NIST Technical Note 1830

The ghost in the machine: don't let it haunt your software performance measurements

Vreda Pieterse
David Flater

http://dx.doi.org/10.6028/NIST.TN.1830

National Institute of
Standards and Technology
U.S. Department of Commerce

NIST Technical Note 1830

The ghost in the machine: don't let it haunt your software performance measurements

Vreda Pieterse
David Flater
Software and Systems Division
Information Technology Laboratory

http://dx.doi.org/10.6028/NIST.TN.1830

April 2014

National Institute of Standards and Technology Technical Note 1830
Natl. Inst. Stand. Technol. Tech. Note 1830, 26 pages (April 2014)
http://dx.doi.org/10.6028/NIST.TN.1830
CODEN: NTNOEF

The ghost in the machine: don't let it haunt your software performance measurements

Vreda Pieterse
David Flater

April 2014

Abstract

This paper describes pitfalls, issues, and methodology for measuring software performance. Ideally, measurement should be performed and reported in such a way that others will be able to reproduce the results in order to confirm their validity. We aim to motivate scientists to apply the necessary rigor to the design and execution of their software performance measurements to achieve reliable results. Repeatability of experiments, comparability of reported results, and verifiability of claims that are based on such results can be achieved only when measurements and reporting procedures can be trusted. In short, this paper urges the reader to measure the right performance and to measure the performance right.

1 Introduction

Metrology is the science of measurement and its application. IT metrology is the application of metrology to computer systems. It entails the measurement of various aspects of hardware components as well as software entities at differing levels of abstraction. Software performance metrology deals specifically with performance aspects of software processes (how software actually runs). Research in this field aims to identify appropriate metrics for measuring aspects of software performance as well as the specification of sound methods and techniques to acquire and analyze software performance data.

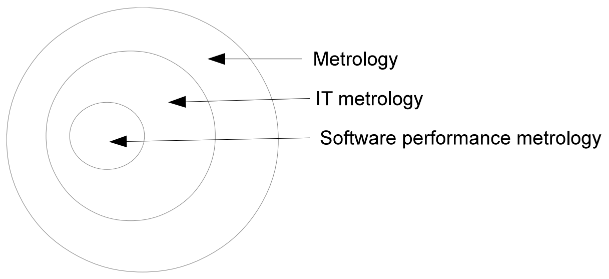


Figure 1: The position of software performance metrology in the metrology field

Software performance measurement is much harder than it is generally believed to be. There are many factors that confound software performance measurements. The complexity of computing systems has tremendously increased over the last decades. Hierarchical cache subsystems, non-uniform memory, simultaneous multi-threading, pipelining and out-of-order execution have a huge impact on the performance and compute capacity of modern processors. These advances have complicated software performance metrology to a large extent. Simple practices and rules of thumb used in software performance metrology in the past are no longer reliable. New metrics, methods and techniques have to be applied to cope with modern hardware and software complexities.

A lack of the necessary rigor in the application of software performance metrology seems apparent. Mytkowicz et al. [2009] observed that measurement bias is commonplace in published papers with experimental results while Flater and Guthrie [2013] noted that the earlier observation by Berry [1992] that statistical design of experiments for the analysis of computer system performance is not applied as often as it ought still seems true more than a decade later. This observation was supported by Vitek and Kalibera [2012] who stated that papers in proceedings of computer science conferences regularly appear without a comparison to the state of the art, without appropriate benchmarks, without any mention of limitations, and without sufficient detail to reproduce the experiments. Kalibera and Jones [2013] investigated papers published during 2011 in prominent conference proceedings and found that the majority of these papers reported results in ways that seem to make their work impossible to repeat.

We are concerned about the current state of affairs and acutely aware of the problems that contribute to the degeneration of the quality of reported software performance experiments as reported in the literature. This paper aims to draw attention to these problems and provide some guidelines to address them. In Section 2 we provide a theoretical framework describing the required content and life cycle of software performance experiments to serve as a foundation on which software performance metrologists should base their work. A number of the issues that are mentioned in the discussion of the framework are further discussed in the sections that follow. In Section 3 we explain the strengths and weaknesses of a selection of available software performance metrics to serve as a baseline when selecting the metrics for software performance experiments. Section 4 provide guidelines to find suitable instruments that can be applied to obtain the required data. Section 5 highlights some aspects that should be considered when designing software performance experiments to ensure that the experiments comply with statistical requirements. Section 6 describes available measurement techniques while Section 7 provides guidance for dealing with some known hard to control factors. The concluding section summarizes the essence of our plea that the state of the practice in experimental evaluation of various aspects of software performance should be improved.

2 Software performance experiments

Software performance experiments are conducted to determine the relative performance of software entities or to assert claims about the compliance of software entities with specified standards (an instance of conformity assessment or conformance testing). A scientific approach is required throughout the life cycle of experiments to achieve credibility.

2.1 Components

A software performance experiment consists of the following components identified by the Evaluate Collaboratory [Evaluate Collaboratory; Blackburn et al., 2012]:

1. Measurement context: software and hardware components to vary (*independent variables*) or hold constant (*controlled variables*) in the experiment.
2. Workloads: the characteristics of the data that are used in the experiment.
3. Metrics: the properties that are measured and how they are measured.
4. Data analysis and interpretation: the methods applied to analyze and interpret the data.

When reporting the outcome of a software performance experiment it is essential to provide detailed information about each of these components.

2.2 Life cycle

The term *life cycle* in context of software performance experiments refers to the process to conduct an experiment. This includes all actions from the identification of a need to measure software performance to the accomplishment of the goal of the experiment. Figure 2 illustrates the activities in the life cycle of a software performance experiment (shown on the left) as well as the components of the experiment (shown on the right). The arrows from the life cycle activities to the components indicate which components are influenced by each of the activities.

2.2.1 Formulate the goal

Every software performance experiment should be conducted with a specific goal in mind. It may be to compare alternative configurations or algorithms, to determine the impact of a feature, to find the configuration that produces the best overall performance, to describe the performance capabilities of a program, to determine why a program is not meeting performance expectations, etc. The goal of the experiment dictates the rest of its life cycle. The dotted arrow in Figure 2 from the *Formulate the goal* activity to the outline that encapsulates all the components of an experiment indicates that this activity influences all the components of the experiment.

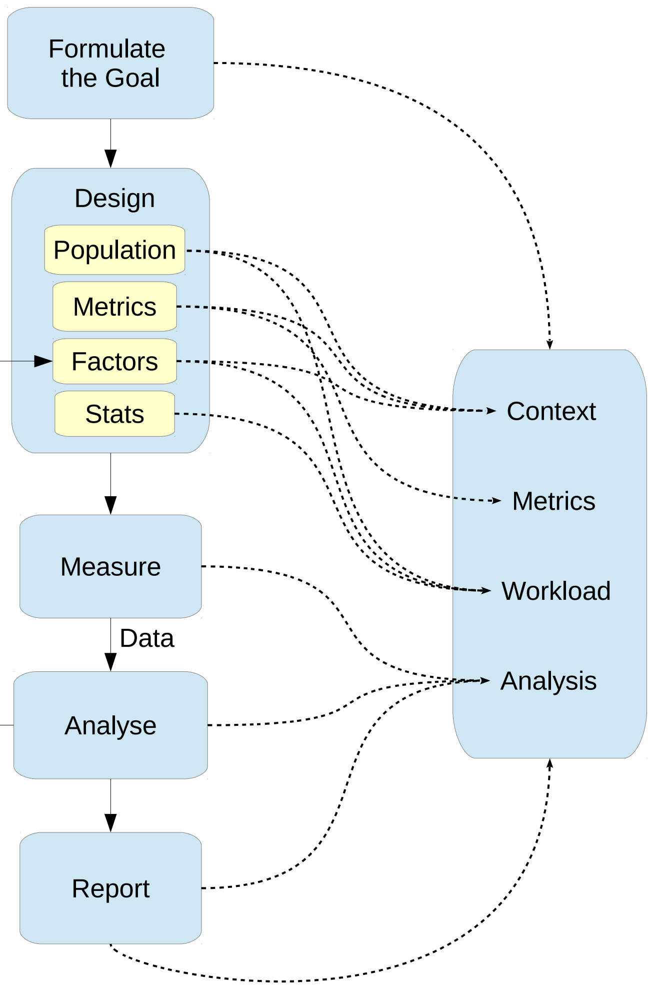


Figure 2: Software performance experiment life cycle

2.2.2 Population

Once the goal is clear, the experiment can be designed. The first step in this regard is to decide what *population* the experiment is intended to reveal information about.

