Performance Analysis of Real-Time Systems

SP_08_SE_655

Real-time Software Performance Monitoring
Approaches, Difficulties, and Analysis Techniques

Christopher Griffis
Embry-Riddle Aeronautical University
4/07/08
Table of Contents

1. ABSTRACT ..................................................................................................................................................................1

2. INTRODUCTION ......................................................................................................................................................1

3. BACKGROUND ..........................................................................................................................................................1

   3.1 MONITORING: OBSERVATION VERSUS MEASURING .......................................................................................1

   3.2 DETERMINISM VERSUS NON-DETERMINISM ...............................................................................................2

   3.3 THE PROBE EFFECT ..............................................................................................................................................2

4. MONITORING SINGLE THREADED, MULTITHREADED, AND DISTRIBUTED REAL-TIME SYSTEMS .................................................................3

   4.1 SINGLE THREADED OR SEQUENTIAL SYSTEMS ...............................................................................................3

   4.2 MULTITHREADED UNIPROCESSOR REAL-TIME SYSTEMS ................................................................................3

   4.3 DISTRIBUTED REAL-TIME SYSTEMS ................................................................................................................3

5. APPROACHES TO NONINTRUSIVE MONITORING ..............................................................................................4

   5.1 HARDWARE PROBES ..............................................................................................................................................4

   5.2 SOFTWARE PROBES ..............................................................................................................................................4

   5.3 RECORD AND REPLAY .......................................................................................................................................5

   5.4 SIMULATION ........................................................................................................................................................5

   5.5 EXAMPLE MONITORING APPROACHES .............................................................................................................5

6. DIFFICULTIES OF REAL-TIME SOFTWARE PERFORMANCE MONITORING .......................................................6

   6.1 THE PROBE EFFECT, REVISITED .........................................................................................................................6

   6.2 QUANTIZATION ERRORS .....................................................................................................................................6

   6.3 DETERMINISM AND PERTURBATIONS ..............................................................................................................7

   6.4 LARGE AMOUNTS OF DATA ...............................................................................................................................7

7. ANALYSIS TECHNIQUES AND METRICS ..............................................................................................................7

   7.1 STATISTICAL PERFORMANCE MONITORING ..................................................................................................7

   7.2 EXAMPLE PERFORMANCE METRICS ............................................................................................................8

8. CONCLUSION ............................................................................................................................................................9

9. REFERENCES ............................................................................................................................................................9

10. PAPER PSP ..........................................................................................................................................................10

List of Figures

FIGURE 1: SAMPLED VALUES OF CPU UTILIZATION MEASURED OVER AN INTERVAL ........................................8
FIGURE 2: “BORING RANGE” VALUES REPLACED WITH THE AVERAGE VALUES ..............................................8

List of Tables

TABLE 1: PAPER PSP .............................................................................................................................................10
Real-time Software Performance Monitoring: Approaches, Difficulties, and Analysis Techniques

1. Abstract

This paper is a discussion of the obstacles associated with real-time software performance monitoring, focusing on the probe effect and its confounding influence. Given is a brief description of the concept of real-time software performance monitoring, the associated approaches to monitoring software in an effort to minimize some of the intrusiveness of the monitoring process, and some of the techniques for analyzing the monitored outcome. The context ranges from sequential or single threaded applications, to multithreaded applications, to real-time distributed systems, focusing on issues of determinism versus non-determinism, and done in light of the difficulties of the probe effect. Key example approaches to real-time software performance monitoring are provided to support the explanations and the discussion is concluded with summarizing observations.

2. Introduction

When monitoring the behavior of software running on computer systems, the concept of performance and performance improvement can be a nebulous one. The difficulty comes in deciding what to observe, how to observe it (via simple observation or measurement), and how to make sense of the results. It turns out that with software, performance monitoring can be more difficult to implement than it seems. Software has both structural and temporal dependencies, which are not only difficult to monitor in a meaningful and sensible way, but the mere act of monitoring can corrupt the observations (as compared to the undisturbed behavior). This disruption can cause behavior anomalies that will non-deterministically skew the behavior being monitored, a dilemma compounded by the complexity of multithreaded and distributed systems. Moreover, once this information is collected, it can often be too much to handle, or so filled with “uninteresting” data that it can be difficult to extract the desired analysis. In some cases, assessing the software performance can be done through statistical analysis of accumulated measurements, and in other cases, the software need not necessarily have its performance measured, but instead simply monitored to identify areas for improvement.

With respect to performance monitoring, real-time software systems not only have the same difficulties as regular software systems, but are also subject issues resulting from timing constraints and the complexity of multitasking (if applicable). These can profoundly compound and bring to pique the issues of non-determinism and probe effect that result from unrestrained software performance monitoring.

In general, it is of great interest to find alternate ways that will allow for effective performance monitoring with the least amount of system perturbation. There is a desire and strive for runtime performance monitoring, but this approach is not always accessible. This paper aims to give a brief description of the concept of software performance monitoring (as opposed to software performance measurement), the associated approaches to monitoring software in an effort to minimize some of the intrusiveness of the monitoring process, and some of the techniques for analyzing the monitored outcome. The scope and context will include a lightweight juxtaposition of single threaded, multithreaded, and distributed systems, and include a discussion on the use of hardware and software probes for performance monitoring.

