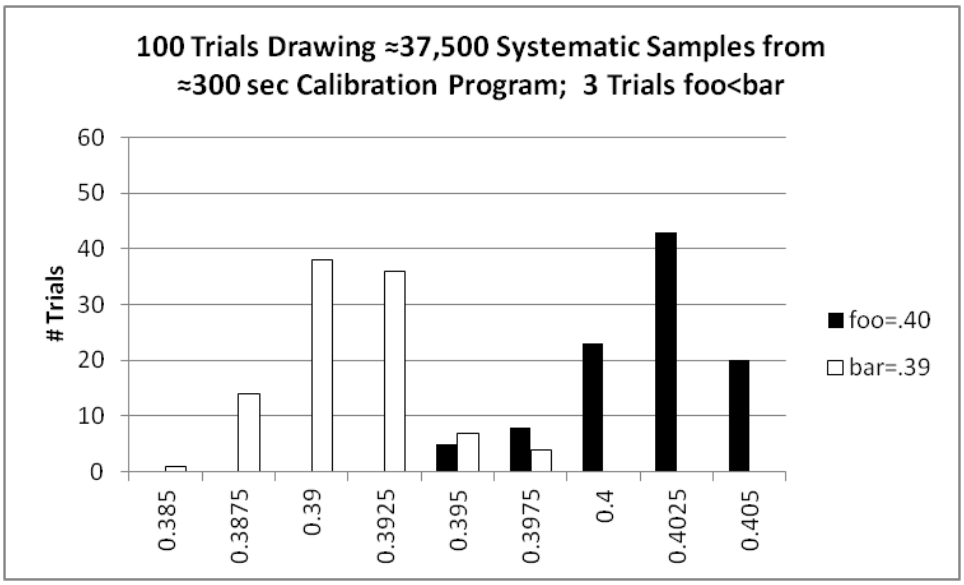

Figure 3.18: Calibration results of foo=.40; bar=.39; ≈37,500 samples

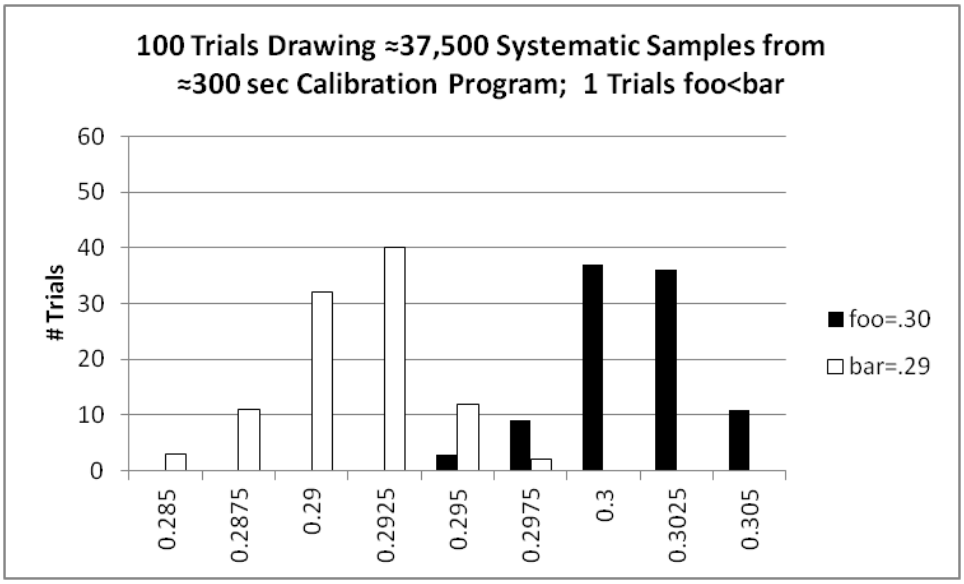

Figure 3.19: Calibration results of foo=.30; bar=.29; ≈37,500 samples

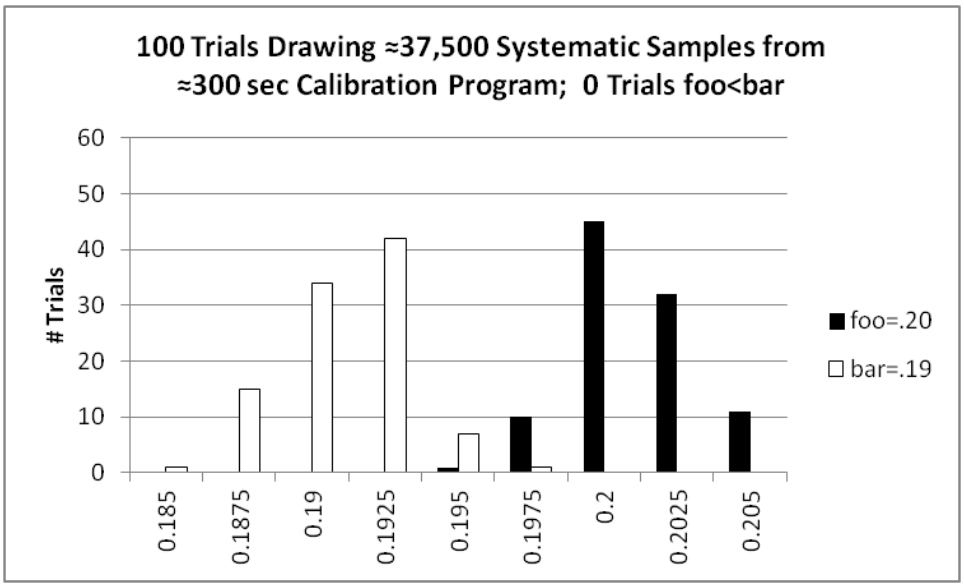

Figure 3.20: Calibration results of foo=.20; bar=.19; ≈37,500 samples

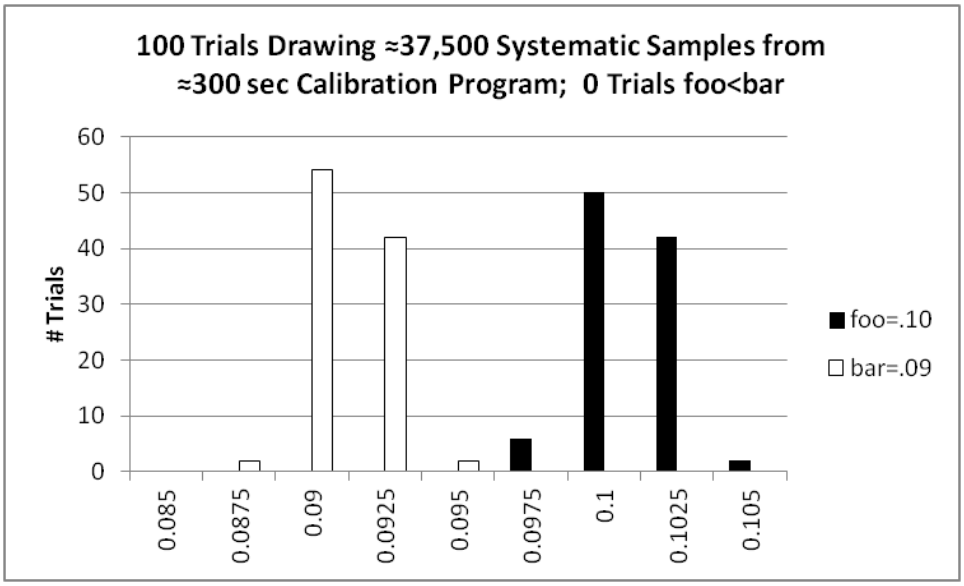

Figure 3.21: Calibration results of foo=.10; bar=.09; ≈37,500 samples

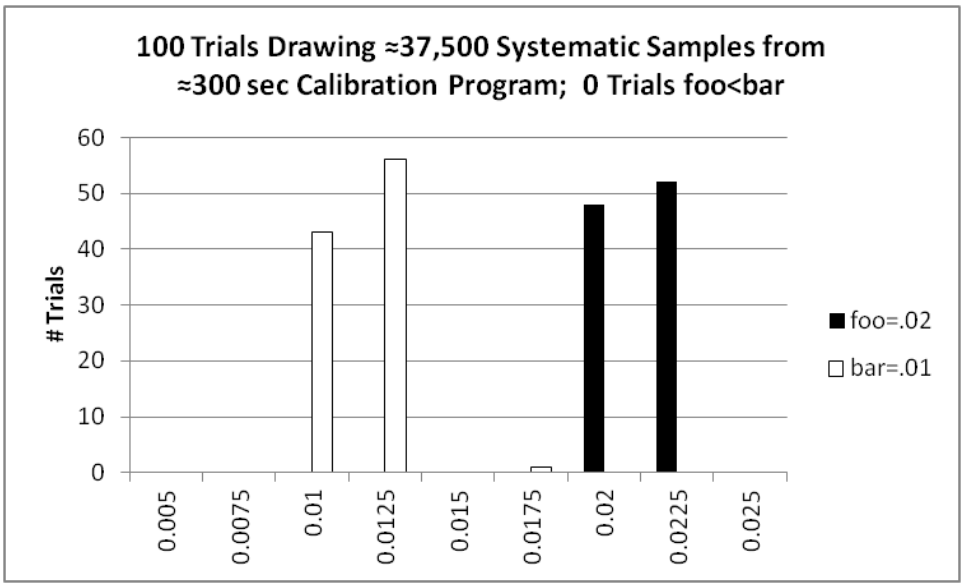

Figure 3.22: Calibration results of foo=.02; bar=.01; ≈37,500 samples

3.2.2 Calculated Sample Size

Similar to our investigation progression with the simulations, we repeated the previous experiments using different sample rates. Once again, the millisecond granularity restriction with the ITIMER_PROF affected the experiments, so experiments with the first two sets of execution percentages were not repeated.

	Execution percentage pairings for six different experiments					
Foo	.49	.40	.30	.20	.10	.02
Bar	.48	.39	.29	.19	.09	.01
# samples ≈	37,500	37,500	33,400	25,000	13,000	2,200

Table 3.3: Execution percentage pairings

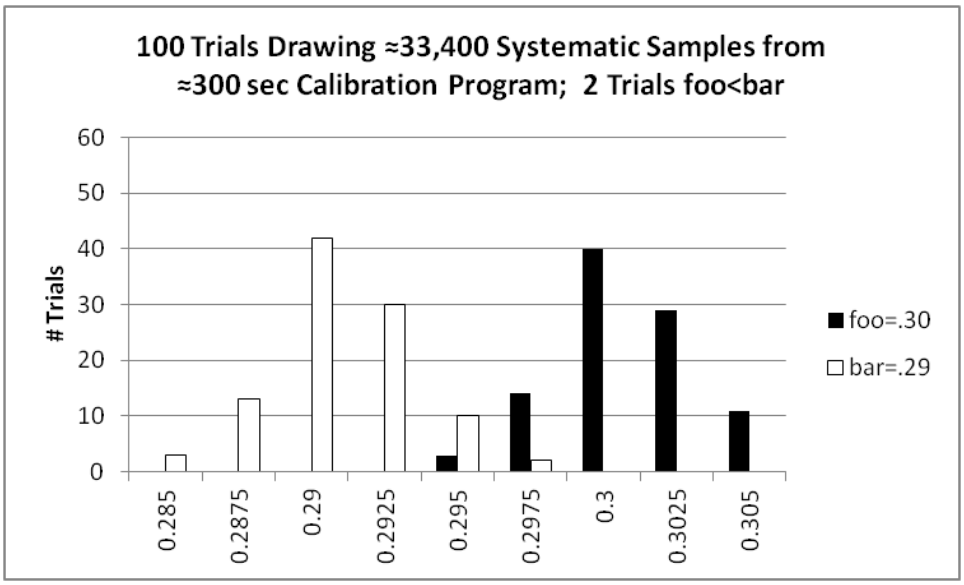

Figure 3.23: Calibration results of foo=.30; bar=.29; ≈33,400 samples

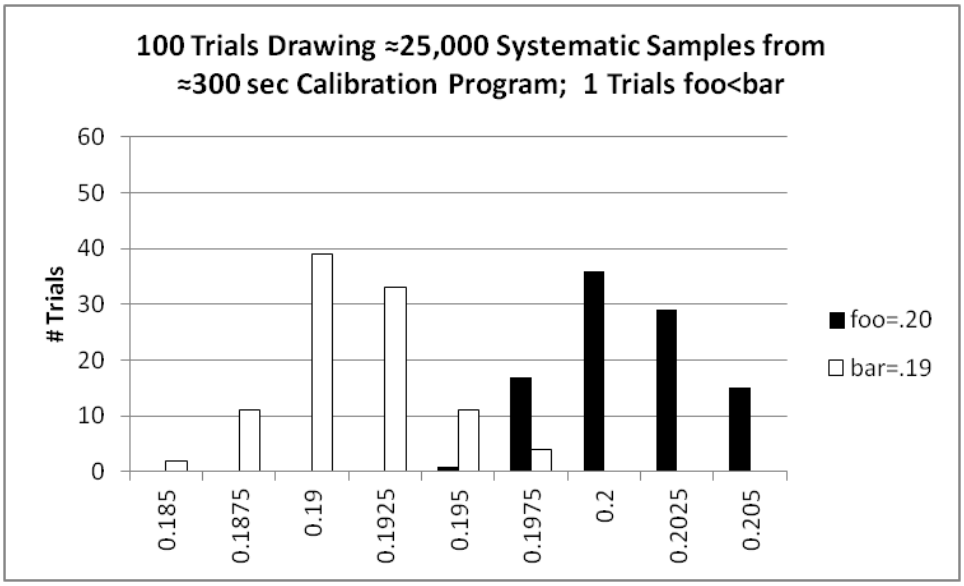

Figure 3.24: Calibration results of foo=.20; bar=.19; ≈25,00 samples

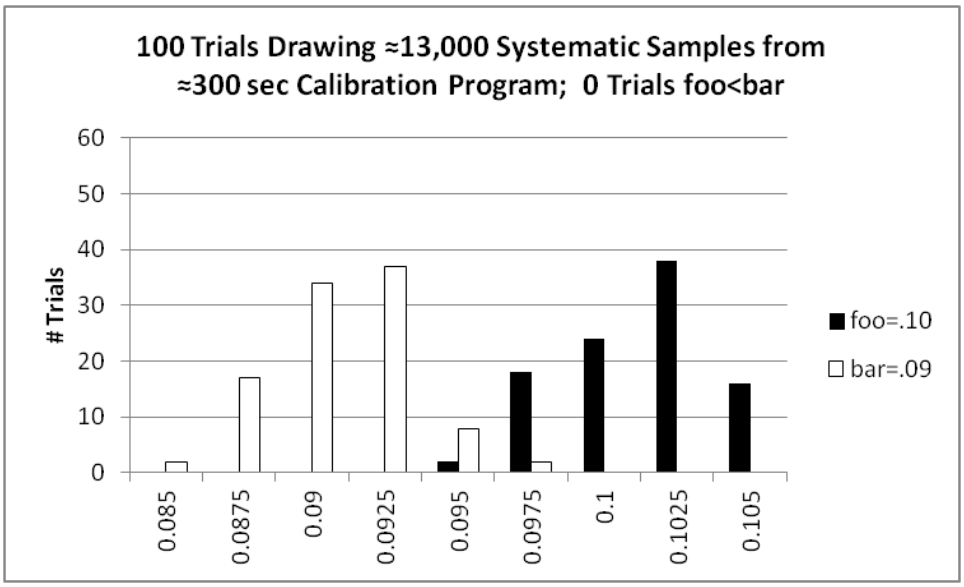

Figure 3.25: Calibration results of foo=.10; bar=.09; ≈13,00 samples

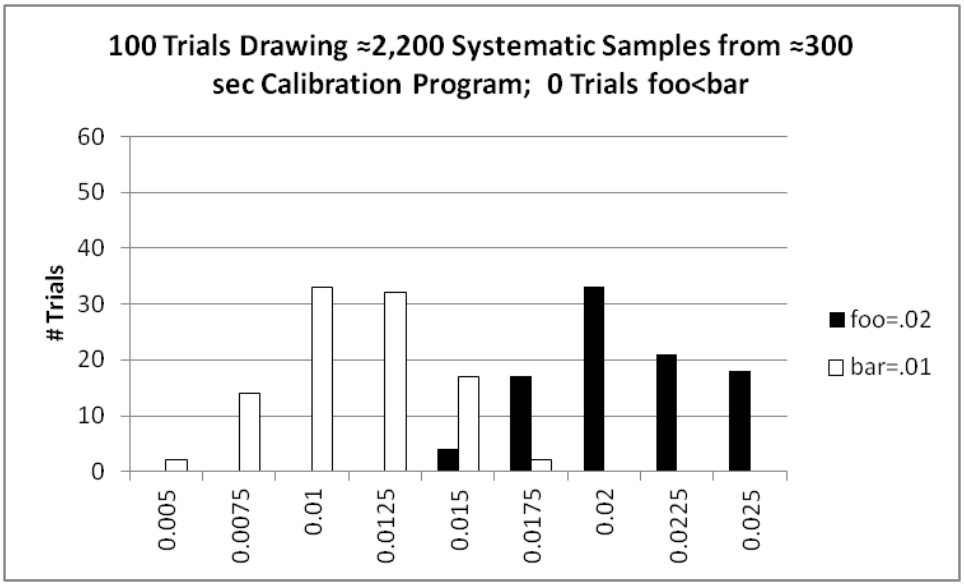

Figure 3.26: Calibration results of foo=.02; bar=.01; ≈2,200 samples

Comparison of # trials foo < bar (per 100)					
		Simulation		Calibration	
Foo	*Bar*	40,000	Various	37,500	Various
.49	*.48*	1.99	2.25	3	3
.40	*.39*	1.07	1.4	3	3
.30	*.29*	.42	.84	1	2
.20	*.19*	.03	.62	0	1
.10	*.09*	0	.34	0	0
.02	*.01*	0	.21	0	0

Table 3.4: Comparison of number of trials where foo < bar (per 100 runs)

Important observations can be made from the experiments conducted in this empirical study. First, our results suggest that far less sampling than is typically collected during dynamic performance analysis is needed to achieve meaningful statistical results for execution profiles, at least when the primary objective is ensuring a proper ordering of functions based on percent of execution time or simply identifying where the majority of execution time is taking place. Second, calculated results for smaller p are likely more accurate than larger p for a given sampling experiment, if perturbation effects have not disproportionately affected their execution.

Chapter 4: Model

4.1 Abstract Model

Taking more and more samples from a population produces a measured value, or *statistic*, that gets closer and closer to the true *population parameter* of interest. Unless we take all possible samples (a census) the statistic will have a confidence interval associated with it within which the actual population parameter lies with some degree of likelihood. Taking more samples narrows the confidence interval in which it is very likely the true population parameter is included, making the measured value more precise and more accurate with each additional sample.

The main problem with sampling a running program is that the process of taking samples perturbs the program execution in a manner that very likely is not fully understood, thus resulting in a statistic that no longer strictly describes only the original, un-sampled running program. Dilation of execution time is typically the easiest perturbation effect to observe, but changes to the spatial and temporal access patterns of cache and memory locations, event and execution pipeline reordering, additional context switching and interrupts, and register interlock stalls [42, 50] are all likely to occur and are much more difficult to confirm. As well, distortion of the accuracy of the measurements designed to provide insight into program executions is a problem of perturbation [40].

This problem is compounded by the diminishing return aspect of sampling on measurement error reduction. Though a statistic becomes more precise (and hopefully more accurate) with each additional sample, the improvement in precision provided with each additional sample is less and less. Consider for example, once

we've taken x samples and have a confidence interval of $\pm$r: How many samples, y, would we need to take to reduce the confidence interval by half, such that once we've made that much progress in measurement error reduction we cannot possibly take enough samples to reduce the remaining confidence interval to 0 without conducting a full census? The answer is relatively straight forward:

$$\frac{1}{2}z\sqrt{\frac{\hat{p}(1-\hat{p})}{x}} = z\sqrt{\frac{\hat{p}(1-\hat{p})}{y}} \tag{4.1}$$

$$\frac{1}{2}z\sqrt{\hat{p}(1-\hat{p})}\frac{1}{\sqrt{x}} = z\sqrt{\hat{p}(1-\hat{p})}\frac{1}{\sqrt{y}} \tag{4.2}$$

$$\frac{1}{2\sqrt{x}} = \frac{1}{\sqrt{y}} \tag{4.3}$$

$$y = 4x \tag{4.4}$$

To make this more concrete, assume z=1.96, $\hat{p}$=.5, and we take 100 samples to produce a statistic with a confidence interval of $\pm$ .098. To cut that confidence interval in half ($\pm$ .049) we need to take 300 additional samples or 400 total samples ($y=4x$). At this point, reducing the remaining confidence interval to 0 *cannot* be accomplished as easily as taking another 300 samples; it would take a complete census. Thus, the diminishing return aspect of sampling on measurement error reduction is a significant consideration and cannot be ignored.

We can abstractly describe the competing effects of decreasing measurement error and increasing perturbation error on our performance measurement statistic with the diagram in Fig. 4.1. The intuition we gain from this abstract model is that it seems likely there exists a sample size that provides the best overall balance between reducing measurement error and limiting perturbation error effects. If we can

determine this point we can calculate the appropriate sampling period to use to achieve a performance measurement statistic with the overall least residual error.

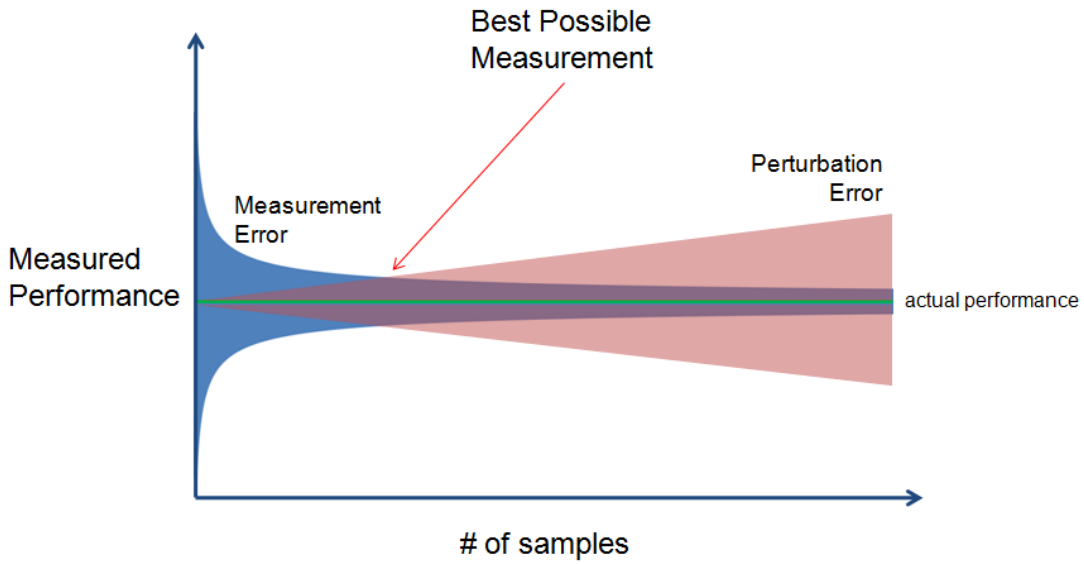

Figure 4.1: Abstract model of the effect on measurement error and perturbation error as an increasing number of samples is taken during a program's execution.

4.2 Analytical Model

4.2.1 Background

We begin constructing our analytical model from our abstract model by considering a likely performance measurement context. There exists a program in which there is a function *foo*. Though it's not critical to the discussion just yet, we will consider *foo* to be the function which takes the greatest proportion of the given program's execution time. We would like to know:

How much execution time is spent in function *foo*?

We imagine a simple statistical profiling system that halts the execution of our program and notes which function is currently in execution. A total of n samples are taken, m of the samples are found to be function *foo*, and $\boldsymbol{T}$ is the measured execution

time of the entire program. We can calculate the measured proportion of execution time for function *foo* as:

$$\hat{p} = \frac{m}{n} \tag{4.5}$$

We can then answer our performance analysis question where $t_{foo}(n)$ is the measured execution time for *foo* after *n* samples are taken as follows:

$$t_{foo}(n) = \boldsymbol{T}\hat{p} \tag{4.6}$$

We know that the statistic $\hat{p}$ calculated with m and *n* is likely not equivalent to the actual population parameter calculated with *M* and *N*, where *N* is the total number of all possible samples and *M* is the total number of samples that would be taken during the execution of *foo*. So we could generate a confidence interval around the statistic $\hat{p}$ into which we expect to find the true population parameter with some established confidence level. For this calculation we use the estimated standard deviation or standard error formula,

$$\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{4.7}$$

to create a confidence interval of:

$$\left(\boldsymbol{T}\left(\hat{p} - \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right), \boldsymbol{T}\left(\hat{p} + \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right)\right) \tag{4.8}$$

Into this confidence interval we would expect the actual execution time of *foo* to be included with roughly 68% confidence. Because we only use one standard error we would be limited to the 68% confidence level, but we can introduce the well-known *z* term in front of the standard error to calculate confidence intervals at a greater range of confidence levels:

$$\left(\boldsymbol{T}\left(\hat{p} - z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \right), \boldsymbol{T}\left(\hat{p} + z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \right) \right) \tag{4.9}$$

At this point we expand our term ***T***, the measured total execution time of the program which contains *foo*, to explicitly show that it includes both the true (un-sampled) execution time, T, and the additional execution time attributable to the sampling which will be the number of samples, *n*, multiplied by the overhead, *o*, or execution time required per sample. Thus, equation (4.2) becomes:

$$t_{foo}(n) = (no + T)\hat{p} \tag{4.10}$$

4.2.2 Applying the Intuition

Given these intermediate steps and the intuition from our abstract model, we can now create an analytic function to reflect the changes to a measured statistic with respect to measurement error and perturbation error that occur with each additional sample. The notation for our analytic function is in Table 4.1.

$t_{foo}(n)$	Execution time calculated for *foo* based on *n* samples
n	Number of samples
o	Overhead time cost per sample
$\hat{p}$	Measured proportion of total program execution time spent in *foo*
T	Un-sampled (true) total program execution time
z	Standard score (z-value) for confidence level used

Table 4.1: Analytic function notation

$$t_{foo}(n) = no\hat{p} + T\left(\hat{p} \pm z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \right) \tag{4.11}$$

The analytic function combines the two curves from the abstract model. Increasing perturbation error is captured in the first part of the equation by multiplying together the number of samples taken, the overhead time required for each sample, and the proportion of samples that interrupted *foo*. Decreasing

measurement error is represented by the common formula for a confidence interval of a proportion multiplied by the total program execution time.

Recall that this analytic function is created to answer the specific question, how much execution time is spent in function *foo*? In this context, *foo* is the function taking the largest proportion of execution time in the program being analyzed. We choose to frame the question in this particular manner for two primary reasons. First, we wish to explicitly focus on the dilated execution time that is an indication of the perturbation caused by sampling. Most dynamic performance analysis typically results in the generation of a list of functions described chiefly by proportion of attributed execution time. Because the primary focus is on proportion of execution time it is probable that perturbation effects are masked as there is usually no accompanying comparison to actual (or expected actual) execution time.

The second reason we narrowly frame the question is that the reliability of statistical results is greatly affected by the size of the proportion of the measured function (see Chapter 3). The same number of samples provides a different level of precision for each different-sized proportion. The larger a function's proportion of execution time, the greater the width of the confidence interval, thus the less precise the statistic for agiven confidence level. Observe that the sum $p(1-p)$ ranges from 0 (when $\hat{p} = 1$ or $\hat{p} = 0$) to .25 (when $\hat{p} = .5$), but also note that this fact doesn't invalidate our previous sentence because although a sampled statistical proportion of .5 will have a wider confidence interval than one of .6, no program can have function execution proportions add up to a sum greater than 1. For example, if a function *foo* has proportion .6, then function *bar* must have a proportion of the execution time that

is strictly ≤ .4. By focusing on the function taking the largest proportion of execution time we guarantee the results for all other functions will never be less statistically precise and most often they will be more statistically precise. For example, given a program with functions *foo*, *bar*, and *baz* calculated after 1,000 samples to take respectively .70, .20, and .10 of total execution time, they would have corresponding standard error values of .0145, .0126, and .0095. So, the confidence interval around *foo*, (.6855, .7145), would be wider than those around *bar*, (.1874, .2126), and *baz*, (.0905, .1095).

We make the following set of assumptions for our model:

- Because the full spectrum of perturbation effects is difficult to capture, we make the simplifying assumption that time (the additional time the analyzed program spends in execution beyond the unperturbed execution time) is a representative surrogate for all perturbation effects.
- The Hypergeometric distribution $H(m; N, n, M)$ may be appropriately approximated by a Normal (Gaussian) distribution where $n\hat{p} \geq 10$, and M, N are at least an order of magnitude larger than n. See discussion in section 2.2.1.
- Systematic sampling provides results similar to random sampling and occurs asynchronously with respect to any periodic events in the analyzed program.

4.2.3 Analytic Function

Using our current version of the analytic function (4.11), we created the graph in Fig. 4.2 that visually presents the predicted changing behavior of the performance

measurement of a hypothetical function *foo* that is 20% of an executing program of 300 seconds where each sample incurs 250 microseconds of overhead.

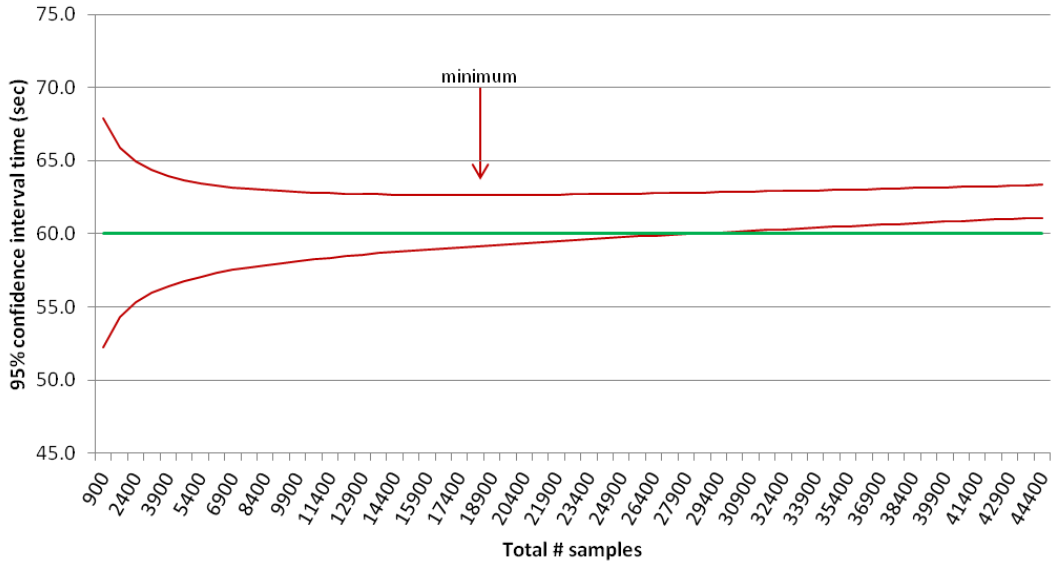

Figure 4.2: This graph presents the predicted results for measurements of the execution of *foo* within a program that executes for a total of 300 seconds and in which *foo* accounts for 20% of the execution time. Of the measurements taken at each sample level, 95% of them are expected to fall within the upper and lower curves.

The top curve depicts the higher end of the confidence interval where the standard error is added to p.

$$t_{foo}(n) = no\hat{p} + T\left(\hat{p} + z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) \qquad (4.12)$$

The bottom curve depicts the lower end of the confidence interval which subtracts the standard error.

$$t_{foo}(n) = no\hat{p} + T\left(\hat{p} - z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) \qquad (4.13)$$

The slightly thicker straight line at 60 seconds represents the true execution time for *foo*. The curves are calculated with a 95% confidence level (z = 1.96) which means we would expect 95% of our measurements of function *foo* for any given

sample size to fall within the two curves. For example, if we ran this hypothetical 300 second-long program 100 times and took 2,400 samples each time we would expect to find about 95 of the measurements of *foo* fell between 55 and 65 seconds while 5 measurements fell outside those bounds.

Displaying the data generated by our analytic function in this way makes it easier to see the change in the statistical precision of the confidence interval as more samples are taken. It is also easier to see there is an intuitive benefit to using the top curve of the model over the bottom curve to identify the sample size at which the combined effect of both measurement error and perturbation error are minimized. Notice there exists a minimum along the top curve that indicates at what point the measurement is no longer making downward progress towards the true execution time for *foo* as the decreasing measurement error begins to be dominated by the increasing perturbation effect of the sampling overhead and the curve adopts a positive slope. If we can calculate the minimum of this upper curve, we can then determine the sample size (and corresponding sampling period) needed to reach this point that strikes the best balance between the two types of error.

So, we simplify our analytic function to only add the standard error because it most clearly displays the point at which the reduction in measurement error begins to be less than the accumulating perturbation error. Our final version of the analytic function is:

$$t_{foo}(n) = no\hat{p} + T\left(\hat{p} + z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) \qquad (4.14)$$

Finding the minimum along this curve is a matter of taking the first derivative of our analytic function (4.15) with respect to the number of samples, *n*, setting it to 0, and solving. The steps we used to find equation (4.15), the first derivative of our analytic function, and equation (4.16), the result of solving for n after setting the derivative to 0, are contained in Appendix A.

$$t'_{foo}(n) = o\hat{p} - \frac{Tz\sqrt{\hat{p}(1-\hat{p})}}{2\sqrt{n^3}} \tag{4.15}$$

$$n = \sqrt[3]{\left(\frac{Tz\sqrt{\hat{p}(1-\hat{p})}}{2o\hat{p}}\right)^2} \tag{4.16}$$

Using (4.12) and the parameter values mentioned earlier (T = 300, z = 1.96, $\hat{p}$ = .2, o = .00025) we get the following result:

$$n = \sqrt[3]{\left(\frac{300(1.96)\sqrt{.2(1-.2)}}{2(.00025).2}\right)^2}$$

$$n = \left(\frac{235.2}{.0001}\right)^{\frac{2}{3}} = 17{,}686$$

Note that this is the minimum point along the top curve in Fig. 4.2. Beyond this point in the graph the confidence interval continues to narrow as more samples are taken, but the confidence limits are converging on a number getting further away from what would be the actual unperturbed value of *foo*'s execution time. Beyond 28,000 samples the lower curve no longer even encompasses the true unperturbed execution time.

4.3 Simulation

In order to validate our ideas and corresponding analytical model we next constructed a program to simulate the process of sampling the execution of a program which spends a specific fraction of time in a single function. The program was designed in such a manner to allow us to corroborate the statistical results predicted by our analytical model with the range of actual results from the simulation. The program is written in C. We simulate a time series of sampled values as an integer array of size 1,000,000. This array represents the execution of a program running for 300 seconds where each integer represents a notional 300 microseconds of execution time. We assigned 200,000, exactly 20%, of the integers the value 1 to represent the function *foo* and assigned the other 800,000 integers the value -1, representing all other non-*foo* functions.

Once generated, the entire array was shuffled with the Fisher-Yates shuffle algorithm making use of the C standard library function *rand*. The result was the 1,000,000 integer array with the 200,000 values of 1, representing foo, spread randomly (not uniformly) throughout. We used this array technique in our simulation rather than directly employing the *rand* function to generate samples as we might in other statistical analysis simulations in order to guarantee that the execution percentage p was precisely 20% and to provide a more intuitive equivalence between the statistical simulation and an actual statistical analysis of an executing program.

With each integer representing a notional 300 microseconds of execution time, this design is relatively coarse-grained compared to the number of execution steps that could be sampled in an actual 300 second-long program. However, we felt

a total population size of 1,000,000 sufficient for simulation purposes in light of the fact that a more fine-grain representation of execution time with a larger array would not greatly impact the statistical calculations and therefore provide little gain. This is because the standard error term we're using, equation (3.3), is really a simplification based on the central limit theorem assumption that all samples are randomly chosen with replacement from an infinitely large population. Our simulation takes samples without replacement from a finite population as does all sample-based dynamic performance analyses of which we're aware. A more fully correct formula for error would include the finite population correction factor [7] making the population proportion standard error from which confidence errors are calculated as follows:

$$\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\sqrt{\frac{N-n}{N-1}} \qquad (4.13)$$

We chose to make use of the simplified version because for very large populations N and sample sizes n of 5% or less of the total population, the finite population correction factor typically evaluates to something very close to 1 and has minimal impact on the calculated error measurement. Additionally, determining N for a program execution is likely impossible.

Once the array was prepared we ran 1,000 iterations of simulated 300 second executions for each of the sampling sizes from 900 to 44,400 at increments of 1,500. So, we ran 1,000 different iterations which took 900 random samples, then 1,000 iterations of 2,400 random samples, etc., all the way to 1,000 iterations of 44,400 random samples. The sampling scheme was random, not systematic as is typically done in dynamic performance analysis and meant to approximate systematic sampling rates of 3 per second through 148 per second at increments of 5. To factor in the

perturbation error each sample was assessed an overhead of 250 microseconds. The results of our simulation are graphed in Fig. 3.3.

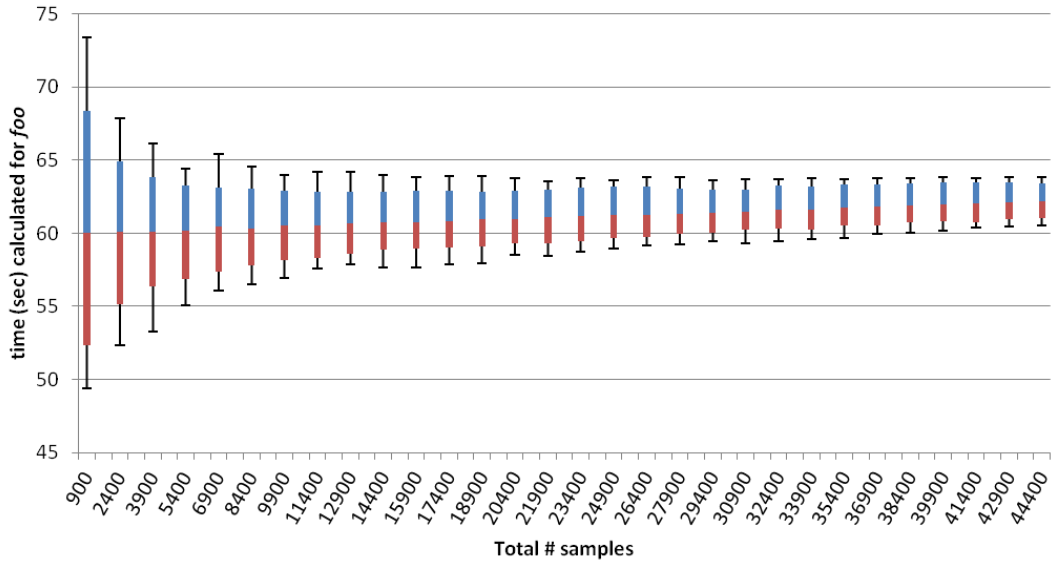

Figure 4.3: This graph presents the simulation results for measurements of the execution of *foo* within a simulated program that executes (when unperturbed) for a total of 300 seconds and in which *foo* accounts for 20% of the execution time. 1,000 simulations were run per each sample size with the middle 95% of measurements shown as the solid part of the bar. The whiskers above and below each bar indicate the remaining 5% (2.5% above and 2.5% below).

The results of our simulation compare very favorably with the predicted results of our analytic function from Fig. 4.2. As predicted, 95% of the simulation results generated with 2,400 samples fell between 55 and 65 seconds. As well, the lower curve of our prediction graph crossed over the 60 second line after the 28,000 sample mark which matches what we see with the simulation results. So, the simulation results indicate the model has validity.

Chapter 5: Sequential Execution

Given the intuition of our abstract model, the results from our analytic model, and the outcome of our simulation, we next progress to examining executing programs. For this part of our research, we conduct experiments with a simple calibration program with characteristics very similar to those used to evaluate the analytic model and conduct the simulation, then do extensive experiments with several programs from the SPEC CPU 2006 benchmark suite [29, 58]. Our model of the sampling-induced effects of perturbation on the execution of these sequential programs holds up reasonably well. Though unable to predict the best sampling period for every experiment, our model sometimes picks the best period, when not picking the best period, chooses something that close to the best period, and, importantly, avoids bad choices. Compared to the 28 sampling intervals used during our experiments, our technique guides us to better results than all but two intervals, and our results are equally good compared to these other two intervals.

5.1 Calibration Program

5.1.1 Experiment Design and Environment

To further validate our idea we designed an experiment where a program running for approximately 300 seconds with a function *foo* accounting for 20% of the execution time was periodically interrupted and the function name at the top of the call stack recorded. We ran the experiment 100 times for each of 23 different specific systematic sampling rates from 3 samples per second up to 166 samples per second. The results of those experiments are presented in Fig. 5.1.

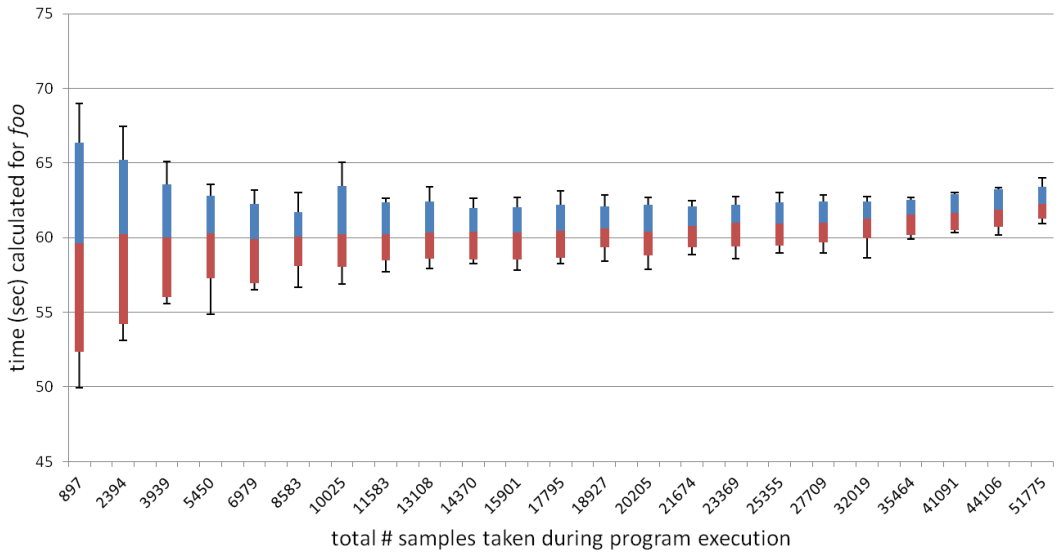

Figure 5.1: This graph presents the results for measurements of the execution of *foo* within our calibration program that executes (when unperturbed) for a total of approximately 300 seconds and in which *foo* accounts for approximately 20% of the execution time. 100 experiments were run per each sample size with the middle 95% of measurements shown as the solid part of the bar. The whiskers above and below each bar indicate the remaining 5% (2.5% above and below).

The sampling rates used for the experiments were chosen to match the resolution available with ITIMER_PROF, the software interval timer we used. We used rates of 3, 8, 13, 18, 23, 28, 33, 38, 43, 47, 52, 58, 62, 66, 71, 76, 83, 90, 100, 111, 125, 142, and 166 samples/second. Because the resolution of the timer used to generate interrupts is approximately 1 millisecond, the number of sampling rates possible are greatly reduced. Fig. 5.2 delineates the relationship between the target and actual sampling rates from 1 per second through 500 per second. Instead of the seemingly 500 possible sampling rates, only 61 unique sampling rates are actually possible when using the straightforward approach of dividing 1 second by the desired sampling rate and using the resultant fractional part of a second as the timer interval. The millisecond resolution of ITIMER_PROF creates a rounding effect that gets more pronounced the greater the number of samples we try to take.

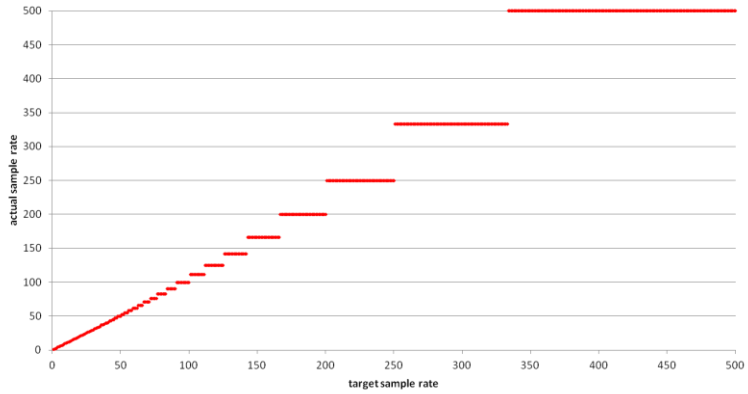

Figure 5.2: Sample rate limitations

Development of our sampling tool and execution of all experiments were done on a machine code-named Mashie. It is a quad duocore Intel Xeon 2.33GHz with 4GB of main memory. In runs the Linux operating system, version 2.6.9-89.0.25-ELsmp, Red Hat 3.4.6-11. The measured program was written in C and compiled with gcc 4.4.3 using compiler flags -Wall –g -fPIC -rdynamic. The first part of the measurement tool was written in C++ and compiled with g++ using compiler flags -Wall -g -fPIC and library flags -ldyninstAPI -lsymtabAPI –lcommon -liberty -lelf -ldwarf. The library part of the measurement tool was written in C and compiled with gcc using flags - shared -lm -lrt.