At the conclusion of an experiment there will be some general inference made based on observations of a *sample*. The utility of a study has to do not only with whether a question is answered for the system as tested, but also whether that conclusion is robustly valid over all of the different kinds of software applications, environments, use cases, loads, etc., that the system may encounter or that the experimenter is interested in. If the robustness issue is not dealt with, then one can end up optimizing for an overly specific scope and actually making things worse on average for the scope that applies in reality.

A careful enumeration of what the robustness factors are, followed by what levels those robustness factors should be set at so as to be representative of actual usage, is a good start to identifying the set of factors for the experiment and selecting representative workloads.

2.2.3 Metrics

Next, an appropriate metric has to be chosen. The formulation of the goal should indicate the metrics that can be used to reach the goal. For example, if you want to determine if a new system consumes more power than your old system, it is obvious that you need metrics to measure and express power consumption.

When choosing a metric, you need to know how it can be measured as well as the unit of measure that is appropriate for the experiment. It is important to use metrics that are relevant rather than simply choosing metrics that are easy to obtain. Also use units that are suited to achieve the goal of the experiment. Whether a metric is appropriate is determined by the goal of the experiment as well as the cost of gathering the data. The choice of the metric, the procedure to measure it as well as the chosen unit constitutes the metrics component of the experiment. This is shown with a dotted arrow from the *Metrics* aspect in the design activity in Figure 2 to the metrics component. The metric that is chosen may influence the measurement context because the process to measure the chosen metric may require instrumentation which is part of the measurement context.

Section 3 discusses some performance measures and their characteristics to support decisions regarding the most appropriate performance measure for a given problem. Section 4 points to some instruments that can be used to obtain these measures.

2.2.4 Factors

An important aspect of the design of the experiment is the identification of factors that play a role in the phenomenon under investigation. One should be aware of the various factors that may influence the metric of choice and find ways to minimize bias that may be caused by such factors.

The identification of factors is crucial because the omission or misunderstanding of factors may invalidate the results and conclusions of the experiment or severely limit the scope within which those conclusions are valid. The identification of the appropriate factors is arguably the most difficult aspect of the design of a software performance experiment. To do so correctly without experimentation requires a deep understanding of the underlying system. A more reliable approach is to perform a separate experiment just to identify the most important factors. This is called a sensitivity analysis experiment or a *screening experiment*—screening for those factors that have the largest impact and letting the less important factors fall through. A two-level fractional factorial experimental design [Schatzoff, 1981; Gunst and Mason, 2009; NIST/SEMATECH, 2012; Montgomery, 2013] is usually both efficient and effective for this purpose provided that there are not conflicting constraints on the combinations of factor levels (often called *treatments*) or the testing sequence.

Measures to control the influence of factors that are kept constant should be described in the experimental context while the values of factors that are varied in the experiment should be described in the workload component. This is illustrated in Figure 2 by the arrows pointing from the factors component in the life cycle to the experimental context and workload components of the experiment.

The discussion about metrics in Section 3 mentions some pertinent factors that should be addressed when using each of the metrics while Section 7 discusses a few obscure factors. Often, unidentified factors lead to anomalies in the results, ultimately contributing to the discovery of new factors. In Figure 2 there is a transition from the analysis stage back to the factors stage to indicate that the discovery of a new factor requires the metrologist to incorporate the newly identified factor in the experimental design and repeat the experiment in order to be able to reach the goal of the experiment. Sometimes factors may be discovered earlier in the process; for example, Flater [2013] identified several factors while validating profiling instruments.

2.2.5 Statistics

Errors are inherent in measurements of real computer systems. Established and trusted statistical methods should be applied to deal with variability and increase the confidence levels with which one can draw conclusions. The data gathered for an experiment need to contain sufficient samples where the role of each of the contributing factors is understood. The experimental system should be designed to have each factor either be carefully controlled or realistically varied to be representative in the observed context.

Georges et al. [2007] point out that many performance evaluation studies seem to neglect statistical rigor. They advocate increased statistical rigor for experimental design and data analysis because statistics, and in particular confidence intervals, enable one to determine whether differences observed in measurements are due to random fluctuations in the measurements or due to actual differences in the alternatives compared against each other.

It is beyond the scope of this article to provide detailed guidelines on proper statistical procedures to compute confidence intervals, to analyze variance and to compare multiple alternatives. The reader is advised to collaborate with statisticians and consult textbooks such as Jain [1991]; Lilja [2000]; Maxwell and Delaney [2004] and Johnson and Wichern [2007] for guidance. A more technical, freely available reference is the "GUM" (Guide to the expression of Uncertainty in Measurement) and its supplement [JCGM 100, 2008; JCGM 101, 2008]; see also the NIST/SEMATECH e-Handbook of Statistical Methods [NIST/SEMATECH, 2012].

Some statistical considerations feature prominently in Section 5 where different aspects of the design of software performance experiments are discussed.

2.2.6 Measure

The validity of data that are gathered relies on the use of trustworthy instruments. The selection and validation of instruments is discussed in Section 4. It is also assumed that the measurements are performed according to a well-designed plan as suggested in Section 5.

The application of wrong or inappropriate methods when measuring, even when applying a good design and using trusted tools, can still invalidate the measurements and consequently the conclusions drawn from such results. It is therefore essential that performance measures be validated before they are used in analysis.

Different measurement techniques that can be applied are discussed in more detail in Section 6.

2.2.7 Data

The data resulting from experiments are also subject to validation through inspection. To begin with, it is always a good idea to examine data with scatter plots to see whether anomalies such as time-dependency are obvious. More sophisticated analytical validations that can be done include normalcy tests to see whether results are described by a normal distribution and autocorrelation tests to see whether they are independent. The gamut of applicable methods falls in the scope of Exploratory Data Analysis (EDA), a field famously championed by Tukey [1977].

Data can be "anomalous" by being nonrandom or by having a drifting location, drifting variation, or drifting distribution. The goal of this examination is to gain confidence that the process is in "statistical control" and therefore satisfies the assumptions of the statistical methods that will be applied to make inferential statements about a population.

Often, measurements gathered during a *warm up* phase are substantially different from data gathered after a system has reached a stable state where benchmark iterations are statistically independent and identically distributed. This phenomenon can be attributed to once-off initial overhead required by dynamic linking, filling I/O buffers for data/code, or just-in-time compilation.

If a system reaches a stable state within reasonable time, valid conclusions can be drawn by using only the data gathered during the stable state (*assuming* that this optimized performance, and not the once-off performance, is the measurand of interest). However, Kalibera and Jones [2013] observed that it may often happen that a system fails to reach a stable state. They recommend that the metrologist use data from the same iteration of each of multiple runs in such cases.

2.2.8 Analyze and report

The assumption at this point is that the metrologist has implemented the steps needed to ensure that the measurements obtained are as accurate as possible as described in Section 4.2 and have applied the sound statistical precautions described in Section 5 in the design of the experiment to ensure the validity of the observations. Having sufficiently characterized and validated the data, one is in position to choose applicable and appropriate methods to summarize and visualize them for use in drawing conclusions.

An analysis will usually involve graphical methods like main effects plots and quantitative methods like Analysis of Variance (ANOVA). Generically speaking, some parameters are estimated and then a comparison is performed or a hypothesis is tested. The body of scientific and statistical knowledge that is available to assist the experimenter at this point is both broad and deep, covering general cases as well as a plethora of special cases. One could not do justice to it with any survey that would fit within the scope of this paper.

When reporting the results, one has to be able to demonstrate that the data that were gathered truly reflect the phenomenon that is observed. Reporting should contain sufficient information to substantiate the claims that are made and support repeatability of experiments. For this reason, each report should discuss each of the components of an experiment that is outlined in Section 2.1. Reporting that complies with these expectations contributes to the comparability of results and the verifiability of claims that are based on such results.

3 Performance measures

The selection of the appropriate metric depends on the goal of the measurement experiment; however, the choices are limited by the services provided by the systems being benchmarked and available tooling (measurement instruments). The measures that are discussed here are not comprehensive. A few commonly used metrics that can be measured with the aid of existing profiling tools are covered. This provides a starting point for the creative metrologist to find or derive the ideal metric to reach his goal.

3.1 Elapsed ("wall clock") time

Elapsed time is simply the time that elapses while the process being measured is executed. It can be determined by starting a timer before the process is started and stopping the timer when the process signals completion. This time is also known as *wall clock* time. Elapsed time can also be determined by reading a clock time just before the process is started and again at completion time. The absolute difference between these times is the elapsed time.

Elapsed time includes time that may have been spent on other processes on the system while the process being measured was executed. This is very common on most modern systems. Even on stand-alone machines the OS is likely to time-share many processes that are constantly working towards improving the overall performance of all the applications that are used on the machine. For this reason the influence of other processes is an important factor that should be considered when measuring elapsed time. Other factors that may play a role are the processor make and model, the number of processors and their frequencies.

Lilja [2000, page 19] deems elapsed time, when performed using a statistically sound experimental design, one of the best metrics to use when analyzing computer system performance.

3.2 CPU time

CPU time (or process time) is the amount of time, usually measured in seconds or clock ticks, that was used while executing the process being measured. This excludes the time that the CPU was processing other processes or was in low-power (idle) mode. It also excludes time the process being measured was waiting for input/output (I/O) operations. When a system has multiple processors, it may be necessary to acquire the CPU time of each of the different processors individually and add them to calculate the total CPU time of the process being measured; however, up-to-date tooling will do this automatically.

The CPU time of a function is sometimes expressed as a percentage indicating the proportion of time consumed by the function relative to the total time consumed by the process that contains the function.

The CPU time of a function may be measured as self time or as total time. The self time of a function is the time spent executing the function itself, excluding time spent by any sub-functions it may have invoked. The total time of a function is the time spent executing the function as well as any sub-functions invoked by it. The calculation of total time is complicated by situations where a function may be called by two or more different functions, or when functions are called recursively. Figure 3 illustrates a problem arising with a recursive call. In the diagram on the left, function f calls functions f_1 and f_2. The total time of f is calculated as the sum of its self-time plus the total times of the functions it calls. Since the call graph is hierarchical this does not pose a problem. In the diagram on the right the function f also calls itself. Owing to the cyclic call graph, the application of the proposed formula is infinite.

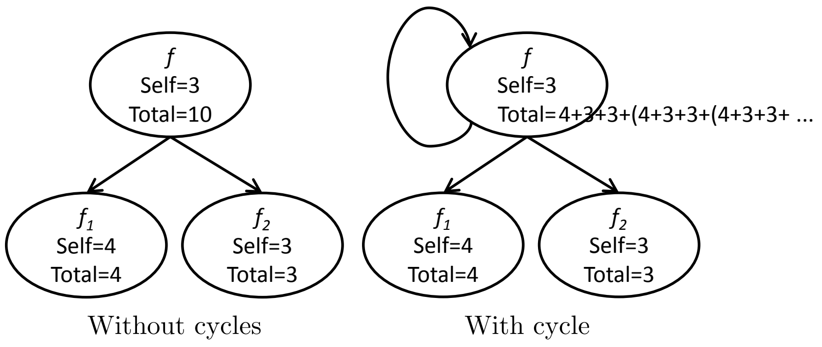


Figure 3: Recursive definition to calculate total time

A more robust way to determine the total time of a function is to use information on the call stacks. The total time of a function can be defined as the sum of the times when the function is either executing or was on the stack. Table 1 shows a hypothetical call stack of the execution of the functions shown in Figure 3 at times $1, 2, 3, \ldots 10$ and the total times of these functions calculated through the application of this definition. This robust approach is supported by the combination of Linux perf [Perf, 2013] with the Gprof2Dot visualization tool (option `--total=callstacks`) [Fonseca, 2013]. For GNU gprof [Fenlason, 1988] and other tools that do not separate the call stack information from the call graph, workarounds and heuristics must be used instead to estimate total times.

The designs for experiments measuring CPU time have to describe how prominent factors that influence these measurements such as processor make and model, the number of processors and their frequencies are controlled or varied. While the measurement of CPU time does not include the time spent on other processes in the system while the process being measured was executed, as is the case when measuring elapsed time,

1	2	3	4	4	6	7	8	9	10
f	f	f	f	f	f	f	f	f	f
	f_1	f_1	f_2	f_2	f_2	f	f	f	f
							f_1	f_1	f

Function	Self time	Total time
f	3	10
f_1	4	4
f_2	3	3

Table 1: Calculating total time by counting the number of times a function was observed executing or on the call stack

all of the factors that should be considered when measuring elapsed time are still relevant when measuring CPU time, albeit for different reasons. Competing loads may indirectly increase the CPU time used by a process by filling up cache space or creating additional overhead. Zaparanuks et al. [2009] noted significant variability of cycle counts owing to perturbation associated with performance counters. This variability was reproduced in Flater and Guthrie [2013].

3.3 CPU utilization and memory latency

CPU utilization is a term used to describe how much the processor is working [Robbin, 2009], while latency is the time delay experienced in a system. These concepts are related. Latencies directly influence the CPU utilization: reducing wait time can be more important than increasing execution speed. A CPU waiting for 100 clock ticks for data would run at 1/100 (1 %) of theoretical performance [SiSoftware staff, 2012]. Conversely, excessive CPU utilization for a given process can starve other processes and lead to overall non-responsiveness of the system.

CPU utilization can be used to track CPU performance regressions or improvements and is a useful data point for performance problem investigations. Originally, CPU utilization was a concern only for scheduling algorithms, but it is increasingly important for the design of software in general because nearly all CPUs are now multi-core. An application that cannot use all available CPU cores will suffer in the elapsed time measure, and a low CPU utilization measurement indicates the root cause of that suffering.

In modern systems it is likely that CPU frequency is changed dynamically to optimize performance, heat dissipation, and power consumption. To prevent the processor clock frequency from changing during measurements, frequency scaling should be disabled [Zaparanuks et al., 2009]. Hopper [2013] points out that theoretical hardware performance values do not always reflect actual application performance due to many factors, including caching effects, data locality, and instruction sequences, among other things.

3.4 Memory usage

The memory usage of a process is simply how much memory the process is using. It is, however, a complex matter to determine the memory usage of an application or process accurately since it can fluctuate arbitrarily as the application executes. Furthermore, memory management algorithms implemented at both the kernel and application levels can exhibit system-dependent, context-sensitive, irreproducible behavior. Even the memory page size, and therefore the granularity of memory allocation, can be varied by the operating system [Giraldeau et al., 2002; Torrellas et al., 2005].

This physical memory usage observed at the kernel level may not be an accurate reflection of memory operations performed by an application. Since it is common for applications to allocate and free memory in cycles or in chaotic patterns, memory management algorithms will hang onto nominally freed memory in order to avoid the relatively expensive process of fully releasing memory back into the operating system's pool only to have to fully allocate it again [Giraldeau et al., 2011].

Another aspect contributing to confusion when determining the memory usage is the various ways in which operating systems may implement the use of virtual memory. Most profiling tools distinguish between resident set size (RSS) and virtual set size (VSZ). RSS is how much memory this process currently has in main memory (RAM), whereas VSZ is how much virtual memory the process has in total, i.e., memory in RAM as well as memory that has been swapped out. Numerous other distinctions are possible. For example, in Linux one can distinguish between shared and unshared memory, between memory requested and memory actually used, between regular memory and *transcendent* memory [Magenheimer, 2011], and possibly other variations. Depending on the goals of the measurement, one might include or exclude different types of memory from the total.

Going below the application level of granularity, one can distinguish between code, heap, and stack memory and between addressable and unaddressable (i.e., *leaked*) memory. Profiling instruments that are capable of reporting detailed statistics at this level are likely to slow down the operation of the processes under test. Bruening and Zhao [2011] measured the execution times while using two different profiling instruments relative to native times on a number of benchmarks and observed considerable slow down in some cases. To avoid variability of time measurements that may be caused by the profiling instrument, it may be a good idea to measure CPU usage or CPU time in benchmarks separate from the benchmarks applied to profile the memory usage at this lower level of granularity.

3.5 Power consumption

Despite incomplete and uncertain information, it is estimated that the generation and distribution of electricity comprises nearly 40 % of U.S. CO_2 emissions [Weber et al., 2010]. Recent trends in achieving a sustainable future pay due attention to energy efficiency in order to reduce environmental impacts and risks. While energy-efficient hardware design seems to dominate energy-efficient computing literature, future applications and system software also need to participate in power-saving efforts [Esmaeilzadeh et al., 2012]. In many applications, such as portable devices, low power is more important than performance. Kin et al. [1997] believe that low power researchers should be open to sacrificing some performance for power savings. For these reasons, interest in measuring the relative power consumption of software has grown.

One way to measure the power consumption of a computer is the use of a power meter installed at the wall socket. This allows the analyst to measure the overall wattage used by the system. This method of measuring is not useful in cases where the power consumption of separate components is needed (memory, hard drives, I/O boards, disk controllers, etc.). With more effort, hardware instrumentation can be attached to the power inputs of the components one wants to measure [Carroll and Heiser, 2010].

Alternative approaches for estimating power consumption based on a power model and activity counters that can be used in place of instrumented power measurements have been proposed [Flinn and Satyanarayanan, 1999; Goel et al., 2010; Lewis et al., 2008; Singh et al., 2009; Stockman et al., 2010]. Software instruments that may be used to estimate power consumption at varying levels of detail are mentioned in Section 4.