5.1.2 Program

The program we analyzed during our experiments was written to mimic as closely as possible the behavior of the program used during simulation. We used the same 1,000,000 integer array, shuffled it in the same manner, and seeded *rand()* with the same value. The array was then used to call functions in the same order as was done in the simulation. We created identical loops executing simple mathematical operations in each function in order to ensure each took as close as possible to 300 microseconds to execute and thereby provide a program that runs very nearly 300

seconds. Due to the imprecise nature of this technique the program ends up running approximately 300 seconds.

5.1.3 Measurement Tool

Our measurement tool works in the following way. We invoke our tool at the command line with three arguments: the length in seconds the analyzed program executes when unperturbed, the sampling rate desired, and the executable name of the program to be analyzed (plus whatever command line arguments are required by that program). The tool forks a process for the program to be measured and inserts our tool library into that address space. It then calls an initialization function from our tool library that establishes some data structures to collect performance analysis data, installs a signal handler, saves the start time, and sets the interval timer. The program to measure is then started and it executes in the normal way. Whenever the interval timer expires, a signal is raised that is dealt with by the installed signal handler. Within the signal handler the program's stack is accessed and the currently executing function name recorded. Upon program termination the contents of the performance analysis data structures are printed to a file and the tool then exits.

The measurement tool we constructed consists of two main parts. Both are compiled against Dyninst [11], a run-time code-patching library. The first part executes in relationship to the analyzed program much like a parent process. It is responsible for setting up the analyzed program's runtime environment, adding the tool library, calling the library initialization function, and then starting the program with its required arguments. The tool front-end waits until the measured program completes execution, and then calls the final library clean-up functions and exits. The

second part of our tool is the library that is added to the process space of the measured program and encapsulates the functionality for computing the sample period, handling signals, storing the performance data, and writing it to an output file.

5.1.4 Result Comparison

Our results from this experiment with the calibration program are presented in Fig. 5.3 alongside the results from our simulation and our model prediction from Chapter 4. The upper and lower curves of the confidence interval at the 95% confidence level from Fig. 4.2 are changed to the format of the results from Fig. 4.3 and Fig. 5.1 in order to make comparison easier.

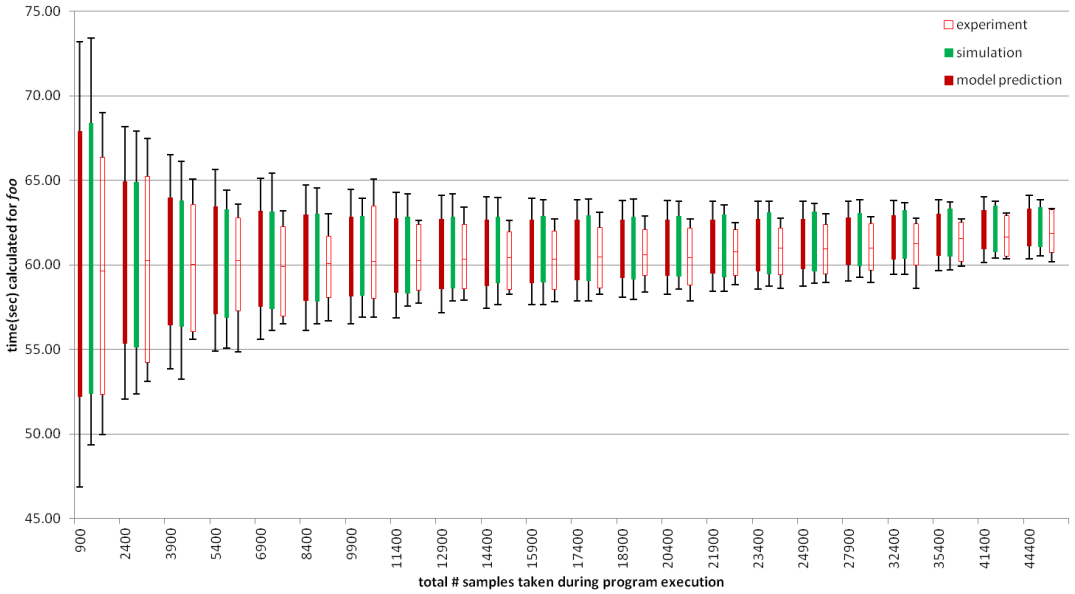

Figure 5.3: This graph presents a consolidated view of the results depicted in Figs 4.2, 4.3, and 5.1. Each three bar group pictured per sample count are, left to right, model prediction, simulation result, and experiment result.

The results are promising in that they show an outcome for the simulations and experiments that is generally consistent with that predicted by our analytic model. There is more variability in the results of the experiments than the simulations, which is expected partly due to the nature of running on a real system, but may also be

partly attributed to the order of magnitude difference in the number of runs conducted - 100 per experimental sampling rate vs. 1000 per simulated sampling rate.

5.2 SPEC CPU Programs

To further test our idea, we conducted extensive ITIMER-based sampling experiments with the five SPEC CPU 2006 benchmark programs: bzip2, mcf, milc, omnetpp, and sjeng. We will use omnetpp for the initial explanation of our experiments and for more in-depth analyses of the results because it is most close in runtime to our simulation and calibration programs.

5.2.1 Experiment Design and Environment

Running experiments with the SPEC CPU 2006 benchmark programs presented a new challenge to the method of analysis we had been using. When evaluating our analytic function and running simulations, we knew truth for *foo*. With the calibration program, we could carefully and directly affect truth for *foo* such that we had high confidence with the value we used. However, determining truth for the various functions in each SPEC CPU program was part of the very problem we are investigating. So, we decided on the following weak measurement method for determining the values for execution proportion of a given program's functions: execute a program multiple times and sample with a low perturbative sampling rate, then treat the median result for function execution time as "truth." Determining the number of times to execute a program in order to gather the samples necessary to calculate "truth" was somewhat arbitrary. For example, with omnetpp we determined that the function that took the most execution time, *cMessageHeap::shiftup(int)*,

accounted for roughly 15% of it. So, using p=.15, z=1.96 (for 95% confidence level), and r=.0025 we used Jain's equation (2.4) to determine 78,369 samples were required. We settled on a sample rate of two per second to provide balance between minimizing perturbation and keeping the overall experiment runtime to something not unduly burdensome. We then executed omnetpp 115 times accumulating a total of 79,021 samples (an average of 687 samples per run). The measured values of *cMessageHeap::shiftup(int)* across all runs were distributed as depicted in Figs. 5.4 and 5.5. Fig. 5.4 shows the distribution of the value of *p* for *cMessageHeap::shiftup(int)* by the number of samples collected on a per run basis. Fig. 5.5 shows the distribution of the value of *p* when sorted from the smallest to the largest calculated value.

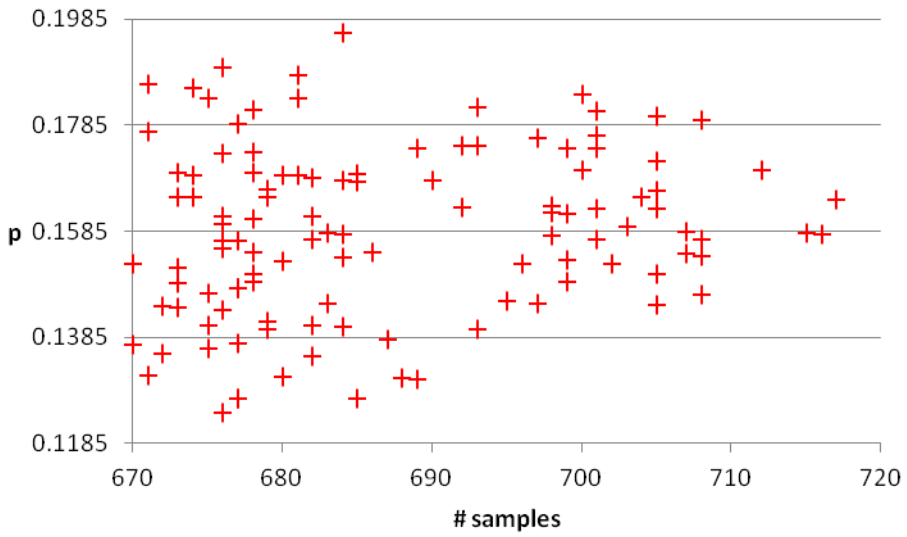

Figure 5.4: This graph presents the distribution of the calculated value of *p* for *cMessageHeap::shiftup(int)* arranged by number of samples taken.

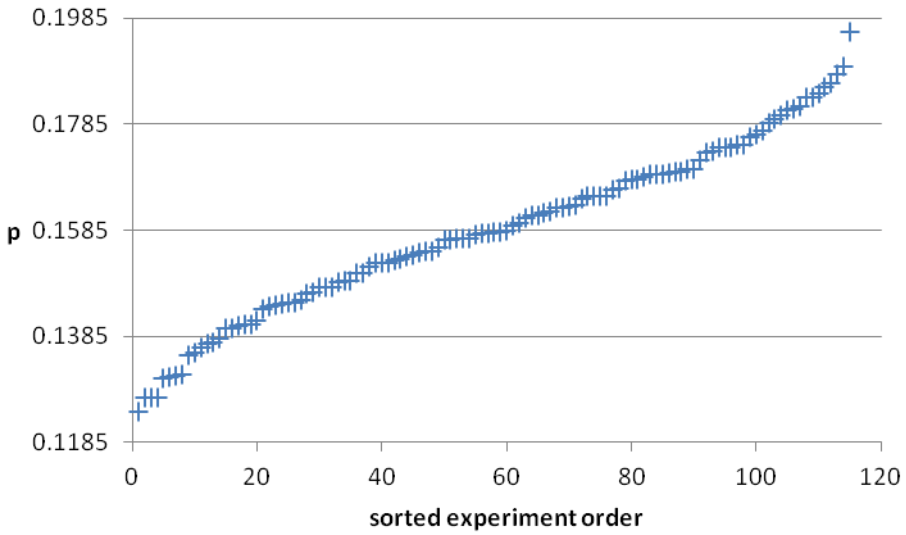

Figure 5.5: This graph depicts the distribution of the calculated value of *p* for *cMessageHeap::shiftup(int)* ordered from smallest to largest.

After inspecting the distributions of these values we reasoned that using the median value as truth, rather than the mean, was appropriate because the larger estimated values in the distribution likely reflected more perturbation error caused by transient system behavior and would therefore unnecessarily skew the mean. We thus determined the "true" value of each function in each of the SPEC CPU programs we used in a similar manner. The following table indicates for each of the SPEC CPU 2006 benchmarks analyzed, how many times we executed each while taking two samples/second; how many total samples were collected aggregated across all runs; the average number of samples collected per run; the function (*foo*) that accounted for the greatest percentage of execution time; and the percent of execution time (*p*) attributed to *foo*.

SPEC	# Runs	# Samples	Avg samples/run	Foo	p
bzip2	352	61,858	176	BZ2_blockSort	38.5%
mcf	99	77,596	784	primal_bea_mpp	43.6%
milc	95	87,537	921	mult_su3_na	18.0%
omnetpp	115	79,021	687	cMessageHeap:: shiftup(in)	15.8%
sjeng	76	79,236	1043	std_eval	15.3%

Table 5.1: Calculation of largest function per benchmark.

All experiments were done on a machine named Niblick. It is a quad duocore Intel Xeon 2.93GHz with 4GB of main memory. It runs the Linux operating system, version 2.6.9-89.0.25-ELsmp, Red Hat 3.4.6-11.

5.2.2 Programs

We chose this particular subset of SPEC programs for several reasons. First, we avoided using the benchmarks that were aggregations of shorter duration program runs scripted to run consecutively. Our concern with them was the potential introduction of additional overhead and the possibility of further complicating the profile results with an additional level of sample combining. We did make use of bzip2 which is provided by SPEC as an aggregation, but only ran it with the single input that caused it to execute the longest. Second, we focused on programs that had execution times nearest to our simulation and calibration run times of 300 seconds. Third, we had a slight personal preference for benchmarks written in C and C++ as compared to those written in Fortran.

5.2.3 Measurement Tool

Rather than use the simple measurement tool we created, we used Rice University's HPCToolkit [56], an integrated suite of tools used for measurement and analysis of program performance. This switch was motivated by the maturity of the

Rice tool suite and its acceptance in the systems research community as well as its additional functionality, like tracking calling context. The workflow is depicted in Fig. 5.6 which is copied from their site. We very slightly modified the *hpcrun* shell script utility to output the run time of each program. We created a small dynamically-loaded shared library so that *atexit()* the combined processor time charged for the execution of a program's instructions (tms_utime) and the processor time charged for execution by the system on behalf of the program (tms_stime) were output to a file. Otherwise we used the HPCToolkit components as provided.

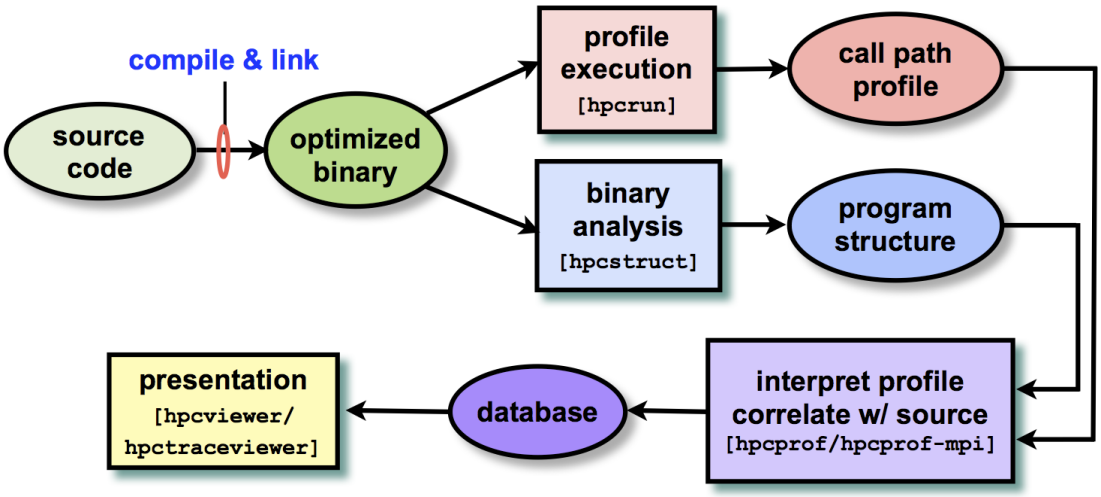

Figure 5.6: HPCToolkit component work flow.

We used the *hpcstruct* and *hpcprof* components of HPCToolkit as designed to associate calling context measurements with source code structure and to overlay call path profiles respectively. However, because we intended to analyze the measurement results in ways other than *hpcviewer* or *hpctraceviewer* are designed, we inserted our own data processing steps into the workflow. We wrote a parser to handle the XML file that results from running *hpcprof*, and then two programs to take the combined output from the parser program and output per function data for either execution percent or execution time.

5.2.4 Execution Details

When invoking hpcrun we made use of the wall clock timer functionality, e.g., `hpcrun -e WALLCLOCK@500000 omnetpp omnettpp.ini`, runs the SPEC CPU program omnetpp with input parameter omnettpp.ini and samples omnetpp every 500,000 microseconds.

In addition to collecting tens of thousands of samples at a sampling period of 500 milliseconds across tens of runs to establish "truth", we also conducted sampling runs with each SPEC program 10 times at each of 28 different sampling rates (200, 100, 67, 50, 33, 29, 25, 22, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, and 1 millisecond intervals) for a total of 280 runs per SPEC benchmark. The 280 runs were scripted to execute consecutively iterating over the 28 sampling rates 10 times in order to create some separation in time among each of the 10 runs of any particular sampling rate thus minimizing the impact of any temporary spike in system noise. At the end of the 28x10 experiment runs we parsed the output files, aggregated and sorted the results, analyzed them in a spreadsheet, and presented them in a graph.

5.2.5 Results

Though we make use of program and function time in our analytic equation and in our previous experiments, we will present the results in this section using percent of execution (*p*) because it makes the comparison to "truth" in the analysis section more meaningful. Sampling a running program results in us being able to calculate a statistic for the proportion (or percent) of execution time during which a given function was executing. If we convert that to time we would simply be multiplying the calculated proportion by the complete execution time of the sampled

program. As well, picking a best sampling rate when considering more than one function creates a situation where using time can cause an unbounded outlier based on time to disproportionately affect performance evaluation. Using proportion provides a degree of bounding that limits the effect of outliers.

Omnetpp

The next four figures display the results of the 28x10 experiments for each of the top four functions of the omnetpp benchmark. The center gridline in each chart is the proportion "truth" for that function as determined during our weak measurements run. The red dashed lines indicate the 95% confidence interval with respect to decreasing measurement error only; so, they are computed solely based on the proportion "truth" and the number of samples taken. The red and blue markers are intended to make it easier to differentiate among runs of adjacent sample intervals.

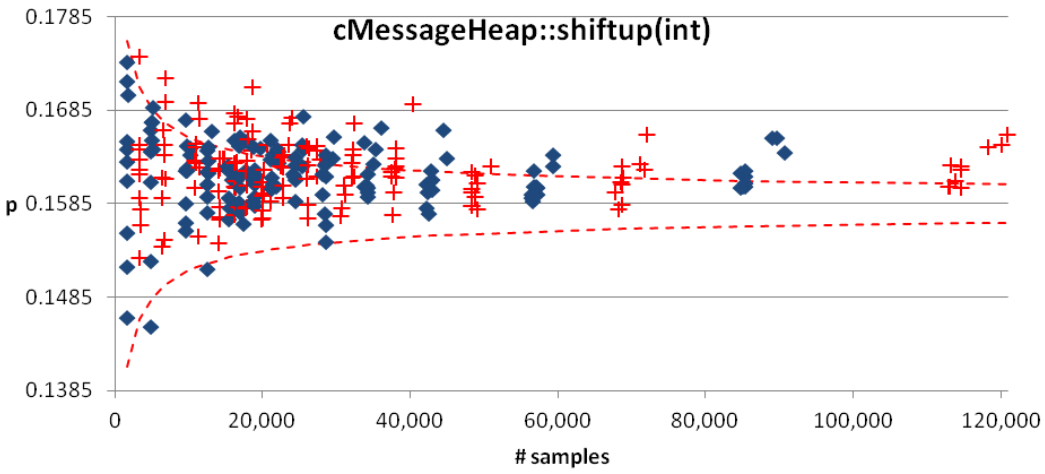

Figure 5.8: Plot of proportion calculation results for *cMessageHeap::shiftup(int)*, the function taking the most execution time for omnetpp, across the 28x10 execution runs.

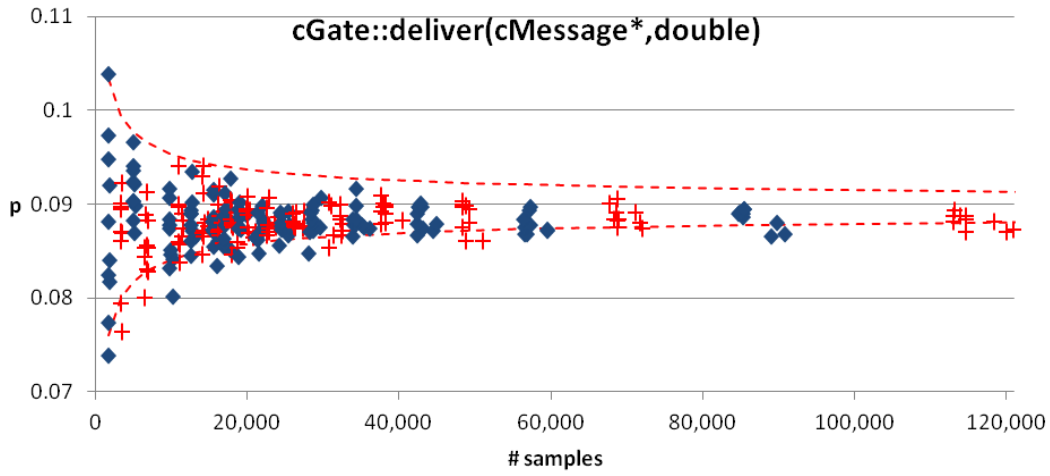

Figure 5.9: Plot of proportion calculation results for *cGate::deliver(cMessage*,double)*, the function taking the 2nd most execution time for omnetpp, across the 28x10 execution runs.

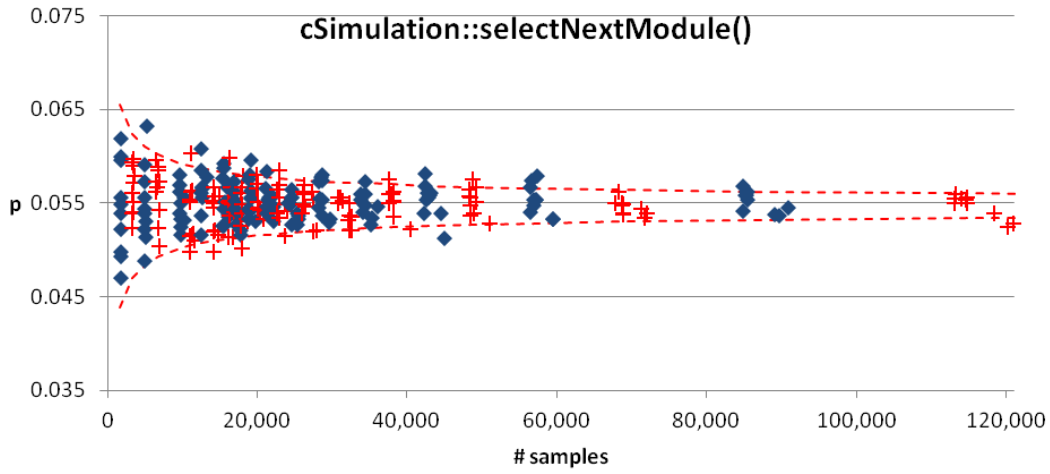

Figure 5.10: Plot of proportion calculation results for *cSimulation::selectNextModule()*, the function taking the 3rd most execution time for omnetpp, across the 28x10 execution runs.

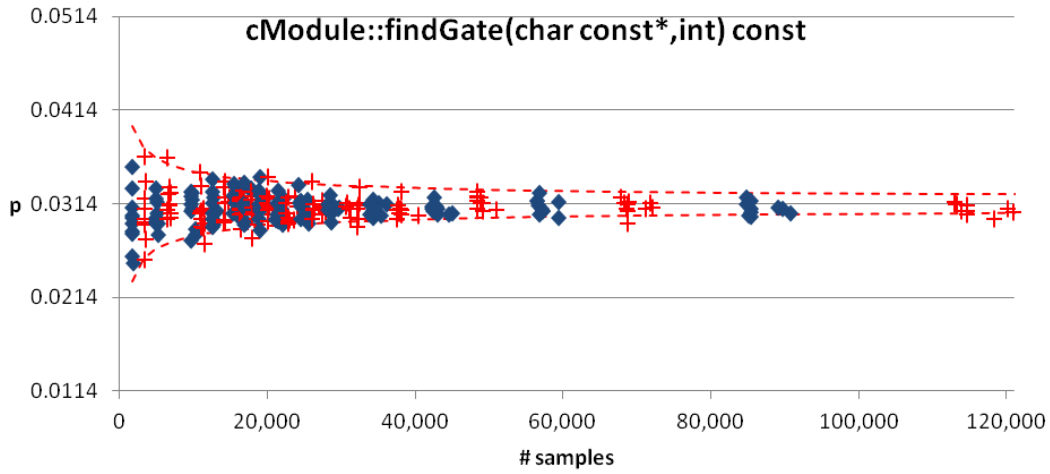

Figure 5.11: Plot of proportion calculation results for *cModule::findGate(char const*,int) const*, the function taking the 4th most execution time for omnetpp, across the 28x10 execution runs.

To provide a visual comparison between the execution proportion results and the execution time results and illustrate the reason for looking at proportion as opposed to time when evaluating the results for these experiments we've included in Fig. 5.12 one graph of our results with execution time on the y axis. The graph is of the function *cMessageHeap::shiftup(int)*, the omnetpp function that takes the most execution time.

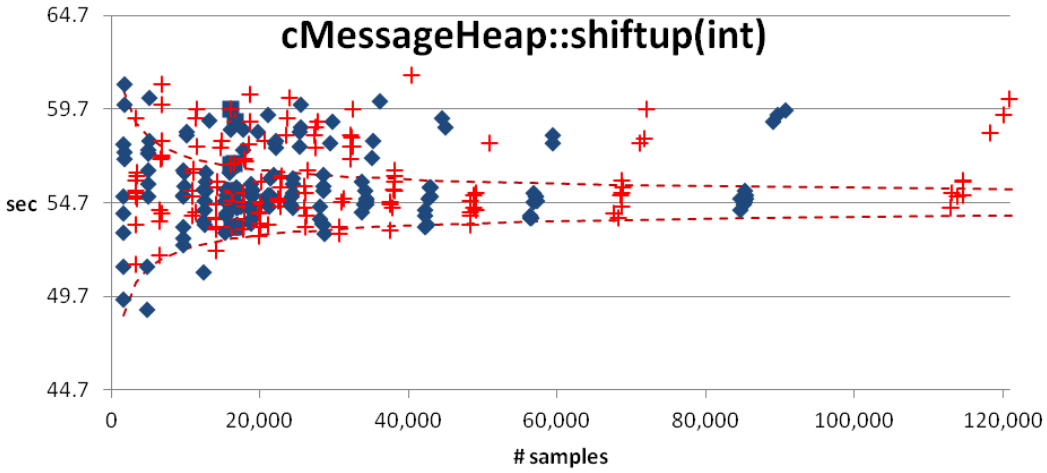

Figure 5.12: Plot of execution time calculation results for *cMessageHeap::shiftup(int)*, the function taking the most execution time for omnetpp, across the 28x10 execution runs.

To evaluate the accuracy of the results of the 10 runs at each of the 28 different sampling rates, we decided to calculate the mean absolute percentage error (MAPE) for each of the top four functions as compared to "truth," combine the four values (equation 5.1), and compare them for each of the 28 sampling rates.

$$\text{cMAPE} \qquad \sum_{f \in top\ 4\ functions} \frac{|p_t(f) - p(f)|}{p_t(f)} \qquad (5.1)$$

Whichever sampling rate produced the smallest combined MAPE (cMAPE) value was considered to be the best. The decision to use the top four functions as

opposed to some other number of functions or all of the functions was based on an inspection of the data generated from all the experiments run on the SPEC programs. A seemingly small difference from "truth" of a function that accounts for a small percentage of execution time overall can have a disproportionate impact on the aggregate MAPE value, so including all functions seemed unwise. It also seemed important to look at more than only the top function, so we decided, based on intuition and experience, upon what seemed a reasonable compromise. We noticed that for the five SPEC programs on which we conducted extensive experiments and detailed analysis, almost all of the top four functions of each program took up a proportion of execution time above 5%; an arbitrary threshold, but one that seemed likely to be of interest.

We determined the APE for each of the 10 runs at the 28 different sampling rates for the top four functions of each SPEC program. Figure 5.13 graphically presents the APE results for each of the 28x10 runs, the calculated MAPE (blue line), and the approximate expected MAPE (red dashed line) calculated statistically, for the top function of omnetpp, *cMessageHeap::shiftup(int)*. We calculated the approximate expected MAPE by taking a 95% confidence interval and dividing by 2. We consider it approximate because with a bell-shaped normal distribution, more measurements should be closer to "truth" than are far away, so dividing by 2 puts the red dashed line a little further from "truth" that it should be; however, we present it here only as a guide, not for measurement purposes, as the approximate nature of the calculation has no impact on our results.

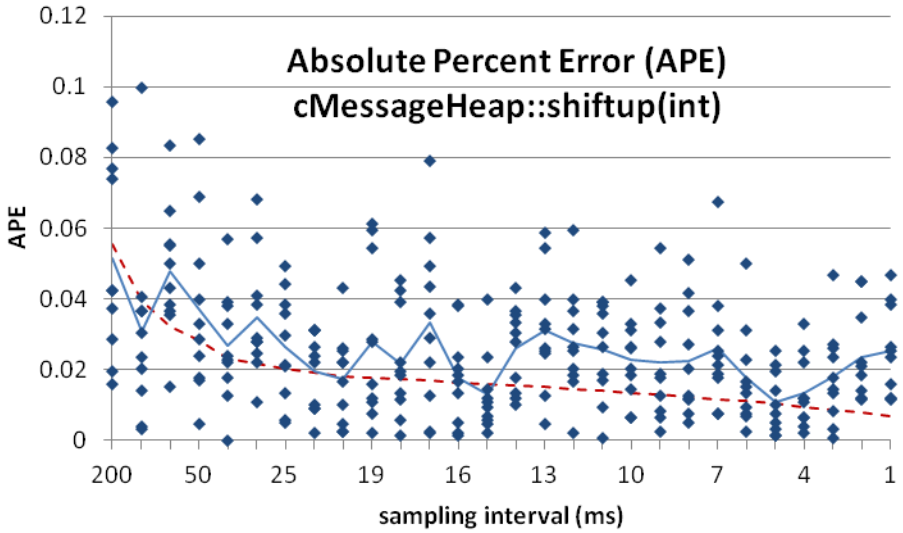

Figure 5.13: Results for the 28x10 runs compared to "truth" for *cMessageHeap::shiftup(int)*. The solid blue line tracks the MAPE for the set of 10 runs at each sample rate and the red dashed line indicates the approximate expected MAPE.

We then combined the MAPE values for the top four functions to determine the best overall sampling rate. Fig. 5.14 presents the MAPE for each of the top four functions in omnetpp while Fig. 5.15 presents the aggregated MAPE results from the same.

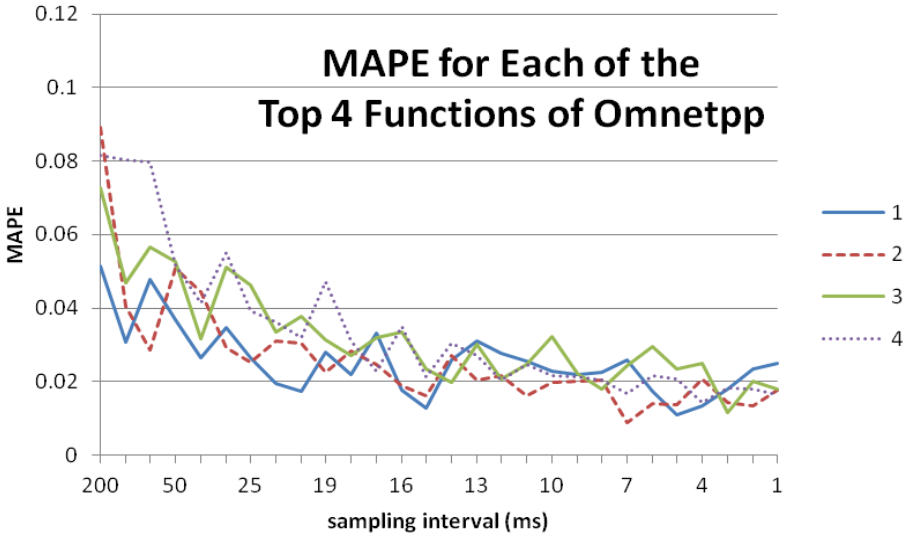

Figure 5.14: MAPE results for each of the top four functions in omnetpp.

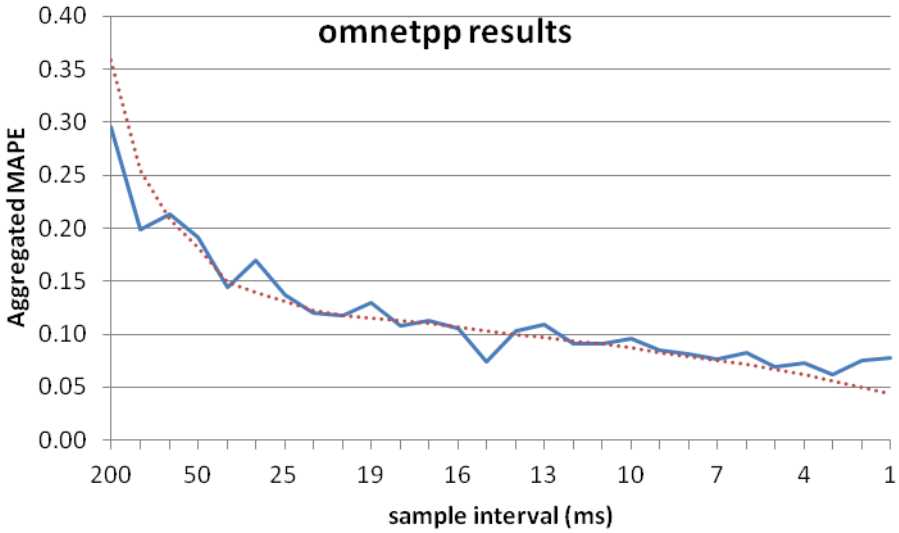

Figure 5.14: cMAPE results for the top four functions in omnetpp (blue solid line) and the calculated approximate expected cMAPE (dotted red line).

The sampling total that resulted in the overall best result for the omnetpp benchmark was the 48,829 samples taken using a 5 ms interval. Our technique (eqn. 4.16) predicted that the best results would occur with 91,830 samples:

$$n = \sqrt[3]{\left(\frac{345(1.96)\sqrt{.1585(1-.1585)}}{2(.000028).1585}\right)^2}$$

$$n = \left(\frac{247}{.000008876}\right)^{\frac{2}{3}} = 91{,}830$$

When used with the runtime of omnetpp (345 sec) to calculate sample rate, it predicts that 3.75 ms would be the best sampling interval:

$$\frac{345}{91{,}830} = .00375$$

This calculation brings to light another challenge when using software timer based sampling. In addition to the sample rate limitation illustrated in Fig. 5.2 which makes using a sample interval of 3.75 ms infeasible, we see here that the effective sampling outcome of the 5 ms sampling interval generated a total number of samples

that would have been expected with the larger interval of 7 ms. It varies somewhat from one program to another, but it is true in a general way that as the sampling interval gets smaller, the expected number of total samples generated is further from the actual number of total samples generated. For example, with a sampling interval of 100 ms we would expect to generate 3,450 samples, but on average generated 3,352, a difference of 98 or about 2.8%; at 10 ms expected is 34,500, actual was 28,665 on average for a difference of 5,835 or 16.9%; and at 1 ms expected is 345,000, actual was 115,623 on average for a difference of 229,377 or 66.5%. Thus, we must compare total samples as opposed to the calculated sample interval.

The following four graphs are of the aggregated MAPE results (cMAPE) for the other SPEC benchmarks we used. The graphs of the results of the 28x10 runs for the top four functions in each of these other benchmarks is included in Appendix B.

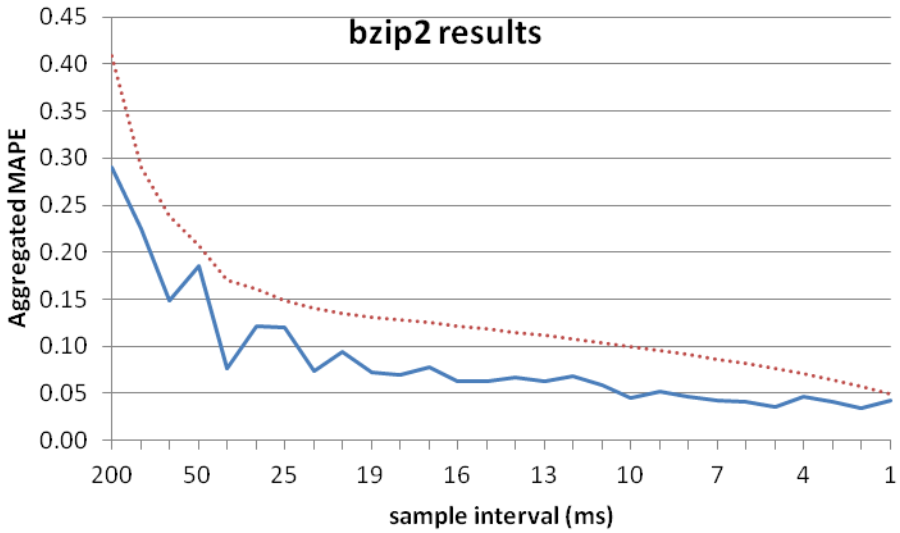

Figure 5.15: cMAPE results for the top four functions in bzip2 (blue solid line) and the calculated approximate expected cMAPE (dotted red line).

The sampling total that resulted in the overall best result for the bzip2 benchmark was the 21,837 samples taken using a 2 ms interval. Our technique predicted that the best results would occur with 21,418 samples.

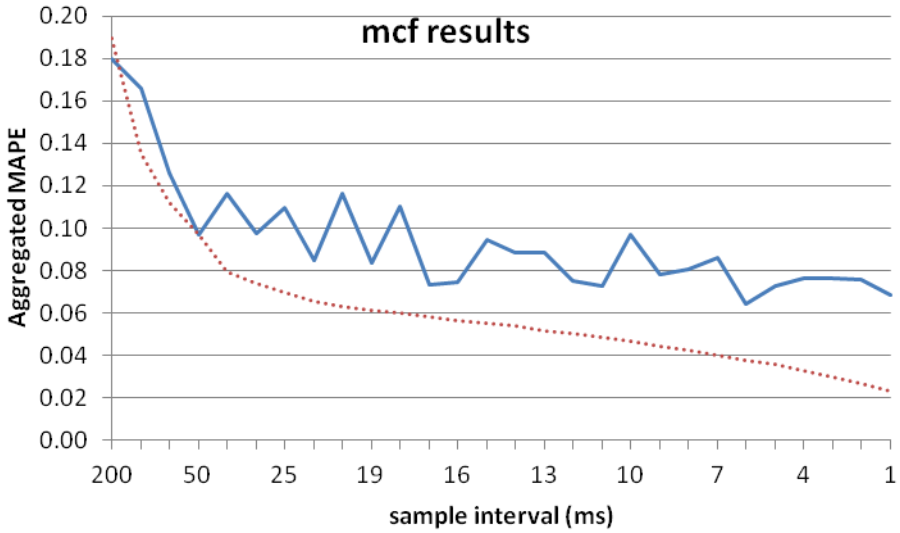

Figure 5.16: cMAPE results for the top four functions in mcf (blue solid line) and the calculated approximate expected cMAPE (dotted red line).

The sampling total that resulted in the overall best result for the mcf benchmark was the 49,106 samples taken using a 6 ms interval. Our technique predicted that the best results would occur with 38,472 samples.

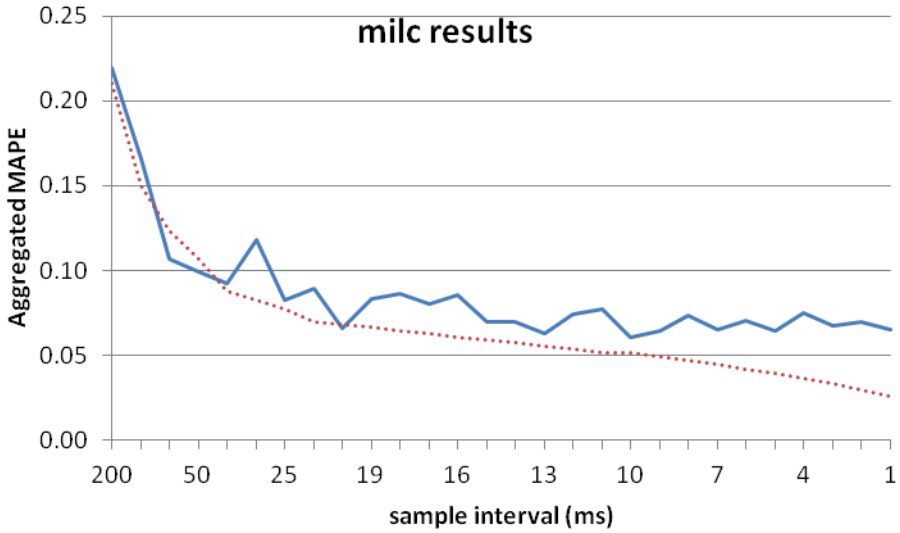

Figure 5.17: cMAPE results for the top four functions in milc (blue solid line) and the calculated approximate expected cMAPE (dotted red line).

The sampling total that resulted in the overall best result for the milc benchmark was the 36,622 samples taken using a 10 ms interval. Our technique predicted that the best results would occur with 60,555 samples.

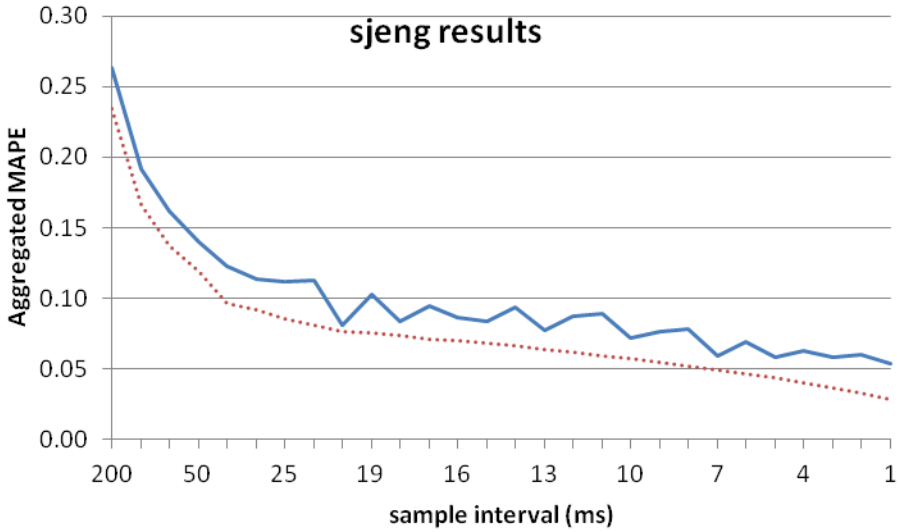

Figure 5.18: cMAPE results for the top four functions in sjeng (blue solid line) and the calculated approximate expected cMAPE (dotted red line).

The sampling total that resulted in the overall best result for the sjeng benchmark was the 175,387 samples taken using a 1 ms interval. Our technique

predicted that the best results would occur with 153,811 samples; this is nearly midway between the number of samples possible with a 2 ms sample rate and a 1 ms sample rate, but is slightly closer to the 1 ms rate.

5.2.6 Analysis of Analytic Equation Results

Our analytic equation guided us to the sampling interval with the best outcome twice out of the five experiments conducted. Table 5.2 presents the overall results. The calculated cMAPE value is shown to provide a sense of how our predicted best outcome compares to the measured best outcome as well as to the measured worst outcome. This aggregation of the mean absolute percentage error (MAPE) for each of the top four functions as compared to "truth" is calculated per equation 5.1.

	Predicted Best Outcome			Measured Best Outcome			Measured Worst Outcome		
	Sample Total	Nearest Interval	cMAPE	Sample Total	Interval	cMAPE	Sample Total	Interval	cMAPE
Omnetpp	91,830	2 ms	9.78%	48,829	5 ms	8.75%	1,665	200 ms	38.77%
Bzip2	21,418	2 ms	3.38%	21,837	2 ms	3.38%	431	200 ms	28.96%
Mcf	38,472	8 ms	8.04%	49,106	6 ms	6.40%	1,954	200 ms	11.29%
Milc	60,555	6 ms	7.02%	36,622	10 ms	6.05%	2,164	200 ms	18.78%
Sjeng	153,811	1 ms	5.37%	175,387	1 ms	5.37%	2,580	200 ms	19.00%

Table 5.2: Predicted vs. actual outcomes.

On the surface, this is a modest outcome; however, when compared to an experimental setup that chooses a single sampling interval to use across all five experiments (as is commonly done) our analytic equation does better than all tested intervals except two. And against those two intervals of 5 ms and 2 ms our equation is tied, 2-1-2 with 5 ms and 1-3-1 with 2 ms; meaning that twice 5 ms is a better choice than what our equation recommends, once they tie, and twice our equation

recommendation is better. As well, our equation did a good job at avoiding bad choices.

Chapter 6: Parallel Execution

In the next part of our investigation, we examine our idea for predicting the "sweet spot" sample count needed to balance the effects of decreasing measurement error and increasing perturbation error in the context of evaluating shared-memory parallel program executions. We set out to verify the applicability of our technique through extensive experiments with several programs from the SPEC OMP 2001 benchmark suite [59, 6] and we report that our model does reasonably well. Our model once again sometimes exactly predicts the best sampling rate, but is generally within close proximity to the best sampling rate, and avoids bad choices.

6.1 Preliminary Notes on Experiments

Running experiments with the SPEC OMP 2001 benchmark programs presented two new challenges over those noted earlier in Chapter 5. The first challenge was the difference in how much more the parallel program executions varied among runs compared to the variance of sequential executions. The second challenge encountered was the odd behavior of the software timer (ITIMER_PROF) used with the profiling tool, HPCToolkit, for performance analysis of shared-memory multiprocessing programs.

6.1.1 Variance of Execution

Program execution times have been studied and results have been reported that indicate non-trivial variations in program behavior are relatively common [64, 37, 54]. Somewhat surprisingly, it has been recently shown that in addition to execution variance rooted in the inherent non-determinism of computer systems,

measurement bias can also be a more common factor that anticipated [47]. In fact, Mytkowicz et al claim that measurement bias can manifest when relatively insignificant aspects of a computer system are changed. Their results indicate the bias can lead to performance analysis that over-states an effect or even leads to incorrect conclusions, and that the bias was evident across all architectures, both compilers, and most programs they tested. Further empirical study has shown that parallel execution times are even more variable than sequential execution times [44]. Mazouz et al specifically use experiments with and comparisons among the SPEC CPU 2006 benchmark suite and the SPEC OMP 2001 benchmark suite to demonstrate their findings.

Compared to the overall run time variance seen in our experiments with the SPEC CPU 2006 benchmarks used, we also saw a marked increase in the variance of the run times of the parallel benchmark executions. With all SPEC CPU 2006 benchmarks, the average standard deviation was 1.7% of execution time compared to 10.5% for SPEC OMP 2001 benchmarks. As an example, Fig. 6.1 shows the distribution of run times from some of our experiments on the SPEC OMP 2001 benchmark equake. It is easy to see the large degree of variance possible among each set of 20 runs and the greater variance exhibited by 12 core runs compared with 8 core and 4 core runs; as well as that exhibited by 8 core runs compared with 4 core runs. In comparison, Fig. 6.2 shows a similar sampling of runs of our experiments with the SPEC CPU 2006 benchmarks we used. Though variance is present, it is much less of a factor than with equake. NOTE: the scale of the y axis of Fig. 6.2 is set to match Fig. 6.1 so the comparison is more meaningful. For both figures, the

legend is at the top of the graph with different shaped/colored markers differentiating among the different number of cores (Fig. 6.1) or the different benchmarks (Fig. 6.2). The alternating yellow and white background is intended simply as an aid to make it easier to see which data points are associated with which sampling intervals.

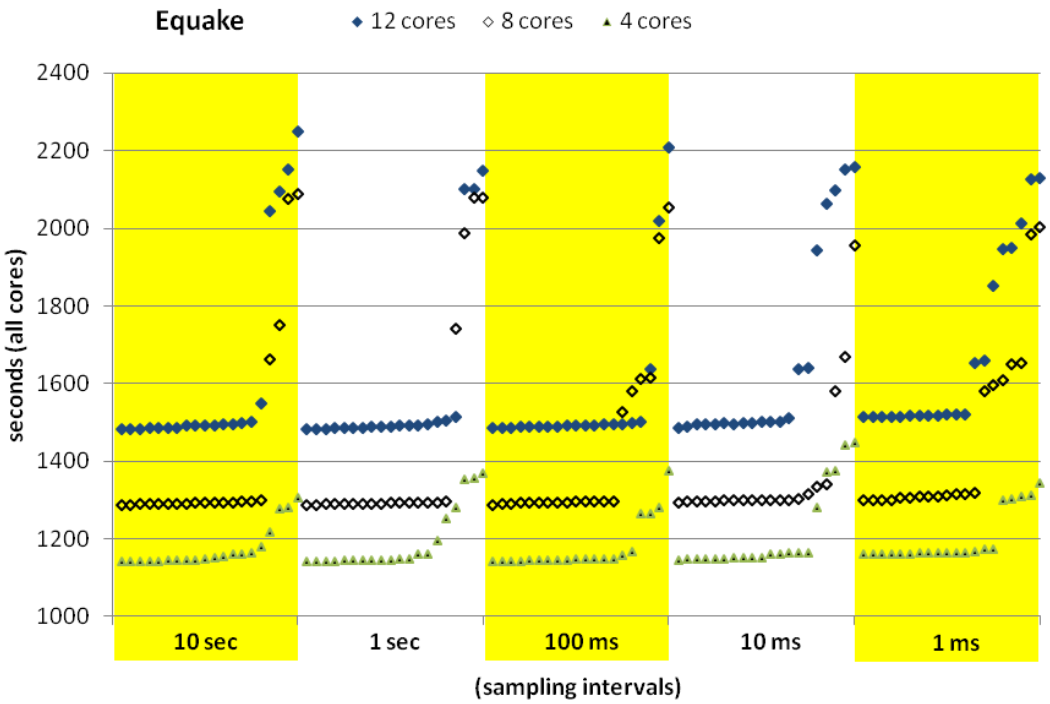

Figure 6.1: Distribution of run times, 20 each, sorted shortest to longest within each sampling interval, for the SPEC OMP 2001 benchmark equake.

Similar charts showing the variance of run times for the other SPEC OMP 2001 benchmarks we used are contained in Appendix C. The benchmark swim exhibited the most variance and is the only chart using a different y axis scale in order to ensure all runs can be displayed.

In light of the expected and observed increase in variance of run times for the parallel executions, we altered our standard experimental setup to include 20 iterations of each program execution for any given sample rate explored rather than 10 iterations as we did with the sequential program executions.

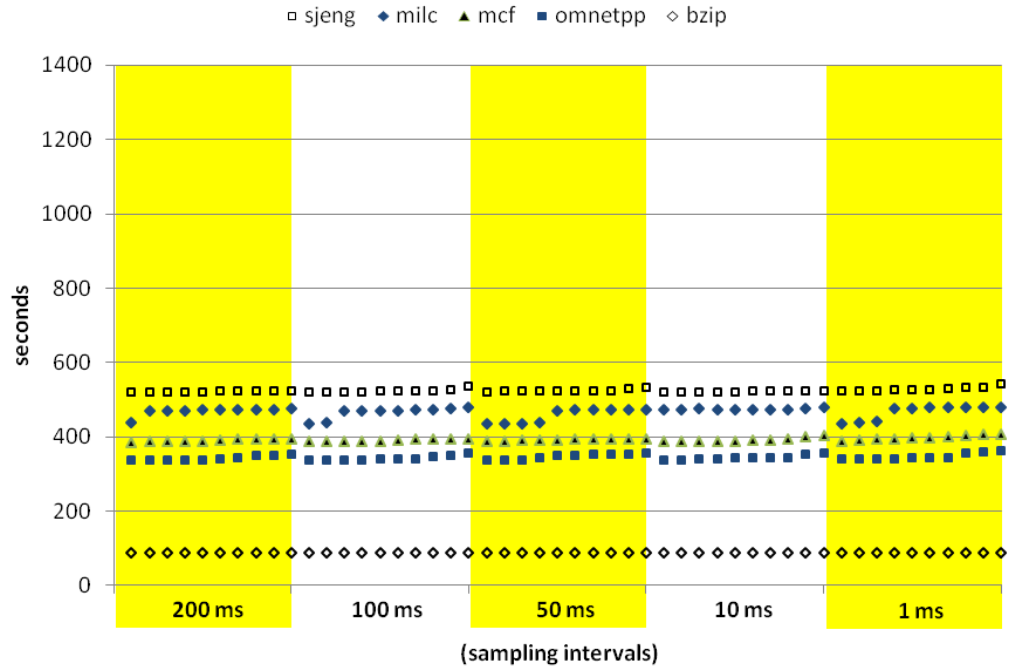

Figure 6.2: Distribution of run times, 10 each, sorted shortest to longest within each sampling interval, for the SPEC CPU 2006 benchmarks sjeng, milc, mcf, omnetpp, and bzip.

6.1.2 Software Timers

For experiments with sequential executions in Chapter 5, we made use of the HPCToolkit hpcrun call to WALLCLOCK which specifies ITIMER_PROF, indicating that the Linux profiling timer should be used for timing the sampling interval. We switched to using the HPCToolkit hpcrun call to REALTIME, specifying the CLOCK_REALTIME Linux timer, for our experiments evaluating shared-memory parallel program executions in light of very inconsistent results we were getting with ITIMER_PROF. It seems that a problem with the ITIMER_PROF interval timer is that it is not thread-specific and profiles only the main thread when profiling a multithreaded program, thereby producing incorrect and highly variable results.

Software sample-based profiling uses software clocks or timers to set the sample rate. Each process in Linux is provided with three interval timers: ITIMER_REAL, ITIMER_VIRTUAL, and ITIMER_PROF. ITIMER_REAL measures wall-clock time, decrements at all times regardless of whether the profiled program is executing, and delivers SIGALRM upon expiration. ITIMER_VIRTUAL measures process time, decrements only when the profiled process is executing, and delivers SIGVTALRM upon expiration. ITIMER_PROF is intended for use with profilers, decrements both when the profiled process is executing as well as when the operating system is specifically executing on behalf of the profiled process, and delivers SIGPROF upon expiration.

The ITIMER_PROF is the preferred of the three available process-specific software interval timers because it should provide the most accurate accounting of the profiled process. We experienced difficulty achieving consistent results during our initial profiling experiments with the SPEC OMP 2001 benchmarks and were initially very puzzled by the results. Upon investigation into the cause of the odd results, we discovered that the HPCToolkit authors also identified this clock problem and in their latest user manual [45] recommend using the hpcrun call to REALTIME, which is based on the CLOCK_REALTIME Linux timer that counts wall clock time. They also recommend trying the hpcrun call to CPUTIME, which is based on the CLOCK_THREAD_CPUTIME_ID Linux timer, but we found some benchmarks tended to hang when using this timer. In light of the inconsistent results we were getting with the ITIMER_PROF interval timer and our subsequent discovery of the updated advice in the HPCToolkit user manual, we switched to using REALTIME in

all of our SPEC OMP 2001 benchmark experiments. All results presented in this chapter use calls to REALTIME.

6.2 SPEC OMP 2001 Programs

We chose to evaluate five of the SPEC OMP 2001 benchmark suites in our experiments: applu, equake, fma3d, swim, and wupwise. These particular five benchmarks were chosen for a combination of reasons – we experienced no difficulty or errors when building them and they had run times that were of reasonable length, and thus manageable, for the number of experiments we wanted to conduct We choose to exclude apsi, mgrid, galgel, and gafort because of compiling difficulties and some early error-terminating run results; art because it contained a single function that dominated overall execution time with >95%; and ammp due to its significantly longer runtime than all other benchmarks.

6.2.1 Experiment Design and Environment

Execution of all experiments was done on a machine code-named Pygmy. During experiment runs the machine was dedicated to a single user. Pygmy is a 12 core, Intel Xeon running at 2.53 GHz. It runs Ubuntu version 12.04.4 LTS with the Linux kernel 3.2. The programs we evaluated were each executed using 4, 8, and 12 cores of Pygmy. Since we compiled the benchmarks with GNU's gcc, we had the option of specifying during runtime the environment variable GOMP_CPU_AFFINITY in order to bind the OpenMP threads to specific processing units. For every run using 4 cores we specified GOMP_CPU_AFFINITY=8-11;

using 8 cores we specified GOMP_CPU_AFFINITY=4-11; and using 12 cores we specified GOMP_CPU_AFFINITY=0-11.

6.2.3 Programs

Of the SPEC OMP 2001 benchmarks evaluated, equake is written in C and the others are written in Fortran. We used the 'M' input set for all the benchmarks which is designed to be used on shared-memory systems with between 4 and 32 processors (as opposed to the 'L' input set which can be used with up to 512 processors) [58]. We used the reference input data set (as opposed to the test or train data sets) for all experiments. The benchmarks were compiled with gcc 4.5.2 within the build system provided by SPEC OMP, though all benchmark runs were conducted outside of the SPEC scripting system.

6.2.4 Execution Details

For each experiment run, we used our analytic function (4.16) to predict where we expected to find the sampling "sweet spot" given the experiment parameters. Then, because of the relatively long experiment runtimes involved with executing each benchmark multiple times at each sampling interval for each of the three thread-core settings of interest (4, 8, and 12), we decided to bracket the results with sampling intervals differing by an order of magnitude, compare them to our prediction in a qualitative manner, and then allow those results to guide more focused investigation. For each benchmark at each of the three designated thread-core counts, we started by conducting 20 runs at each of 5 different sampling intervals: 10,000 ms (10 sec), 1000 ms (1 sec), 100 ms, 10 ms, and 1 ms. Once that was done, we conducted comparative qualitative analysis between graphs we created of function

proportions and our sampling equation prediction. Based on the outcome from that analysis, we determined which broad interval was of the most interest, conducted 20 more runs at each of 6 new different sampling intervals, and then carried out more in-depth quantitative analysis between the proportion results at our predicted value and the results at the new sampling intervals.

6.2.5 Results

In this section we present some results from experiments with the SPEC OMP 2001 benchmark equake executing with 4 threads on 4 cores to demonstrate how we arrived at our findings. When we use our analytic function (4.16) for predicting the number of samples we should take during an execution, using preliminary data for smvp.omp_fn.5 and equake run with 4 threads on 4 cores, we get the following predicted sample count:

$$n = \sqrt[3]{\left(\frac{310(1.96)\sqrt{.32(1-.32)}}{2(.00002).32}\right)^2}$$

$$n = \left(\frac{283.4}{.0000128}\right)^{\frac{2}{3}} = 78{,}854$$

Taking samples at the 5 order-of-magnitude differing intervals resulted in median sample counts of 116; 1171; 11,737; 117,992; and 1,179,613 respectively across the 5 sets of 20 runs. In Fig. 6.3 are the results from these bracketing runs of the calculated percentage of execution for the function taking the most execution time, smvp.omp_fn.5. The data within each set of 20 runs are sorted by run length,

shortest to longest, left to right. The results in Fig. 6.4 are the calculated percentage execution values for the top four functions in equake presented the same way.

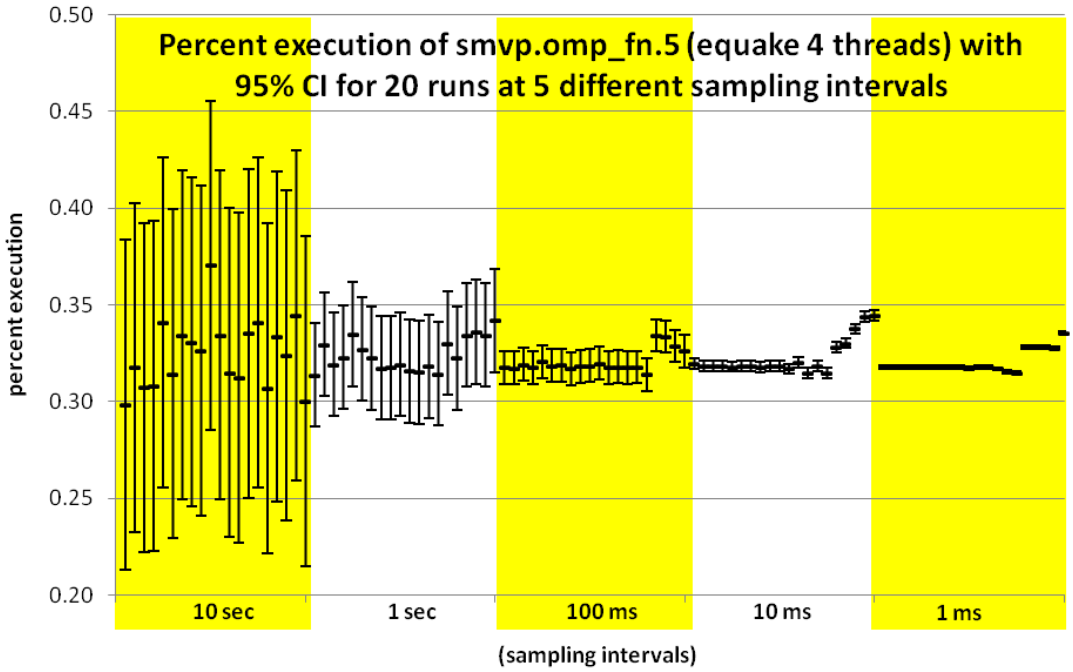

Figure 6.3: Distribution of the percent execution taken by smvp.omp_fn.5 from 100 runs of equake with 4 threads on 4 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

With qualitative analysis of the two charts, we note that the 10 sec sample interval results in calculated proportions that are quite variable and have 95% confidence intervals that are far wider than desired. As can be seen in Fig. 6.4, for almost all runs of GOMP_taskwait the CIs overlap with those of both smvp.omp_fn.5 and main.omp_fn.10, so statistically the correct order of most computationally expensive to least cannot be determined. As confidence intervals indicate precision, rather than accuracy, the 1 ms sample interval provides an outcome that is not particularly useful. About half of the runs have a very tight grouping among the calculated proportions, but the 95% confidence intervals are so narrow that the slightest variation indicates there are statistically significant differences among many

of the runs. Somewhere between the 100 ms sampling interval and the 10 ms sampling interval seems to provide the best balance between narrowing the confidence interval and getting consistent results. Our prediction falls between the median counts for the 100 ms sampling interval (11,737 samples) and the 10 ms sampling interval (117,992 samples), so we examined this interval more closely.

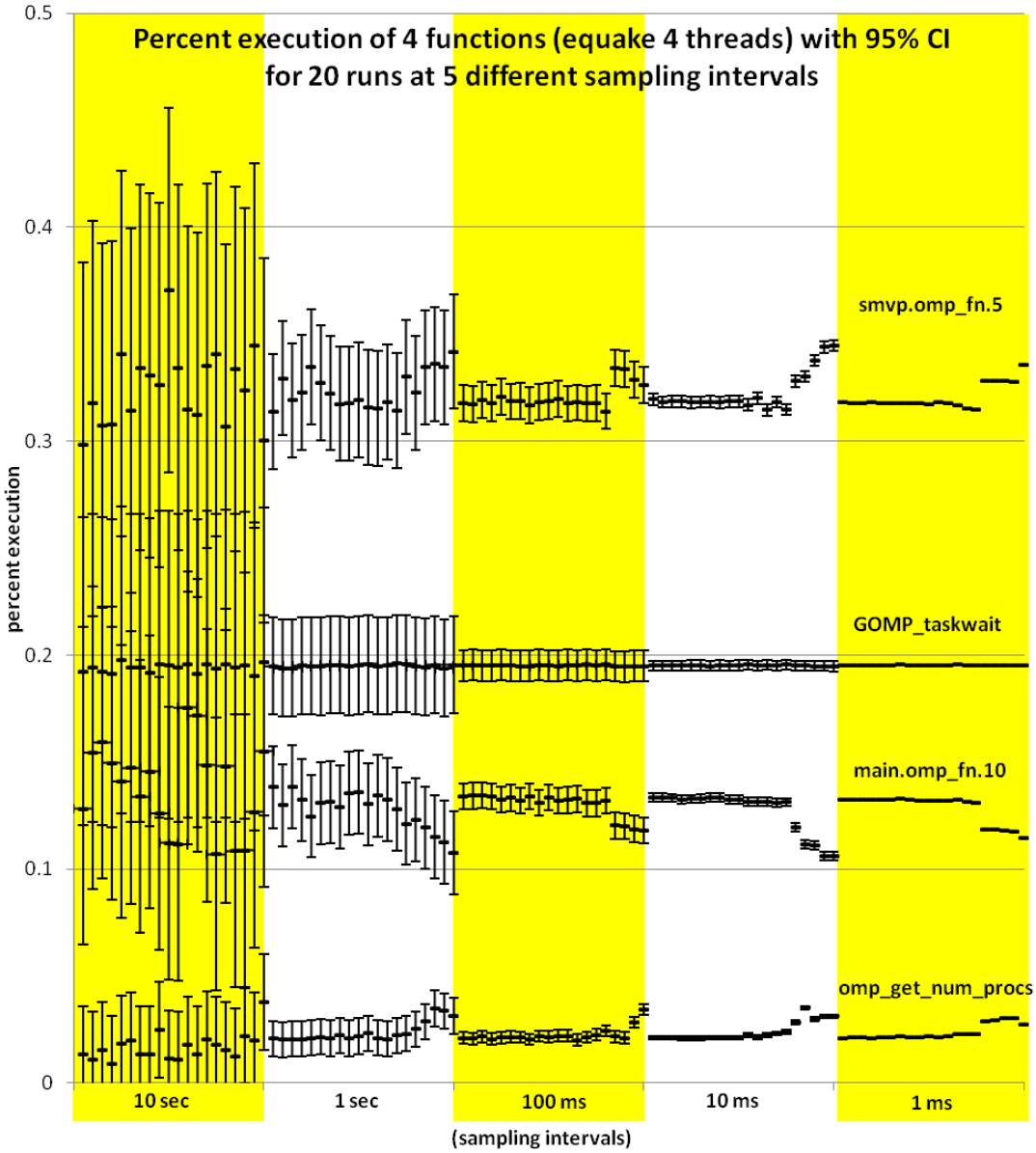

Figure 6.4: Distribution of the percent execution taken by 4 different functions from 100 runs of equake with 4 threads on 4 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

Note that our use of function 4.16 treats the parallel shared-memory program wall-clock time in the same manner as for a sequential program. We make no modification to the equation based on the number of threads or number of cores being used. At first glance, this may seem odd since the total execution time (the aggregate of core execution times across all cores) seems analogous to the wall-clock time for a sequential program. However, when we explored that technique we found that sticking with the wall-clock time provides better results. Using the longer total execution time resulted in taking more samples which unsurprisingly turned out to be more disruptive to parallel programs and provided poorer results.

To evaluate the effectiveness of our prediction, we used a technique similar to the broad bracketing used earlier. We calculated six sampling intervals at which we could collect sampling counts that would more narrowly bracket the desired sampling count of 78,854. In addition to running an experiment set at the predicted "sweet spot," we used a sample count interval of 20,000 (for predicted sample counts >=100,000 we used an interval of 30,000) to calculate the timing intervals required to collect 3 intervals below the predicted count and 2 above. This technique does not exhaustively test all sampling intervals around our prediction, but it offers the benefit of a check on the qualitative analysis of the results from the large bracket runs while also providing an effective trade-off between result granularity and the resource burden of intensive experimentation. The six sampling counts and the respective sampling intervals used for the smaller bracket experiments with equake, 4 threads on 4 cores, are in Table 6.1.

Expected approximate sample counts						
Interval (ms)	63	30	20	15	12	10
Expected sample count	18,000	38,000	58,000	78,000	98,000	118,000

Table 6.1: Sampling intervals and expected approximate sample counts for targeted experiments with equake using 4 threads on 4 cores.

The next two charts present the results of the calculated percentage execution of smvp.omp_fn.5 for the 120 runs conducted with equake, 4 threads on 4 cores, 20 runs per sampling interval. Fig. 6.5 displays the experiment runs grouped by sampling interval with 95% confidence intervals for the expected "true" value depicted as the red dashed lines.

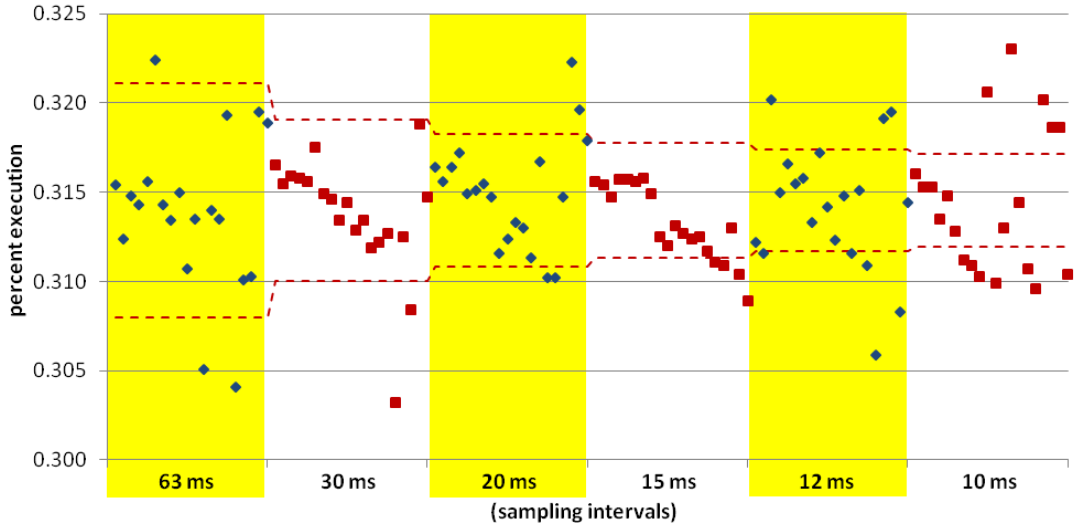

Figure 6.5: Results of the percent execution of smvp.omp_fn.5 for the 120 runs conducted with equake, 4 threads on 4 cores, 20 runs per sampling interval. The 95% confidence intervals of the "true" value are depicted as red dashed lines.

Fig. 6.6 displays the same experiment runs using sample count as the x-axis – in this format, runs that encountered some kind of non-deterministic system event that caused deviation from "normal" execution stand out as having run noticeably longer and consequently collected many more samples than peer runs within a particular sample interval cohort. In both graphs, red and blue markers are simply intended to more easily distinguish the runs of adjacent sampling intervals.

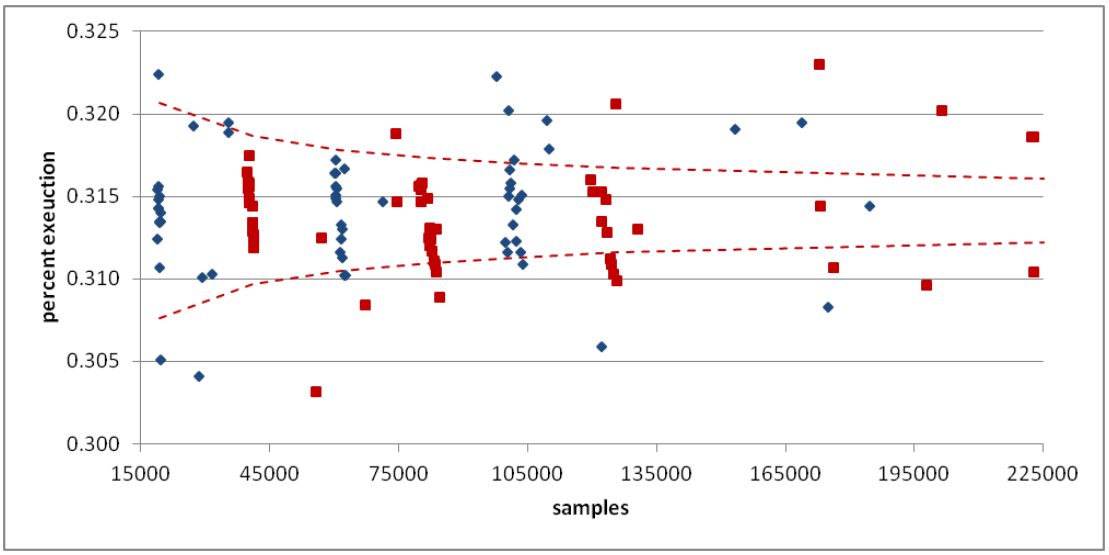

Figure 6.6: Results of the percent execution of smvp.omp_fn.5 for the 120 runs conducted with equake, 4 threads on 4 cores, 20 runs per sampling interval. The 95% confidence intervals of the "true" value are depicted as red dashed lines.

Additional charts showing the experimental results for the other three functions we looked at for equake, 4 threads on 4 cores, in addition to all charts for the other benchmarks we studied are in Appendix C.

Statistical science indicates to us that with 95% confidence intervals around what we believe to be "truth" for a given experiment, we could expect 19 of 20 runs to fall w/in the interval. Considering the number of runs that fall outside the 95% CIs provides us a granular indication of the effect of sampling perturbation on measurement results. As can be seen in both figures above, more than 1 experiment per group of 20 appears to fall outside the interval, with the incidence of variation increasing as more samples are taken. The table below provides the count of runs per sampling interval that are outside the confidence interval.

Number of runs outside CI						
Interval (ms)	63	30	20	15	12	10
# runs outside CI	4	3	3	3	6	11

Table 6.2: Sampling intervals and corresponding number of runs (out of 20) with calculated proportion (percent execution) outside the 95% CI for the "true" value of smvp.omp_fn.5 from equake, 4 threads on 4 cores.

It's important to note at this point that determining the "truth" to compare with the calculated proportions for each of our experiments, already a difficult process in Chapter 5, is more difficult in light of the greater variability of parallel program execution. Given this greater difficulty, we tried to ensure greater rigor in this process by applying a chi-square test of homogeneity to the calculated proportions from the many different runs. We used the chi-square test of homogeneity to determine to what degree profiles from the various executions of the same experimental context (e.g., equake, 4 threads on 4 cores) differed. We used the results from the chi-square test to determine which runs from our initial bracketing runs could be combined to create a "true" proportion value with high confidence. We then did the same for the subsequent executions done with the more targeted brackets. After examination of the experimental results, we proceeded in our chi-square test by first calculating the chi-square test-statistic with all data. If the result was statistically significant (at a significance level of .05) to reject the hypothesis of homogeneity, we sorted all experiments by total runtime within their sample-interval cohorts and then re-calculated the chi-square test-statistic using the 10 shortest running experiments per cohort. Our rationale is that the longer-running experiments are more likely perturbed by a rare, but higher impact event than the shorter-running experiments. If we did not find statistically significant evidence for homogeneity with the 10 shortest runs, we then re-calculated with the 5 shortest running experiments per cohort. The results of all the chi-square tests for equake, 4 cores on 4 cores, are in Table 6.3 below.

Chi-Square Test Results						
Function	**Brackets**	**X^2**	**DF**	**P-value**	**$\hat{p}$**	**95% CI**
smvp.omp_fn.5	Large	11.04	49	1.0000	.3179	(.3176, .3181)
	Small	36.1	59	.9919	.3147	(.3143, .3151)
GOMP_taskwait	Large	4.1	99	1.0000	.1953	(.1951, .1954)
	Small	1.3	59	1.0000	.1957	(.1953, .1961)
main.omp_fn.10	Large	16.4	49	1.0000	.1324	(.1322, .1326)
	Small	26.9	59	.9999	.1309	(.1306, .1312)
omp_get_num_procs	Large	5.7	39	1.0000	.0210	(.0208, .0213)
	Small	19.5	29	.9065	.0228	(.0226, .0230)

Table 6.3: Chi-square test results for determining "truth" for the percent execution of equake functions.

Except for GOMP_taskwait, the results of "truth" for percent of execution for the aggregation of the initial bracketing runs don't match "truth" for the second set of more tightly bracketed runs despite both sets having "passed" chi-square tests for homogeneity among their own bracketed runs. For example, smvp.omp_fn.5 has a "true" value of p=.3179 with 95% confidence interval (.3176, .3181) after the initial large bracketing runs using order-of-magnitude difference sampling intervals and a "true" value of p=.3147 with 95% CI (.3143, .3151) after the small bracketing runs around our predicted best sampling count. These results, not entirely unexpected, illustrate the problem of large numbers of samples leading to small CIs and further back-up the parallel run variation problem noted earlier in this chapter and reported by others [47, 44].

We needed to choose one of the values as "truth" to compare performance analysis results quantitatively, and we decided upon using the "truth" derived with the set of runs to which we were comparing. Our rationale is rooted in the observation that both "true" values for each function differ only after the third significant digit, so the difference is relatively small; and is based on the points made earlier about inherent execution variability and both data sets passing the chi-square test for homogeneity.

Given our choice of "truth" we evaluated the second set of smaller bracketing runs for equake, 4 threads on 4 cores, once again making use of equation 5.1 for cMAPE, and determined that the best sampling interval was, in fact, the one our model predicted. We then did the same analysis for all experiments. The results for the analysis of all parallel shared-memory experiments are in Table 6.4 below. The calculated cMAPE value is shown to provide a sense of how our predicted best outcome compares to the measured best outcome as well as to the measured worst outcome.

	Predicted Best Outcome			Measured Best Outcome			Measured Worst Outcome		
	Sample Total	Nearest Interval	cMAPE	Sample Total	Interval	cMAPE	Sample Total	Interval	cMAPE
	Equake								
4-core	78,854	15 ms	17.87%	80,120	15 ms	17.87%	116	10 s	99.37%
8-core	53,450	24 ms	17.07%	13,603	100 ms	7.03%	129	10 s	171.75%
12-core	48,114	31 ms	56.00%	36,922	45 ms	27.27%	156	10 s	188.53%
	Applu								
4-core	174,420	14 ms	7.84%	230,588	12 ms	7.05%	240	10 s	85.43%
8-core	126,297	21 ms	9.67%	207,060	17 ms	7.56%	256	10 s	96.41%
12-core	81,319	37 ms	2.42%	82,764	37 ms	2.42%	288	10 s	152.30%
	Fma3d								
4-core	185,236	15 ms	3.33%	237,635	13 ms	1.46%	276	10 s	27.52%
8-core	138,683	22 ms	2.64%	149,203	28 ms	1.61%	304	10 s	35.82%
12-core	96,618	36 ms	1.91%	159,281	23 ms	1.20%	348	10 s	46.63%
	Swim								
4-core	131,209	17 ms	144.54%	102,292	23 ms	80.74%	220	10 s	214.14%
8-core	76,634	31 ms	82.24%	72,584	52 ms	80.05%	232	10 s	249.07%
12-core	60,505	45 ms	118.92%	145,893	27 ms	65.70%	2,724	1 s	182.09%
	Wupwise								
4-core	188,336	18 ms	2.71%	159,761	21 ms	2.58%	336	10 s	18.08%
8-core	159,459	23 ms	5.48%	170,625	23 ms	5.48%	368	10 s	37.70%
12-core	99,049	39 ms	2.69%	192,689	20 ms	0.73%	384	10 s	52.46%

Table 6.4: Parallel predicted vs. best outcome.

The results in Table 6.4 show that our model once again does moderately well. In addition to avoiding bad choices for the sampling interval, our technique led to the selection of 3 optimal predictions and 10 total predictions within 30% of the

best sample count of the targeted bracket. With all n-core experiments of applu, fma3d, and wupwise, our predicted best outcomes compare well with all measured best outcomes. For equake the comparison is somewhat mixed while the results for swim, the most memory-intensive of the OMP benchmarks, are not nearly as good. Both the measured best outcome and our predicted best outcome results for swim are noticeably worse than the results for the other experiments. This is caused by significant cache thrashing of the large arrays swim uses during its shallow water modeling being further exacerbated by the sampling process.

When compared to an experimental setup that chooses a single sampling interval to use across all 15 experiments (as is commonly done) our analytic equation does better than all tested intervals except one and it does at least as equally well as that interval. Thus, using adaptive instrumentation based on our analytic function provides more accurate results than picking a single sampling interval.

Our analytic equation predicts the optimal sample count in our shared-memory parallel execution experiments with the same efficacy as in our sequential execution experiments. We demonstrated that the number of samples required is proportional to time of execution regardless of number of cores used. Predicting the best sample count with wall clock time turns out to provide several benefits. It mitigates the increased perturbation potential observed with larger numbers of cores when taking very large numbers of samples. And, it moderates the decreasing utility value of each successive sample as the sample size gets very large helping avoid the misleadingly small confidence intervals that very large sample sizes imply.

Chapter 7: Future Work

In this chapter, we describe future work motivated by the research results presented in this dissertation.

7.1 Process Refinement

Our current technique for predicting the optimal sample interval for minimizing overall error relies on initial experiments to determine a starting value for p and a value for o (overhead). There are many ways that we could make the start up process less cumbersome and costly. Statically determining these values or values "close enough" would be ideal, but even selectively starting and prematurely ending runs could save time and resources with programs of moderate length and would likely be necessary for programs long enough that experimenters do not have the luxury of running them multiple times.

7.2 Analytic Function Refinement

An attractive aspect of the statistical science technique for computing confidence intervals, given that certain conditions hold, is the reliance on the number of samples collected regardless of many underlying aspects of the population under investigation. Investigating the feasibility of evolving our analytic function's measure of perturbation, to something more like the statistical science community's function for decreasing measurement error, based on the number of samples taken would greatly improve its usefulness. Even some intermediate improvement of the development of different functions for programs with generally equivalent levels of

perturbation sensitivity would be a big improvement and something we intend to delve into.

7.3 Other Application Domains

We studied portions of the class of programs related to sequential execution and shared-memory parallel executions for 12 cores and below. There are more classes of programs where we believe our technique or something very similar can be used to improve performance measurement. Exploring programs running with much greater parallelism, especially, programs where some cores are used exclusively for program execution while others are used strictly for measurement and management tasks is one such area of future research. Given the continued reemergence of virtualized machines based on their practical benefits, exploring the experiment space of profiling applications designed to run on virtual machines is another area we would like to investigate.

7.4 Beyond Functions

The effects of perturbation we examined were primarily on the calculated proportion or percent execution of functions – a very common and useful, but singular and high-level point of analysis. We want to conduct further studies below the function level; e.g. basic block, source line, instruction; as well as the data centric perspective, to determine the impacts of perturbation error and how our model might be modified to capture it.

7.5 Computer Universe

In the course of our investigation, it was not uncommon to brush up against references to the Hisenberg Uncertainty Principle with hints at the idea that computer systems can be viewed in a manner similar to that of a mini universe where there are limits to the accuracy with which we can know or predict anything. We would like to identify and model other aspects of these systems for inclusion with our model to gain a better understanding of the limits of accuracy. Some examples include the clock resolution error; the duration or reach of a sample's perturbation effect; and the residual "noise" in a system – e.g., interrupts, system calls, context switches.

Chapter 8: Conclusions

With this research, we introduced and explored four ideas related to dynamic performance analysis. We started with the creation of a simple abstract model designed to demonstrate the general effects of measurement error and perturbation error on a program's execution during the process of statistical profiling. From this abstract model, we developed a mathematical model, constructed with some simplifying assumptions, to describe program execution performance; the derivative of which can be used to predict the sample count needed to find the "sweet spot" where the combined effects of residual measurement error and expanding perturbation error are lowest. In conjunction with the development of our mathematical model, we conducted an extensive set of experiments to validate the idea that a sampling "sweet spot" existed and to confirm the notion that a model might be used to find or predict the "sweet spot." This confirmation is an important (first?) step that undergirds the final larger idea that we should begin to move beyond qualitative guidance to experimenters by providing some quantitative guidance similar to what is available from the field of statistical science.

The starting point of our research, the abstract model, is a simple, but useful, general model of the interaction of measurement error and perturbation error that provides a valuable and constructive manner of thinking about the interaction of the two error types and their combined effect on measurement results across the spectrum of potential sample counts. It also serves as a reliable indicator of how the interaction of the errors affects overall experimental measurement accuracy of software statistical profiling. It helps challenge the thinking, often implicit in statistical performance

profiling, that collecting a large number of samples is a good and useful thing to do, often motivated by the simple fact that it is easy to do so.

Our mathematical model proved to be reasonably accurate at predicting the "sweet spot" sample count which allowed us to often calculate a sample interval at or near that which we determined to be optimal, while regularly avoiding those that were truly bad. This model is based on some simplifying assumptions, focusing only on cases where performance measurements dilate in response to perturbation; however, across all experimental contexts, our predictions performed better than the choice of using a single sampling interval for all experiments.

We demonstrated the high degree of likelihood for the common existence of a measurement accuracy "sweet spot" through experimental results from simulations and from experiments with our calibration program, with sequential execution on a subset of SPEC CPU 2006 benchmarks, and with shared-memory parallel execution on a subset of SPEC OMP 2001 benchmarks. Further, we showed our analytic model to be sufficiently accurate in predicting the "sweet spot" and calculating the corresponding sample interval that it could stand against any singularly favored and unchanging sample interval.

The systems performance analysis research literature is replete with qualitative suggestions and guidance like that provided in a 1969 paper by W.R. Deniston: "A prime consideration in developing a software measurement technique to obtain internal data is that a suitable compromise between resolution and system degradation must be achieved." [14] The main thrust of our research is in the direction of providing experimenters with some useful quantitative guidance related

to this problem of balancing measurement and perturbation errors. To that end, we believe we have provided an important step intended to help experimental procedures move beyond the well-founded suggestions to use "reasonable" or "appropriate" sampling intervals, towards solid quantitative guidance that helps experimenters calculate sampling intervals resulting in execution profiles with greater accuracy.

Appendix A

In this appendix we provide the sequence of steps by which we progress from our mathematical analytic function to our formula for calculating the ideal number of samples for a given performance analysis run. The following notation is used:

$t_{foo}(n)$	Execution time calculated for *foo* based on *n* samples
n	Number of samples
o	Overhead time cost per sample
$\hat{p}$	Measured proportion of total program execution time spent in *foo*
T	Un-sampled (true) total program execution time
z	Standard score (z-value) for confidence level used

Table A.1: Analytic function notation

Our analytic function is as follows:

$$t_{foo}(n) = no\hat{p} + T\left(\hat{p} + z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) \tag{A.1}$$

To begin, we take the first derivative of our analytic function with respect to *n* as follows:

$$t'_{foo}(n) = \frac{d}{dn}\left(no\hat{p} + T\left(\hat{p} + z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right)\right) \tag{A.2}$$

$$t'_{foo}(n) = \frac{d}{dn}(no\hat{p}) + \frac{d}{dn}\left(T\left(\hat{p} + z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right)\right) \tag{A.3}$$

$$t'_{foo}(n) = o\hat{p}\frac{d}{dn}(n) + T\frac{d}{dn}\left(\hat{p} + z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right) \tag{A.4}$$

$$t'_{foo}(n) = o\hat{p} + T\left(\frac{d}{dn}(\hat{p}) + \frac{d}{dn}\left(z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right)\right) \tag{A.5}$$

$$t'_{foo}(n) = o\hat{p} + Tz\sqrt{\hat{p}(1-\hat{p})}\left(\frac{d}{dn}\sqrt{\frac{1}{n}}\right) \quad \text{(A.6)}$$

$$t'_{foo}(n) = o\hat{p} - \frac{Tz\sqrt{\hat{p}(1-\hat{p})}}{2}\left(\sqrt{\frac{1}{n^3}}\right) \quad \text{(A.7)}$$

$$t'_{foo}(n) = o\hat{p} - \frac{Tz\sqrt{\hat{p}(1-\hat{p})}}{2\sqrt{n^3}} \quad \text{(A.8)}$$

We can now take the simplified first derivative, set it equal to 0, and solve to find the minimum point of our top curve as expressed by our analytic function.

$$0 = op - \frac{Tz\sqrt{\hat{p}(1-\hat{p})}}{2\sqrt{n^3}} \quad \text{(A.9)}$$

$$op = \frac{Tz\sqrt{\hat{p}(1-\hat{p})}}{2\sqrt{n^3}} \quad \text{(A.10)}$$

$$\sqrt{n^3} = \frac{Tz\sqrt{\hat{p}(1-\hat{p})}}{2o\hat{p}} \quad \text{(A.11)}$$

$$n = \sqrt[3]{\left(\frac{Tz\sqrt{\hat{p}(1-\hat{p})}}{2o\hat{p}}\right)^2} \quad \text{(A.11)}$$

We use this final equation (which is also equation 4.10 from section 4.2.2) to calculate the sample size and then determine the sampling rate that provides the best balance between achieving measurement precision and limiting the effects of perturbation error in the following manner. Solve for *n* using known or estimated values for terms *T*, *z*, $\hat{p}$, and *o*, then divide *n* (the total number of samples) by the number of minutes the program will run.

Appendix B

Additional detailed results relevant to the analysis of the SPEC CPU 2006 sequential program executions we conducted are contained in this appendix.

Bzip2

Following are the graphs of the bzip2 experiments.

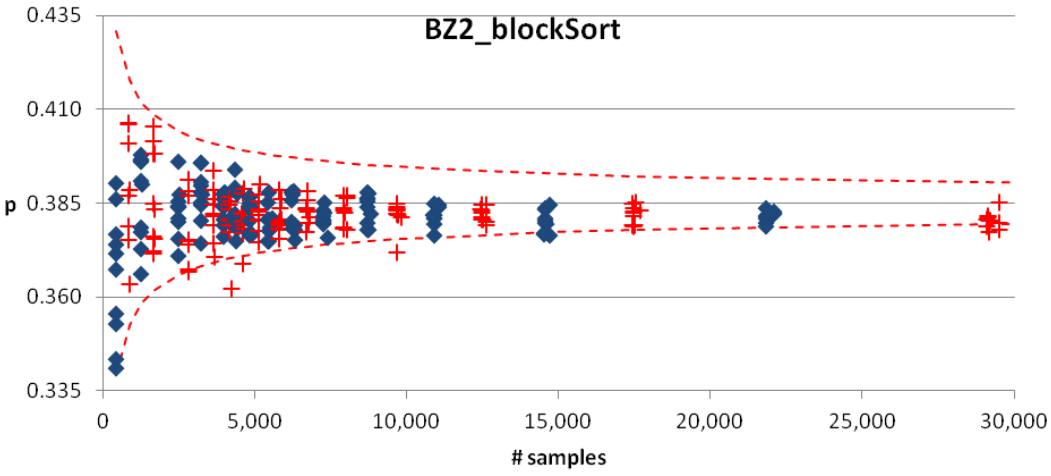

Figure B.1: Plot of proportion calculation results for *BZ2_blockSort)*, the function taking the most execution time for bzip2, across the 28x10 execution runs.