Owing to complex architectural setup when using multiple cores, the power used by a given process may not be directly associated with the measured power consumption. Tudor and Teo [2013] observed large imbalances between cores, memory and I/O resources leading to under-utilized resources and energy waste. Environmental and electrical variations such as temperature and electro-migration may contribute to unwanted variability in energy consumption and hence need to be controlled when measuring in-system power consumption. Brooks et al. [2007] explain the power, thermal, and reliability modeling problems. Muroya et al. [2010] remarked that higher room temperature can affect power consumption in information and communications technology equipment through increasing cooling fan speed. An AnandTech forum participant (IDC) [2011] reported an experiment showing a striking correlation between operating temperature and the power consumption of a CPU *itself*. Both effects must be considered in context of the total system, which includes the power for air conditioning to maintain lower ambient temperatures.

3.6 Power cost

Although the cost of power is closely correlated with the amount consumed, it is a different concern that leads to different conclusions. In some cases power is dynamically priced so that simply shifting power consumption from one time of day to another can decrease costs. Thus, a software configuration that performs more poorly according to other measures such as response time may be better according to this metric. To maximize profit, cloud service providers must account for such effects when pricing out service options with differing guarantees on response time and other quality of service metrics [Wang et al., 2013; Urgaonkar et al., 2013; Samanthula et al., 2013].

Of course, the cost of power is only one of the costs that are incurred by or amortized over software processes, but it has for now become the focus of attention. Not long ago, the focus was on the cost of labor (human time and attention) needed to keep software systems operational, and metrics for the maintainability of software were highly sought after. Before that, it was the cost of procuring or developing software; before that, it was the cost of hardware.

4 Instruments

In Section 3 some key performance measures were discussed. After deciding what metrics are to be used the metrologist has to find instruments to determine these metrics and identify those that are suitable to use for each particular case. A plethora of performance monitoring and profiling instruments exists. An understanding of the availability and techniques of the different monitoring tools is essential for choosing the proper tool to suit the goal of each experiment.

4.1 Finding an instrument

This section only points to sources of information to serve as a starting point to find the appropriate instruments for software performance experiments.

Most operating systems have command-line utilities that can be used to report the time used by a given executable. Examples are `time` on Linux and Mac and `Measure-Command` on Windows. These commands are available by default, easy to use and require no modification of the software under inspection. They report elapsed time as well as CPU time of a complete program. It may not be fine-grained enough to provide all the required information. The GNU 1.7 version of the `time` command outputs additional information on other resources like memory, I/O and IPC calls. The Linux perf tools [Perf, 2013] allow user-level access to per thread hardware counters.

A list of performance monitoring and profiling tools can be found in Wikipedia. Many of these instruments are complete suites that can be used to collect most of the performance measures mentioned in Section 3 and more.

Popular instruments that support measuring different levels of memory usage are Valgrind [Valgrind], Pin [Berkowits, 2012] and Dr. Memory [Dr. Memory]. Generic system-level monitoring tools such as the Unix **ps** and **top** commands can also be used for this purpose, albeit with a limited sampling rate. A few software instruments that can be applied to measuring or estimating power consumption are PowerTOP [PowerTOP] on Linux, Joulemeter [Microsoft Research] on Windows, Sleep Monitor [Dragon Systems Software Ltd.] on a Mac, and Intel Power Gadget [Vega, 2013] on all three platforms.

4.2 Validating an instrument

One should validate an instrument before using it. Flater [2013] demonstrates the importance of validating configurations before they are relied upon. It is a good idea to read the technical documentation of the instruments as well as some independent reviews of a product before using it. Wun [2006], Helvick [2008] and Prinslow [2011] have compiled surveys of available instruments that may provide valuable insight about some of the popular ones. However, the tools evolve rapidly, so it is important to search for similar surveys that are not yet outdated at the time you need this information.

Surveys and documentation are always vulnerable to being outdated or simply wrong; therefore, operational validation is always advisable. Even a single, simple, inexpensive test of the features most needed can return surprising results that will drastically affect your plans. There is no test that is too small or too simple to fail. Among other things, a validation should attempt to confirm that the accuracy and resolution of the measurement are suitable for the intended use.

When a measurement has imperfections that give rise to an error, the error is either random or systematic [JCGM 100, 2008]. If the error is random and small it can be dealt with through taking large samples; however, if it is so significant as to be practically insurmountable, the instrument is obviously unsuitable for use. Discovering this fact up front is both easy and important. If the error is systematic, it is a kind of bias that can and should be corrected for. For example, if under given conditions the instrument is known to read low by a fixed amount or fixed factor, then one can add this fixed amount or factor to each result.

Bias, *precision*, *accuracy*, and *resolution* are frequently confused but important to distinguish. These concepts are further explained in Figure 4, Figure 5, and the following subsections.

4.2.1 Bias

The bias of an instrument is an indication of the instrument's closeness to a set standard. It is not always easy to determine the bias of an instrument. It is usually trusted that the instrument was properly calibrated by the manufacturer. Such an assumption is not unreasonable; however, it is not impossible to encounter biased instruments. See, for example, the news item by Anthony [2013].

4.2.2 Precision

When measuring the same characteristic of an object multiple times under the same conditions, an instrument should produce the same result. In other words, the measurement process should have little variability. A tight distribution of repeated measurements is an indication of precision [Lilja, 2000] (see Figure 4). The International Vocabulary of Metrology [JCGM 200, 2012] uses measurement precision to define measurement repeatability and measurement reproducibility.

There are several ways to report the precision of results. The simplest is the range (the difference between the highest and lowest results). It can also be quantified by calculating the standard deviation of the results or other statistical measures to express variance.

4.2.3 Accuracy

If a process has both a small bias and little variability, then it is called *accurate*. Equivalently, an accurate process is one that has both low bias and high precision; or one could say that a process is accurate if it is both unbiased and precise.

As Figure 4 illustrates, there are three ways that a process can be inaccurate: it can be biased, it can be imprecise, or it can be both.

4.2.4 Resolution

The resolution of an instrument is the smallest incremental change that can be observed by the instrument (see Figure 5). If one is interested in differences that are likely to be orders of magnitude smaller than the resolution of a given instrument, that instrument is not suitable to measure it. In cases where the differences are close enough to the resolution of the instrument and the measurement is unbiased, the mean of a series of measurements should converge to the correct value through having the correct proportion of x and $x + 1$ results. If it is biased (e.g., always rounds down or always rounds up) then this doesn't work.

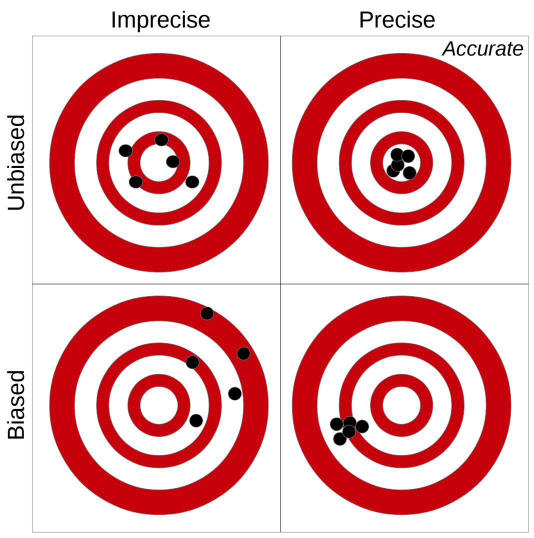
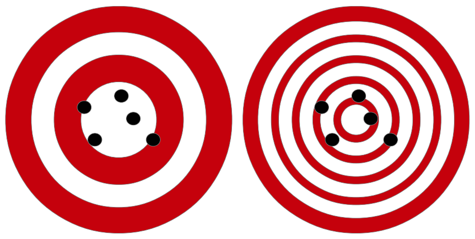


Figure 4: Precision, bias, and accuracy



Lower resolution, Higher resolution

Figure 5: Resolution

Keep in mind that the resolution of an instrument may vary depending on external factors. For example, the resolution of user and system time results reported by the bash shell's builtin **time** command can vary from 0.01 s to 0.001 s depending on the Linux kernel's HZ timer value.

4.2.5 Limitations

One of course attempts to choose instruments so that bias is minimized and precision is maximized (hence the accuracy is maximized), the resolution is to the desired level, and uncertainty is minimized, but this is an optimization problem where some of these aspects may be compromised to avoid unacceptable levels of other aspects. The burden is then on experimental design to perform the analysis in a manner that controls and corrects for the known uncertainties and biases and attempts to detect unknown factors.