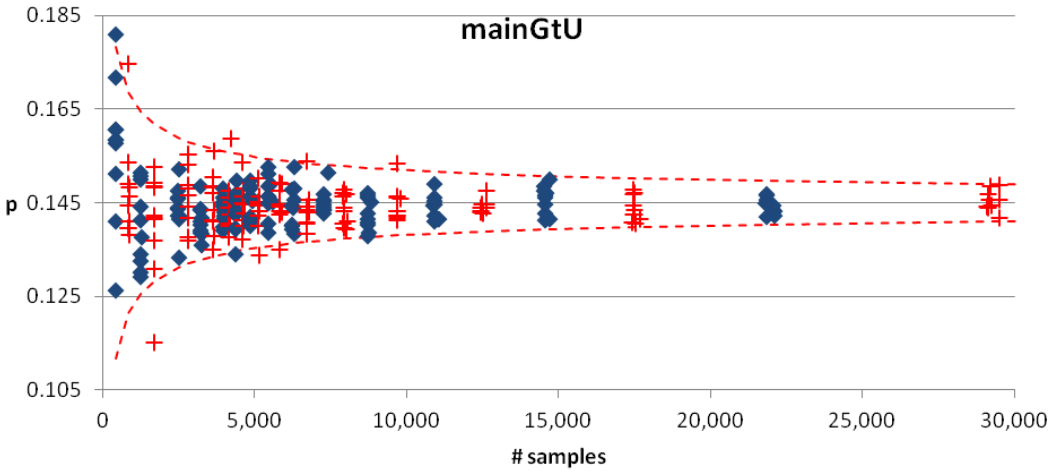

Figure B.2: Plot of proportion calculation results for *mainGtU*, the function taking the 2nd most execution time for bzip2, across the 28x10 execution runs.

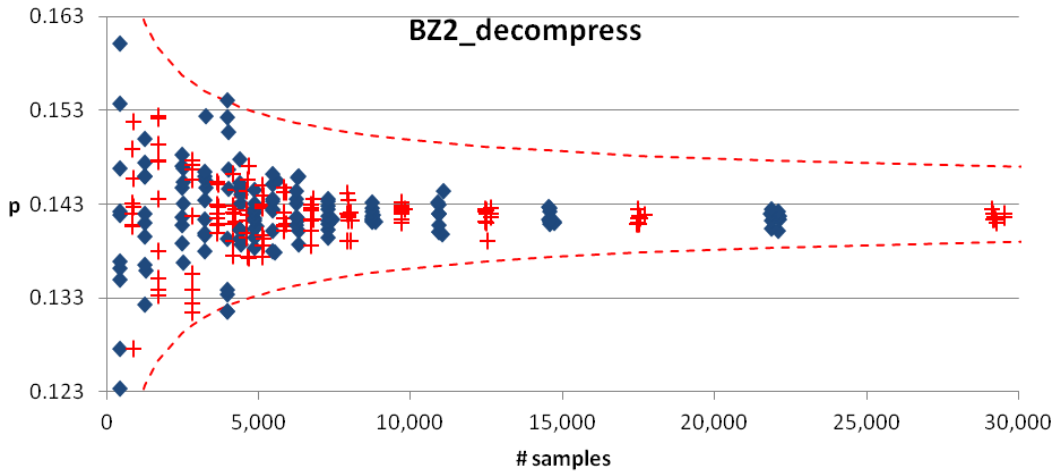

Figure B.3: Plot of proportion calculation results for *BZ2_decompress*, the function taking the 3rd most execution time for bzip2, across the 28x10 execution runs.

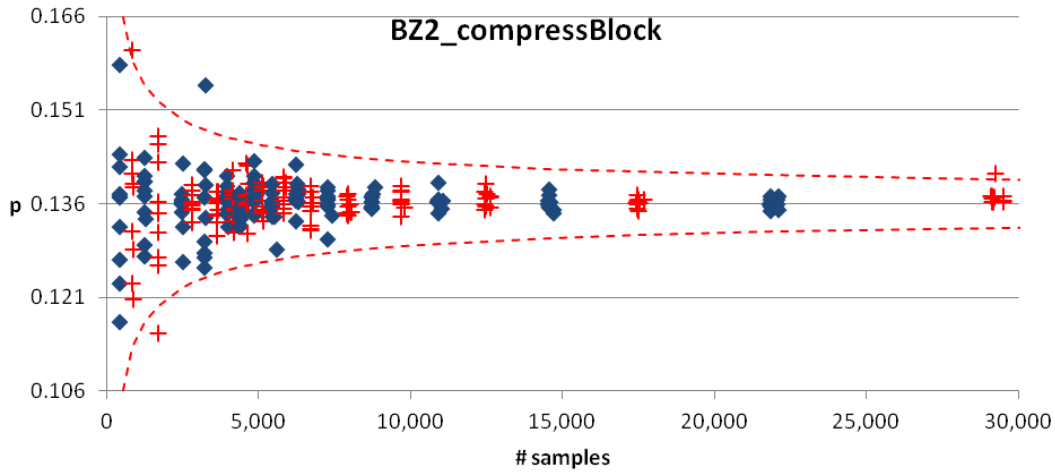

Figure B.4: Plot of proportion calculation results for *BZ2_compressBlock*, the function taking the 4th most execution time for bzip2, across the 28x10 execution runs.

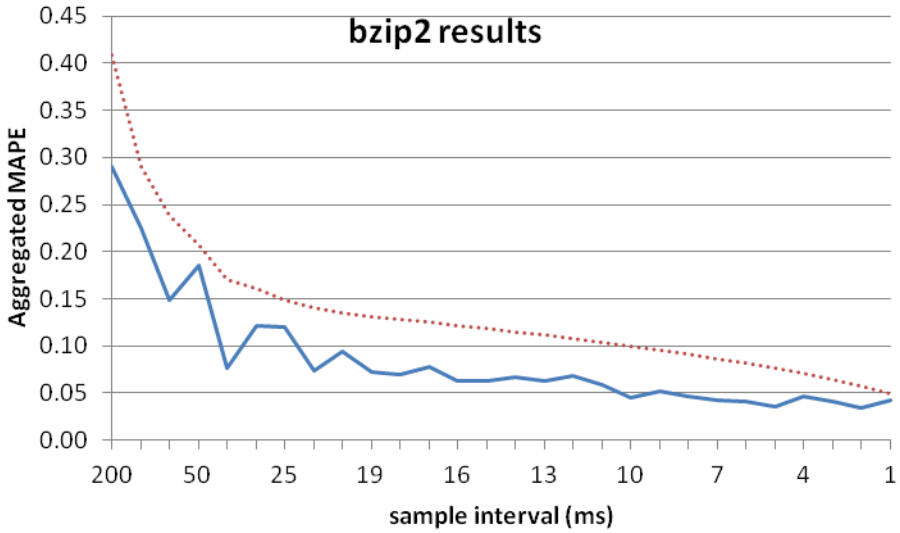

Figure B.5: The aggregated MAPE results for the top four functions in bzip2 (blue solid line) and the calculated approximate expected MAPE (dotted red line).

The sampling interval that resulted in the overall best result for the bzip2 benchmark was 2 ms.

Mcf

Following are the graphs of the mcf experiments.

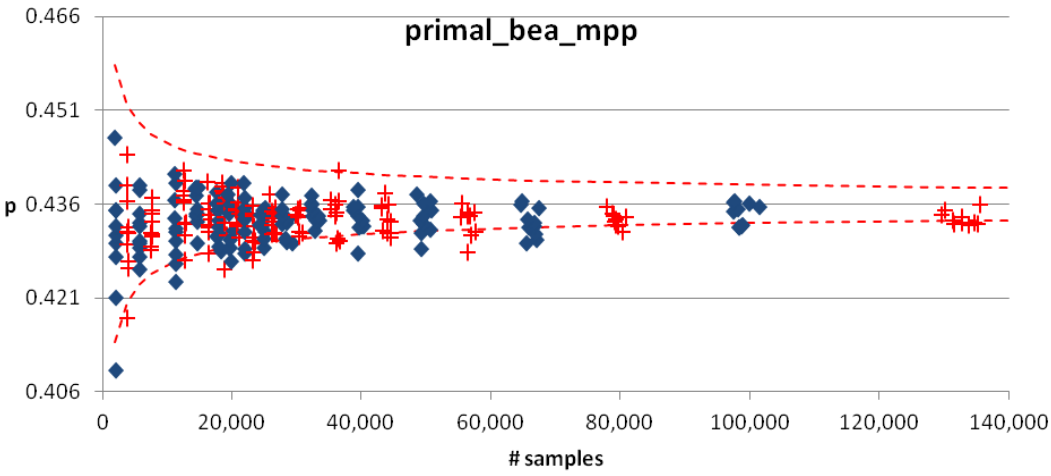

Figure B.6: Plot of proportion calculation results for *primal_bea_mpp*, the function taking the most execution time for mcf, across the 28x10 execution runs.

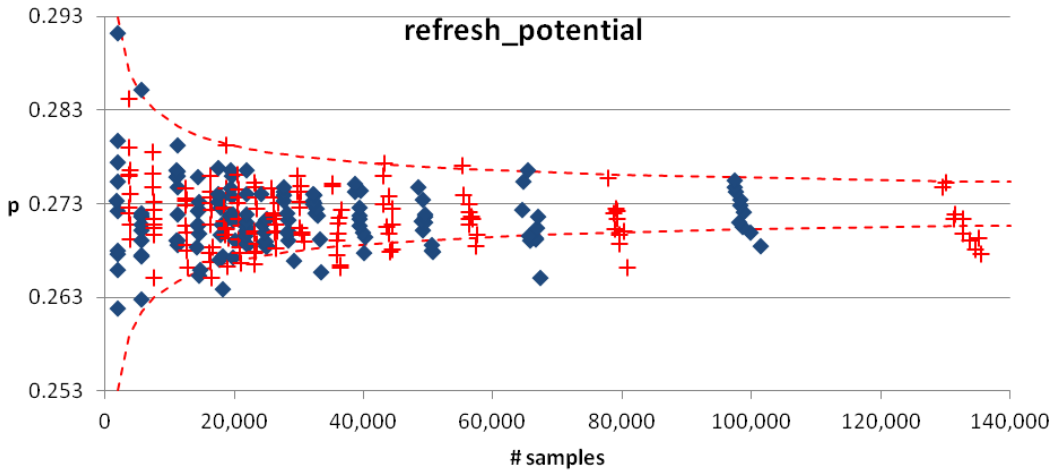

Figure B.7: Plot of proportion calculation results for *refresh_potential*, the function taking the 2nd most execution time for mcf, across the 28x10 execution runs.

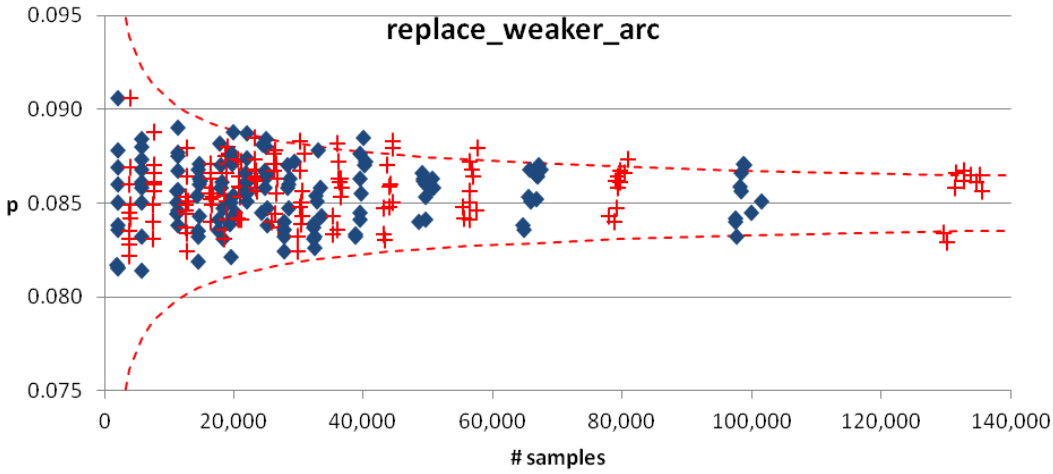

Figure B.8: Plot of proportion calculation results for *replace_weaker_arc*, the function taking the 3rd most execution time for mcf, across the 28x10 execution runs.

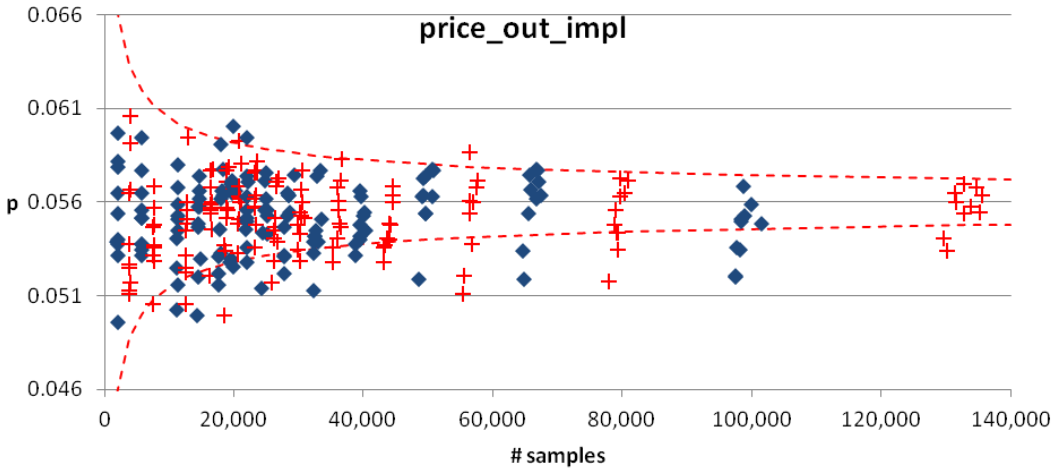

Figure B.9: Plot of proportion calculation results for *price_out_impl*, the function taking the 4th most execution time for mcf, across the 28x10 execution runs.

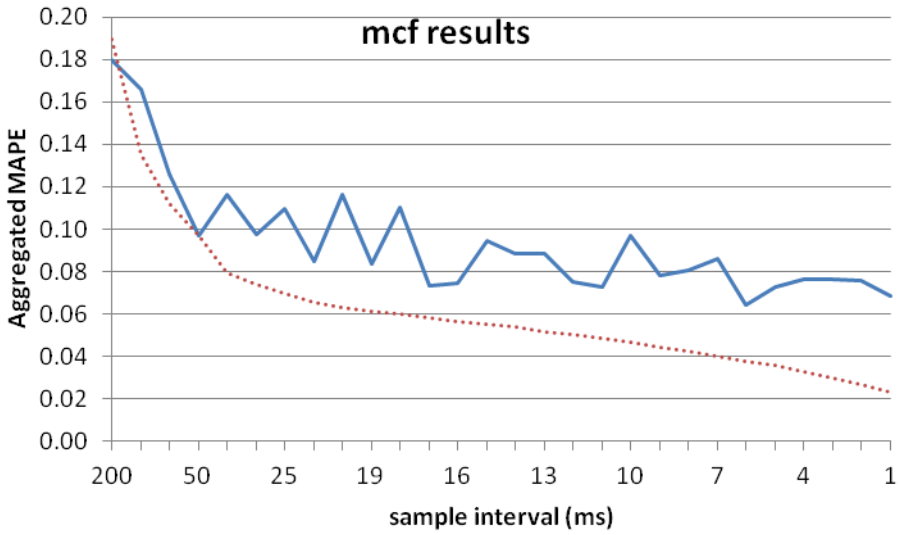

Figure B.10: The aggregated MAPE results for the top four functions in mcf (blue solid line) and the calculated approximate expected MAPE (dotted red line).

The sampling interval that resulted in the overall best result for the mcf benchmark was 2 ms.

Milc

Following are the graphs of the milc experiments.

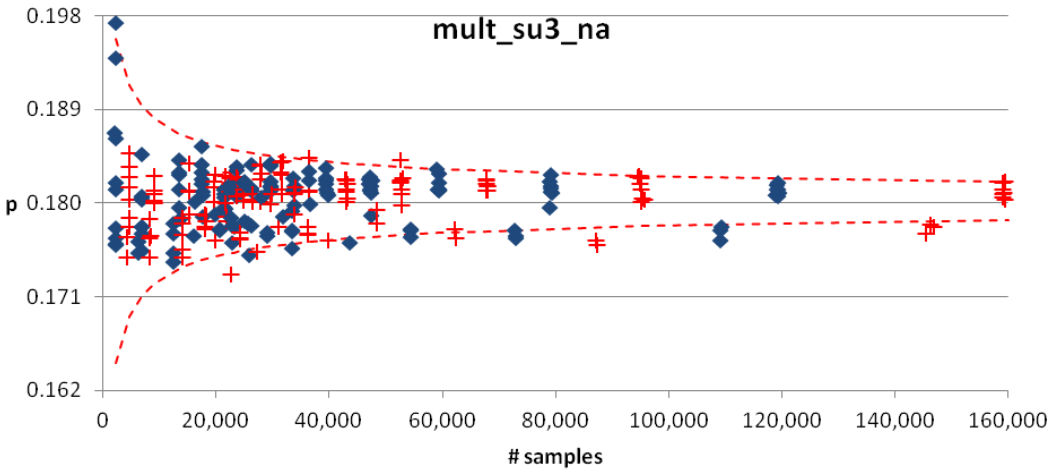

Figure B.11: Plot of proportion calculation results for *mult_su3_na*, the function taking the most execution time for milc, across the 28x10 execution runs.

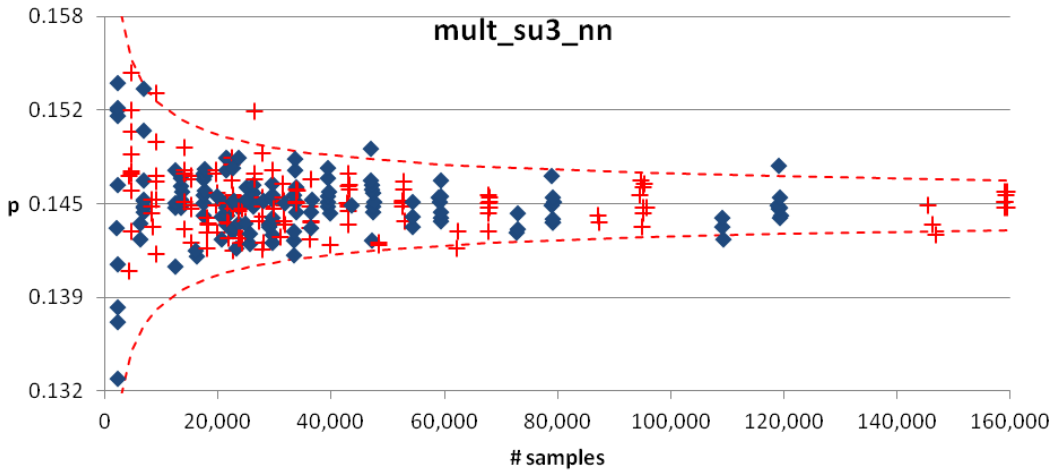

Figure B.12: Plot of proportion calculation results for *mult_su3_nn*, the function taking the 2nd most execution time for milc, across the 28x10 execution runs.

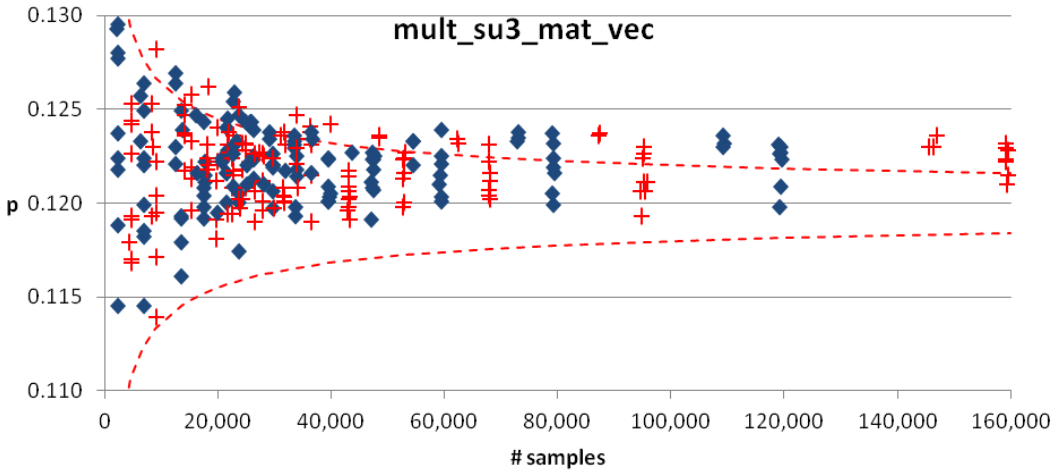

Figure B.13: Plot of proportion calculation results for *mult_su3_mat_vec*, the function taking the 3rd most execution time for milc, across the 28x10 execution runs.

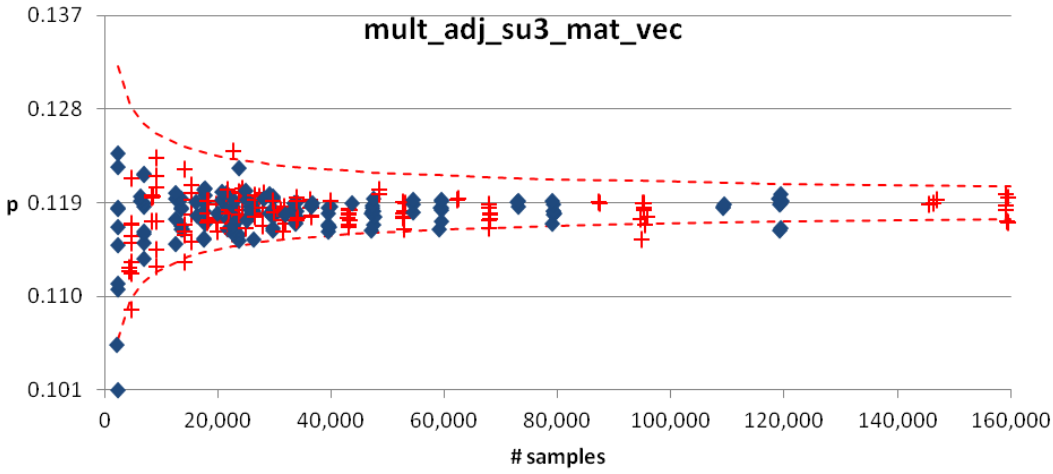

Figure B.14: Plot of proportion calculation results for *mult_adj_su3_mat_vec*, the function taking the 4th most execution time for milc, across the 28x10 execution runs.

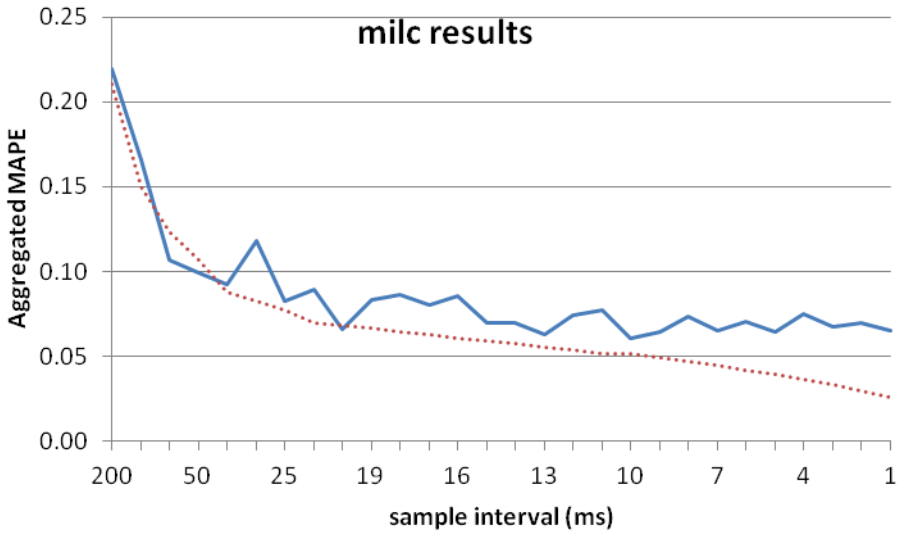

Figure b.15: The aggregated MAPE results for the top four functions in milc (blue solid line) and the calculated approximate expected MAPE (dotted red line).

The sampling interval that resulted in the overall best result for the milc benchmark was 9 ms.

Sjeng

Following are the graphs of the sjeng experiments.

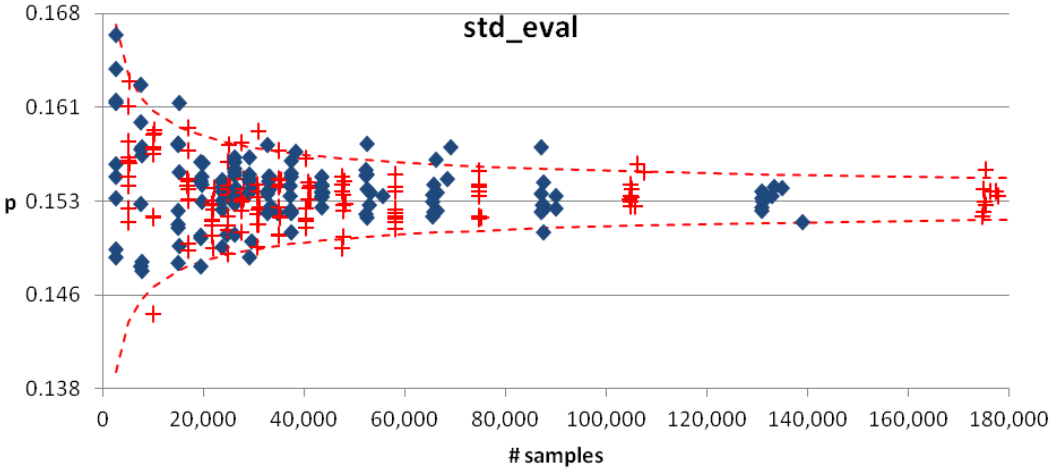

Figure B.16: Plot of proportion calculation results for *std_eval*, the function taking the most execution time for sjeng, across the 28x10 execution runs.

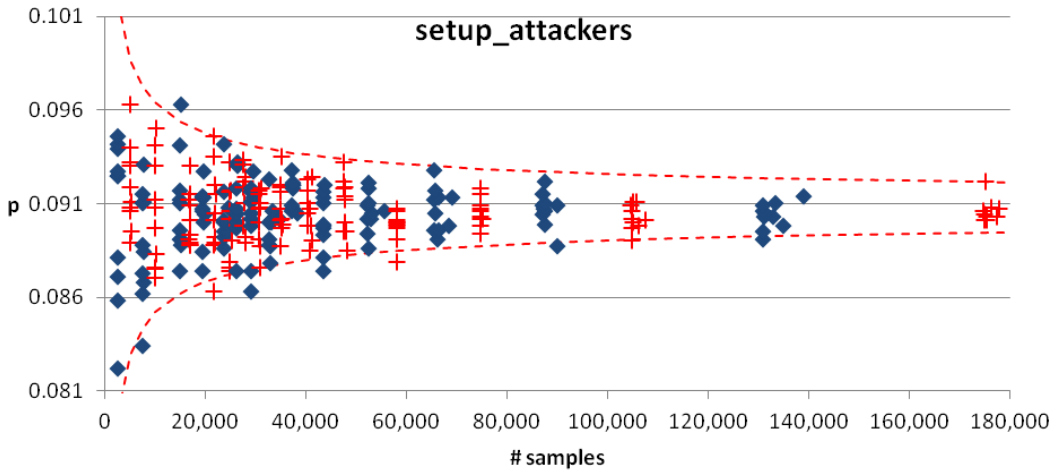

Figure B.17: Plot of proportion calculation results for *setup_attackers*, the function taking the 2nd most execution time for sjeng, across the 28x10 execution runs.

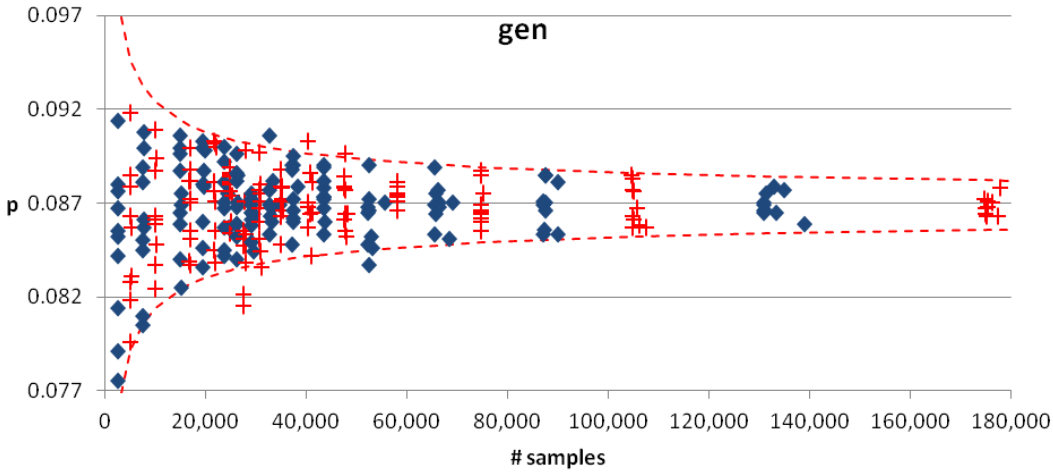

Figure B.18: Plot of proportion calculation results for *gen*, the function taking the 3rd most execution time for sjeng, across the 28x10 execution runs.

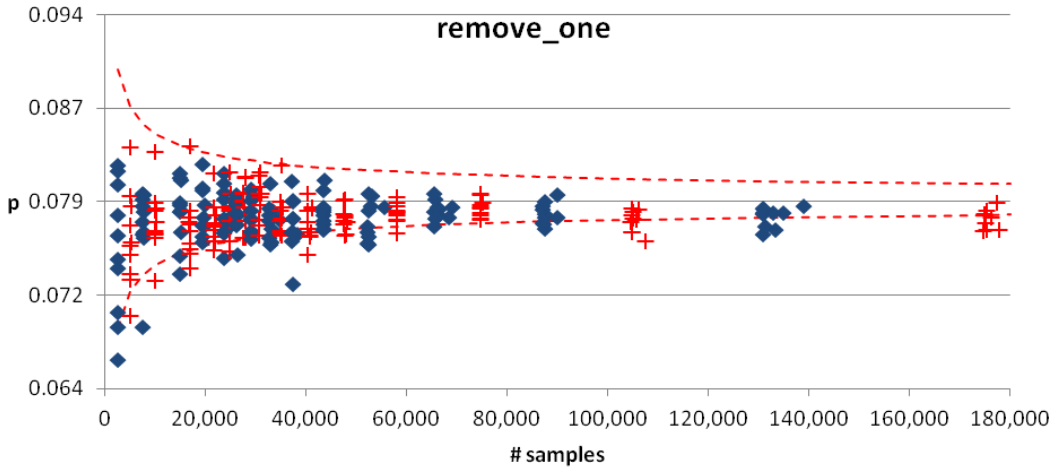

Figure B.19: Plot of proportion calculation results for *remove_one*, the function taking the 4th most execution time for sjeng, across the 28x10 execution runs.

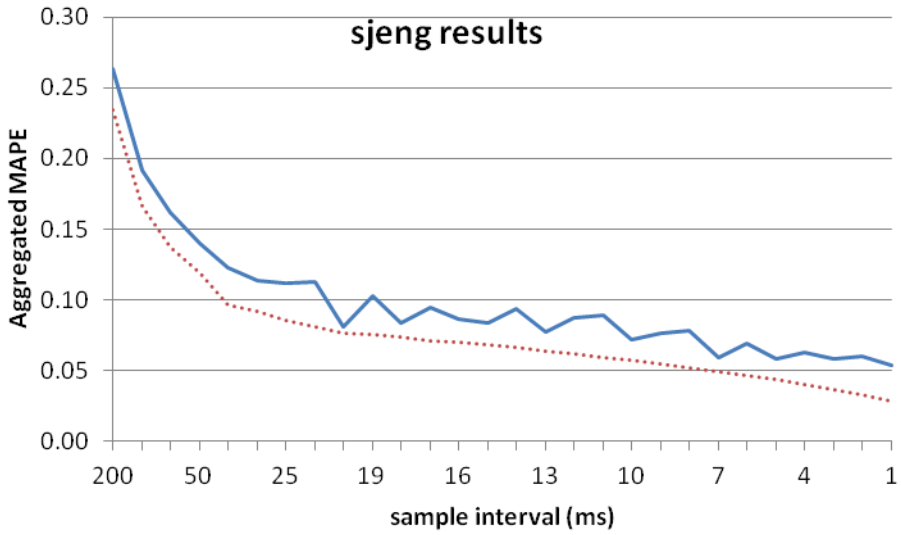

Figure B.20: The aggregated MAPE results for the top four functions in sjeng (blue solid line) and the calculated approximate expected MAPE (dotted red line).

Appendix C

Additional detailed results relevant to the analysis of the SPEC OMP 2001 shared-memory parallel program executions we conducted are contained in this appendix. The following four graphs show the distribution of run times observed across all experiments. Legends for these four graphs are across the top, indicating the name of the SPEC OMP 2001 benchmark and which graph marker corresponds with which core setting.

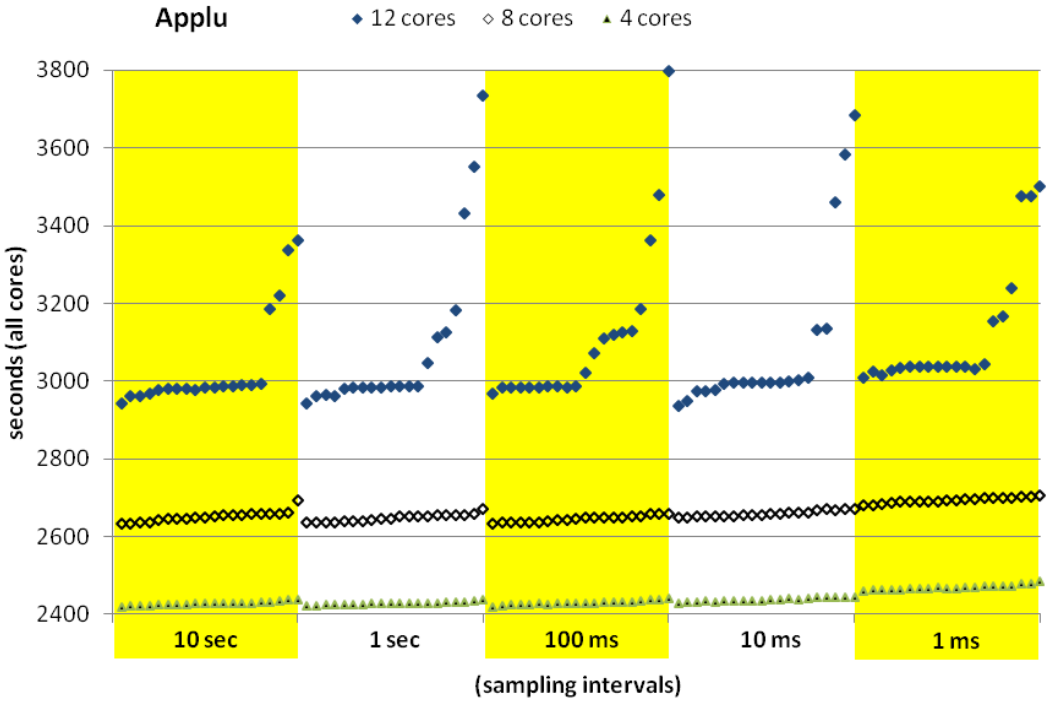

Figure C.1: Distribution of run times, 20 each, sorted shortest to longest within each sampling interval, for the SPEC OMP 2001 benchmark applu.

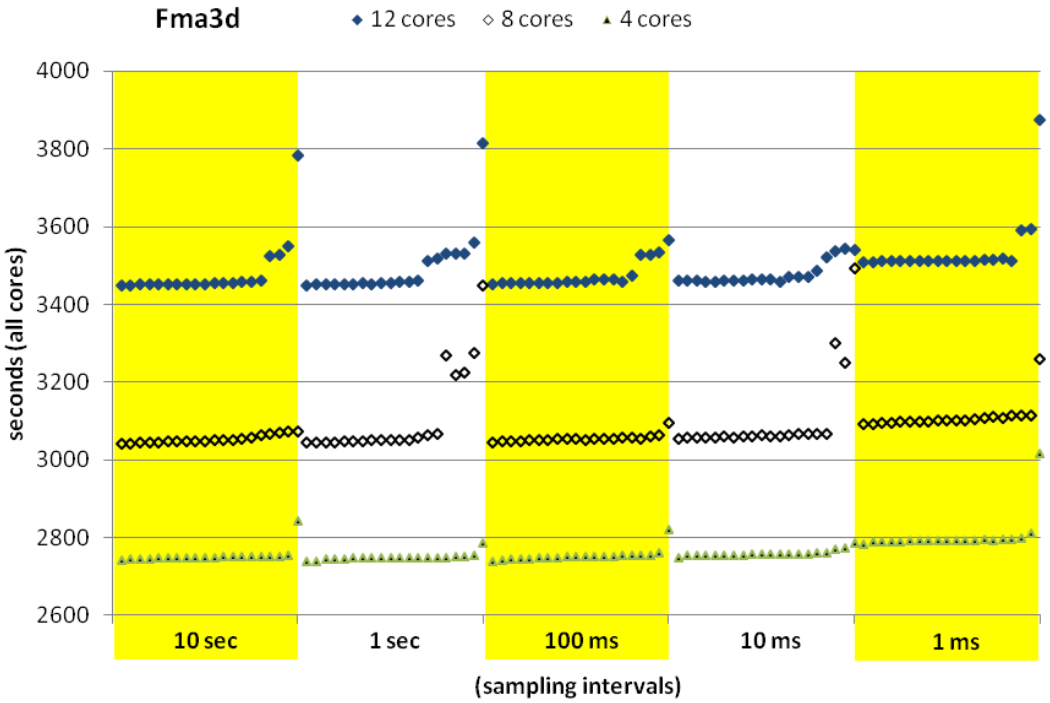

Figure C.2: Distribution of run times, 20 each, sorted shortest to longest within each sampling interval, for the SPEC OMP 2001 benchmark fma3d.

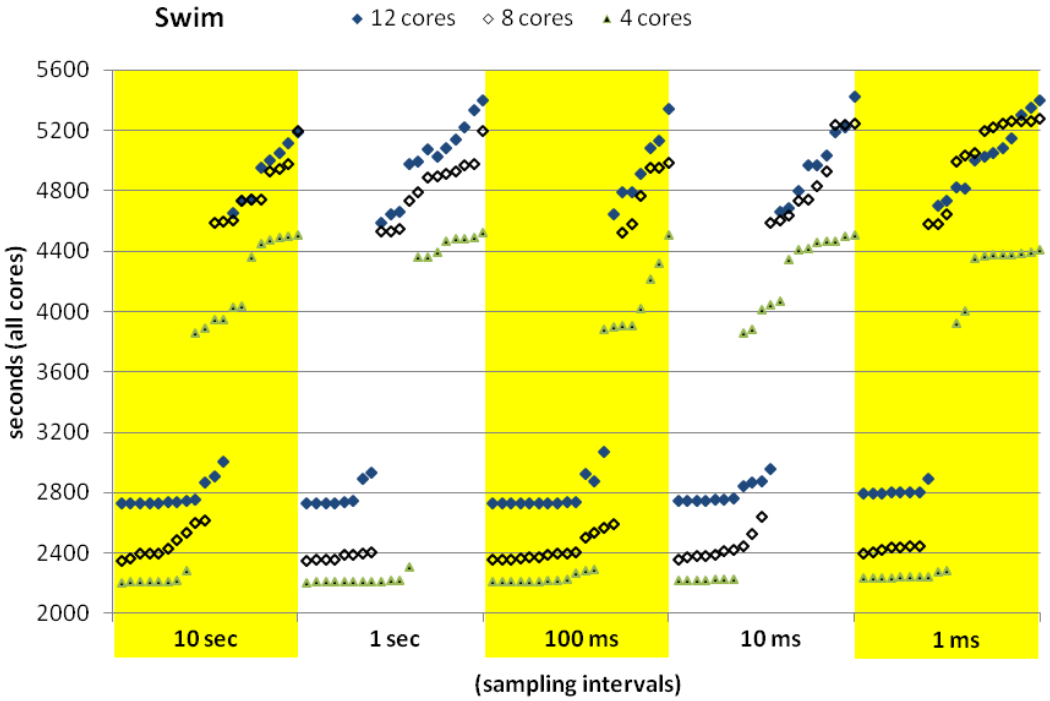

Figure C.3: Distribution of run times, 20 each, sorted shortest to longest within each sampling interval, for the SPEC OMP 2001 benchmark swim.

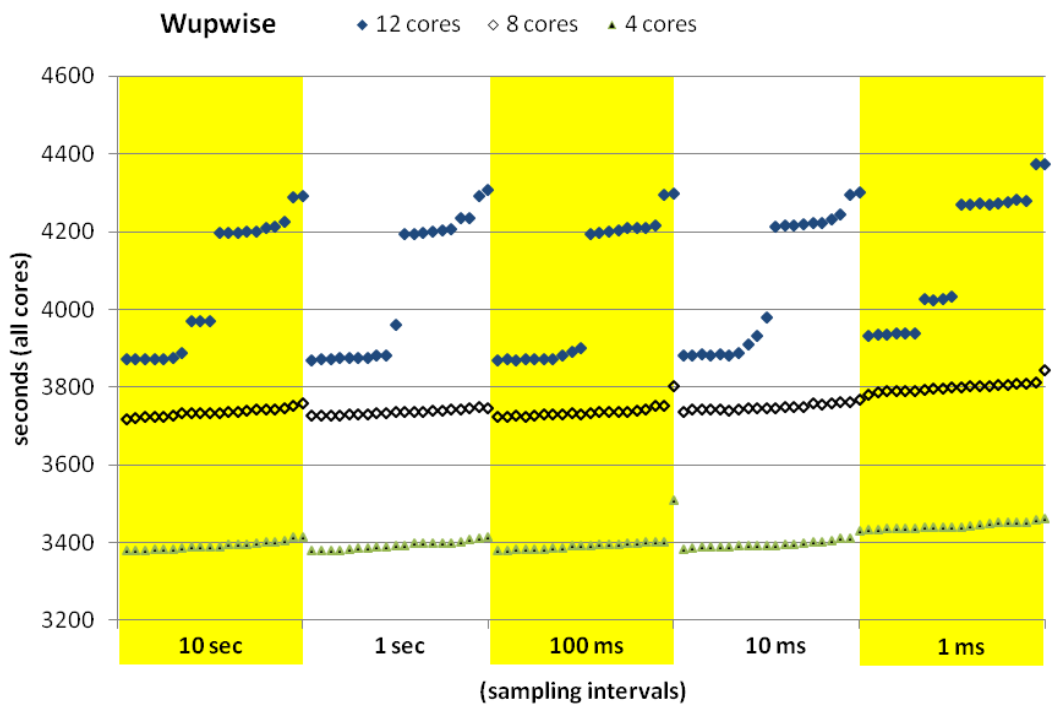

Figure C.4: Distribution of run times, 20 each, sorted shortest to longest within each sampling interval, for the SPEC OMP 2001 benchmark wupwise.

The graphs from this point to the end of the appendix present either the results of individual functions for a specific benchmark or a composite result of the top 4 functions of a benchmark with 95% confidence interval whiskers. These first 3 graphs complement those presented in Chapter 6 for equake 4 threads on 4 cores.

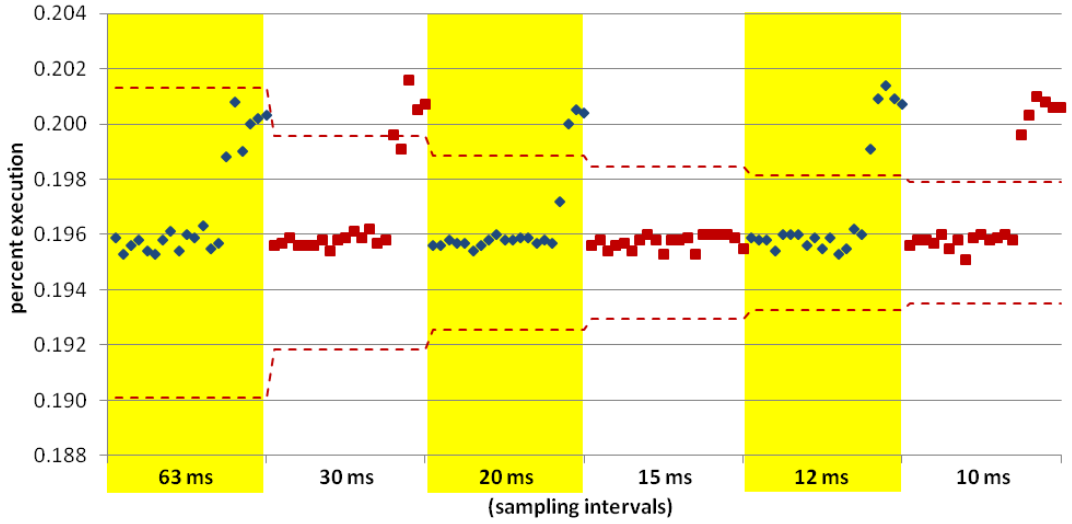

Figure C.5: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with equake, 4 threads on 4 cores, 20 runs per sampling interval.