Even after validating instruments to the maximum practical extent, one can never be certain that unexpected measurement error will not creep in. Such error can result from latent faults in the tooling or from incomplete or erroneous documentation of tool usage and functionality. This leads to problematic, epistemic uncertainty that is particularly acute whenever measurement results seem odd yet plausible. Without additional experiments, one cannot be certain whether the odd behavior is actual performance or is an artifact; additional experiments may contrariwise produce yet more unexpected behaviors without resolving the original question. However, despite the human tendency to be less anxious about normal-looking results, there is no reason to believe that they would be immune to measurement error.

4.3 Automation

The process to gather data in situations where certain factors are kept constant while others are changed in predictable ways lend itself to automation, especially if the factors that are varied can be scripted. Metrologists are encouraged to create such automation and share it with the community. Curtsinger and Berger [2013] created software [Berger] to automate randomization of memory configurations. It forces executions to sample over the space of all memory configurations of C++ programs by efficiently and repeatedly randomizing the placement of code, stack, and heap objects at runtime. Similarly, Georges et al. [2007] created software [Georges] that runs Java benchmarks automatically until a sufficiently narrow confidence interval is obtained for a given performance metric.

5 Experimental design

When reviewing available resources for experimental designs, one will find that techniques are grouped by broad categories of experimental goals. Popular design categories include:

- Screening or sensitivity analysis;
- Comparisons;
- Optimization;
- Regression, modelling, estimation;
- Reliability assessment.

The issues discussed in this section need to be addressed to achieve an experimental design that fits the goal of the experiment given the instruments that were chosen to reach the goal. Additional overview of design of experiments in the context of computer performance evaluation can be found in Schatzoff [1981].

5.1 Constraints

An important part of the design procedure is the formal elicitation of constraints on the number of runs, number of configurations, time, equipment, personnel, money, etc., that can be devoted to the experiment. Experiment design as a discipline is the structured tradeoff between scientific goals and practical constraints.

5.2 Uncertainty

Simply comparing the average result of a number of repeated results often leads to incorrect conclusions, particularly if the variability of the result is high. A lack of statistical rigor has been a major weakness in the field of computer systems performance analysis [Lilja, 2000].

One can deal with uncertainty empirically or analytically. With an empirical approach, uncertainty is estimated based on repeated measurements, whereas with an analytical approach this estimation is based on a mathematical model and the application of probability theory.

Ultimately, the degree of uncertainty determines the confidence level of the conclusions that can be drawn from the measurements. However, the conclusions may be based on arbitrarily complex functions of the measurement values, and those functions affect the propagation of uncertainty from the measurements to the output. When measurement values are subjected to multiple comparisons the uncertainty is amplified, necessitating higher confidence levels for each individual measure in order to attain an adequately high confidence level for the conclusions. Methods applicable to these problems are introduced in Flater [2014] and described in detail in JCGM 100 [2008] and JCGM 101 [2008].

5.3 Random errors

Random errors are errors that cannot necessarily be explained. They may be the result of the measuring tool, the observer reading the output of the tool, or random processes within the system being studied [Lilja, 2000]. Random errors affect the precision of the measurements and thereby determine the repeatability of the results.

The metrologist should apply an appropriate statistical model to describe the effect of random errors on the experimental results in order to quantify the precision of the measurements. Most experimental errors can reasonably be modeled using a Gaussian distribution centered around the true value, but there are exceptions. When random errors are not Gaussian, either the results must be normalized or a more robust uncertainty method must be applied before conclusions can be made based on the measurements.

Traditionally, a measurement value with random error is expressed in the form $y \pm u$. Ultimately, however, the confidence interval matters more than the value of u. In the general case, the output of the measurement can take on any probability distribution, and the distribution need not be symmetrical around the chosen estimate (which is usually the mean of some number of repeated measurements). Furthermore, there is more than one reasonable way to derive a confidence interval for a given coverage probability [JCGM 101, 2008, §5.3] and the different approaches often yield different intervals when the distribution is not Gaussian.

Flater [2014] includes examples of non-Gaussian error distributions in software performance measurements and the use of alternative methods to deal with them.

5.4 Systematic errors (bias)

Systematic errors are predictable errors that are intrinsic to the measuring instrument or process. They can be caused by faulty instruments or by the wrong use of instruments. Systematic errors cause a constant bias in the measurements (see Section 4.2) that must be detected, controlled, and adjusted for [Patil and Lilja, 2012]. JCGM 100 [2008, §F.2.4.5] recommends that corrections be applied to measurement results for known significant systematic effects and gives guidelines on how to calculate a mean or average correction when individual correction may be infeasible.

From a practical standpoint, systematic errors can be more difficult to deal with than random errors because they cannot be detected just from the variability of results and cannot be reduced by merely increasing the sample size and averaging the outcomes. On the other hand, correcting for known systematic errors does not necessitate an increased sample size.

Bias adjustments are frequently difficult. In the physical sciences, bias can be addressed by measurement standards, otherwise-derived ground truth, multiple measurement methods, and models (sometimes called "phantoms") that are specially designed for the purpose. In experimental computer science, there are few applicable measurement standards, ground truth is frequently unobservable (e.g., CPU time is entirely a function of internal state), and the independent validity of different measurement methods is difficult to establish. The design of valid "phantom programs" that provide the transparency required to calibrate software measures is an interesting research challenge.

5.5 Designing the workload

The success and validity of experiments rely heavily on the design of the workloads used. A well-designed workload requires the application of deep knowledge of the factors contributing to the phenomenon under observation along with sound statistical planning. The workload should be designed to be representative of conditions that may occur when the system under test is used for its intended purpose. The characteristics of real workloads can be determined through sampling typical workloads, whereafter a synthetic workload can be designed that displays characteristics similar to that of the sample.

When the number of contributing factors is large, a representative workload may be too large for practical application. In such cases, fractional factorial design is recommended [Schatzoff, 1981; Gunst and Mason, 2009; NIST/SEMATECH, 2012; Montgomery, 2013]. For screening, a carefully selected subset of the experimental runs of a full factorial design is used to identify those factors that have the most notable impact. After identifying the notable factors, a smaller workload can be designed that varies only the identified factors and interactions, but with better coverage.

One should strive to create a benchmark suite that is diverse enough to reveal measurement bias yet small enough to allow the completion of the experiment within the constraints that were identified. Kalibera and Jones [2013] address the question of how to determine the minimum number of repetitions needed to obtain adequate results in various situations. The minimal sample size needed for a design is a function of the natural variability in the data and the desired tolerance that the researcher needs for the estimates. For the special case of estimating the population mean given the sample mean of well-behaved data, one can apply standard formulae.

When synthetic test data are generated, it is advised that the pseudorandom number generator used for this purpose as well as the generated data be evaluated for randomness. NIST provides a statistical test suite for random number generators for cryptographic applications [Rukhin et al., 2010]; although the generator need not meet cryptographic requirements, those statistical tests are very relevant and useful.

6 Measurement techniques

6.1 "Wall clock"

The simplest measurement technique is sometimes the best. If one is interested only in the total elapsed execution time for an application, one can time it with an independent timer or (less independently) use any of the many simple methods for recording elapsed time that are normally available, such as the bash shell's builtin **time** command.

At this highest level of granularity, the only difference between hardware and software performance measurements is which factor (the hardware platform or the software application) is controlled and which one is varied. Such measurements can be obtained without recourse to sophisticated tools, but the results are correspondingly less useful if it is necessary to improve the software's performance.

6.2 Sampling-based profiling

Measuring software performance at a finer granularity is called *profiling*. The term *profiling* is used to refer to the process of acquiring fine grained information about an application or process while it is being executed. With measurements scoped to the function level or below, it becomes possible to identify and remedy previously unknown performance bottlenecks in the software with less experimentation.

The sampling-based approach to profiling entails interrupting the application under test on specified events to collect data on what it was doing when the interrupt occurred. The metrologist has to decide on what events profiling data should be taken. Events can range from a specified value of a timer count to tracepoints to specific kernel events. Appropriate parameters should be chosen to control the frequency of sampling. The frequency of sampling should be tuned to account for the capabilities of the system and the needs of the measurement.

When timer events are used, the frequency is related to the frequency of the timer and the chosen value. When other events are used, sampling may be irregular and may occur in bursts. When the frequency is too high, the data collection tools may lose samples or the overhead of profiling may skew or overwhelm the application under test. If it is too low, it can result in too few samples taken, rendering the data inadequate to draw any statistically significant conclusions.

It is important to have enough samples to produce reliable results. The number of samples is controlled by the duration of the experiment and the sampling rate. While a higher sampling rate will yield more samples resulting in less statistical uncertainty, it also increases the perturbation of the program caused by the measurement process, which has a negative effect on the reliability of the results. The metrologist has to find a workable compromise.

Another factor that is important to take note of is that the samples should be taken in a manner that does not bias the results. By taking the samples at fixed intervals, it is assumed that these interrupts will be distributed uniformly throughout the execution of the program. However, if there is any synchronism between the program events and the sampling interrupts, it may happen that some areas of the program are sampled less or more often than they should be given their actual frequency of occurrence.

6.3 Tracing

A trace of a program is a chronological list of the events generated by the program as it executes. Traces can be analyzed to characterize the behavior of a program. Traces are often used for debugging but can also be used to gather detailed information about a wide variety of traceable events of interest such as the sizes and destinations of messages sent over a network.

Traces can be generated by adding statements in the code, in linked objects used by the code, or in the interpreter (for interpreted languages) to output the required information. Normally the process is automated by available tooling and infrastructure. For example, the Debug class enables function entry/exit tracing of Java Android apps, and the Linux kernel is already instrumented to trace all kinds of events that are visible from kernel space, such as system calls made by a program. The Java Virtual Machine Tool Interface [Oracle] is an instrumentation of a Java VM that can similarly be applied.

Another popular option for generating program traces is to use an emulator or emulation layer to run the program. An emulator executes the program and can at the same time gather the required information; an emulation layer traces events as operations pass through it. Examples are the Intel Software Development Emulator [Tal, 2012] and DynamoRIO [DynamoRIO].

Tracing has the potential to slow down or influence the behavior of the program or process under observation significantly. The metrologist has to take such changes in behavior into account when analyzing the data and provide adequate arguments to substantiate the claims that are made despite these possible changes.

The volume of data gathered during a trace depends on the experimental design and the needs of the metrologist. When generating detailed information, execution time overhead might be reduced by saving all the gathered information with the intention to summarize and analyze it later; however, the amounts of memory or disk space and associated I/O needed to store the unprocessed data can also have a negative impact. In some cases it may be better to gather fewer data and summarize them on the fly; for example, it will suffice to count the number of times each basic block in the code was executed if one is interested only in finding the code block that is executed most often, regardless of when or why it is executed.

7 Some factors that may be hard to control

All measurement is subjected to some degree of uncertainty and in many cases there also is bias. Understanding the reasons why the measurements that are made are likely to be biased can help the metrologist to avoid some and mitigate other sources of bias. The subsections that follow provide guidelines on how to deal with bias by considering sources of bias when designing benchmarking experiments and when reporting results.

7.1 Monitoring overhead

Monitoring overhead is the undesirable side-effects of the measurement tools. The software that is used to gather data consumes processor, storage and I/O resources, and that may have adverse effect on the measurements, regardless whether sampling or tracing is used.

One can sometimes measure the overhead by running null benchmarks and then subtract it as a form of systematic error. One must be cautious, however, to distinguish constant overhead from variable overhead and normal states from overloaded states. For example, the time overhead of tracing function entries and exits might be constant in a per-invocation sense, but that overhead must then be scaled by the number of function invocations to obtain a correction for the program as a whole. Also, as the amount of data collected increases, at some point some resource can become saturated by the overhead of saving or processing the data, and then performance will be dramatically impacted in a way that no null benchmark will predict.

Figure 6 shows the results of an example "pre-experiment" that was conducted to find the usable range of sampling frequencies. Data were collected from repeated runs of the synthetic test program that was used in Flater [2014]. The linear progression of sample counts versus sampling frequency that appears in the midrange is the correct behavior. At the low end, the sample counts are so low that quantization error skews the result. At the high end, clearly there is a limit beyond which things break down. With sampling periods of (10^4 and $10^{3.5}$) cycles, the sample counts failed to increase as expected, showing that some limit had been reached. With sampling periods of 10^3 and $10^{2.5}$, the resolution of the sample counts unexpectedly fell by a factor of 1000 (they all came back as even multiples of 1000). Finally, with a sampling period of $10^{2.5}$ cycles, 5 of the 200 repetitions failed to return any data at all.

A high sampling frequency or heavy tracing not only escalates the amount of direct overhead for record-keeping, it also increases the amount of cache disruption that occurs, which indirectly worsens the performance of the software under test.

Zaparanuks et al. [2009] conducted a well-designed experiment to determine the monitoring overhead of a number of configurations that allow user-level access to per-thread hardware counters in Linux. They observed that the monitoring overhead differs drastically between configurations. A surprising anomaly observed in these experiments is that disabling the time stamp counter (TSC, a hardware counter) in the perfctr tool when there is no need for the TSC results significantly degrades accuracy. The explanation for this anomaly is that perfctr, when the TSC is enabled, can read counters from user mode without calling into the kernel.

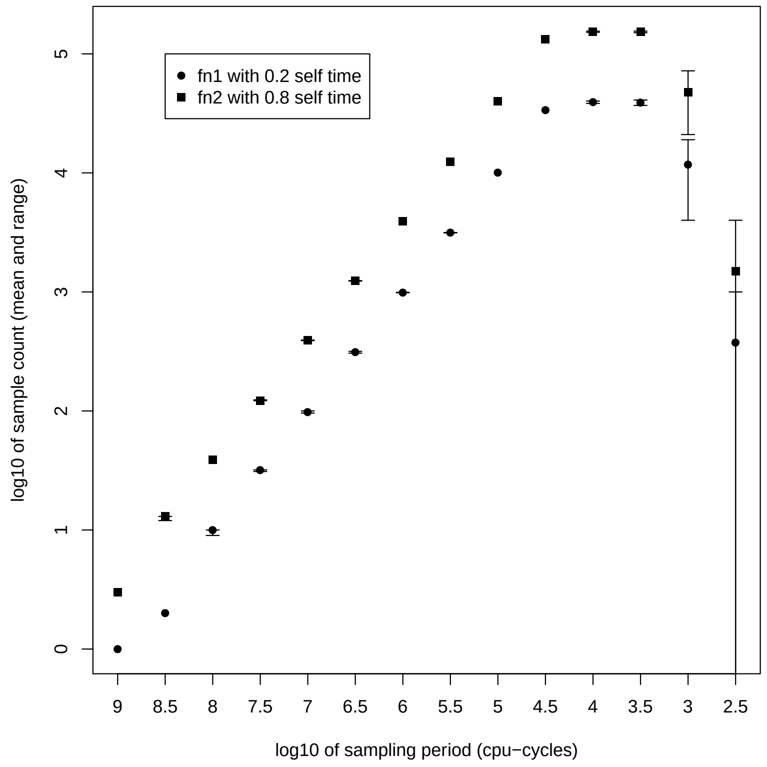


Figure 6: Determining acceptable sampling rates (plot shows mean sample counts with bars indicating the ranges of sample counts)

7.2 Time-dependency and other stateful behaviors

When the execution of a process under investigation—or simply the passage of time, for that matter—changes the state of the system in a nonrandom way that causes the next execution of the same process to behave differently, the statistical assumption that the results of repeated measurements are "independent and identically distributed" (i.i.d.) is violated, necessitating additional care in the design of the experiment and the interpretation of results.

Caching of memory, hard disk sectors, or anything else generally has a significant effect on performance when these resources are used. Another effect is that the processor will heat up during the execution of a CPU-intensive benchmark, resulting in a change of behavior of the processor owing to dynamic thermal management applied by the system [Brooks and Martonosi, 2001].

If a virtual machine is involved, reusing the same virtual machine instance and possibly even the same instance of the app (as is standard practice in Android) creates another entire layer of statefulness. Georges et al. [2007] propose rigorous techniques and have developed tools to automate Java performance evaluation. NIST has a test driver to repeatedly launch Android apps [RTD, 2013].

7.3 Memory layout

The mapping of an application's virtual address space into physical memory can change on every execution of a given program, and this alone is sufficient to impact the effectiveness of caching. Memory layout can further change as a result of deliberate layout randomization for hardening against attacks and, of course, from recompilation with different options.

Mytkowicz et al. [2009] observed that the performance of software can be extremely sensitive to memory layout. The variability reported in Mytkowicz et al. [2009], however, was not found by Kalibera and Jones [2013] to reproduce at such magnitude if the reference sizes of the benchmarks were used.

Zaparanuks et al. [2009] observed that cycle count measurements can be drastically perturbed by code placement. Curtsinger and Berger [2013] maintain that two versions of a program, regardless of the number of runs, are only two samples from the distribution over possible layouts. Performance penalties imposed by cache misses or branch mis-predictions are somewhat unpredictable and may cause substantial variability. Mytkowicz et al. [2009] suggest experimental setup randomization by applying different link orders and environment variable sizes to control for layout effects. Using randomization to combat sources of bias is a standard practice of experimental design; Tsafrir et al. [2007] applied it to a scheduler simulation.