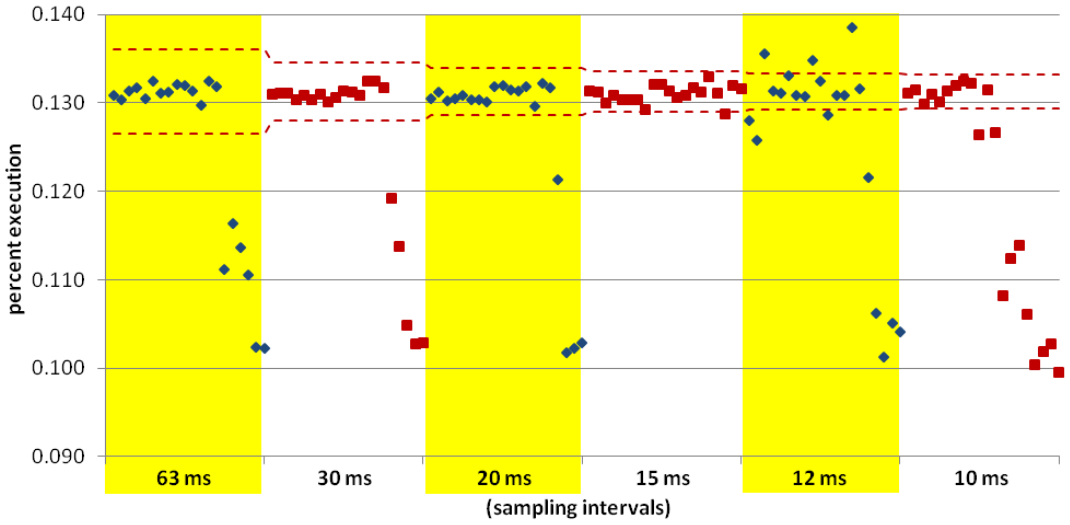

Figure C.6: Results of the percent execution of main.omp_fn.10, the function taking the 3rd most execution time for the 120 runs conducted with equake, 4 threads on 4 cores, 20 runs per sampling interval.

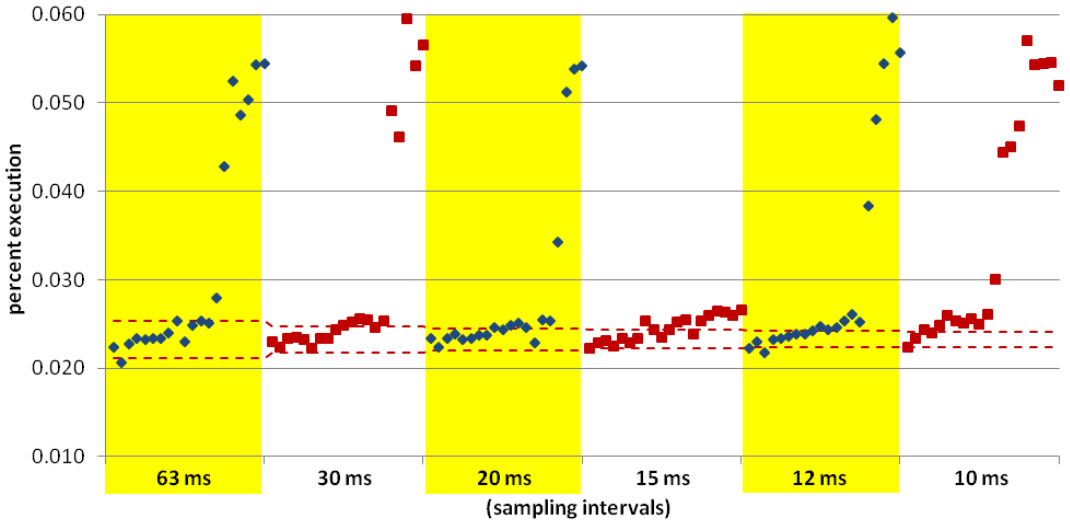

Figure C.7: Results of the percent execution of omp_get_num_procs for the 120 runs conducted with equake, 4 threads on 4 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with equake, 8 threads on 8 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results with 95% CIs for the small interval bracketing runs.

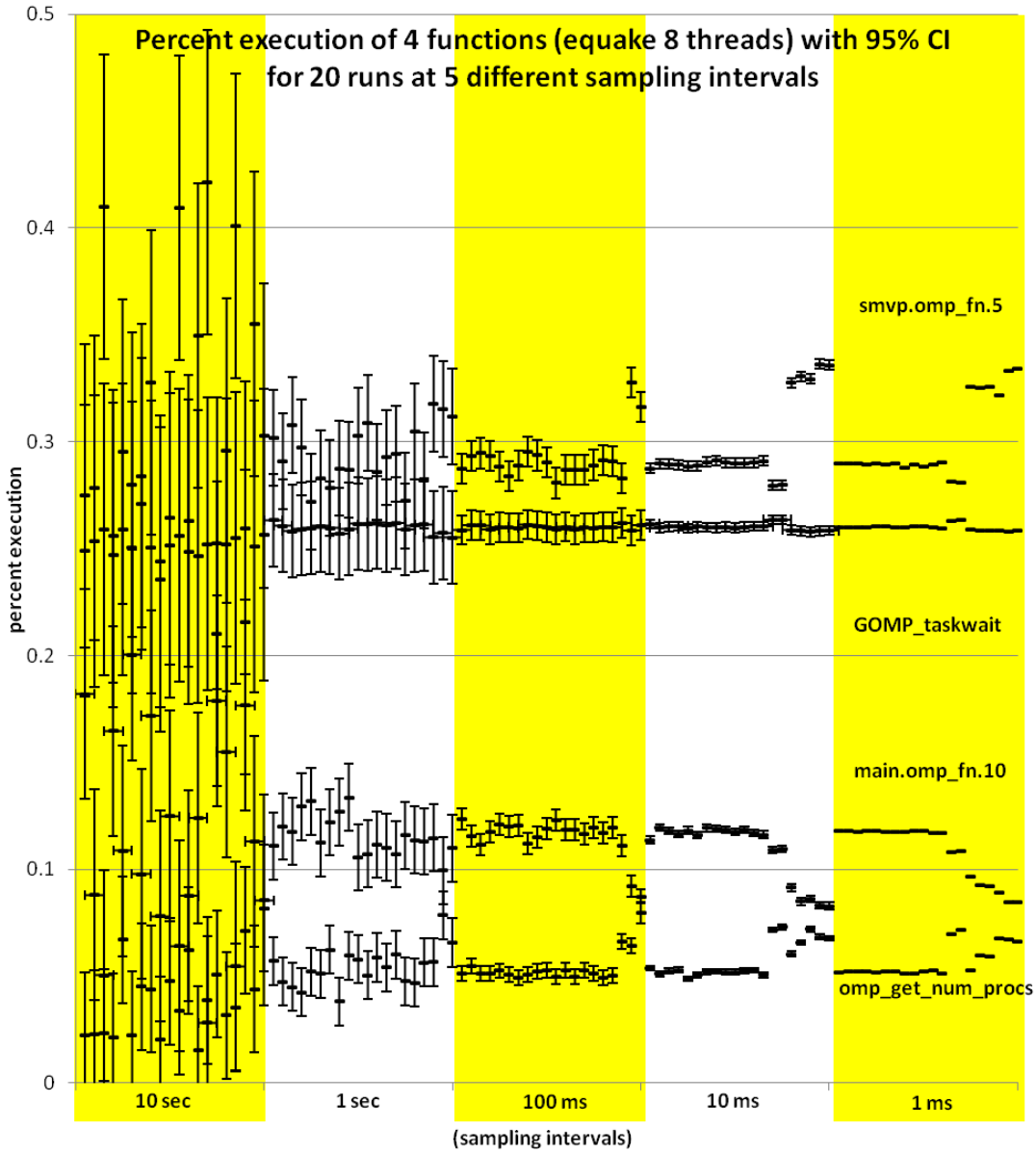

Figure C.8: Distribution of the percent execution taken by 4 different functions from 100 runs of equake with 8 threads on 8 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

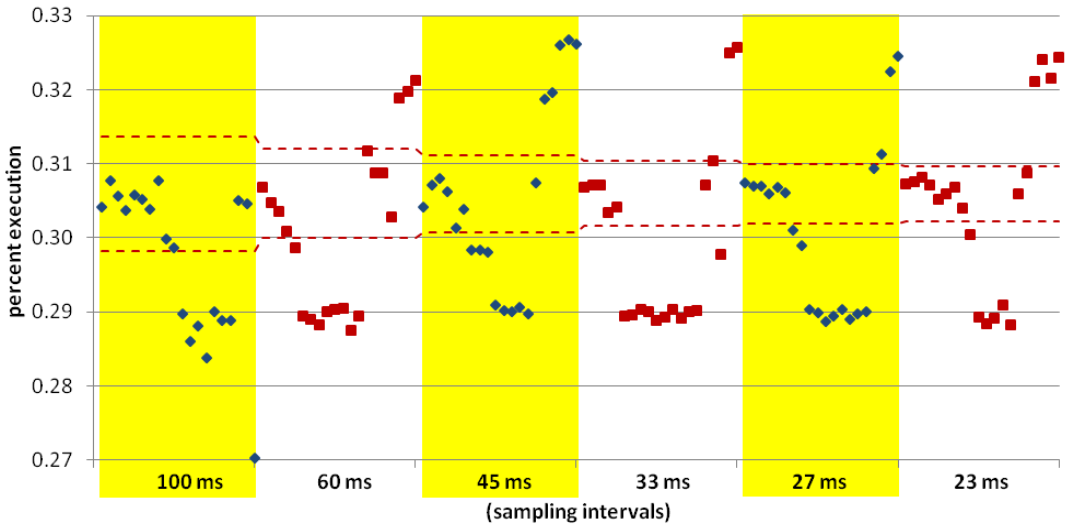

Figure C.9: Results of the percent execution of smvp.omp_fn.5, the function taking the most execution time for the 120 runs conducted with equake, 8 threads on 8 cores, 20 runs per sampling interval.

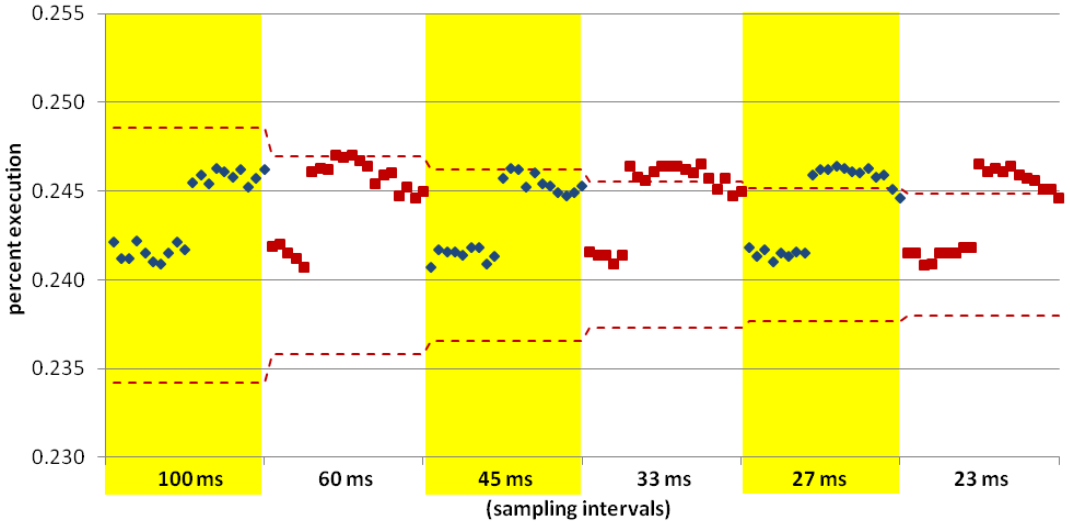

Figure C.10: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with equake, 8 threads on 8 cores, 20 runs per sampling interval.

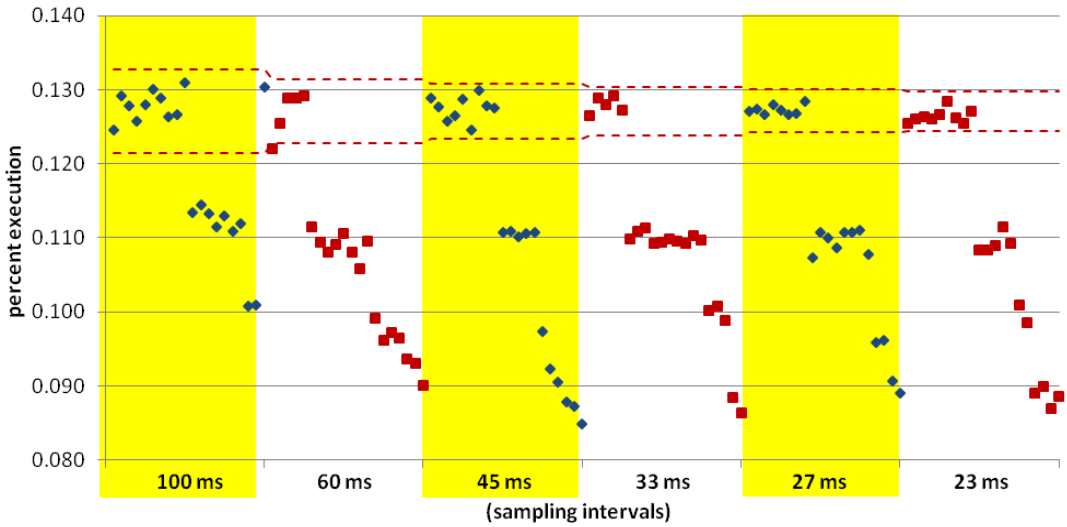

Figure C.11: Results of the percent execution of main.omp_fn.10, the function taking the 3rd most execution time for the 120 runs conducted with equake, 8 threads on 8 cores, 20 runs per sampling interval.

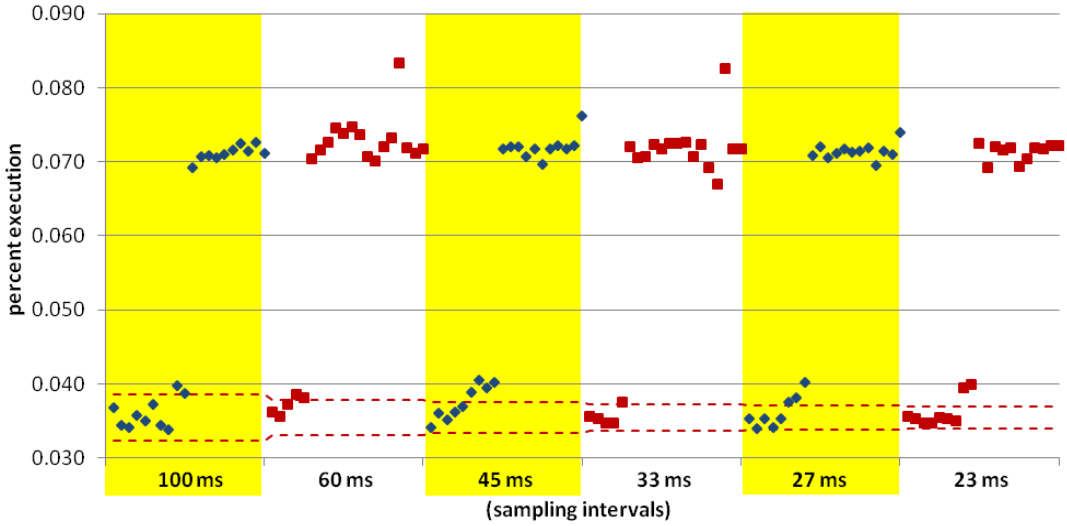

Figure C.12: Results of the percent execution of omp_get_num_procs, for the 120 runs conducted with equake, 8 threads on 8 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with equake, 12 threads on 12 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

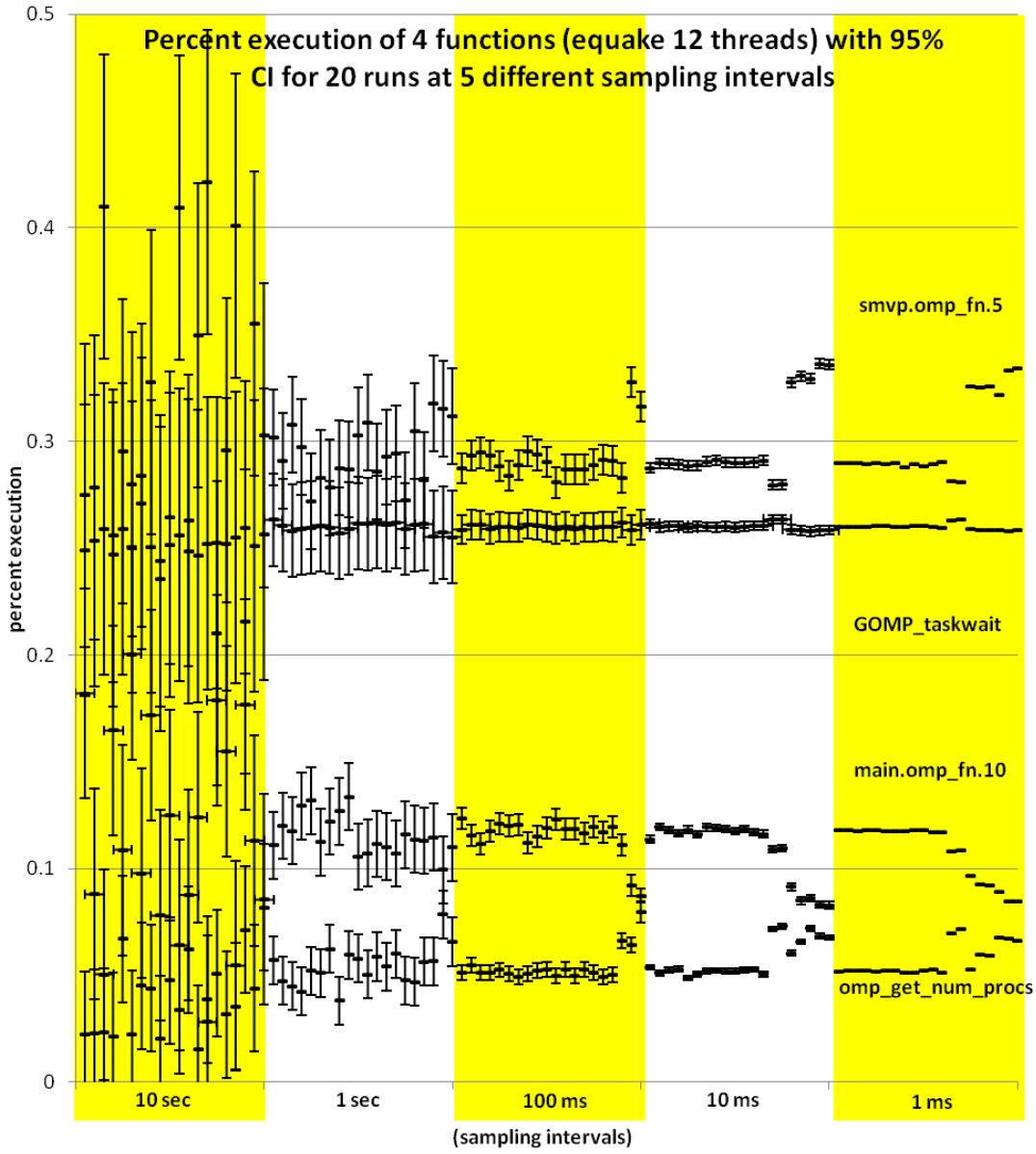

Figure C.13: Distribution of the percent execution taken by 4 different functions from 100 runs of equake with 12 threads on 12 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

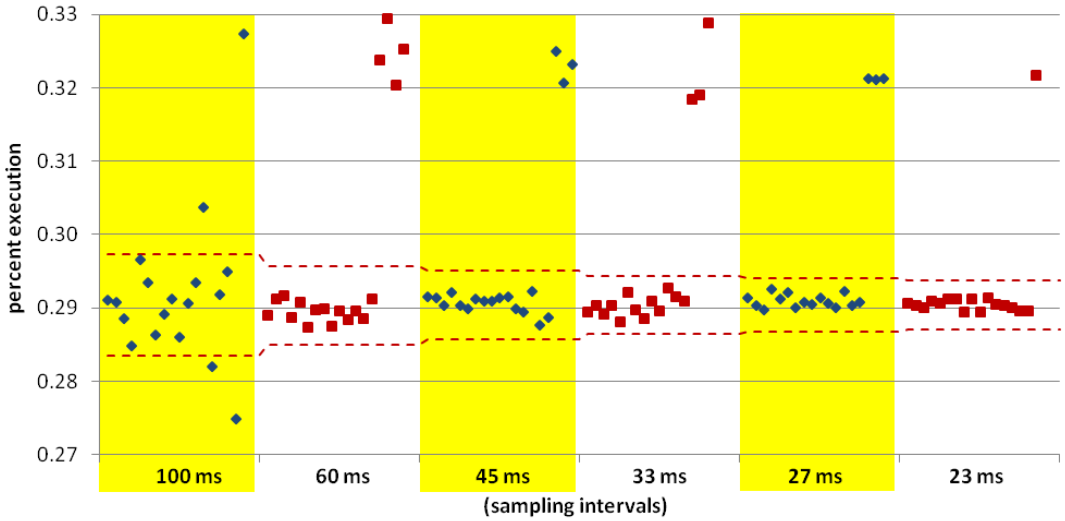

Figure C.14: Results of the percent execution of smvp.omp_fn.5, the function taking the most execution time for the 120 runs conducted with equake, 12 threads on 12 cores, 20 runs per sampling interval.

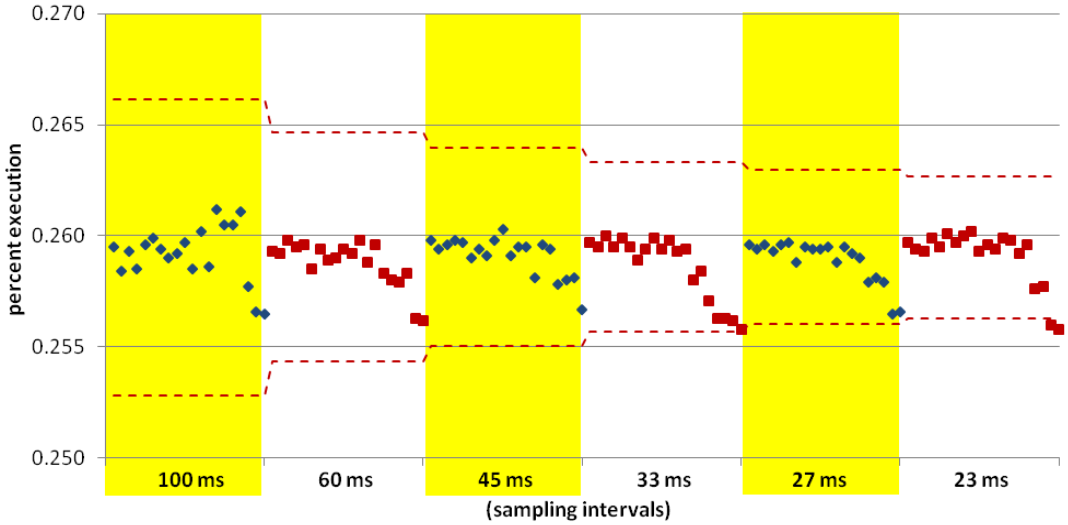

Figure C.15: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with equake, 12 threads on 12 cores, 20 runs per sampling interval.

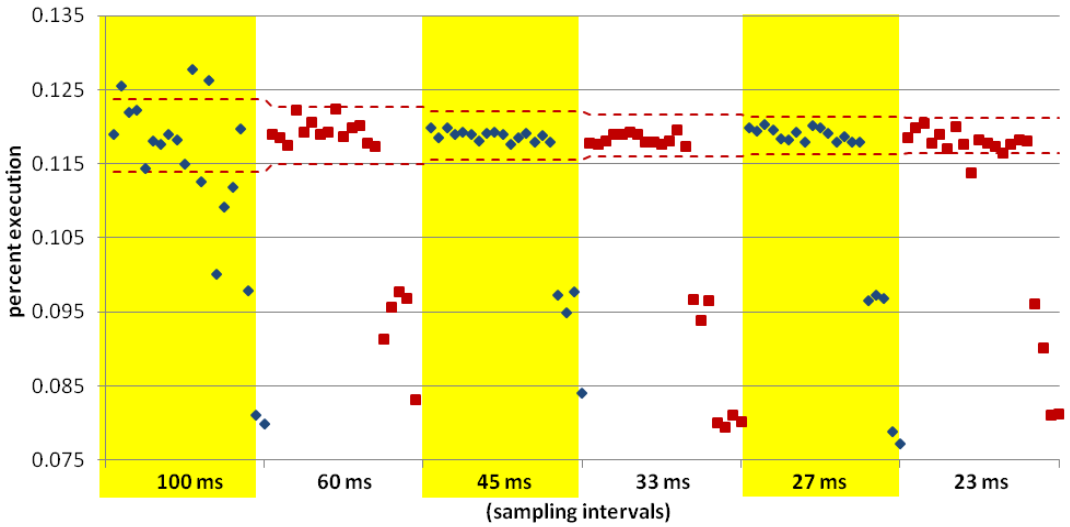

Figure C.16: Results of the percent execution of main.omp_fn.10, the function taking the 3rd most execution time for the 120 runs conducted with equake, 12 threads on 12 cores, 20 runs per sampling interval.

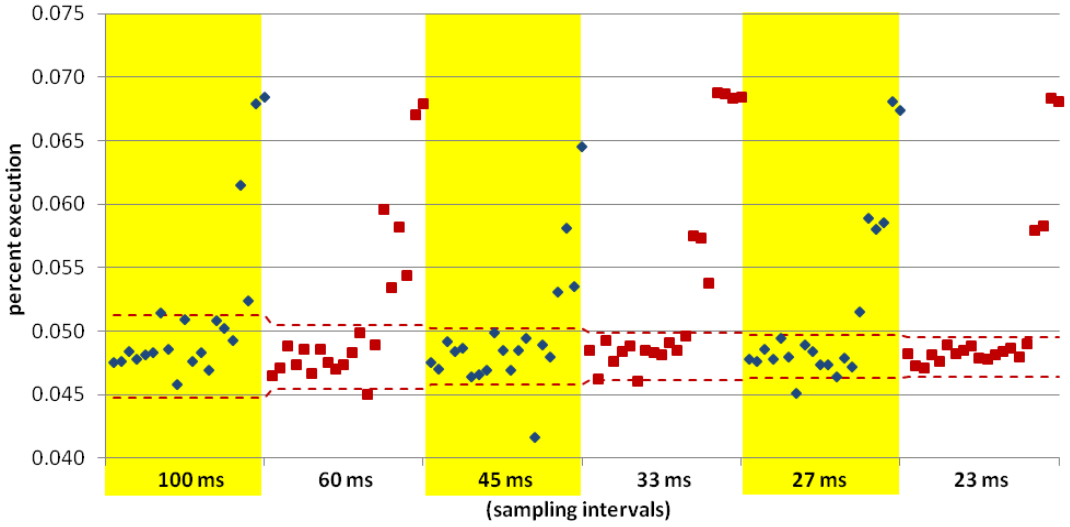

Figure C.17: Results of the percent execution of omp_get_num_procs, for the 120 runs conducted with equake, 12 threads on 12 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with applu, 4 threads on 4 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

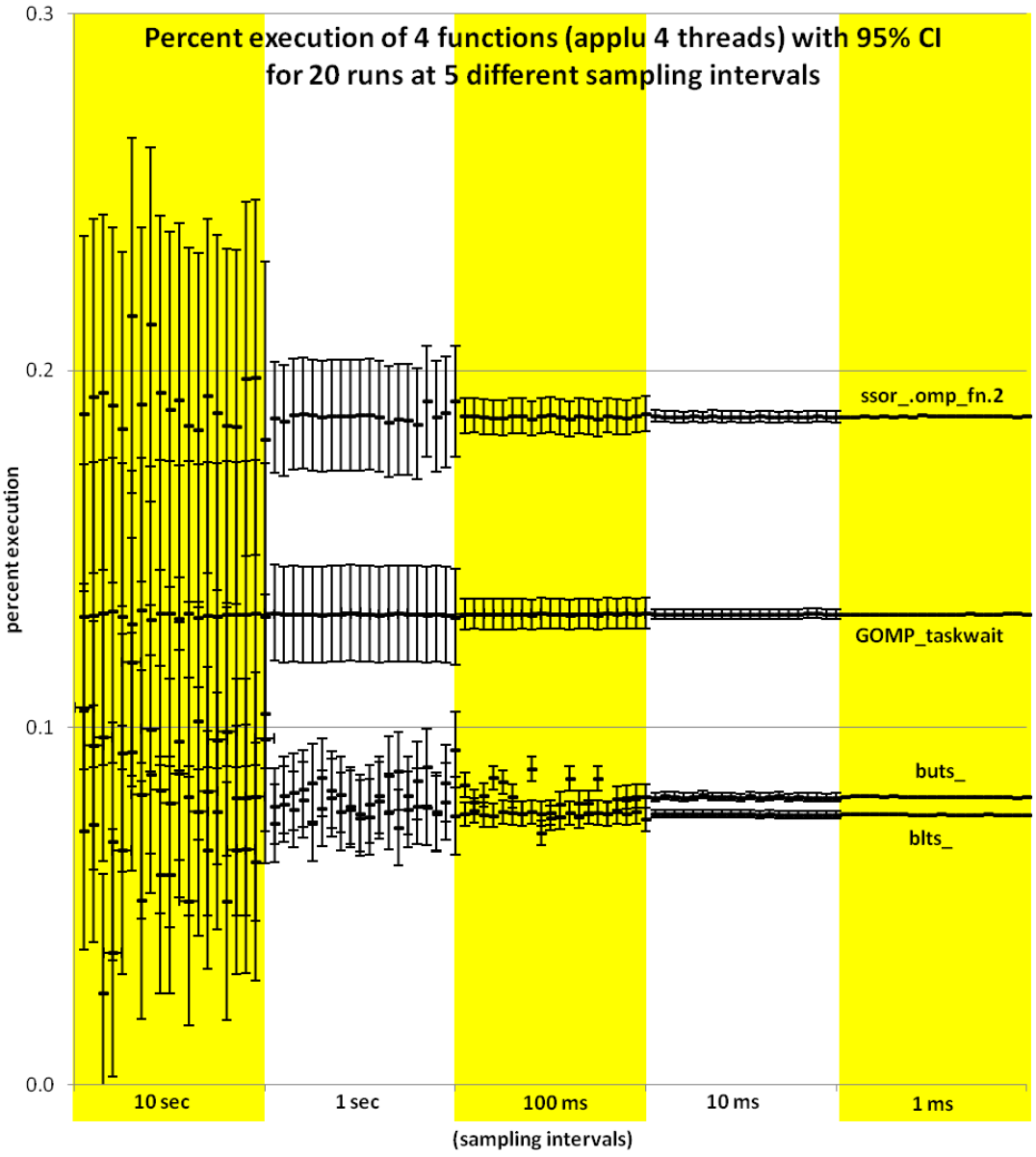

Figure C.18: Distribution of the percent execution taken by 4 different functions from 100 runs of applu with 4 threads on 4 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

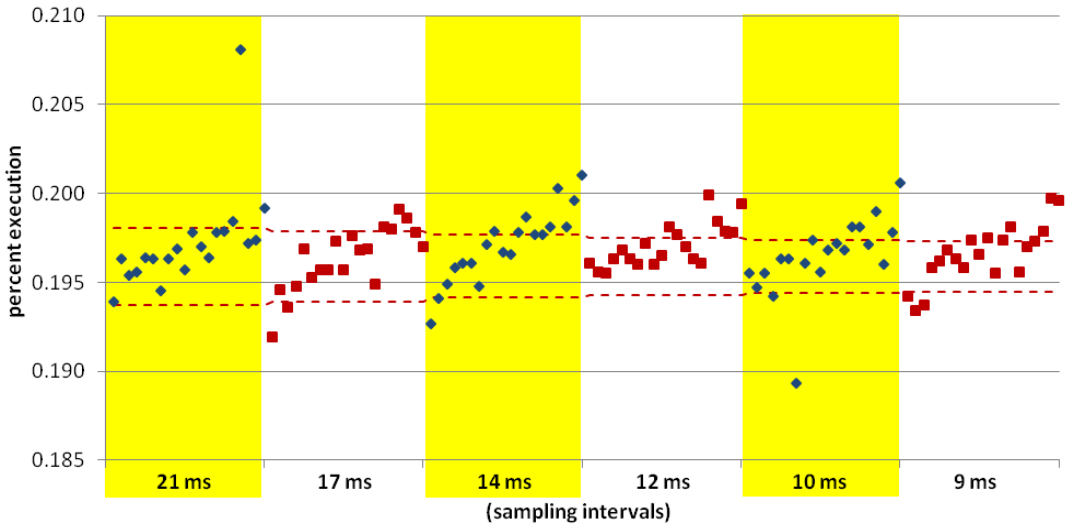

Figure C.19: Results of the percent execution of ssor_.omp_fn.2, the function taking the most execution time for the 120 runs conducted with applu, 4 threads on 4 cores, 20 runs per sampling interval.

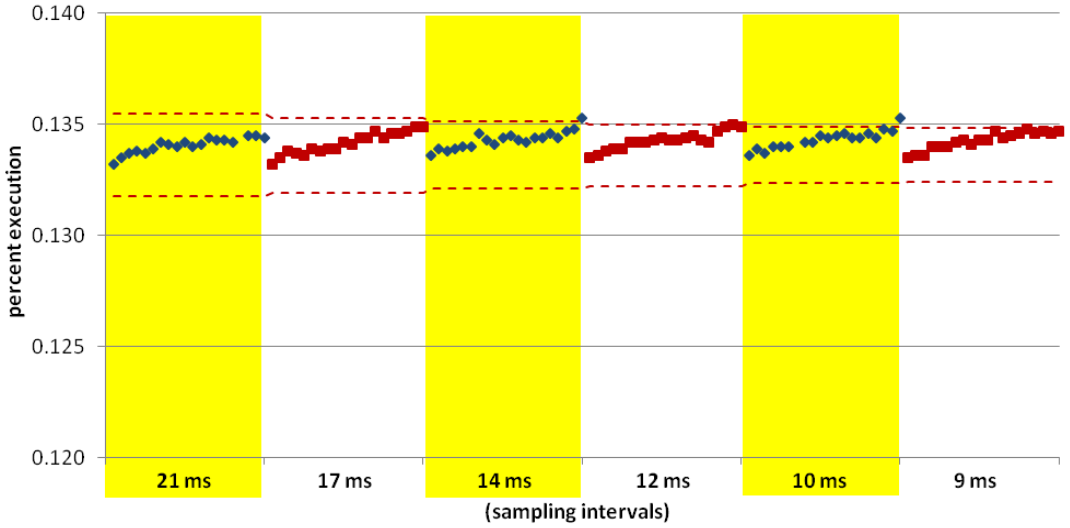

Figure C.20: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with applu, 4 threads on 4 cores, 20 runs per sampling interval.

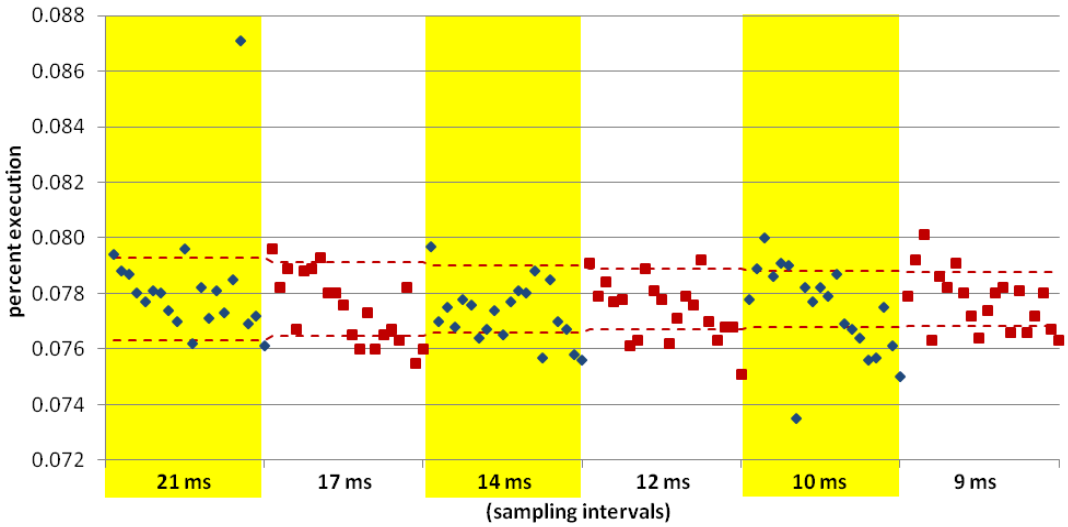

Figure C.21: Results of the percent execution of buts_, the function taking the 3rd most execution time for the 120 runs conducted with applu, 4 threads on 4 cores, 20 runs per sampling interval.

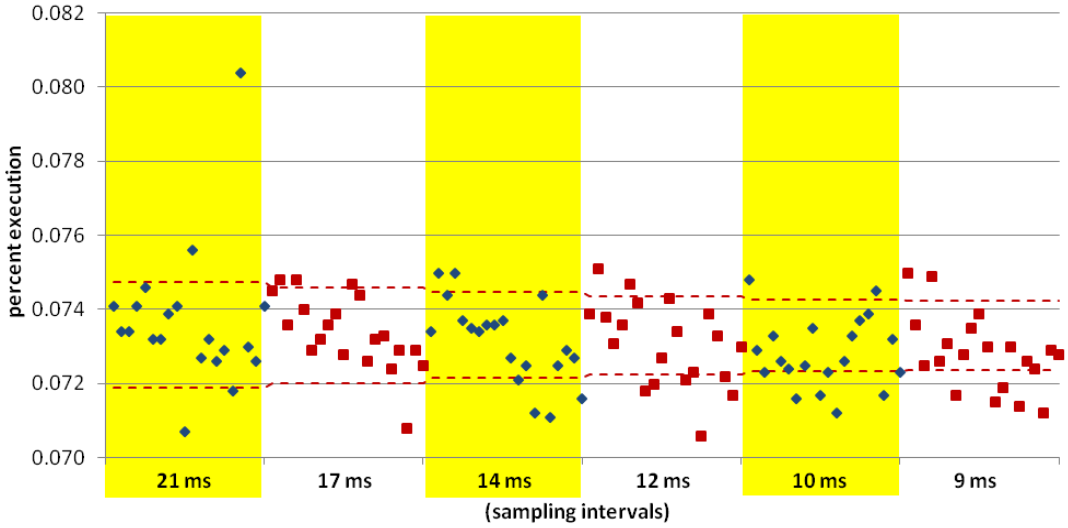

Figure C.22: Results of the percent execution of blts_, the function taking the 4th most execution time for the 120 runs conducted with applu, 4 threads on 4 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with applu, 8 threads on 8 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

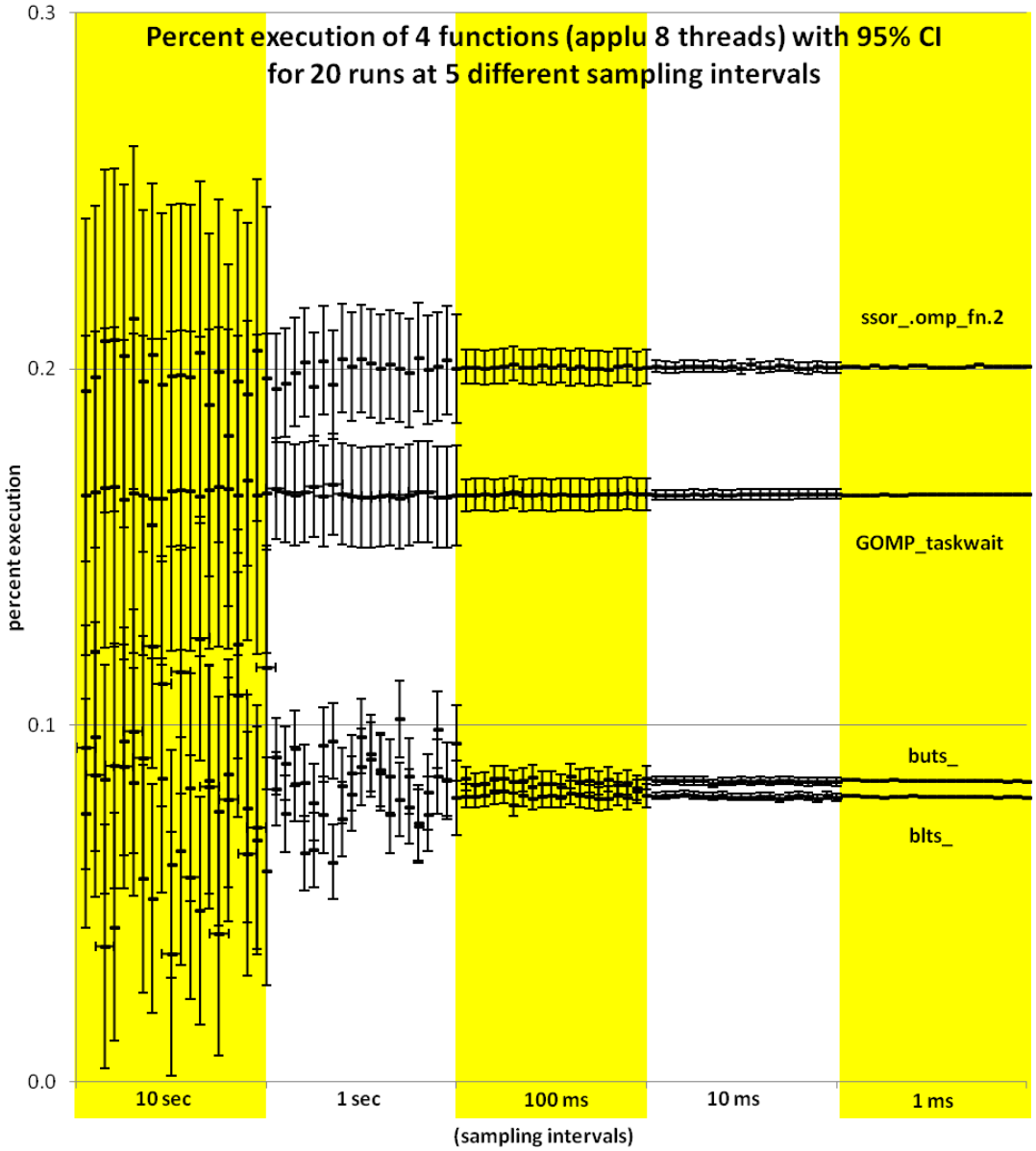

Figure C.23: Distribution of the percent execution taken by 4 different functions from 100 runs of applu with 8 threads on 8 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

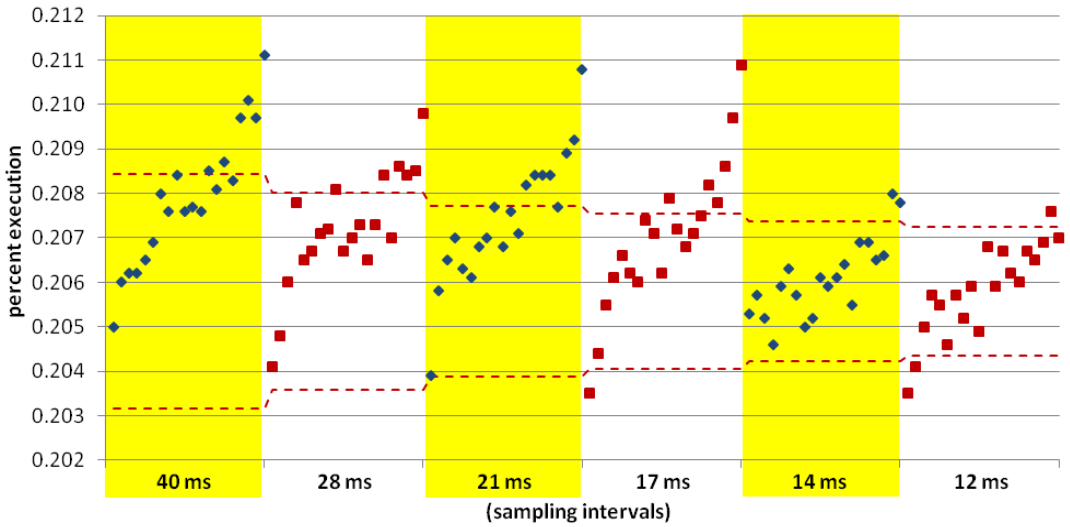

Figure C.24: Results of the percent execution of ssor_.omp_fn.2, the function taking the most execution time for the 120 runs conducted with applu, 8 threads on 8 cores, 20 runs per sampling interval.