7.4 Parallelism effects

With parallelism having been incorporated into multiple levels of every common architecture (including, at least, multiple functional units in the microarchitecture and multiple cores in the CPU), it has become very difficult to avoid perturbations in the sequence and timing of instruction execution. The resulting variability can be chaotic: "The combination of simple caches and multithreading can produce high and unpredictable cache miss rates, even when the compiler optimizes the data layout of each program for the cache." [Sarkar and Tullsen, 2008]

Although taking measurements with all of the parallelism turned off is neither feasible nor representative of actual performance, it is often reasonable to take measurements with the operating system in a single-user mode, minimizing the impact of uncontrolled competing loads. Desktop PCs sometimes further allow one to disable symmetric multiprocessing features in BIOS setup, which may be helpful and appropriate depending on the type of application and measurement being attempted.

7.5 Compiler optimization effects

Differing compiler optimization options can drastically change a program's performance [Flater and Guthrie, 2013; Zaparanuks et al., 2009]. Worse yet, code elimination can produce degenerate results. A compiler will frequently delete variables whose values are not used and cascade this deletion onto the executable statements that generated those values. Any synthetic workload intended to do nothing except create a predictable amount of busy-work is vulnerable to being eliminated in this way, yet turning off optimization entirely yields results that are significantly skewed from the normal case. Flater [2013] applied two techniques to combat this. First, declaring variables with global visibility makes it difficult for the compiler to conclude that their values are never used or altered by external code. Second, printing their values at the end of the program ensures that they cannot be eliminated unless the compiler can determine their final values through static analysis. In test cases that take no external input, it is still theoretically possible for a compiler to optimize away the entire workload and substitute a printout of the final result, but these two techniques have proven effective thus far. Similar challenges were reported in Watson [1995].

7.6 Short function anomalies

Flater [2013] observed statistically significant skewing of the reported self time of a small function relative to the expected results. The realm of plausible explanations for this anomaly includes both measurement bias and actual differences in performance. Regarding the former, the OProfile manual warns that x86 hardware-specific latencies in the delivery of profiling interrupts can skew the results in certain cases [Levon, 2012]. However, it is also plausible that the short functions brought an actual performance feature of the micro-architecture, such as branch prediction, cache, or CPU pipeline scheduling efficiencies, to the forefront.

It is advisable to watch for short, frequently-invoked functions in profile reports and use results with extra caution whenever they are found. The call counts provided by gprof are helpful in determining which functions could be problematic.

8 Conclusion

Despite evidence that benchmarking experiments reported in the literature often lack statistical rigor, awareness of the problem seems to be growing. The establishment of the Evaluate Collaboratory in 2010 and their ongoing work to promote the quality of reported experimental evaluation of computer systems boosted this awareness. A vast body of knowledge covering experimental design exists; however, computer scientists are required to apply these general principles to a specific and ever-changing domain. This paper is but a drop in the ocean to help software performance metrologists to gain better understanding of the work that has to be done. Deeper understanding of the requirements as well as awareness of the problems associated with this work may improve the quality of future benchmarking experiments.

We described the life cycle of performance measurement experiments and the interplay between the different aspects of the life cycle. We discussed the requirements to ensure the quality of all the steps that have to be taken to conduct an experiment to evaluate a computer system. We hope that it will assist software performance metrologists not only to design and implement better software performance experiments, but also to gauge the quality and validity of claims based on the outcome of reported software performance experiments they may encounter.

Acknowledgment

Thanks to Jeffrey Fong of the Applied and Computational Mathematics Division and other reviewers for their contributions. Special thanks to Jim Filliben of the Statistical Engineering Division for helping to make this introductory text more consistent with formal guidance.

References

AnandTech forum participant (IDC). Effect of temperature on power-consumption with the i7-2600k. http://forums.anandtech.com/showthread.php?t=2200205, October 2011. [Online] accessed 2013-12-04.

Sebastian Anthony. Windows 8 banned by world's top benchmarking and overclocking site. http://www.extremetech.com/computing/164209-windows-8-banned-by-worlds-top-benchmarking-and-overclocking-site, August 2013. [Online] accessed 2013-12-04.

Emery Berger. Stabilizer. URL http://plasma.cs.umass.edu/emery/stabilizer.

Sion Berkowits. Pin - A Dynamic Binary Instrumentation Tool, June 2012. URL http://software.intel.com/en-us/articles/pintool.

R. Berry. Computer benchmark evaluation and design of experiments: A case study. *IEEE Transactions on Computers*, 41(10):1279–1289, 1992. ISSN 0018-9340. doi: 10.1109/12.166605.

Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeney, José Nelson Amaral, Vlastimil Babka, Walter Binder, Tim Brecht, Lubomír Bulej, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Robin Garner, Andy Georges, Laurie J. Hendren, Michael Hind, Antony L. Hosking, Richard Jones, Tomas Kalibera, Philippe Moret, Nathaniel Nystrom, Victor Pankratius, and Petr Tuma. Can you trust your experimental results? Technical Report #1, Evaluate Collaboratory, February 2012. URL http://evaluate.inf.usi.ch/technical-reports/1.

D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 171–182, 2001. doi: 10.1109/HPCA.2001.903261.

D. Brooks, R. P. Dick, R. Joseph, and Li Shang. Power, thermal, and reliability modeling in nanometer-scale microprocessors. *IEEE Micro*, 27(3):49–62, 2007. ISSN 0272-1732. doi: 10.1109/MM.2007.58.

Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190067.

Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Annual Technical Conference*, page 21, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855861.

Charlie Curtsinger and Emery D. Berger. Stabilizer: statistically sound performance evaluation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–228. ACM, 2013. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451141. URL http://dx.doi.org/10.1145/2451116.2451141.

Dr. Memory. URL http://www.drmemory.org.

Dragon Systems Software Ltd. Sleep Monitor. URL http://www.dssw.co.uk/sleepmonitor/.

DynamoRIO. URL http://dynamorio.org/home.html.

Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back and looking forward: power, performance, and upheaval. *Communications of the ACM*, 55(7):105–114, July 2012. ISSN 0001-0782. doi: 10.1145/2209249.2209272. URL http://doi.acm.org/10.1145/2209249.2209272.

Evaluate Collaboratory. URL http://evaluate.inf.usi.ch/.

Jay Fenlason. GNU gprof, 1988. In GNU binutils, http://www.gnu.org/software/binutils.

David Flater. Configuration of profiling tools for C/C++ applications under 64-bit Linux. NIST TN 1790, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, March 2013. URL http://dx.doi.org/10.6028/NIST.TN.1790.

David Flater. Estimation of uncertainty in application profiles. NIST TN 1826, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, 2014. URL http://dx.doi.org/10.6028/NIST.TN.1826.

David Flater and William F. Guthrie. A case study of performance degradation attributable to run-time bounds checks on C++ vector access. *Journal of Research of the National Institute of Standards and Technology*, 118:260–279, 2013. URL http://dx.doi.org/10.6028/jres.118.012.

Jason Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, 1999. doi: 10.1109/MCSA.1999.749272.

José Fonseca. Gprof2Dot: Convert profiling output to a dot graph, April 2013. http://code.google.com/p/jrfonseca/wiki/Gprof2Dot.

Andy Georges. JavaStats. URL http://www.elis.ugent.be/JavaStats.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices - Proceedings of the 2007 OOPSLA conference*, 42(10):57–76, October 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297033. URL http://dx.doi.org/10.1145/1297105.1297033.

Francis Giraldeau, Julien Desfossez, David Goulet, Michel Dagenais, and Mathieu Desnoyers. Multiple page size support in the Linux kernel. In *Proceedings of the Ottawa Linux Symposium*, pages 573–593, Ottawa, Canada, 2002. URL https://www.kernel.org/doc/mirror/ols2002.pdf.

Francis Giraldeau, Julien Desfossez, David Goulet, Michel Dagenais, and Mathieu Desnoyers. Recovering system metrics from kernel trace. In *Proceedings of the Ottawa Linux Symposium*, pages 109–115, Ottawa, Canada, 2011. URL https://www.kernel.org/doc/mirror/ols2011.pdf.

B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *Proceedings of the 2010 International Green Computing Conference*, pages 135–146, 2010. doi: 10.1109/GREENCOMP.2010.5598313.

Richard F. Gunst and Robert L. Mason. Fractional factorial design. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(2):234–244, 2009. ISSN 1939-0068. doi: 10.1002/wics.27. URL http://dx.doi.org/10.1002/wics.27.

Scott Helvick. A survey of software monitoring and profiling tools. http://www.cse.wustl.edu/~jain/cse567-08/ftp/hw/index.html, 2008. [Online] accessed 2013-12-04.

Jenifer Hopper. Untangling memory access measurements - lat_mem_rd. https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W51a7ffcf4dfd_4b40_9d82_446ebc23c550/page/Untangling%20memory%20access%20measurements%20-%20memory%20latency, May 2013. [Online] accessed 2013-12-04.

Raj Jain. *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, Inc., New York, USA, 1991. ISBN 0471503363.

JCGM 100. *Evaluation of measurement data—Guide to the expression of uncertainty in measurement.* Joint Committee for Guides in Metrology, 2008. URL http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf.

JCGM 101. *Evaluation of measurement data—Supplement 1 to the "Guide to the expression of uncertainty in measurement"—Propagation of distributions using a Monte Carlo method.* Joint Committee for Guides in Metrology, 2008. URL http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf.

JCGM 200. *International vocabulary of metrology—Basic and general concepts and associated terms (VIM)*. Joint Committee for Guides in Metrology, 3rd edition, 2012. URL http://www.bipm.org/en/publications/guides/vim.html.

Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Statistical Analysis*. Pearson Higher Education, Cranbury, NJ, USA, 6th edition, 2007. ISBN 0131877151.

Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, pages 63–74. ACM, 2013. ISBN 978-1-4503-2100-6. URL http://dl.acm.org/citation.cfm?id=2464160.

Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 184–193, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7977-8. URL http://dl.acm.org/citation.cfm?id=266818.

John Levon. Profiling interrupt latency, in OProfile manual. http://oprofile.sourceforge.net/doc/interpreting.html#irq-latency, August 2012. [Online] accessed 2013-12-04.

Adam Lewis, Soumik Ghosh, and N.-F. Tzeng. Run-time energy consumption estimation based on workload in server systems. In *Proceedings of the 2008 Conference on Power-Aware Computing and Systems*, page 4, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855614.

David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

Dan Magenheimer. Transcendent memory in a nutshell, August 2011. URL http://lwn.net/Articles/454795/.

Scott E. Maxwell and Harold D. Delaney. *Designing experiments and analyzing data: A model comparison perspective*. Psychology Press, 2004.

Microsoft Research. Joulemeter. URL http://research.microsoft.com/en-us/downloads/fe9e10c5-5c5b-450c-a674-daf55565f794/.

Douglas C. Montgomery. *Design and analysis of experiments*. John Wiley & Sons, Inc., New York, USA, 8th edition, 2013. ISBN 1118146921.

K. Muroya, T. Kinoshita, H. Tanaka, and M. Youro. Power reduction effect of higher room temperature operation in data centers. In *2010 IEEE Network Operations and Management Symposium*, pages 661–673, 2010. URL http://dx.doi.org/10.1109/NOMS.2010.5488420.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGARCH Computer Architecture News*, 37(1):265–276, March 2009. ISSN 0163-5964. doi: 10.1145/2528521.1508275. URL http://doi.acm.org/10.1145/2528521.1508275.

NIST/SEMATECH. NIST/SEMATECH e-Handbook of Statistical Methods, 2012. URL http://www.itl.nist.gov/div898/handbook/.

Oracle. Java Virtual Machine Tool Interface (JVMTI). URL http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/.

Shruti Patil and David J. Lilja. Statistical methods for computer performance evaluation. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(1):98–106, 2012. ISSN 1939-0068. URL http://dx.doi.org/10.1002/wics.192.

Perf. Linux profiling with performance counters, 2013. https://perf.wiki.kernel.org/.

PowerTOP. URL https://01.org/powertop/.

Garrison Prinslow. Overview of performance measurement and analytical modeling techniques for multi-core processors. http://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore/index.html, 2011. [Online] accessed 2013-12-04.

Mathew Robbin. Interpreting CPU utilization for performance analysis. http://blogs.technet.com/b/winserverperformance/archive/2009/08/06/interpreting-cpu-utilization-for-performance-analysis.aspx, August 2009. [Online] accessed 2013-12-04.

RTD. Test driver for Android, version 1.0, available at Software Performance Project web page, 2013. http://www.nist.gov/itl/ssd/cs/software-performance.cfm.

Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo, and Lawrence Bassham III. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST SP 800-22 Rev. 1a, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, April 2010. URL http://www.nist.gov/manuscript-publication-search.cfm?pub_id=906762.

Bharath K. Samanthula, Hu Chun, Wei Jiang, and Bruce M. McMillin. Secure and threshold-based power usage control in smart grid environments. *International Journal of Parallel, Emergent and Distributed Systems*, pages 1–26, 2013. doi: 10.1080/17445760.2013.850690. URL http://www.tandfonline.com/doi/abs/10.1080/17445760.2013.850690.

Subhradyuti Sarkar and Dean M. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In Per Stenström, Michel Dubois, Manolis Katevenis, Rajiv Gupta, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 4917 of *Lecture Notes in Computer Science*, pages 353–368. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-77559-1. doi: 10.1007/978-3-540-77560-7_24. URL http://dx.doi.org/10.1007/978-3-540-77560-7_24.

Martin Schatzoff. Design of experiments in computer performance evaluation. *IBM Journal of Research and Development*, 25(6):848–859, November 1981.

Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architecture News*, 37(2):46–55, July 2009. ISSN 0163-5964. doi: 10.1145/1577129.1577137. URL http://doi.acm.org/10.1145/1577129.1577137.

SiSoftware staff. Benchmarks: Measuring Cache and Memory Latency (access patterns, paging and TLBs). http://www.sisoftware.net/?d=qa&f=ben_mem_latency, December 2012. [Online] accessed 2013-12-04.

M. Stockman, M. Awad, R. Khanna, C. Le, H. David, E. Gorbatov, and U. Hanebutte. A novel approach to memory power estimation using machine learning. In *Proceedings of the 2010 International Conference on Energy Aware Computing*, pages 1–3, 2010. doi: 10.1109/ICEAC.2010.5702284.

Ady Tal. Intel Software Development Emulator, June 2012. URL http://software.intel.com/en-us/articles/intel-software-development-emulator/.

J. Torrellas, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *14th International Conference on Parallel Architectures and Compilation Techniques*, pages 339–349, 2005. doi: 10.1109/PACT.2005.32.

D. Tsafrir, K. Ouaknine, and D. G. Feitelson. Reducing performance evaluation sensitivity and variability by input shaking. In *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 231–237, 2007. doi: 10.1109/MASCOTS.2007.58.

Bogdan Marius Tudor and Yong Meng Teo. On understanding the energy consumption of ARM-based multicore servers. In *Proceedings of the ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, pages 267–278. ACM, 2013. ISBN 978-1-4503-1900-3. doi: 10.1145/2465529.2465553. URL http://dx.doi.org/10.1145/2465529.2465553.

John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

Bhuvan Urgaonkar, George Kesidis, Uday V. Shanbhag, and Cheng Wang. Pricing of service in clouds: Optimal response and strategic interactions. In *Proceedings of the Workshop on Mathematical Performance Modeling and Analysis*. ACM, 2013. URL http://www.cse.psu.edu/~bhuvan/papers/ps/cloudnegot.pdf.

Valgrind. URL http://www.valgrind.org.

Jun De Vega. Intel Power Gadget, October 2013. URL http://software.intel.com/en-us/articles/intel-power-gadget-20.

Jan Vitek and Tomas Kalibera. R3: repeatability, reproducibility and rigor. *ACM SIGPLAN Notices*, 47 (4a):30–36, April 2012. ISSN 0362-1340. URL http://dx.doi.org/10.1145/2442776.2442781.

Hao Wang, Jianwei Huang, Xiaojun Lin, and Hamed Mohsenian-Rad. Exploring smart grid and data center interactions for electric power load balancing. In *Proceedings of the Greenmetrics Workshop*. ACM, 2013. URL http://www.sigmetrics.org/greenmetrics/2013/papers/longpaper_F.pdf.

Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Technische Universiteit Eindhoven, 1995. URL http://www.fastar.org/publications/PhD_Watson.pdf.

Christopher L. Weber, Paulina Jaramillo, Joe Marriott, and Constantine Samaras. Life cycle assessment and grid electricity: What do we know and what can we know? *Environmental Science & Technology*, 44(6): 1895–1901, 2010. doi: 10.1021/es9017909. URL http://pubs.acs.org/doi/abs/10.1021/es9017909.

Wikipedia. List of performance analysis tools. URL http://en.wikipedia.org/wiki/List_of_performance_analysis_tools.

Ben Wun. Survey of software monitoring and profiling tools. http://www.cse.wustl.edu/~jain/cse567-06/sw_monitors2.htm, 2006. [Online] accessed 2013-12-04.

Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32, April 2009. URL http://dx.doi.org/10.1109/ISPASS.2009.4919635.