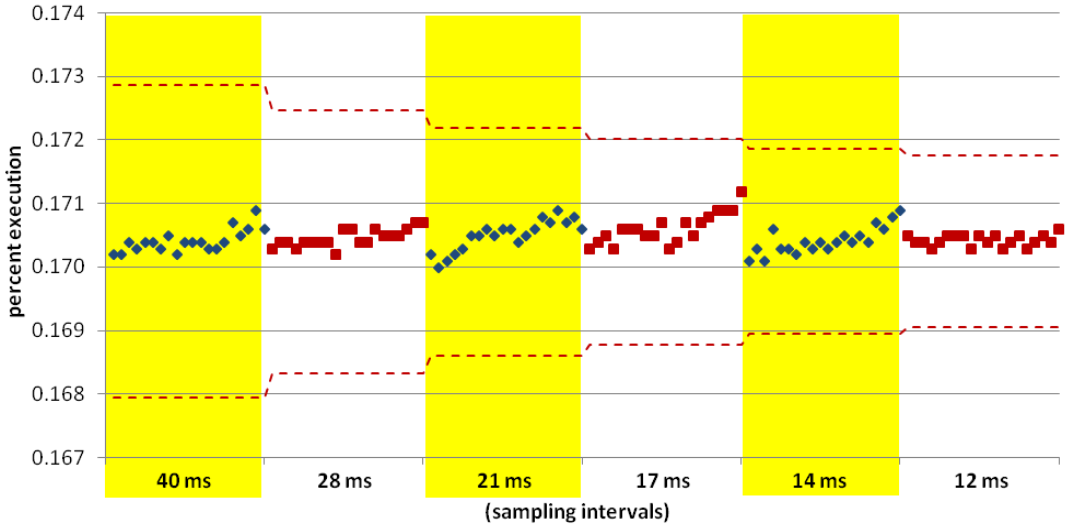

Figure C.25: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with applu, 8 threads on 8 cores, 20 runs per sampling interval.

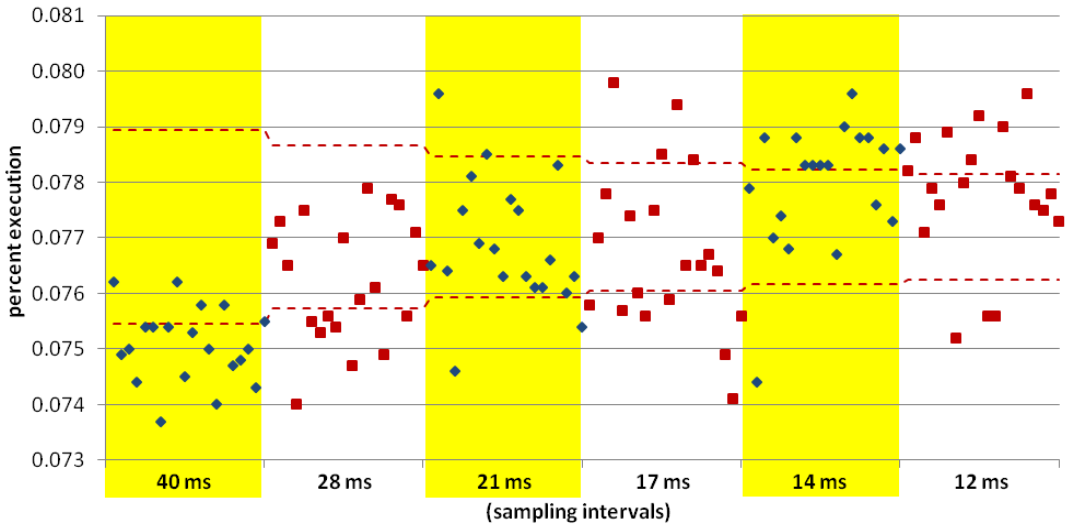

Figure C.26: Results of the percent execution of buts_, the function taking the 3rd most execution time for the 120 runs conducted with applu, 8 threads on 8 cores, 20 runs per sampling interval.

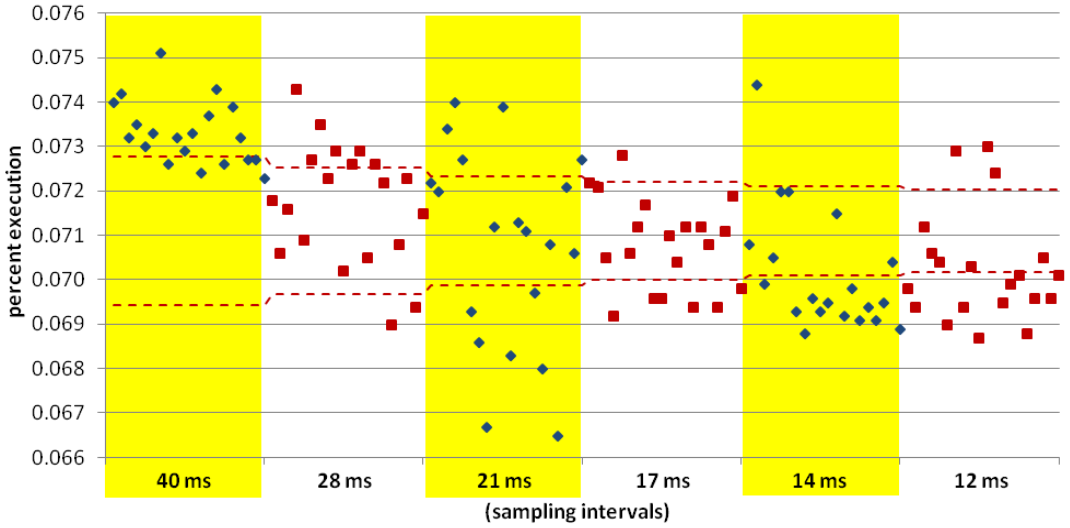

Figure C.27: Results of the percent execution of blts_, the function taking the 4th most execution time for the 120 runs conducted with applu, 8 threads on 8 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with applu, 12 threads on 12 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

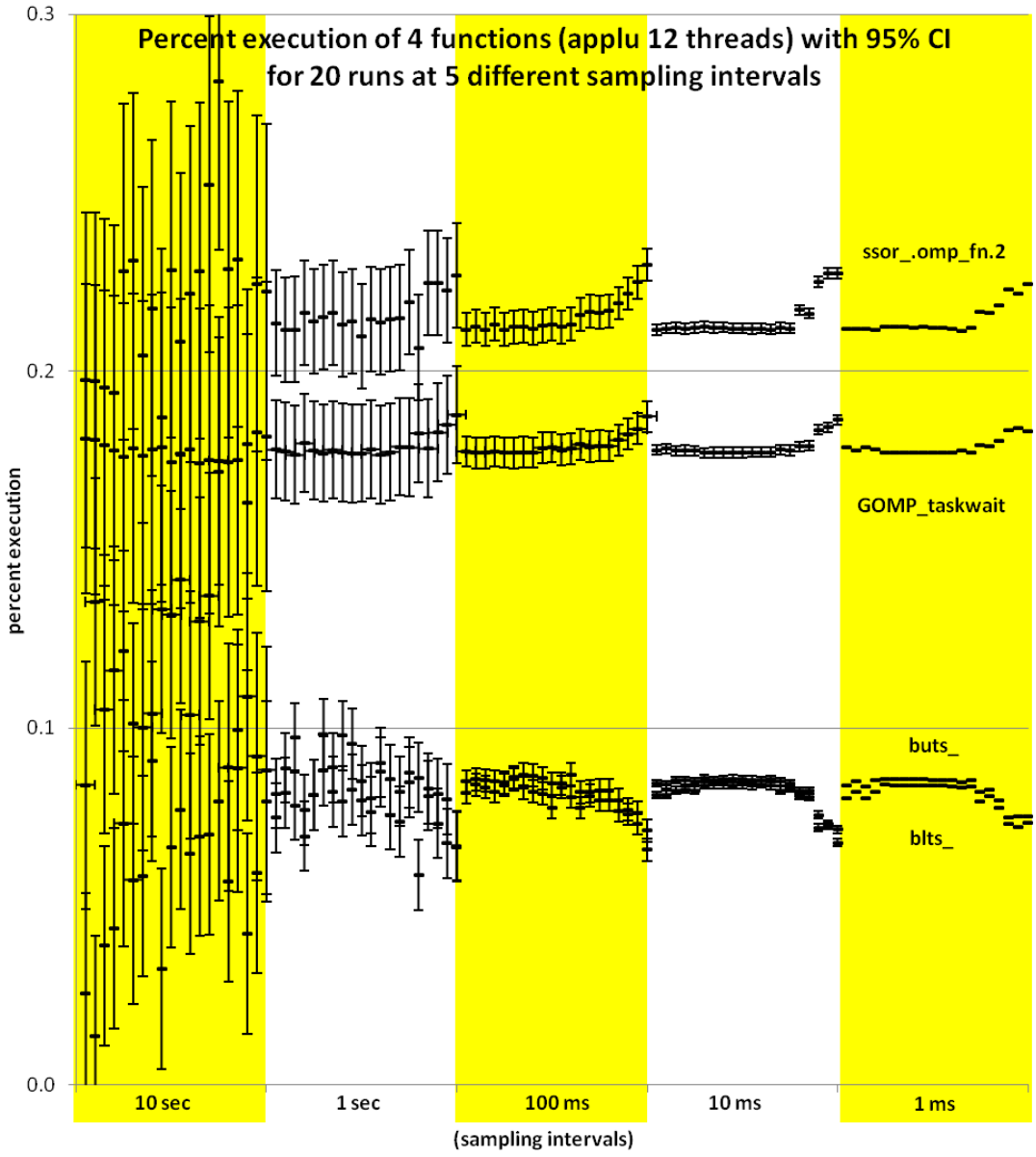

Figure C.28: Distribution of the percent execution taken by 4 different functions from 100 runs of applu with 12 threads on 12 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

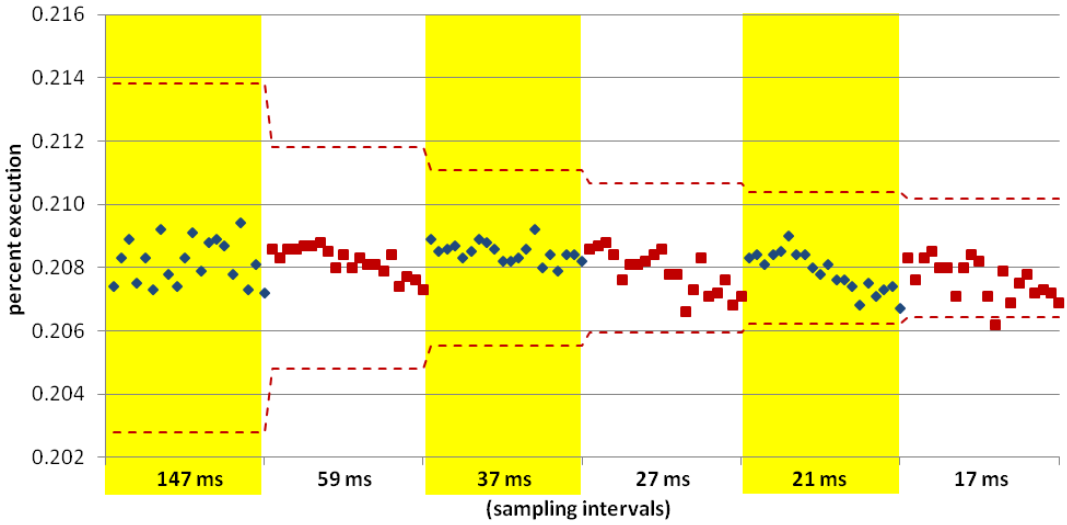

Figure C.29: Results of the percent execution of ssor_.omp_fn.2, the function taking the most execution time for the 120 runs conducted with applu, 12 threads on 12 cores, 20 runs per sampling interval.

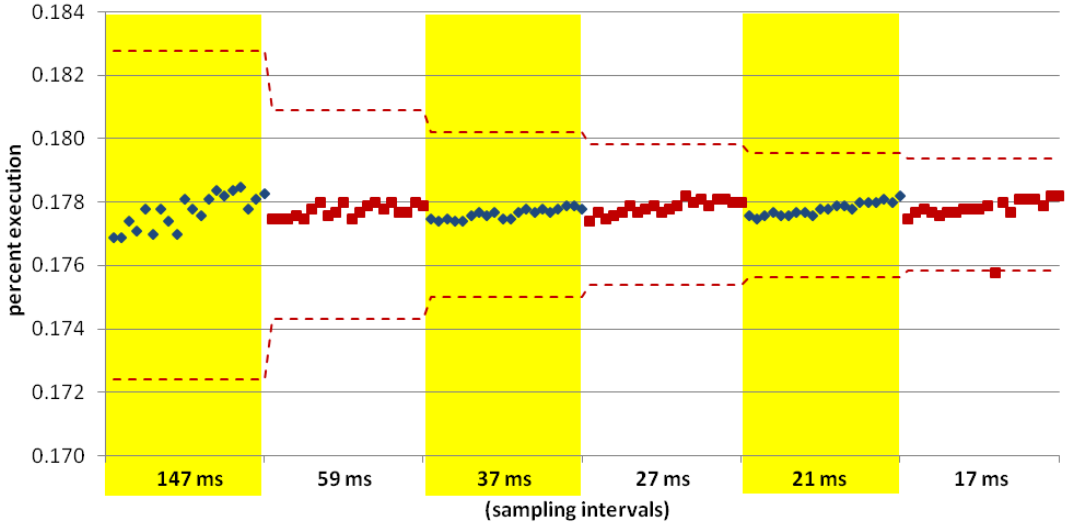

Figure C.30: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with applu, 12 threads on 12 cores, 20 runs per sampling interval.

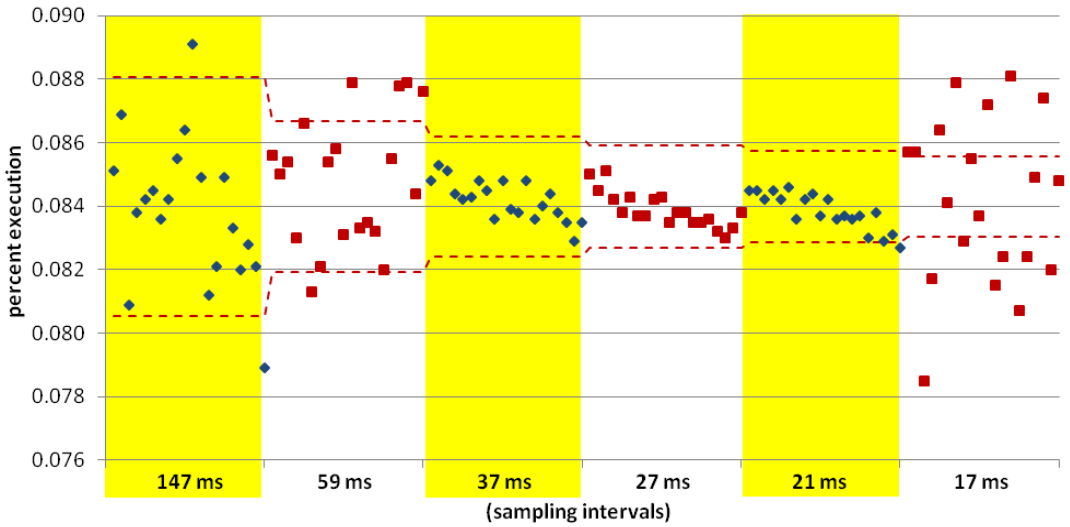

Figure C.31: Results of the percent execution of buts_, the function taking the 3rd most execution time for the 120 runs conducted with applu, 12 threads on 12 cores, 20 runs per sampling interval.

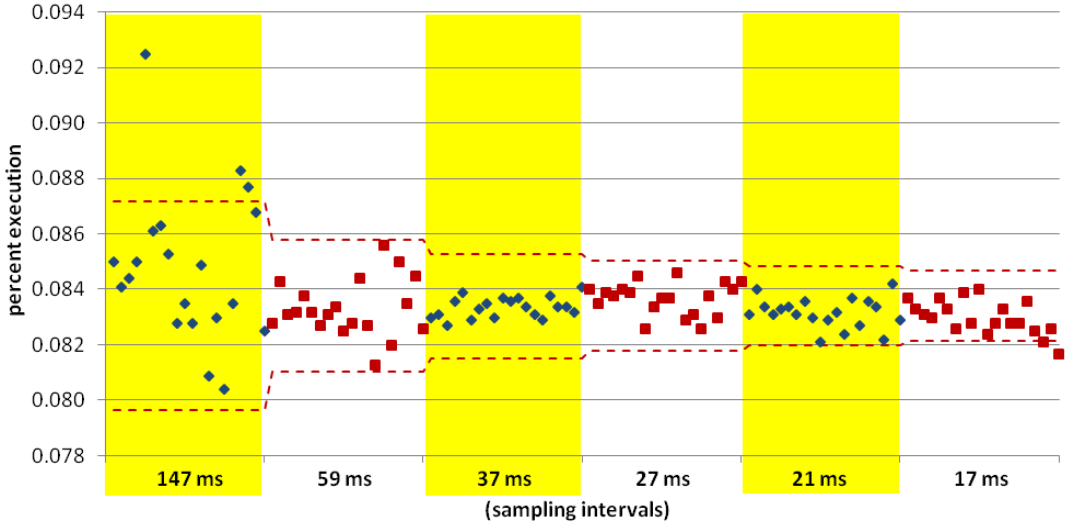

Figure C.32: Results of the percent execution of blts_, the function taking the 4th most execution time for the 120 runs conducted with applu, 12 threads on 12 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with fma3d, 4 threads on 4 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

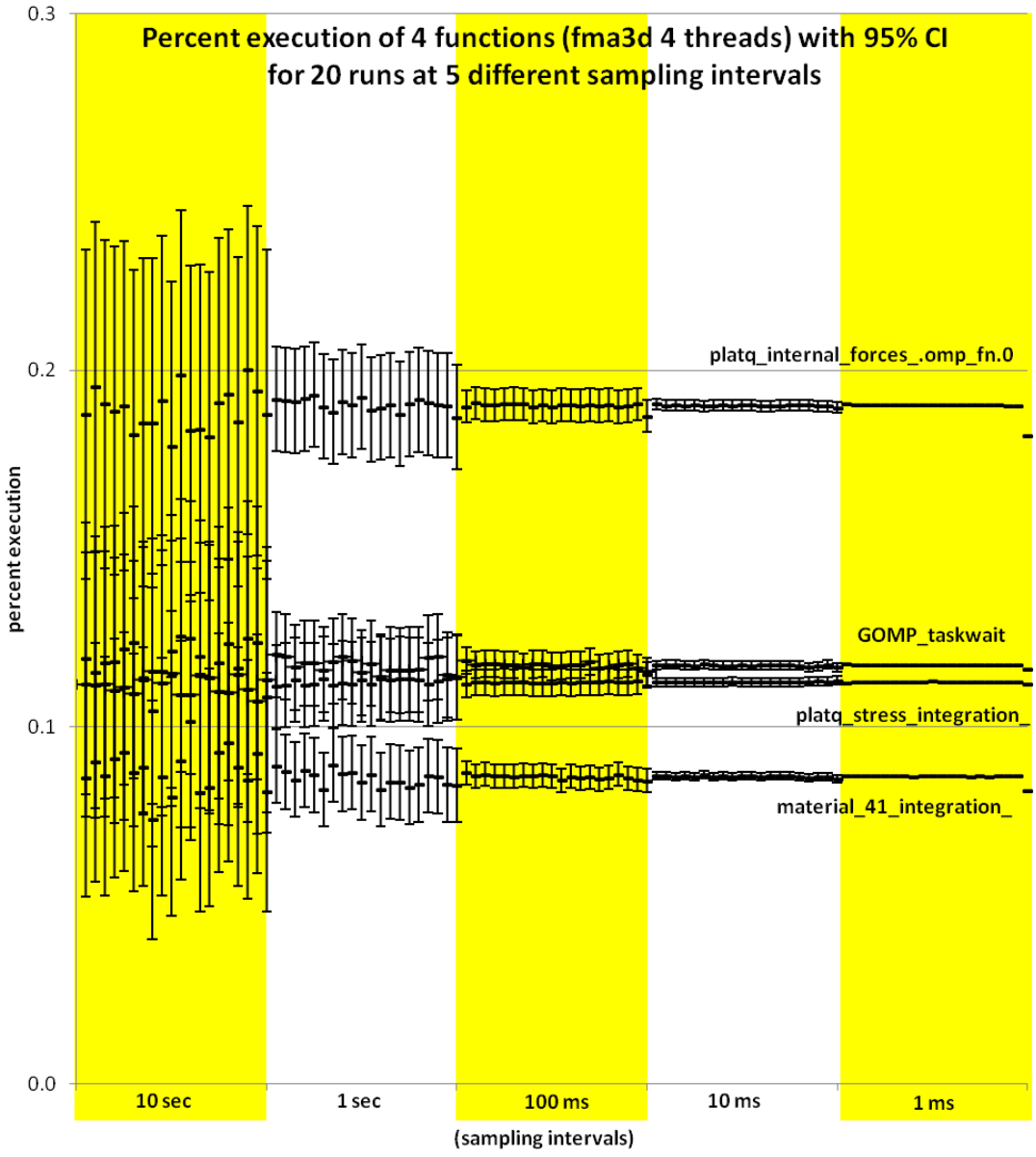

Figure C.33: Distribution of the percent execution taken by 4 different functions from 100 runs of fma3d with 4 threads on 4 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

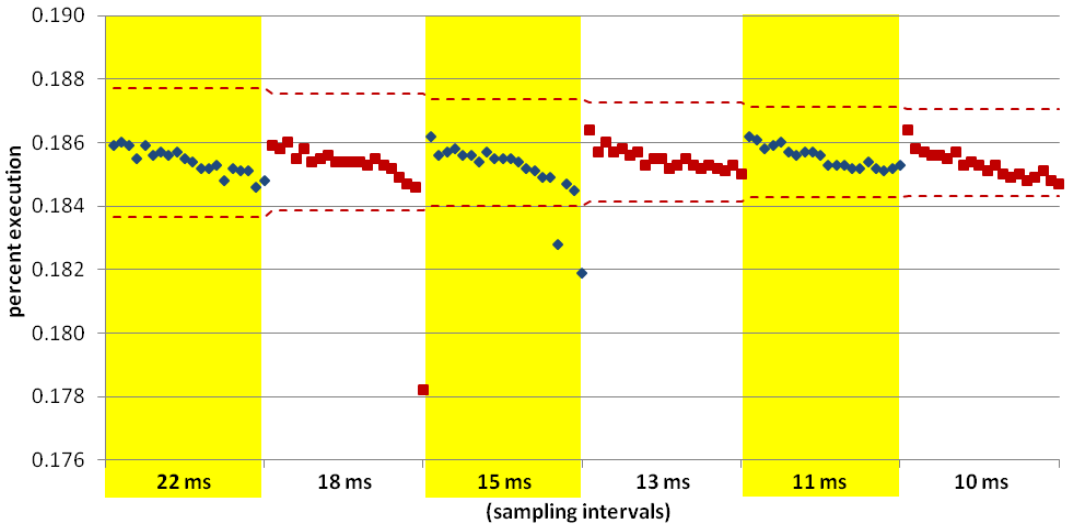

Figure C.34: Results of the percent execution of platq_internal_forces_.omp_fn.0, the function taking the most execution time for the 120 runs conducted with fma3d, 4 threads on 4 cores, 20 runs per sampling interval.

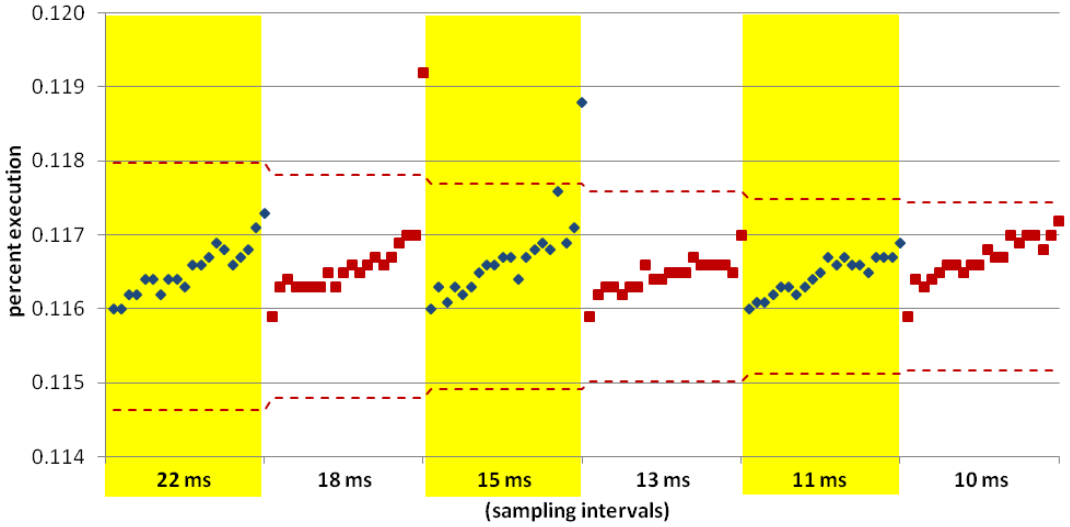

Figure C.35: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with fma3d, 4 threads on 4 cores, 20 runs per sampling interval.

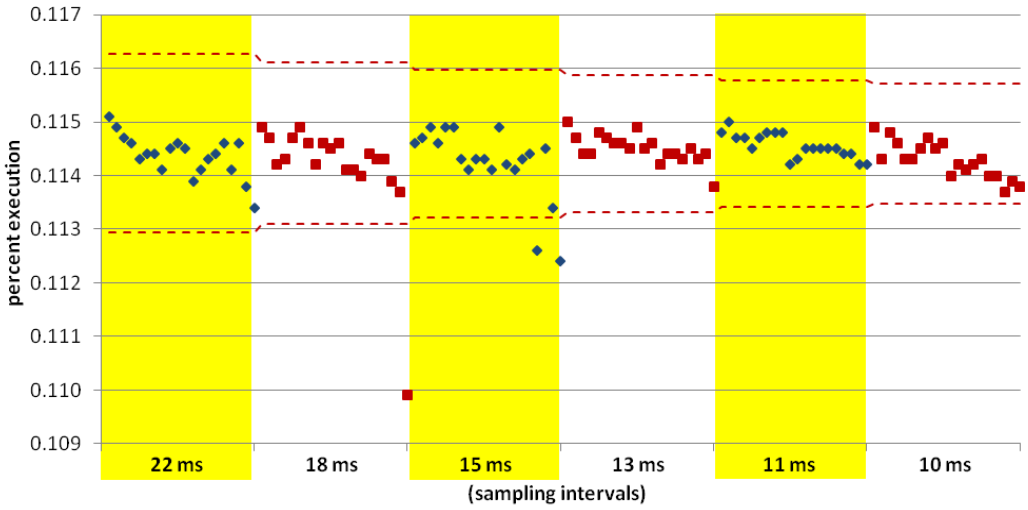

Figure C.36: Results of the percent execution of platq_stress_integration_, the function taking the 3rd most execution time for the 120 runs conducted with fma3d, 4 threads on 4 cores, 20 runs per sampling interval.

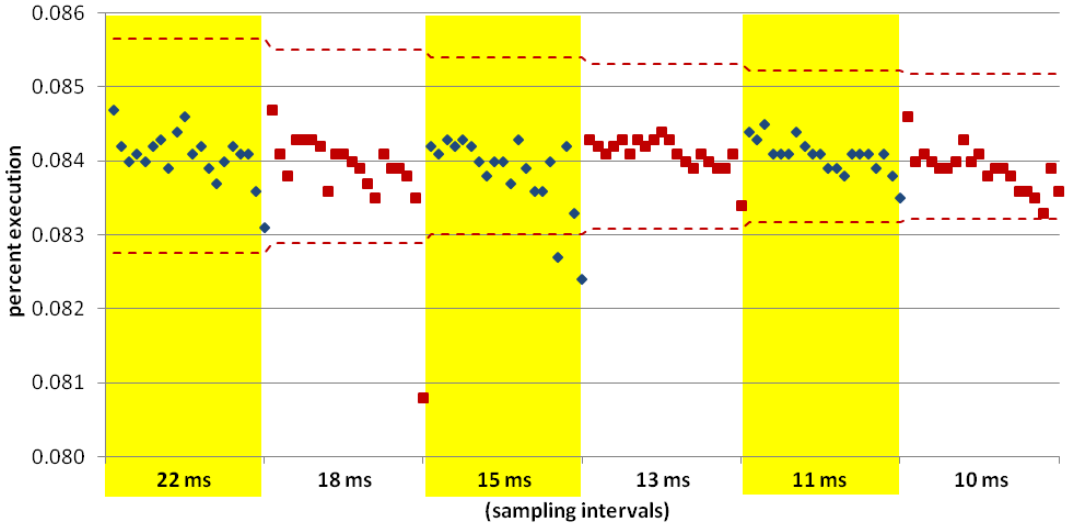

Figure C.37: Results of the percent execution of material_41_integration_, the function taking the 4th most execution time for the 120 runs conducted with fma3d, 4 threads on 4 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with fma3d, 8 threads on 8 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

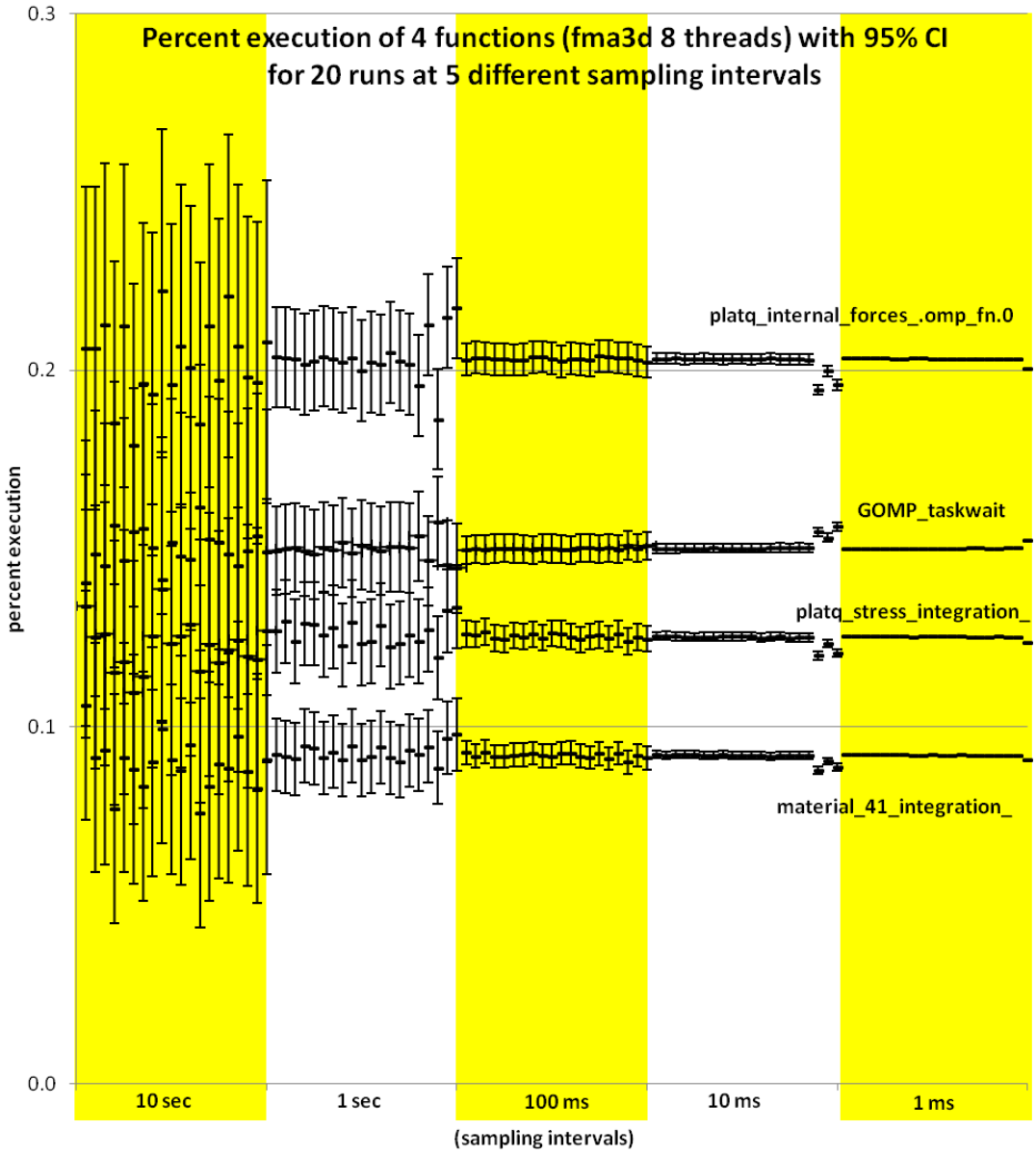

Figure C.38: Distribution of the percent execution taken by 4 different functions from 100 runs of fma3d with 8 threads on 8 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

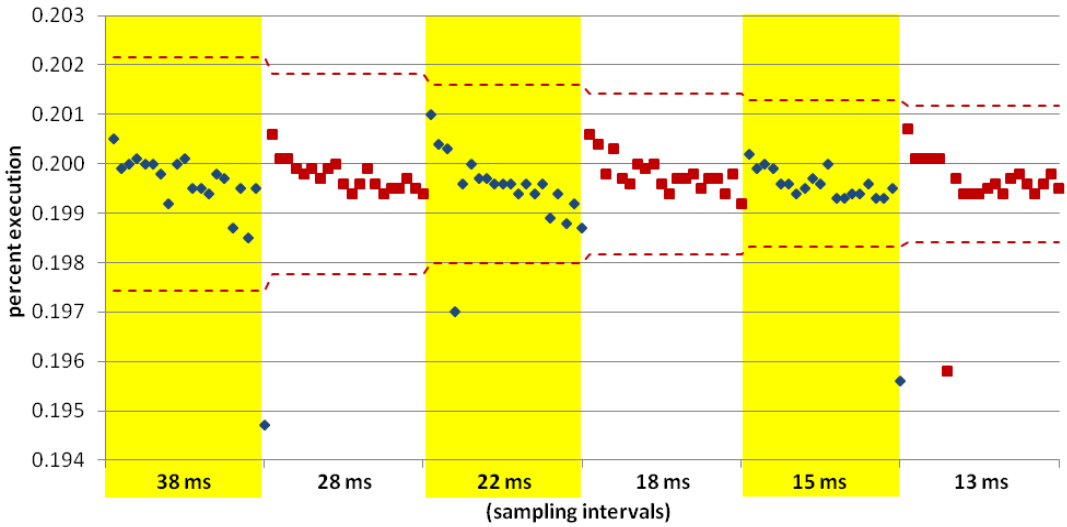

Figure C.39: Results of the percent execution of platq_internal_forces_.omp_fn.0, the function taking the most execution time for the 120 runs conducted with fma3d, 8 threads on 8 cores, 20 runs per sampling interval.

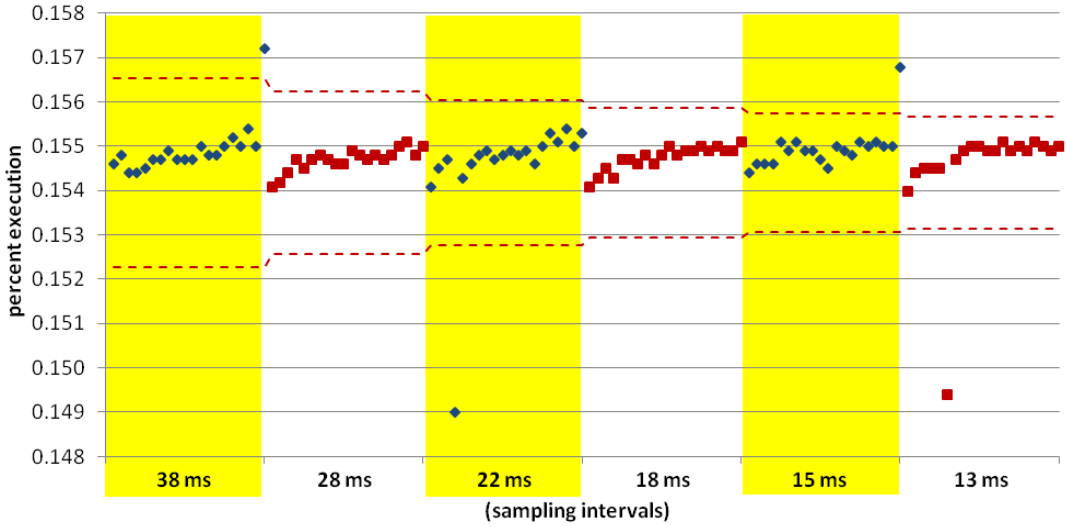

Figure C.40: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with fma3d, 8 threads on 8 cores, 20 runs per sampling interval.

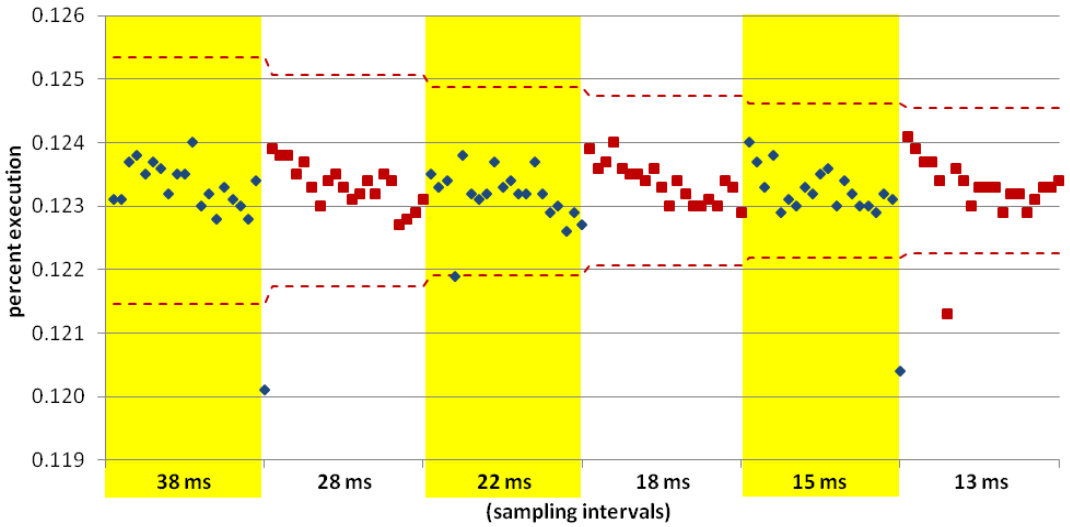

Figure C.41: Results of the percent execution of platq_stress_integration_, the function taking the 3rd most execution time for the 120 runs conducted with fma3d, 8 threads on 8 cores, 20 runs per sampling interval.

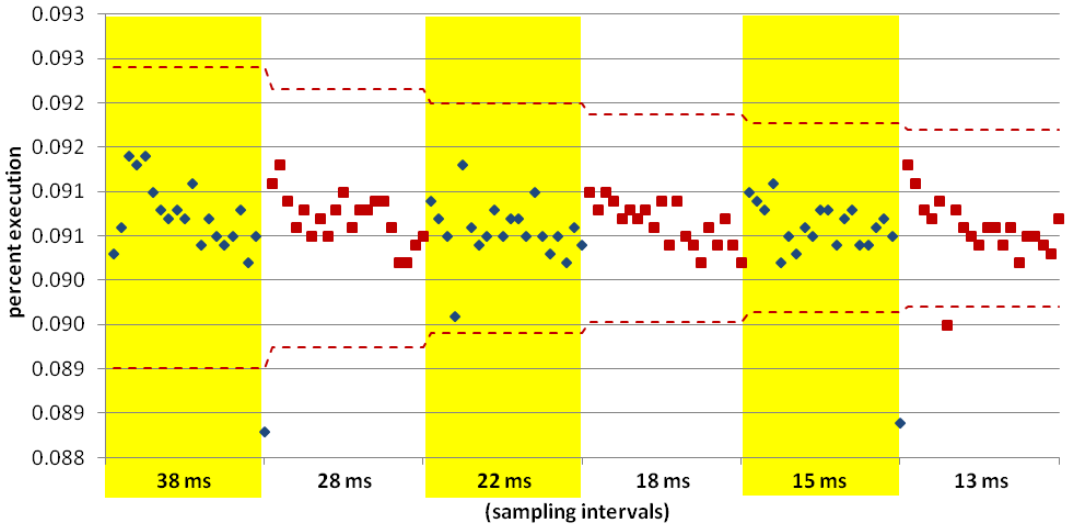

Figure C.42: Results of the percent execution of material_41_integration_, the function taking the 4th most execution time for the 120 runs conducted with fma3d, 8 threads on 8 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with fma3d, 12 threads on 12 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

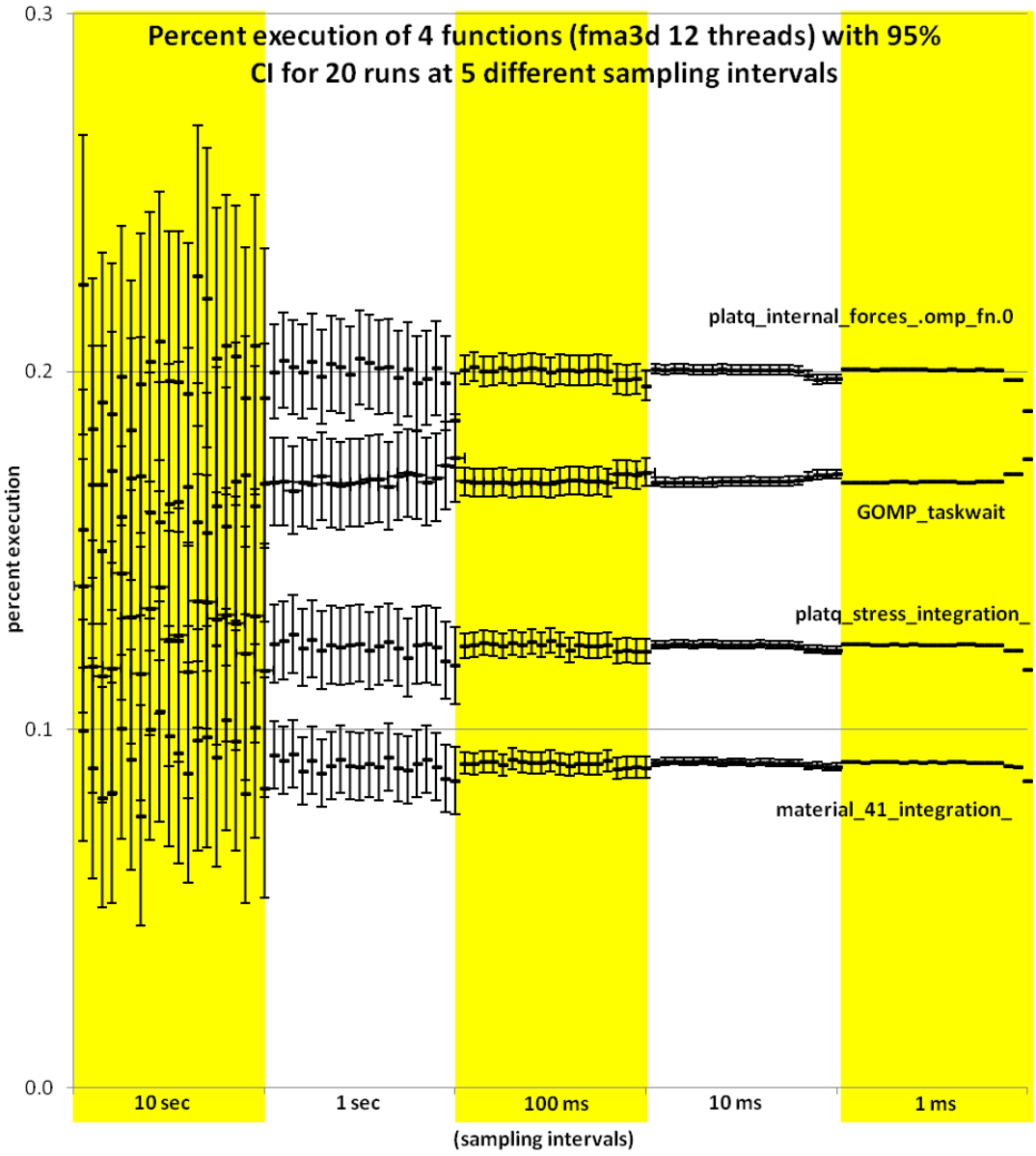

Figure C.43: Distribution of the percent execution taken by 4 different functions from 100 runs of fma3d with 12 threads on 12 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

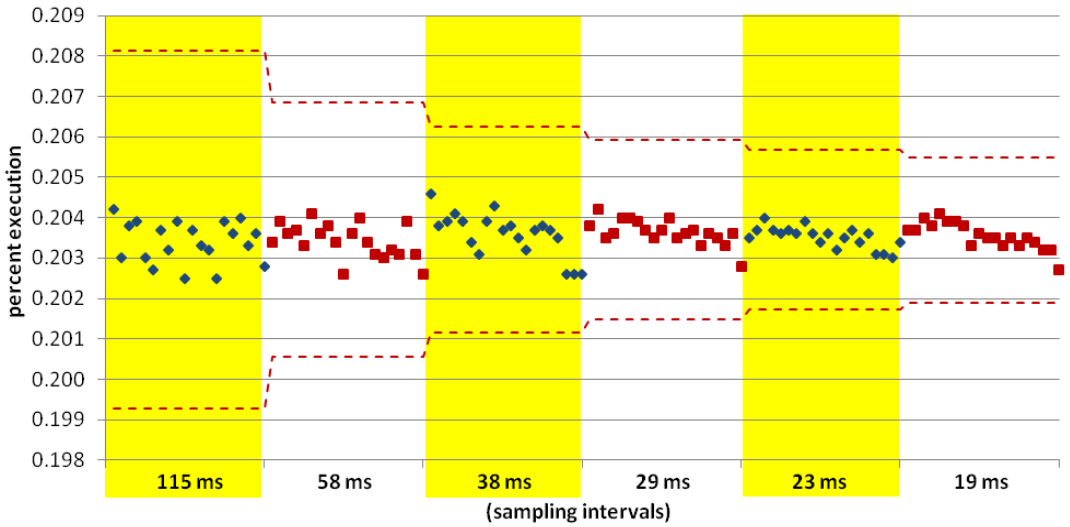

Figure C.44: Results of the percent execution of platq_internal_forces_.omp_fn.0, the function taking the most execution time for the 120 runs conducted with fma3d, 12 threads on 12 cores, 20 runs per sampling interval.

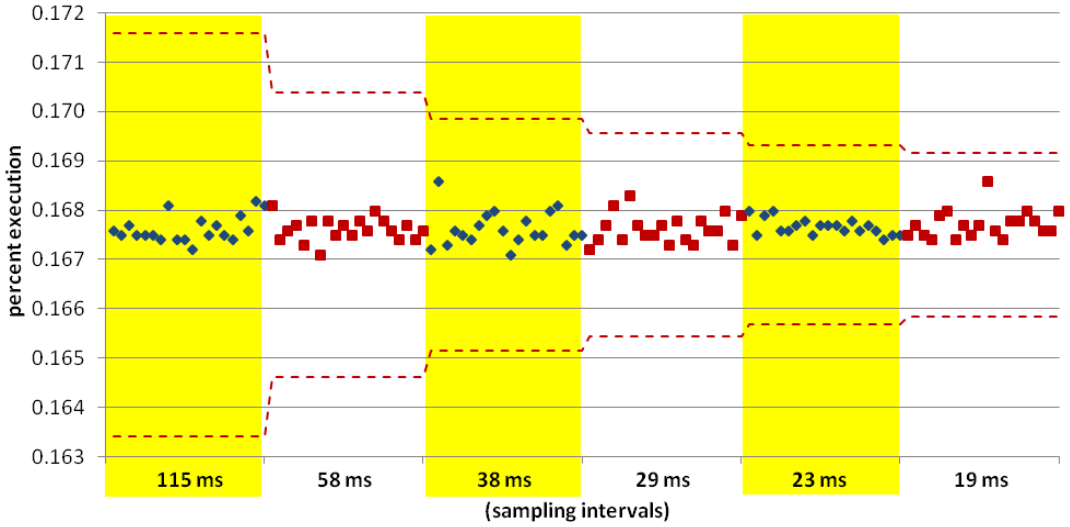

Figure C.45: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with fma3d, 12 threads on 12 cores, 20 runs per sampling interval.

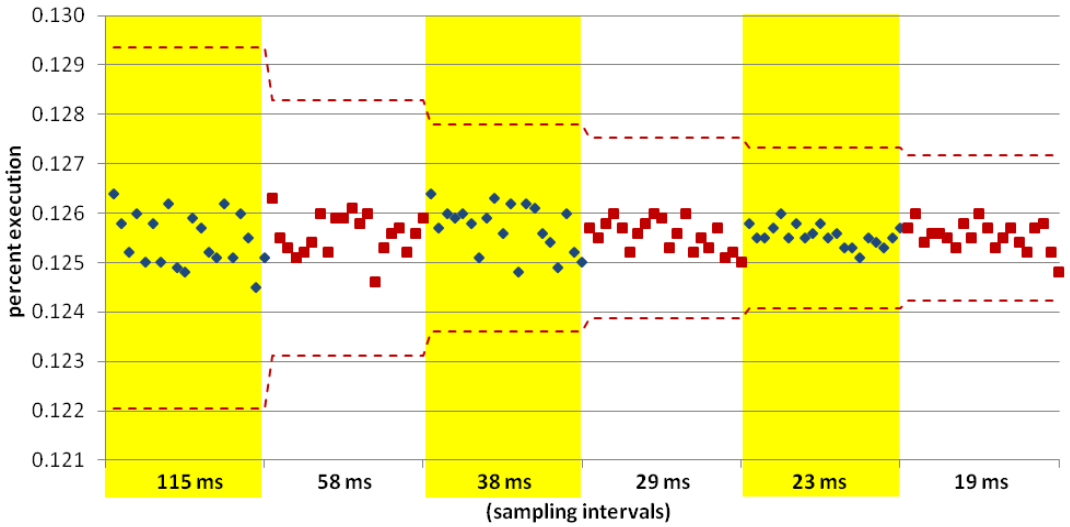

Figure C.46: Results of the percent execution of platq_stress_integration_, the function taking the 3rd most execution time for the 120 runs conducted with fma3d, 12 threads on 12 cores, 20 runs per sampling interval.

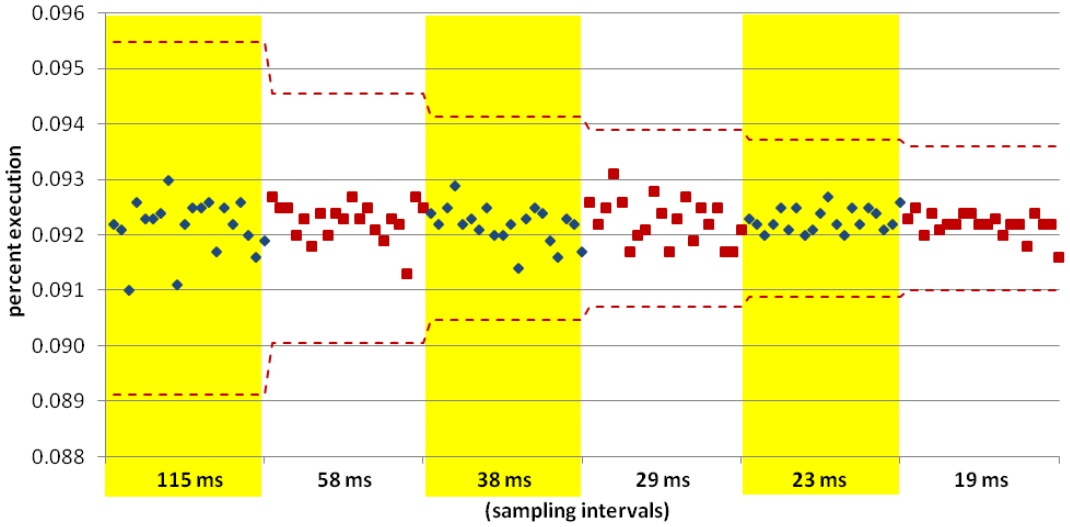

Figure C.47: Results of the percent execution of material_41_integration_, the function taking the 4th most execution time for the 120 runs conducted with fma3d, 12 threads on 12 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with swim, 4 threads on 4 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

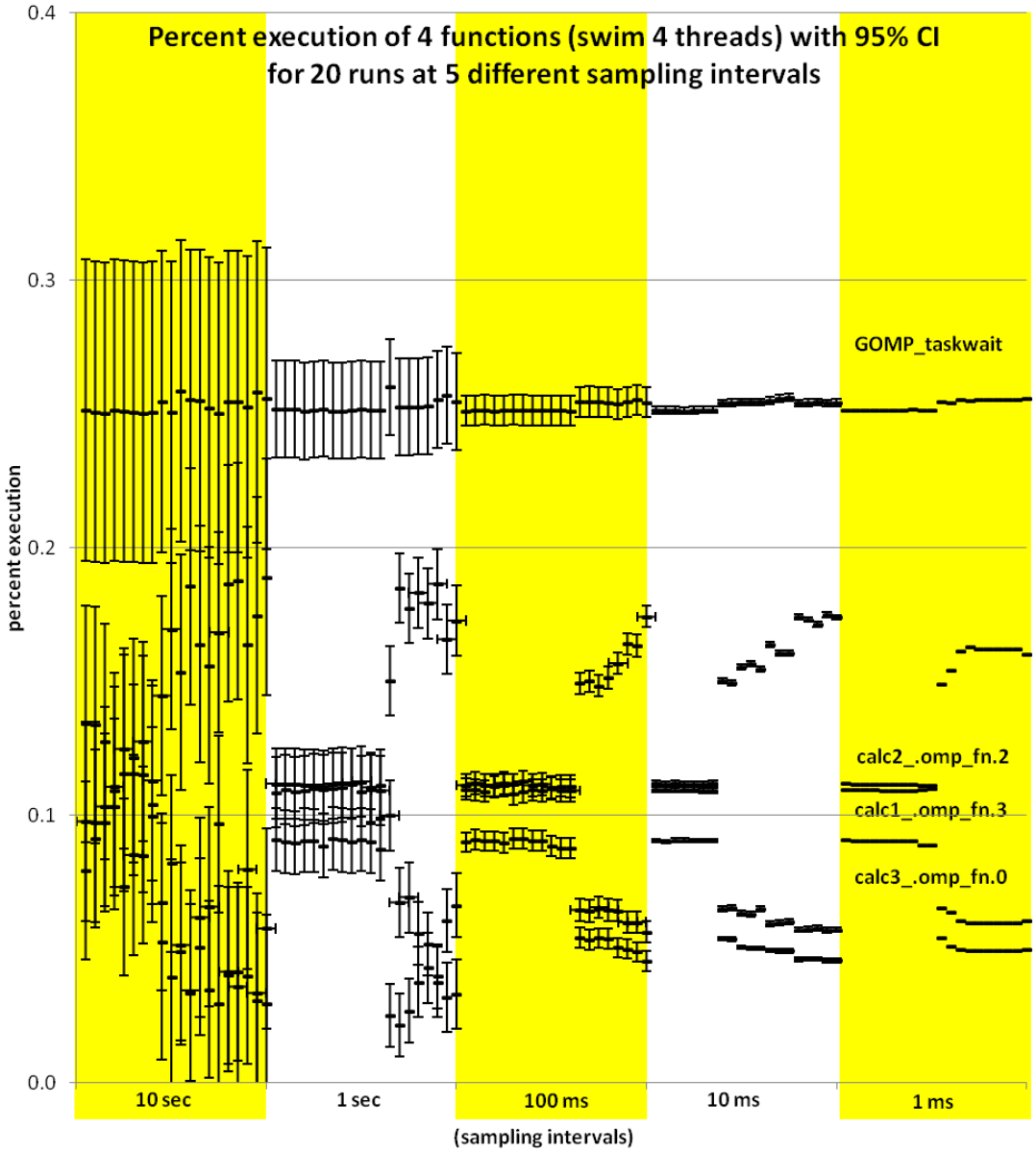

Figure C.48: Distribution of the percent execution taken by 4 different functions from 100 runs of swim with 4 threads on 4 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

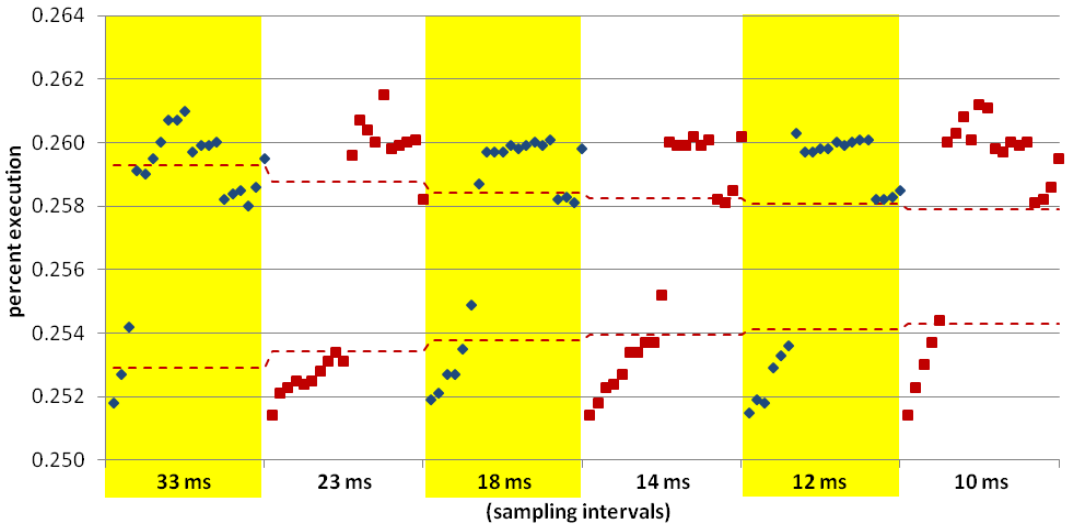

Figure C.49: Results of the percent execution of GOMP_taskwait, the function taking the most execution time for the 120 runs conducted with swim, 4 threads on 4 cores, 20 runs per sampling interval.

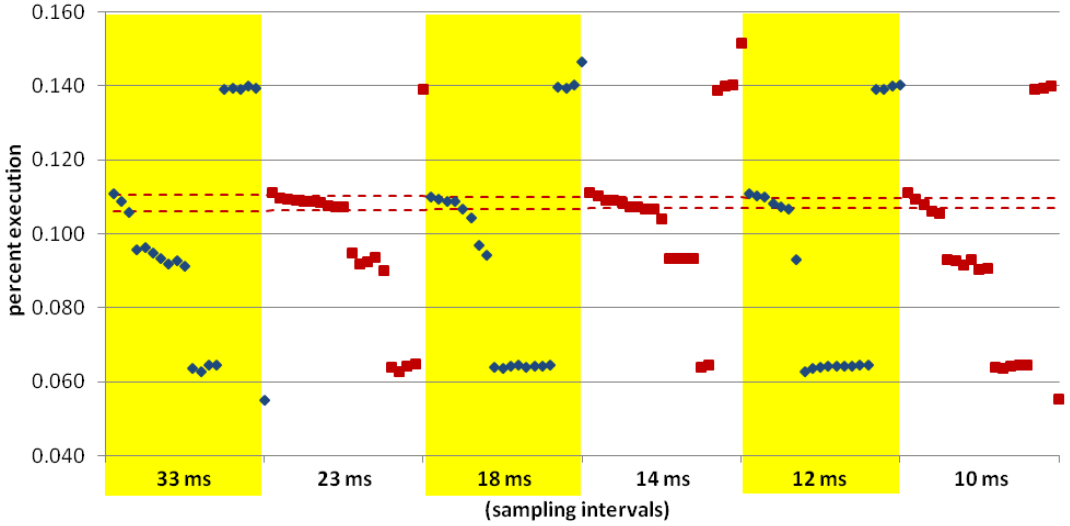

Figure C.50: Results of the percent execution of calc2_.omp_fn.2, the function taking the 2nd most execution time for the 120 runs conducted with swim, 4 threads on 4 cores, 20 runs per sampling interval.

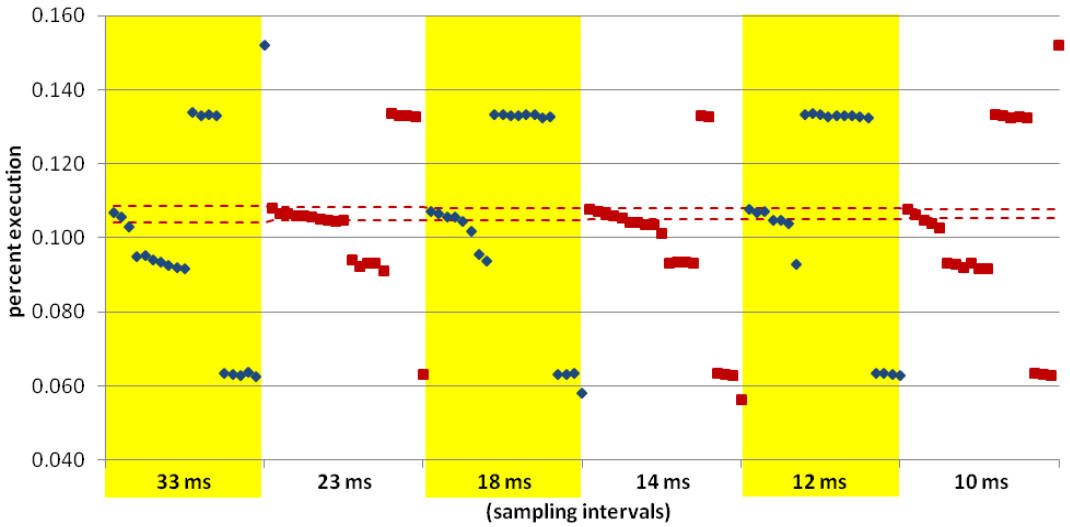

Figure C.51: Results of the percent execution of calc1_.omp_fn.3, the function taking the 3rd most execution time for the 120 runs conducted with swim, 4 threads on 4 cores, 20 runs per sampling interval.

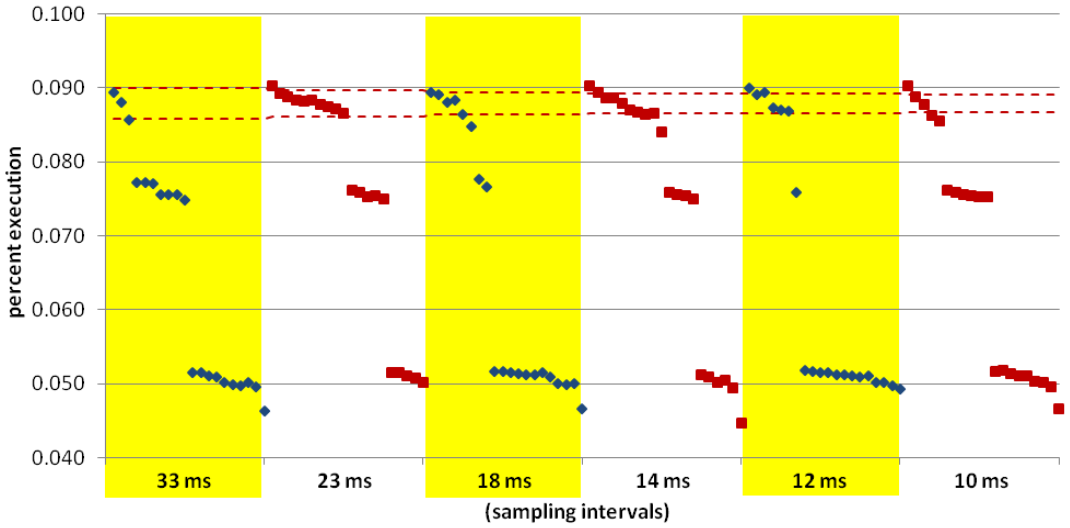

Figure C.52: Results of the percent execution of calc3_.omp_fn.0, the function taking the 4th most execution time for the 120 runs conducted with swim, 4 threads on 4 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with swim, 8 threads on 8 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

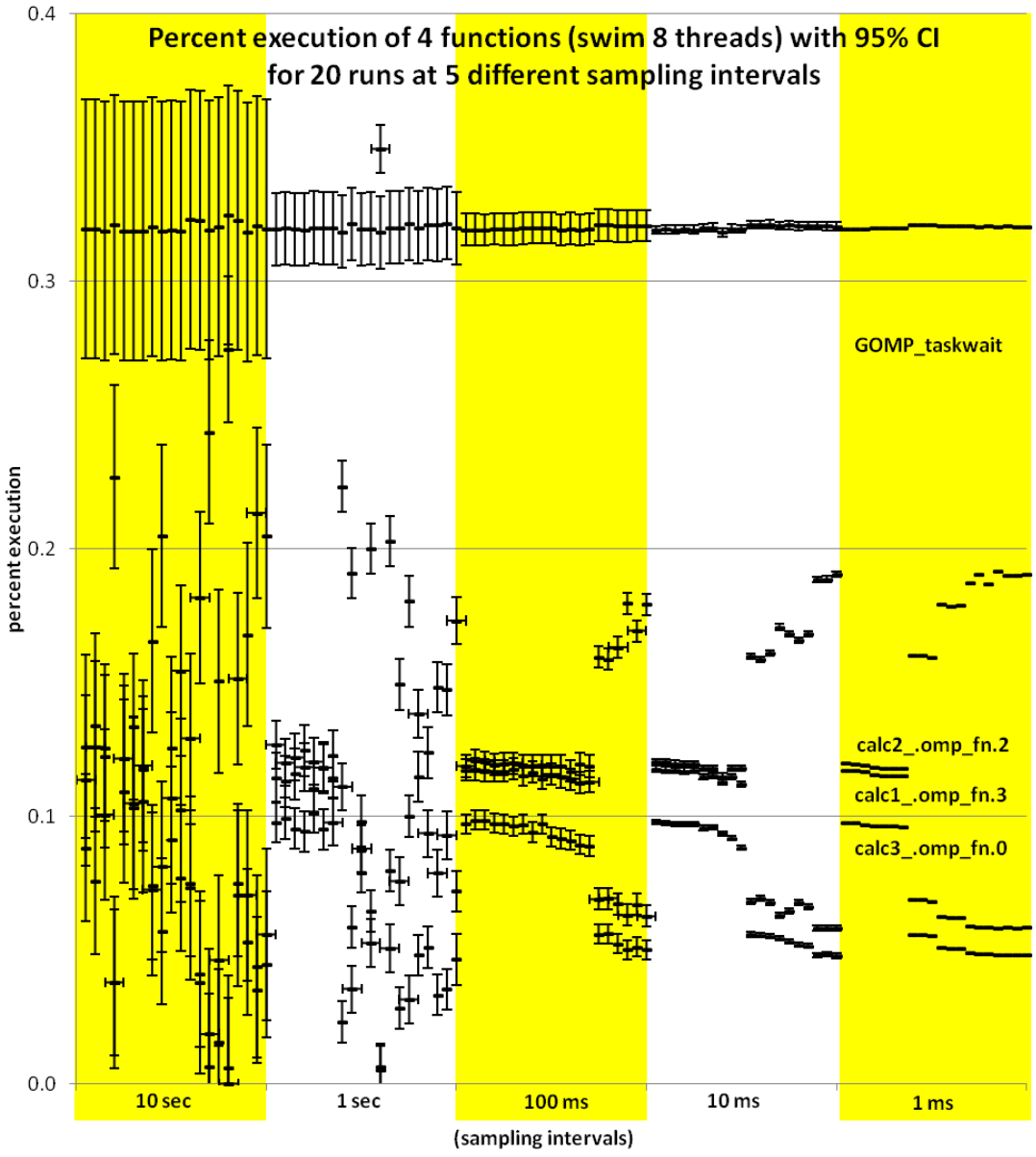

Figure C.53: Distribution of the percent execution taken by 4 different functions from 100 runs of swim with 8 threads on 8 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

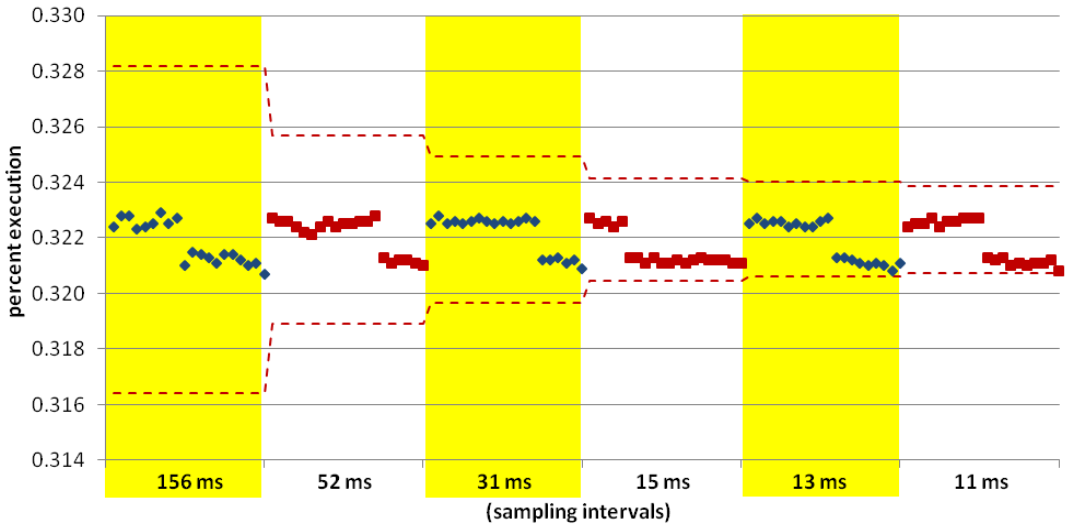

Figure C.54: Results of the percent execution of GOMP_taskwait, the function taking the most execution time for the 120 runs conducted with swim, 8 threads on 8 cores, 20 runs per sampling interval.

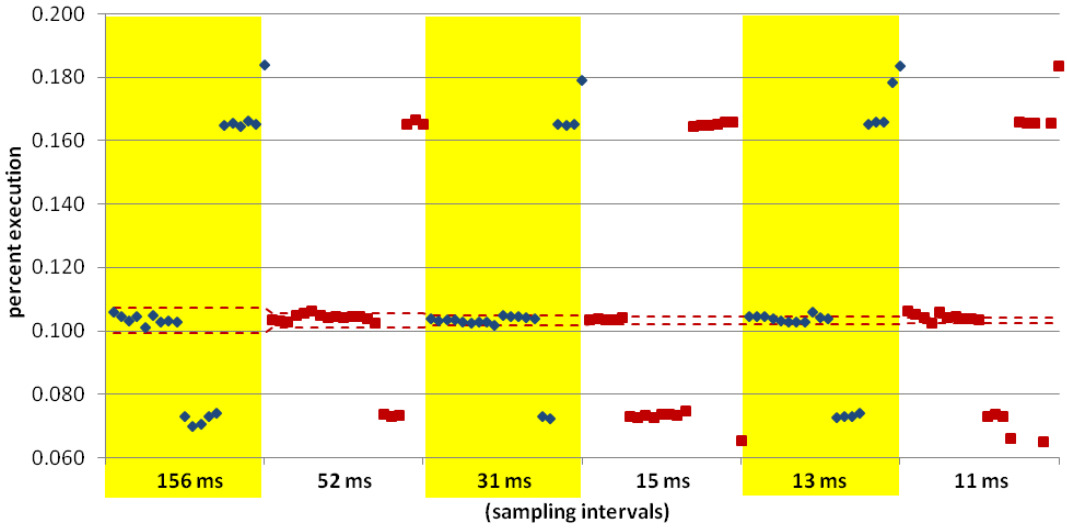

Figure C.55: Results of the percent execution of calc2_.omp_fn.2, the function taking the 2nd most execution time for the 120 runs conducted with swim, 8 threads on 8 cores, 20 runs per sampling interval.

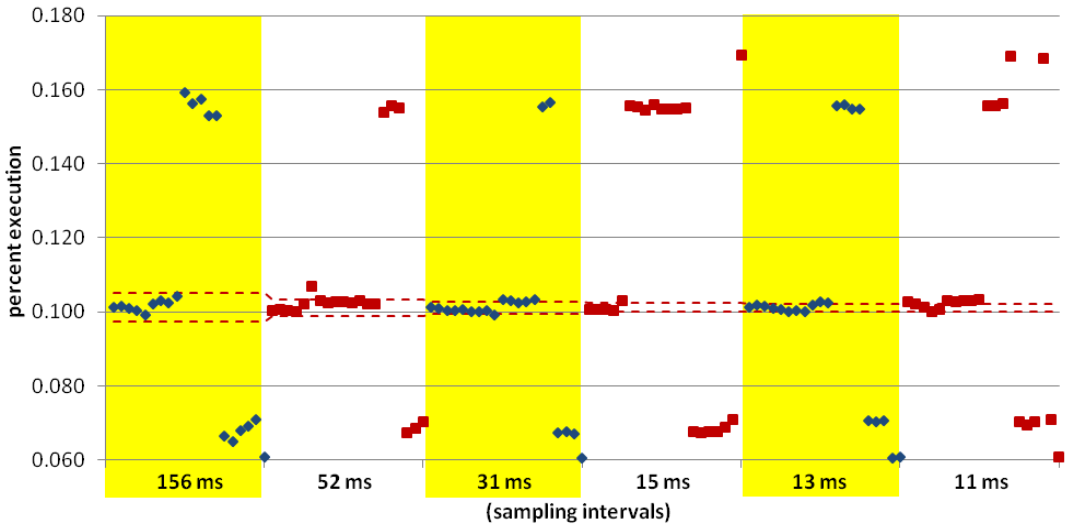

Figure C.56: Results of the percent execution of calc1_.omp_fn.3, the function taking the 3rd most execution time for the 120 runs conducted with swim, 8 threads on 8 cores, 20 runs per sampling interval.

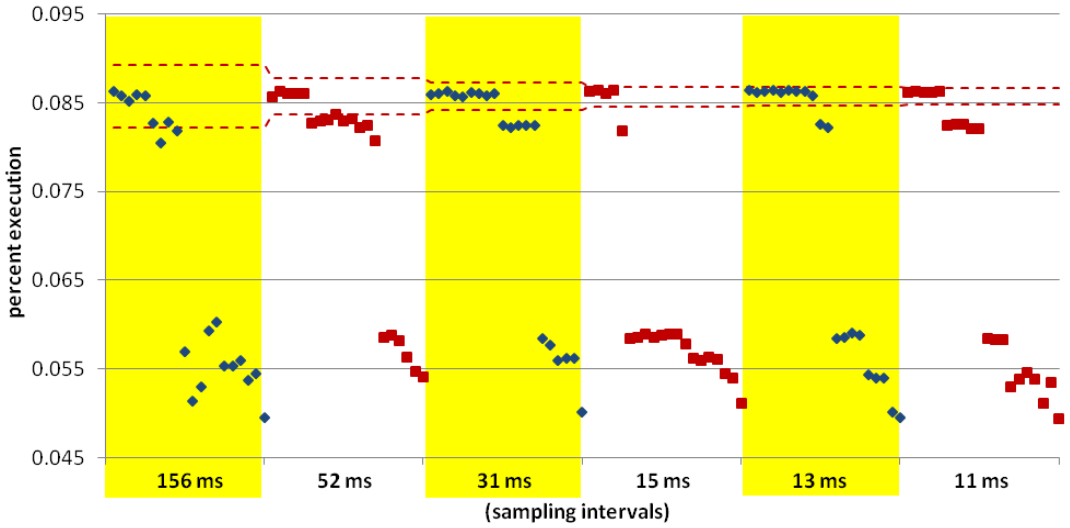

Figure C.57: Results of the percent execution of calc3_.omp_fn.0, the function taking the 4th most execution time for the 120 runs conducted with swim, 8 threads on 8 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with swim, 12 threads on 12 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

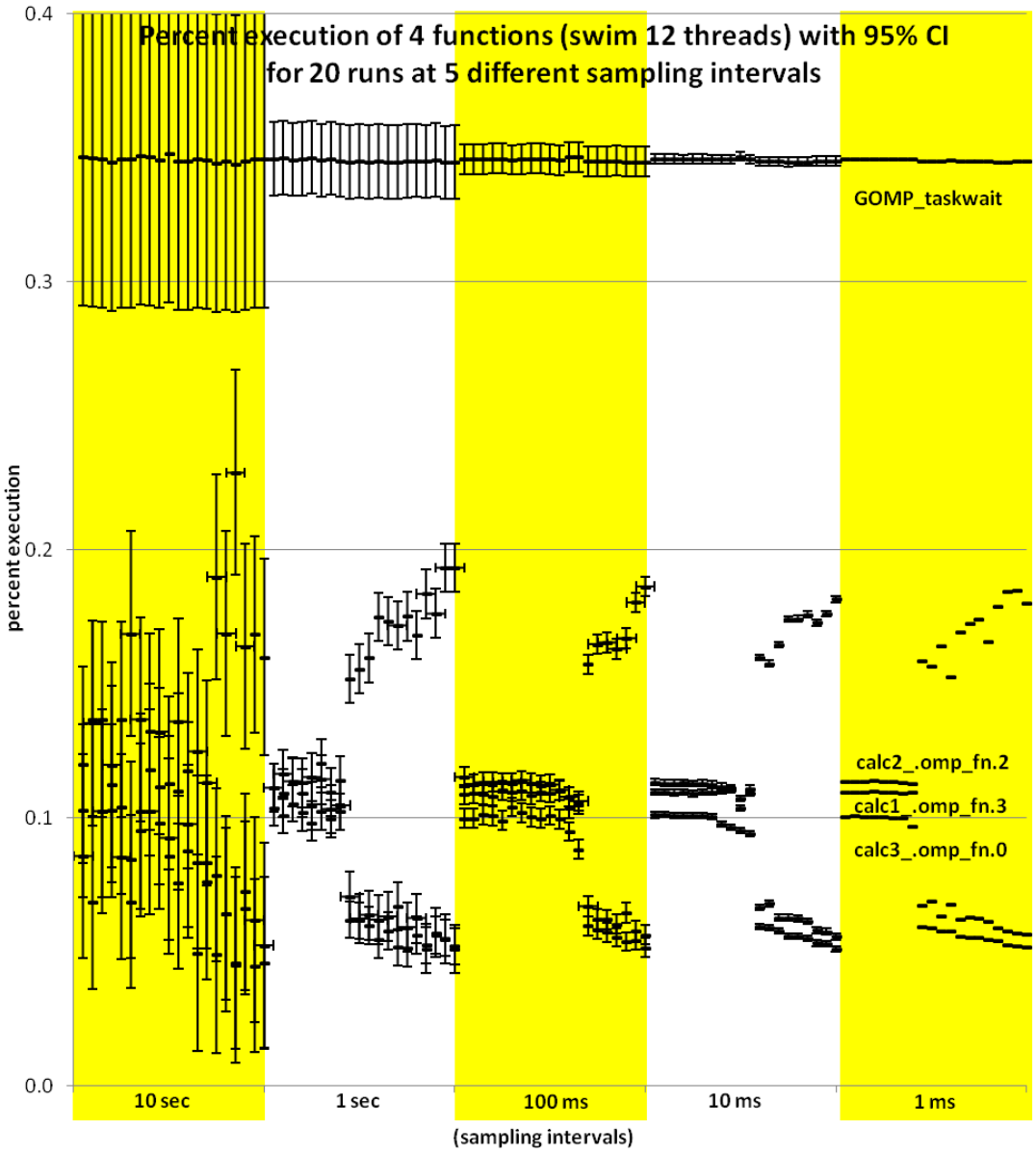

Figure C.58: Distribution of the percent execution taken by 4 different functions from 100 runs of swim with 12 threads on 12 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

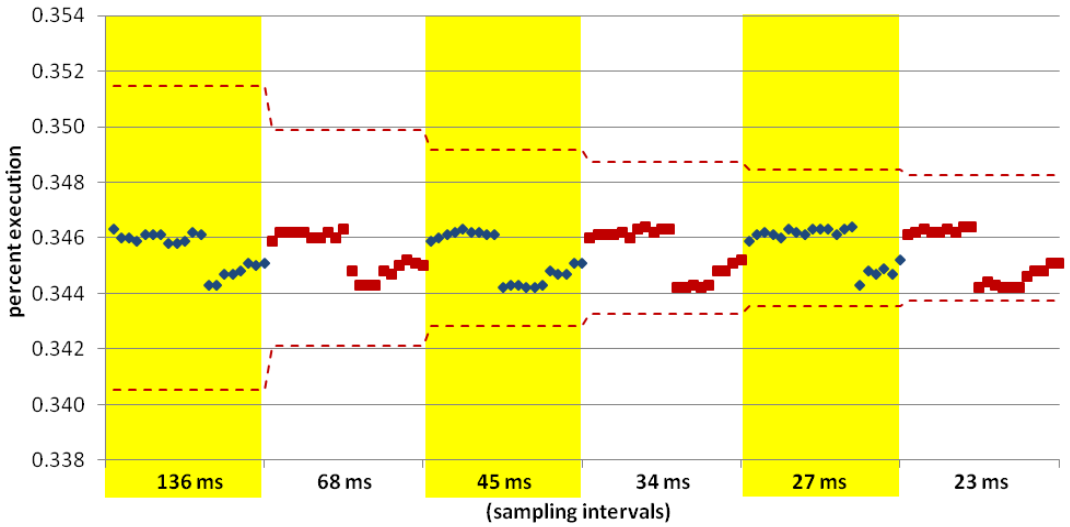

Figure C.59: Results of the percent execution of GOMP_taskwait, the function taking the most execution time for the 120 runs conducted with swim, 12 threads on 12 cores, 20 runs per sampling interval.

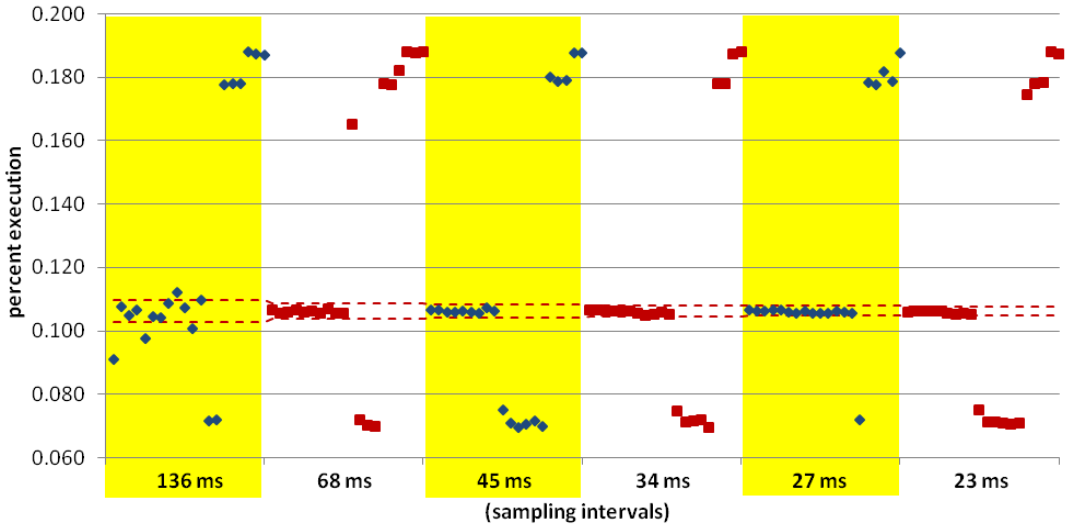

Figure C.60: Results of the percent execution of calc2_.omp_fn.2, the function taking the 2nd most execution time for the 120 runs conducted with swim, 12 threads on 12 cores, 20 runs per sampling interval.

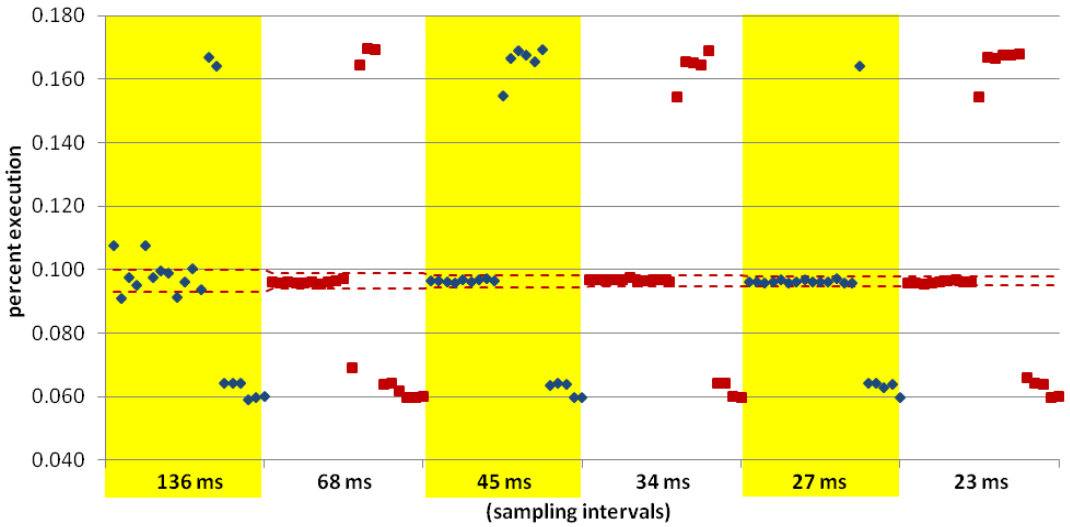

Figure C.61: Results of the percent execution of calc1_.omp_fn.3, the function taking the 3rd most execution time for the 120 runs conducted with swim, 12 threads on 12 cores, 20 runs per sampling interval.

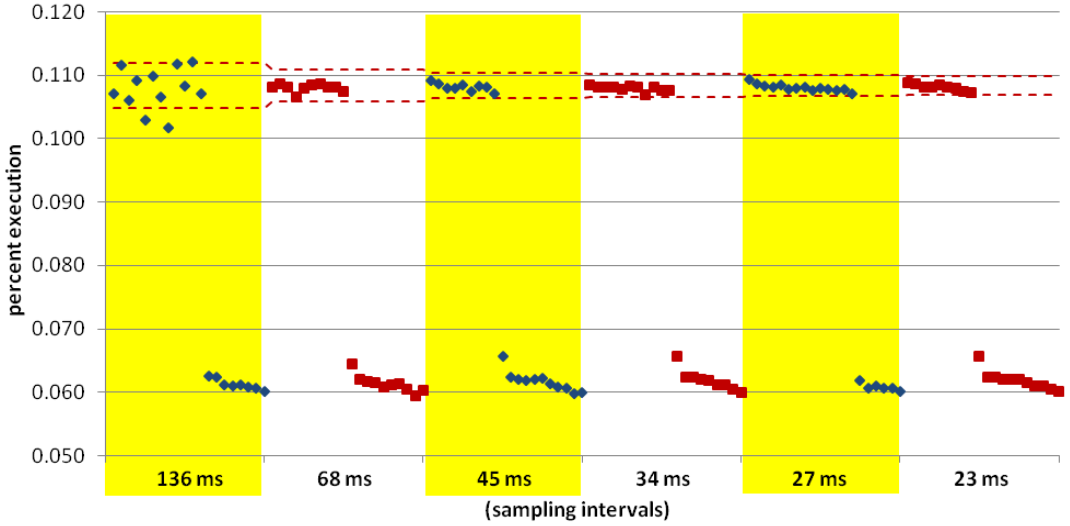

Figure C.62: Results of the percent execution of calc3_.omp_fn.0, the function taking the 4th most execution time for the 120 runs conducted with swim, 12 threads on 12 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with wupwise, 4 threads on 4 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

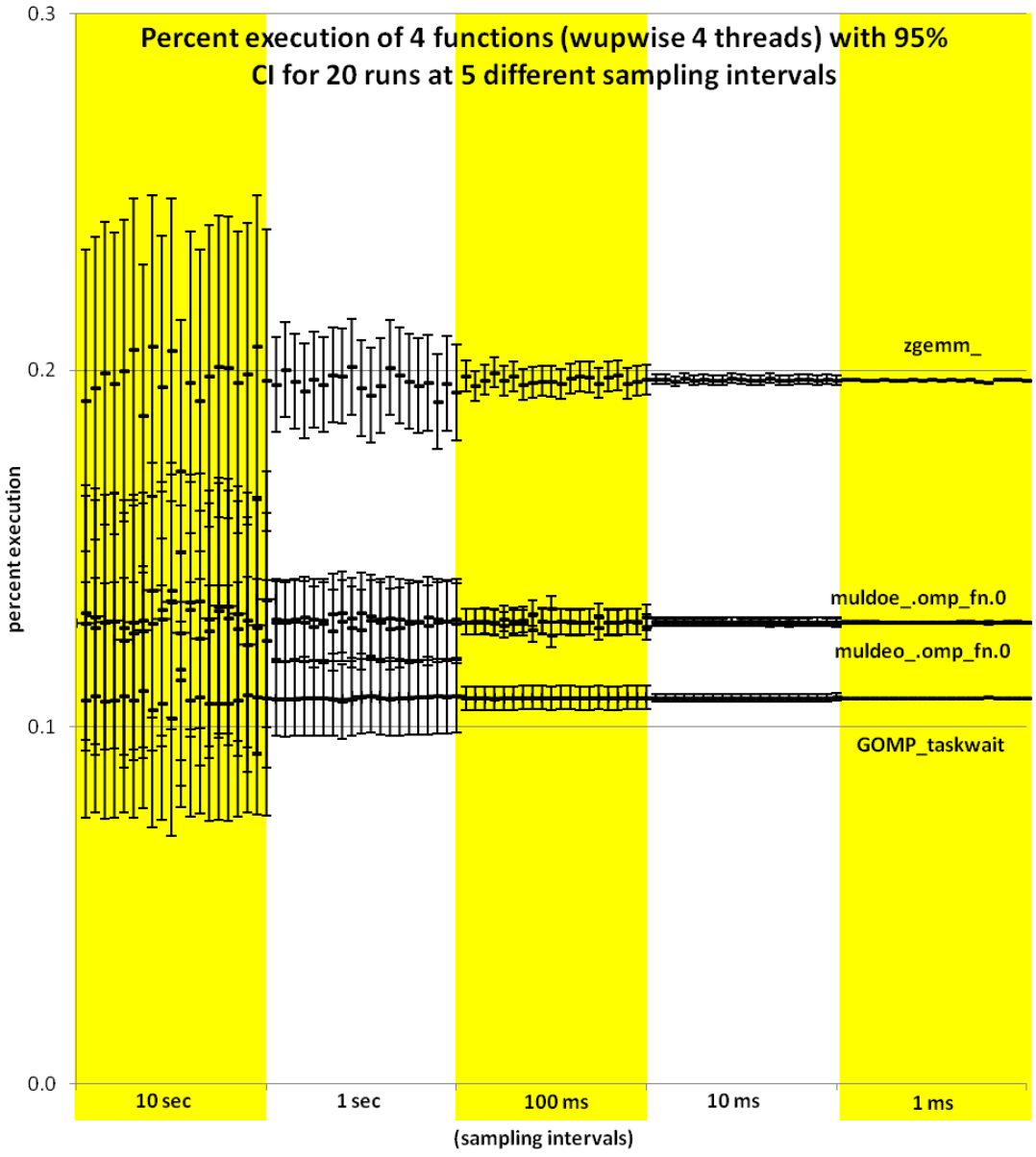

Figure C.63: Distribution of the percent execution taken by 4 different functions from 100 runs of wupwise with 4 threads on 4 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

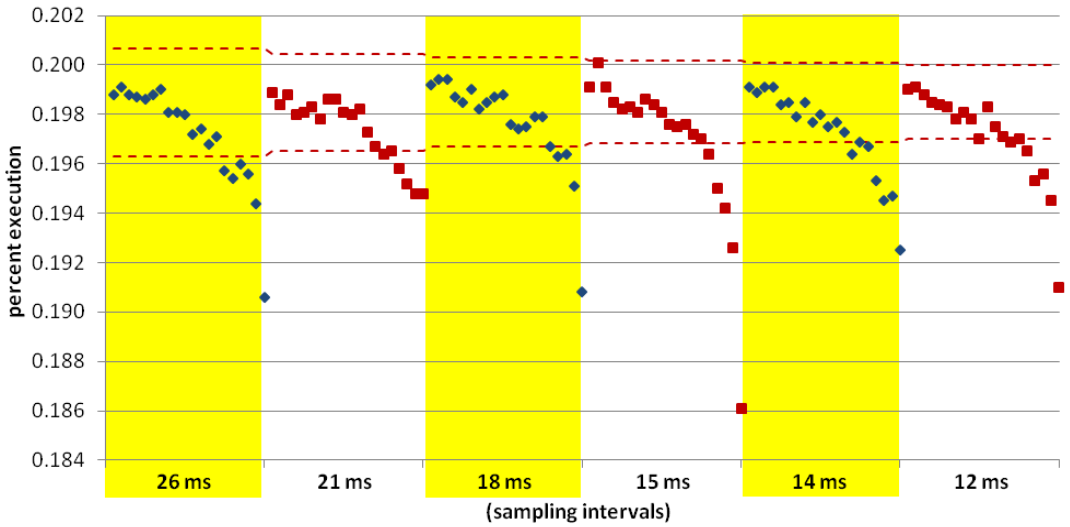

Figure C.64: Results of the percent execution of zgemm_, the function taking the most execution time for the 120 runs conducted with wupwise, 4 threads on 4 cores, 20 runs per sampling interval.

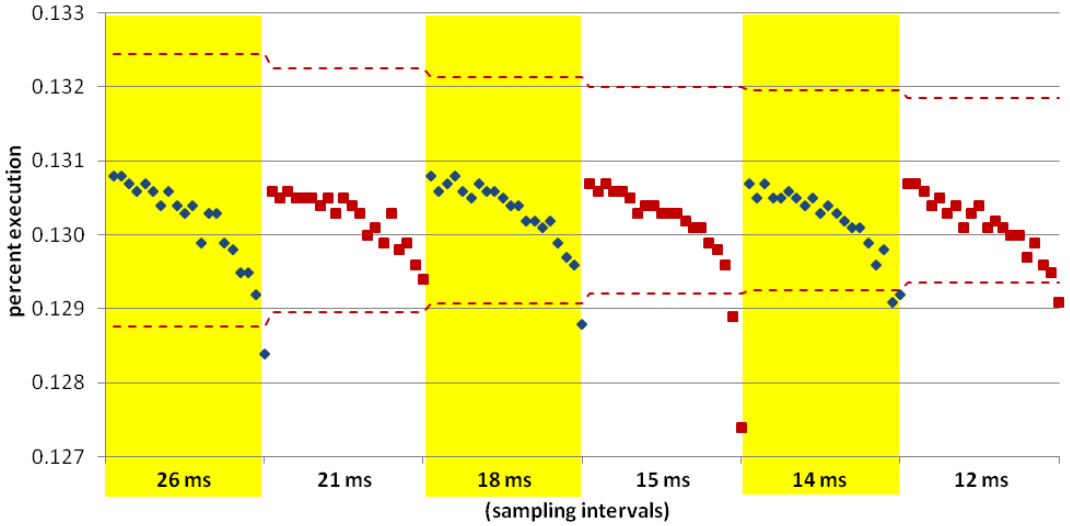

Figure C.65: Results of the percent execution of muldoe_.omp_fn.0, the function taking the 2nd most execution time for the 120 runs conducted with wupwise, 4 threads on 4 cores, 20 runs per sampling interval.

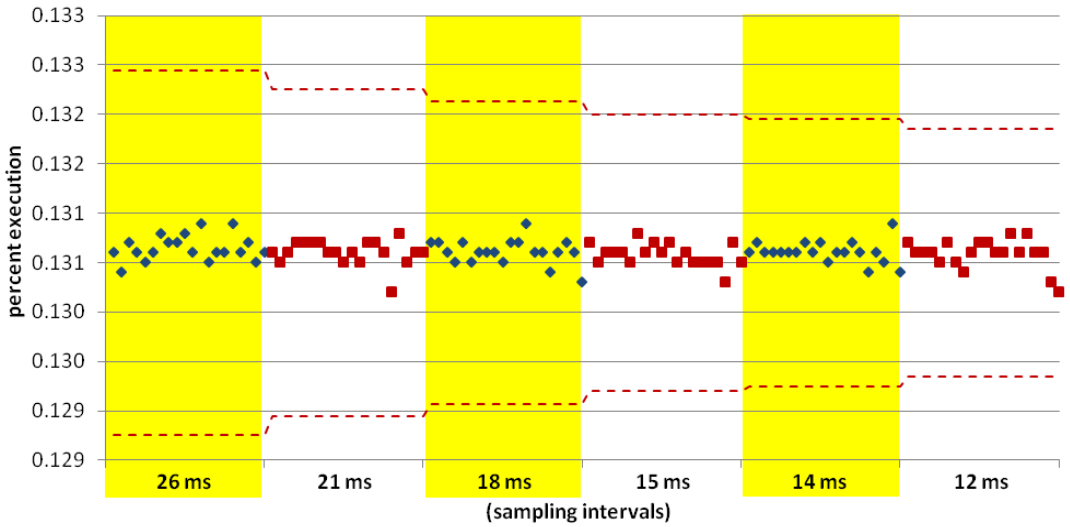

Figure C.66: Results of the percent execution of muldeo_.omp_fn.0, the function taking the 3rd most execution time for the 120 runs conducted with wupwise, 4 threads on 4 cores, 20 runs per sampling interval.

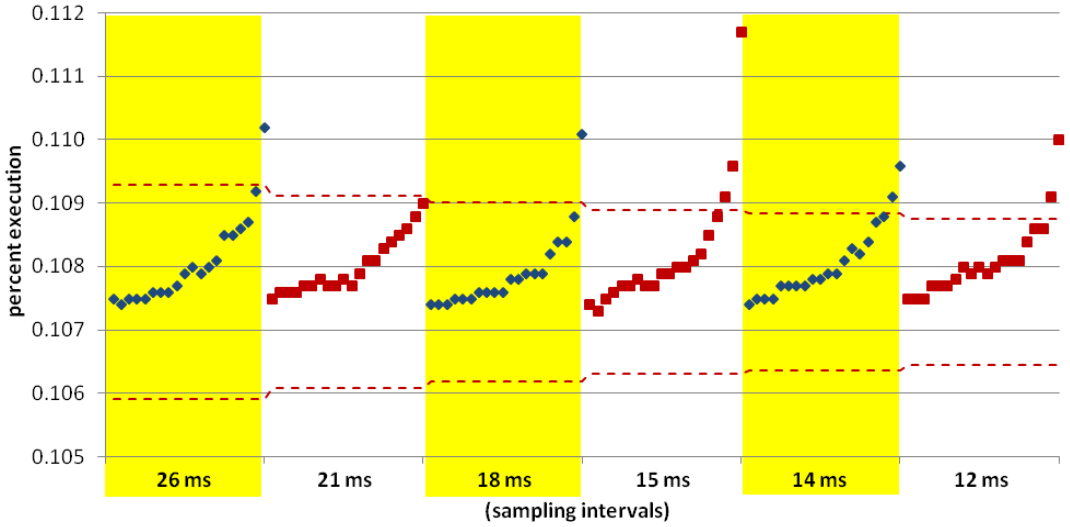

Figure C.67: Results of the percent execution of GOMP_taskwait, the function taking the 4th most execution time for the 120 runs conducted with wupwise, 4 threads on 4 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with wupwise, 8 threads on 8 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

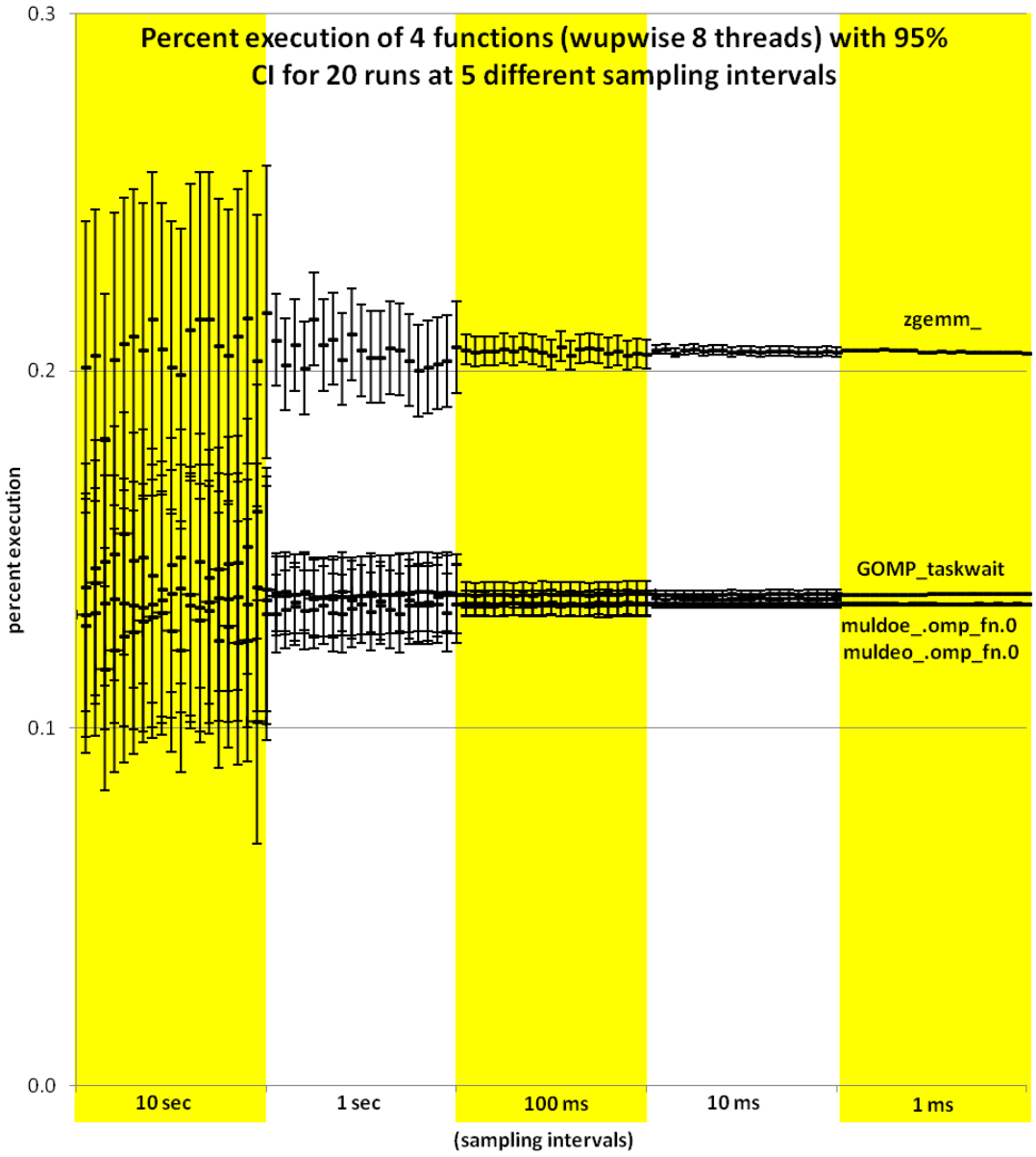

Figure C.68: Distribution of the percent execution taken by 4 different functions from 100 runs of wupwise with 8 threads on 8 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

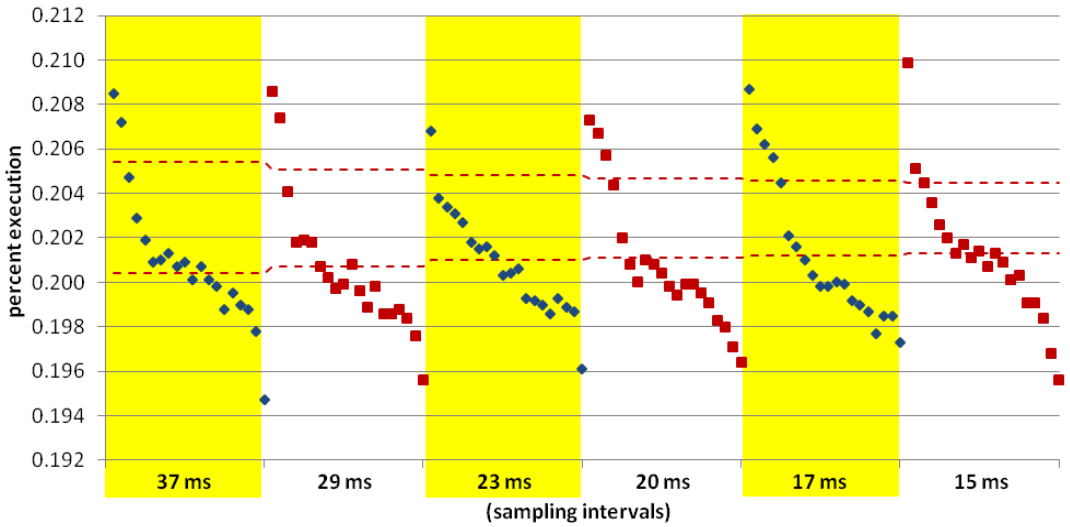

Figure C.69: Results of the percent execution of zgemm_, the function taking the most execution time for the 120 runs conducted with wupwise, 8 threads on 8 cores, 20 runs per sampling interval.

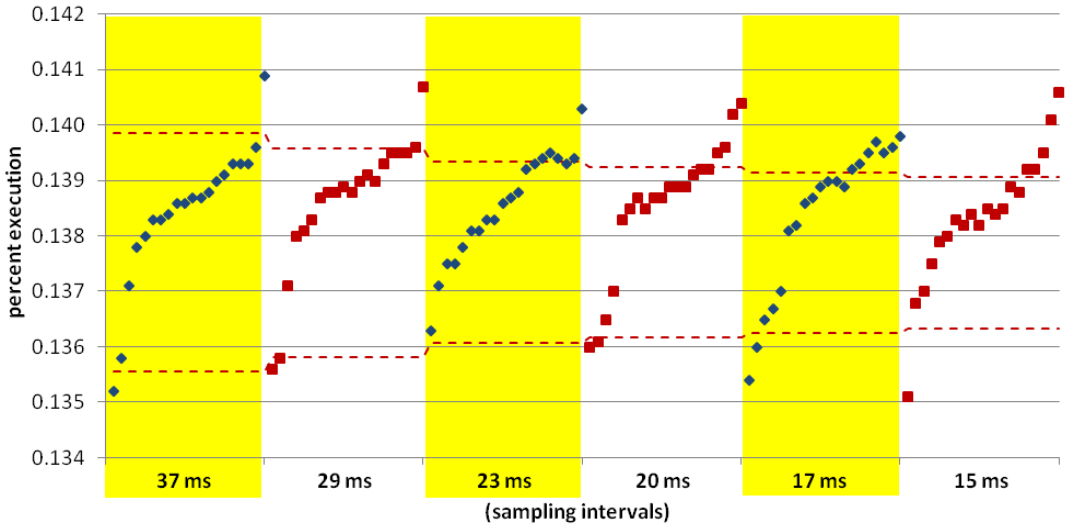

Figure C.70: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with wupwise, 8 threads on 8 cores, 20 runs per sampling interval.

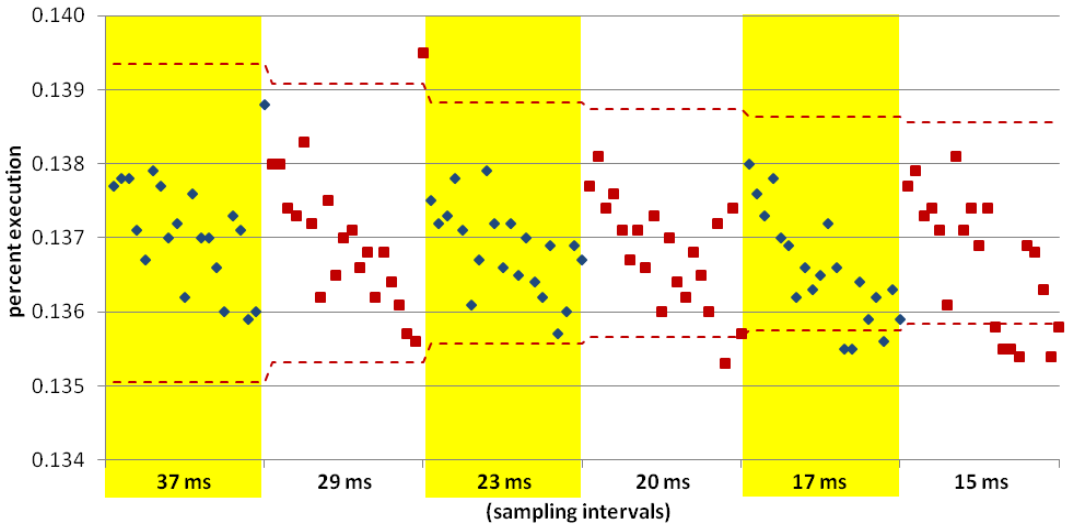

Figure C.71: Results of the percent execution of muldoe_.omp_fn.0, the function taking the 3rd most execution time for the 120 runs conducted with wupwise, 8 threads on 8 cores, 20 runs per sampling interval.

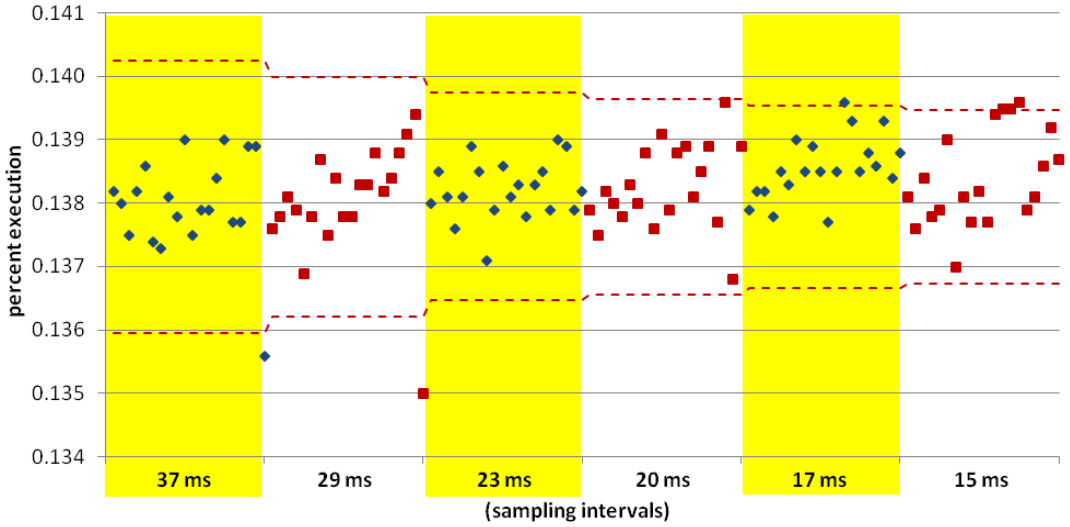

Figure C.72: Results of the percent execution of muldeo_.omp_fn.0, the function taking the 4th most execution time for the 120 runs conducted with wupwise, 8 threads on 8 cores, 20 runs per sampling interval.

The next 5 graphs present results of experiments done with wupwise, 12 threads on 12 cores. First is the composite graph done with the large interval bracketing runs. Following that are 4 graphs showing individual function results for the small interval bracketing runs.

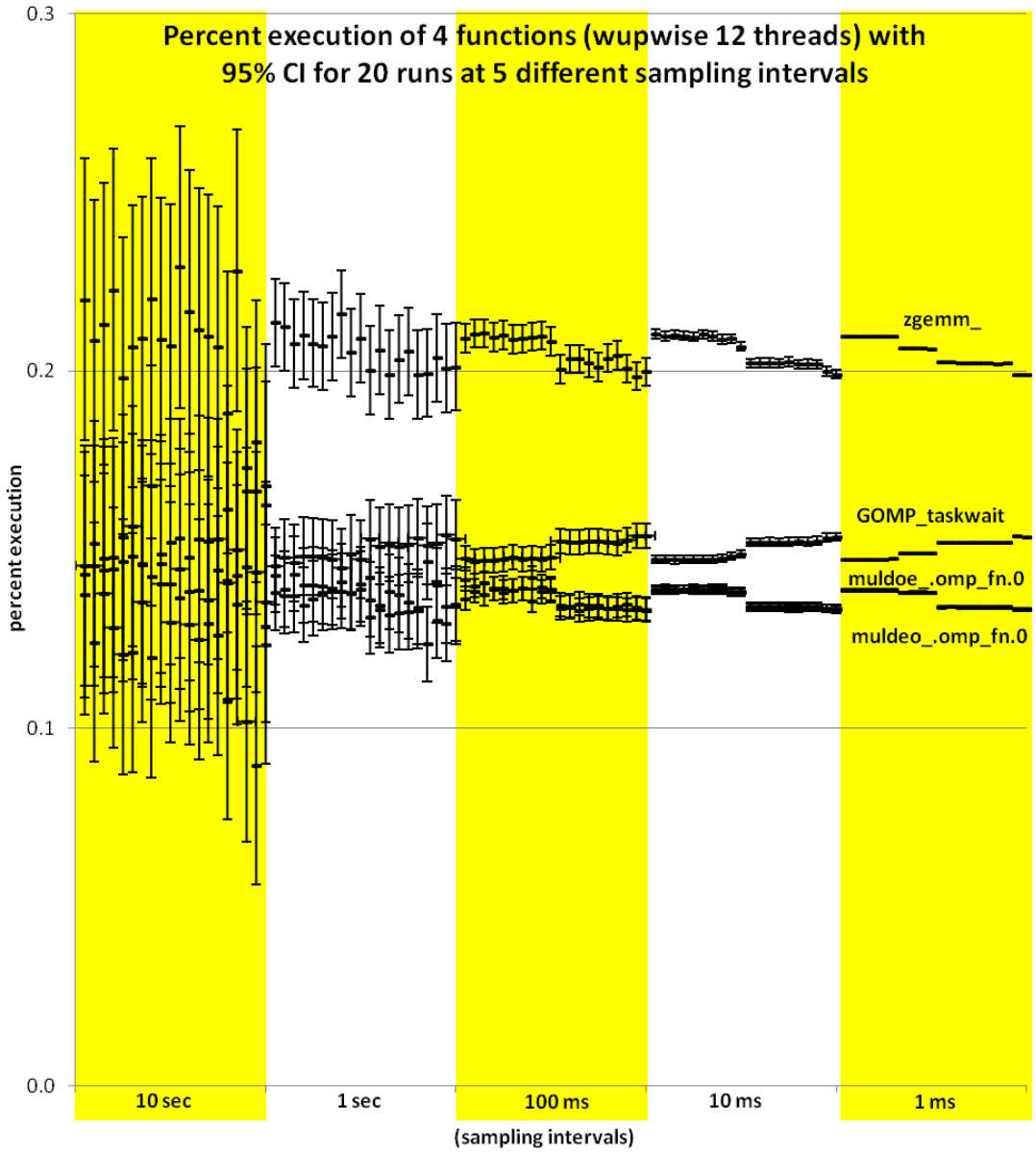

Figure C.73: Distribution of the percent execution taken by 4 different functions from 100 runs of wupwise with 12 threads on 12 cores, 20 each at 5 different sampling intervals. Each data point includes whiskers indicating the expected 95% confidence interval of the calculated proportion.

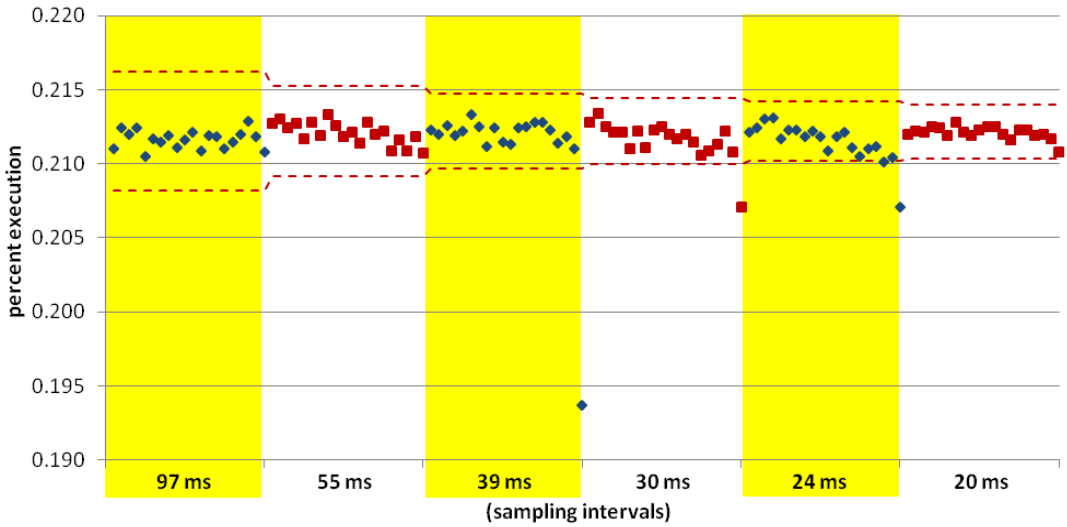

Figure C.74: Results of the percent execution of zgemm_, the function taking the most execution time for the 120 runs conducted with wupwise, 12 threads on 12 cores, 20 runs per sampling interval.

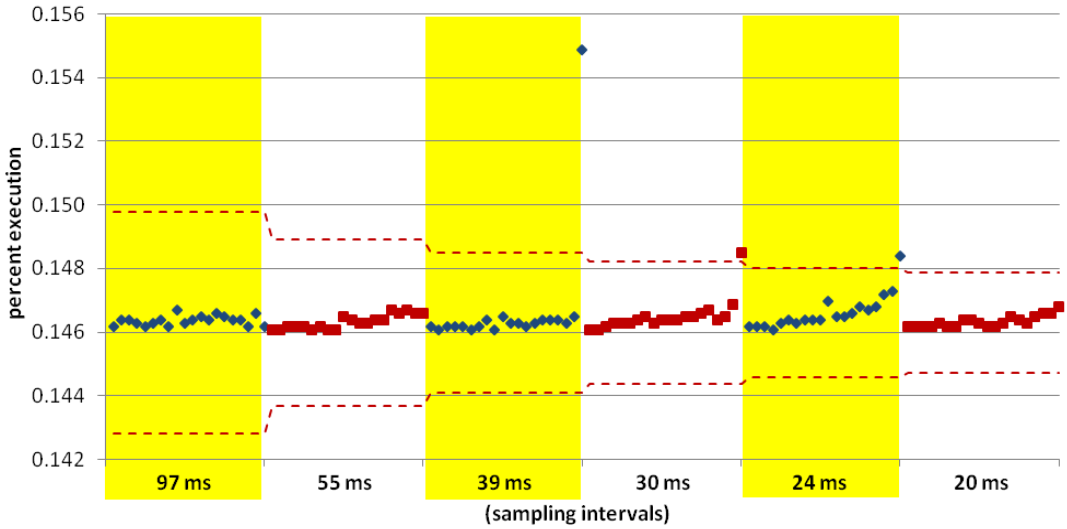

Figure C.75: Results of the percent execution of GOMP_taskwait, the function taking the 2nd most execution time for the 120 runs conducted with wupwise, 12 threads on 12 cores, 20 runs per sampling interval.

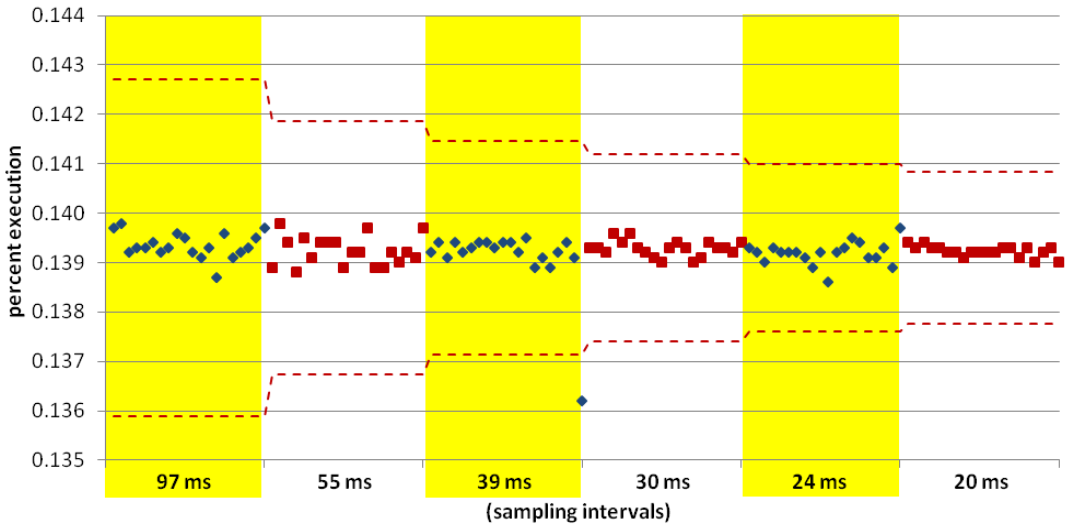

Figure C.76: Results of the percent execution of muldoe_.omp_fn.0, the function taking the 3rd most execution time for the 120 runs conducted with wupwise, 12 threads on 12 cores, 20 runs per sampling interval.

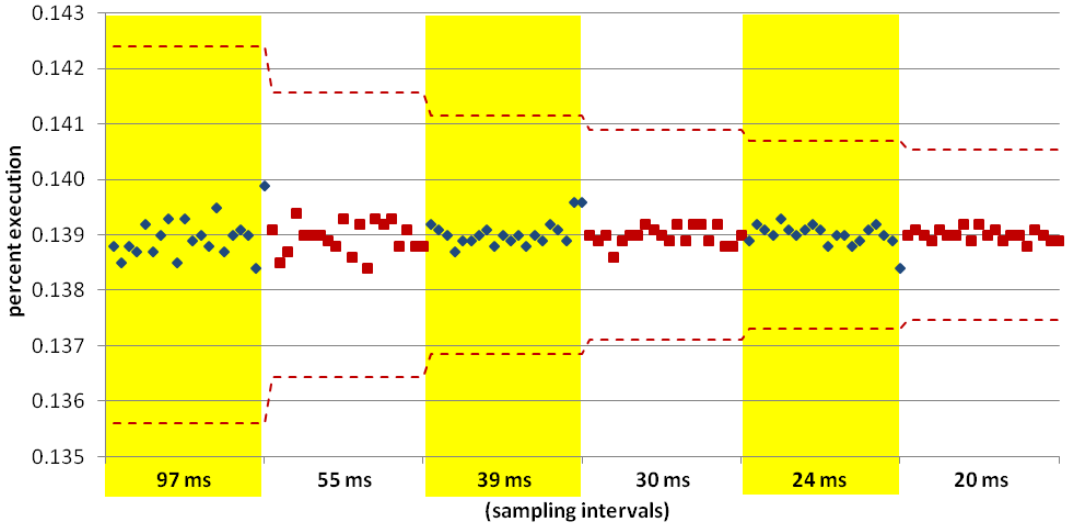

Figure C.77: Results of the percent execution of muldeo_.omp_fn.0, the function taking the 4th most execution time for the 120 runs conducted with wupwise, 12 threads on 12 cores, 20 runs per sampling interval.

Glossary

Accuracy. The absolute difference between a measured value and the corresponding reference value, often the true value [39].

Estimation error (also sampling error). The range of uncertainty between a statistic calculated from a sample and a parameter calculated from an entire population.

Measurement-based profiling (also measured profiling). Instrumenting of a program in order to observe (and record) specific events of interest (e.g., entry and exit of a function) [43].

Perturbation (also interference [12, 23, 16], degradation [14, 26], artifact [52], side effects [63], probe effect [17], intrusion [4, 30], and invasion [48]). The changes in a system's behavior caused by measuring some aspect of its performance [39].

Precision. The amount of scatter in a set of measurements. Corresponds to the repeatability of the measurements [39].

Profile. The collection of frequency counts of program parts (functions, basic blocks, line of code).

Sample-based profiling (also program status sampling [36] or statistical profiling). Periodic interruption of a program, often coordinated via a timer, in order to observe (and record) the current state of an executing program.

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. Concurrency and Computation: Practice and Experience, 22(6):685–701, 2010.

[2] M. Andersland and T. Casavant. Recovering uncorrupted event traces from corrupted event traces in parallel/distributed computing systems. In Proceedings of the 1991 International Conference on Parallel Processing, pages II.108–II.116, 1991.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? ACM Trans. Comput. Syst., 15(4):357–390, 1997.

[4] Z. Aral and I. Gertner. Non-intrusive and interactive profiling in parasight. SIGPLAN Not. 23, 9, 21-30, January 1988.

[5] D. C. Arnold, D. H. Ahn, B. R. D. Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007, IEEE International, pp. 1-10, March 2007.

[6] V. Aslot and R. Eigenmann. Performance characteristics of the SPEC OMP2001 Benchmarks. SIGARCH Comput. Archit. News, December 2001.

[7] M. L. Berenson, D.M. Levine, and T. C. Krehbiel. Basic Business Statistics: Concepts and Applications. Prentice Hall, 10th edition, April 2005.

[8] P. Berube and J.N. Amaral. Combined profiling: A methodology to capture varied program behavior across multiple inputs. Performance Analysis of Systems and Software (ISPASS), 2012 IEEE, pp. 210-220, April 2012.

[9] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. Proceedings of the 2000 ACM/IEEE conference on Supercomputing, pg. 42, 2000.

[10] H.D. Brunk, J.E. Holstein, and F. Williams. A comparison of binomial approximations to the hypergeometric distribution. The American statistician, vol. 22, no. 1, 24-26, February 1968.

[11] B. Buck and J.K. Hollingsworth. An api for runtime code patching. International Journal High Performance Computing Applcations, 14(4):317–329, 2000.

[12] P. Calingaert. System performance evaluation: suvey and appraisal. Comm. ACM Vol 10, No. 1, pp 12-18, January 1967.

[13] M. Casas, H. Savat, R.M. Badia, and J. Labarta. Analyzing the temporal behavior of applications using spectral analysis. Technical Report UPC-DAC-RR-CAP-2009-9, Barcelona Supercomputing Center, February 2009.

[14] W.R. Deniston. SIPE: a TSS/360 software measurement technique" Proc. ACM 24th National Conf. 229–245, 1969.

[15] W. Feller. On the normal approximation to the binomial distribution. The annals of mathematical statistics, vol. 16, no. 4, 319-329, December 1945.

[16] D. Ferrari, G. Serazzi, and A. Zeigner. Measurement and turning of computer systems. Prentice-Hall, 1983.

[17] J. Gait. A probe effect in concurrent programs. Software – Practice and Experience, 16(3)225-233, March 1986.

[18] J. Gannon, K. Williams, M. Andersland, and T. Casavant. Trace recovery: a distributed computing application for perturbation tracking. Proceedings of the 33rd IEEE Conference on decision and control, Vol. 3, pp. 2621 –2626, December 1994.

[19] J. Gannon, K. Williams, M. Andersland, J. Lummp, Jr., and T. Gasavant. Using perturbation tracking to compensate for intrusion in message-passing systems. In Proceedings of the 14th International Conference on Distributed Computing Systems, IEEE, pp. 414 –421, June 1994.

[20] A. Georges. Three pitfalls in java performance evaluation. PhD thesis, Ghent University, Ghent, Belgium, 2008.

[21] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. SIG-PLAN Notices, 42(10):57–76, 2007.

[22] A.J. Goldberg and J.L. Hennessy. Performance debugging shared memory multiprocessor programs with mtool. Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pp. 481–490, 1991.

[23] C.C. Gotlieb. Performance Measurement. In Software engineering, an advanced course, reprint of the first edition, pp. 464-491, Springer-Verlag, London, UK, 1975.

[24] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. SIGPLAN Symposium on Compiler Construction, 17(6):120–126, 1982.

[25] S.L. Graham, P.B. Kessler, and M.K. Mckusick. An execution profiler for modular programs. Software - Practice and Experience, 13(8):671–685, 1983.

[26] U. Grenander and R. Tsao. Quantitative Methods for Evaluating Computer System Performance: A Review and Proposals. Statistical Computer Performance Evaluation, in Statistical Computer Performance Evaluation, W. Freiberger, Ed., Academic Press, N.Y., pp. 3-24, 1972.

[27] G.J. Hahn and W.O. Meeker. Statistical intervals a guide for practitioners. Wiley-Interscience, 1991.

[28] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In Journal of Instruction Level Parallelism 7, September 2005.

[29] J.L. Henning. SPEC CPU2006 Benchmark Descriptions. SIGARCH Comput. Archit. News, September 2006.

[30] J.K. Hollingsworth, J.E. Lumpp Jr., B.P. Miller. Techniques for performance measurement of parallel programs. Parallel computers: theory and practice, IEEE, pp. 225-240, 1995.

[31] HP. Digital continuous profling infrastructure (dcpi). http://h30097.www3.hp.com/dcpi/index.html, August 2010.

[32] IBM. Xprofiler. http://domino.research.ibm.com/comm/research_ projects.nsf/pages/hpct.xprofiler.html, August 2010.

[33] Intel. Vtune performance analyzer. http://software.intel.com/en-us/intel-vtune/, August 2010.

[34] R. Jain. The Art of Computer Systems Performance Analysis. John Wiley & Sons, Inc., 1991.

[35] T. Kalibera and R. Jones. Quantifying performance changes with effect size confidence intervals. Technical report 4-12, University of Kent, June 2012.

[36] D.E. Knuth. An empirical study of fortran programs. Software - Practice and Experience, 1, 1971.

[37] W.T.C. Kramer and C. Ryan. Performance variability of highly parallel architectures. Lecture notes in computer science, v. 2659, p. 560-569, 2003.

[38] N. Kumar, B.R. Childers, and M.L. Soffa. Low overhead program monitoring and profiling. In PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 28–34, 2005.

[39] D.J. Lilja. Measuring computer performance: a practitioner's guide. Cambridge University Press, 2000.

[40] T.F. Lehr. Compensating for perturbation by software performance monitors in asynchronous computations. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[41] A.D. Malony. Performance observability. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1990.

[42] A.D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. IEEE Transactions on parallel and distributed systems, 3(4):433-450, July 1992.

[43] A.D. Malony, S. Shende, A. Morris, and F. Wolf. Compensation of measurement overhead in parallel performance profiling. International journal of high performance computer applications, 21(2):174-194, 2007.

[44] A. Mazouz, S. Touati, and D. Barthou. Study of variations of native program execution times on multi-core architectures. Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), IEEE, pp. 919-924, 2010.

[45] J. Mellor-Crummey, L. Adhianto, M. Fagan, M. Krentel, and N. Tallent. HPCToolkit User's Manual, http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf, January 2013.

[46] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. IEEE Computer, 28(11):37–46, 1995.

[47] T. Mytkowicz, A. Diwan, M. Hauswirth, and P.F. Sweeney. Producing wrong data without doing anything obviously wrong! Proceedings of the 14th international conference on architectural support for programming languages and operating systems (ASPLOS XIV), ACM, pp. 265-276, 2009.

[48] O. Naim and A. J. G. Hey. Invasiveness of performance instrumentation measurements on multiprocessors. IFIP transactions A – computer science and technology Vol. 44, 1994.

[49] H. Najafzadeh and S. Chaiken. Validated observation and reporting of microscopic performance using pentium ii counter facilities. Proceedings of the 4th international workshop on software and performance (WOSP), pp. 161–165, 2004.

[50] H. Najafzadeh and S. Chaiken. Towards a framework for source code instrumentation measurement validation. Proceedings of the 5th international workshop on software and performance (WOSP '05), ACM, pp. 123–130, 2005.

[51] W. L. Nicholson. On the normal approximation to the hypergeometric distribution. The annals of mathematical statistics, Vol. 27, No. 2, pp. 471-483, June 1956.

[52] G. J. Nutt. Tutorial: computer system monitors, IEEE Computer , Vol.8, No.11, pp.51-61, November 1975.

[53] S. Patil and D. Lilja. Statistical methods for computer performance evalutation. Wires computational statistics, Vol. 4, Issue 1, 98-106, 2012.

[54] F. Petrini, D.K. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Vol. 55, pp 15-21, November 2003.

[55] D.A. Reed, R.A. Aydt, L. DeRose, C.L. Mendes, R.L. Ribler, E. Shaffer, H. Simitci, J.S. Vetter, D.R. Wells, S. Whitmore, and Y. Zhang. Performance analysis of parallel systems: approaches and open problems. Proceedings of the joint symposium on parallel processing (JSPP), pp. 239-256, 1998.

[56] Rice University. Hpctoolkit. http://hpctoolkit.org/index.html, August 2010.

[57] S. R. Sarukkai and A. D. Malony. Perturbation analysis of high level instrumentation for spmd programs. SIGPLAN Not., 28(7):44–53, 1993.

[58] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006/

[59] Standard Performance Evaluation Corporation. SPEC OMP2001 Benchmarks. http://www.spec.org/omp2001/

[60] A. Srivastava and A. Eusance. ATOM: A system for building customized program analysis tools. Proceedings of the SIGPLAN '94 conference on programming language design and implementation (PLDI), pp. 196-205, June 1994.

[61] K. Stoodley. Applied and computational statistics. Halsted Press, 1984.

[62] K.V. Subramaniam and M.J. Thazhuthaveetil. Effectiveness of sampling based software profilers. Proceedings of software testing, reliability and quality assurance, December 1994.

[63] L. Svobodova. Performance monitoring in computer systems: a structured approach. ACM Operating system review, Vol. 15, No. 3, July 1981.

[64] V. Tabatabaee, T. Tiwari, and J.K. Hollingsworth. Parallel parameter tuning for applications with performance variability. Proceedings of the 2005 ACM/IEEE conference on Supercomputing, 2005.

[65] N.R. Tallent, J. Mellor-Crummey, and M.W. Fagan. Binary analysis for measurement and attribution of program performance. In proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation (PLDI), pp. 441-452, 2009.

[66] K. Thompson and D.M. Ritchie. Unix Programmer's Manual. Bell Telephone Laboratories, Murray Hill, NJ, June 1974.

[67] UCSD. Simpoint. http://cseweb.ucsd.edu/~calder/simpoint/index.htm, August 2010.

[68] J.S. Vetter and D. Reed. Managing performance analysis with dynamic statistical projection pursuit. Proceedings of the 1999 ACM/IEEE conference on Supercomputing: High performance networking and computing, 1999.

[69] J.J. Yi, D.J. Lilja, and D.M. Hawkins. Improving computer architecture simulation methodology by adding statistical rigor. IEEE Transactions on Computers, Vol. 54, pp. 1360–1373, November 2005.