3. Background

3.1 Monitoring: Observation Versus Measuring

In assessing the performance of some aspect of a software system, one may want to monitor its behavior. This can be done in a variety of ways, e.g. as a user observing the program in operation or, as a different example, using software probes to observe and/or record the mechanics of the software in operation. Regardless, the term monitoring implies the measurement, collection, and processing of information about the execution of tasks in a
computer system [3]. (It is important to note that this at monitoring as both an act of observation and/or an act of quantitative measurement, depending on the discussion).

The general requirements for runtime monitoring systems include low overhead, ease-of-use, and versatility [6], and they should continue to operate in the presence of static or dynamic failures [3]. However, the nature of software and the use of inline code or runtime entity observations by external programs can cause nontrivial perturbations to the system under observation. Ideally, interference caused to the system by the monitor’s presence must be predictable, minimal, and bounded [3].

When trying to make an assessment based on some recorded or observed behavior of the system, it is essential to choose metrics or points of observation that accurately reflect the behavior of interest. In general, measurements for software performance can be summarized as follows [5]:

- A count of how many times an event occurs
- The duration of some time interval
- The size of some parameter

Using quantitative measurements, one can benefit from the numbers to glean system insight for knowledge or comparison sake. Quantitative measurements produce information that can be subject to statistical analysis which may yield interesting relationships between data and allow for mathematically sound comparisons. Moreover, statistical analysis of measured data can provide a way to represent a complex collection of numbers as simple and manageable representations, ones that tend to forgive distracting information and provide summarizing insight. However, with regard to performance analysis, sometimes empirical information is not desired as much as the need to figure out how to improve performance. In this case, it may not be necessary to have all the measurements and data, but only the information necessary to improve performance (this brings to forefront the disparity between measuring a system's performance and simply observing a system's performance).

### 3.2 Determinism Versus Non-Determinism

For a given set of inputs and the same initial state, a deterministic system will always give the same ultimate outputs. This is often the case for many single threaded, sequentially running programs [7]. Determinism is desirable for situations when monitoring with software probes, because in some cases the effects of perturbation in the software probe can be exactly identified and compensated for in the calculated quantitative observations, effectively circumventing the probe effect [7]. Performance monitoring of deterministic systems when the probe effects are known and accounted for can increase reproducibility and decrease ambiguity, increasing confidence in the results.

Alternatively, a non-deterministic system, for various reasons, will not exhibit this property; under identical initial conditions and inputs, the output cannot be assumed to be the same as a previous trial with any certainty; at best its output can only described probabilistically [1][7]. This makes it particularly difficult to test and evaluate non-deterministic software, and even more difficult to observe and properly attribute the negative effects of perturbing the system as a consequence of monitoring its behavior.

### 3.3 The Probe Effect

Monitoring software includes both observation and measurement of its dynamic behavior in order to determine how well the software complies with requirements and meets certain performance expectations. Confidence in the results depends on the reproducibility and non-ambiguity of the results. Consequently, it is also essential that the monitoring process does not corrupt the results through introduced perturbations.

According to Henrik Thane, author of *Design for Deterministic Monitoring of Distributed Real-Time Systems*, “race conditions with respect to order of access to shared resources occur naturally in multi-tasking real-time systems. Different inputs to the racing tasks may lead to different execution paths. These paths will in turn lead to different execution times for the tasks, which depending on the design may lead to different orders of access to the shared resources. As a consequence there may be different system behaviors if the outcomes of the operations on the shared resources depend on the ordering of accesses [7].” Certain approaches to monitoring software include adding probes...
and unless dealt with, these probes can perturb the system in ways that will corrupt the monitored behavior. This accumulated corruption collectively impinges on reproducibility and introduces ambiguity, reducing confidence, and therefore must be dealt with.

4. Monitoring Single Threaded, Multithreaded, and Distributed Real-time Systems
Again, according to Henrik Thane, “there are two significant differences between debugging and testing of software for desktop computers and embedded real-time systems: (1) It is more difficult to observe embedded computer systems, simply because they are embedded, and that they thus have very few interfaces to the outside world, and (2) the actual act of observing a real-time systems or distributed real-time system can change their behavior.” [7] The following section will explain the concepts of sequential or single threaded systems versus multithreaded systems versus Distributed Real-time Systems (DRTS), and do so in the context of the difficulties just described.

4.1 Single Threaded or Sequential Systems
For the purposes of this paper single threaded systems and sequential programs will be addressed as a single idea. Essentially, for sequential or single threaded systems, there is a single thread of control, and there are no concurrent processes involved. Each software instruction is executed one-after-another, and the delays between each are knowable. Also, there are no race conditions or resource contentions in this situation. In many cases, sequential programs exhibit the desirable trait of being deterministic.

For testing of sequential programs it is usually sufficient to monitor inputs, and outputs via predefined interfaces of the programs, and based on that information deem (according to the specification) if a test run was successful or not. However, for distributed and multithreaded real-time systems we need also observe the timing and order of the executing and communicating tasks, since the outputs depend on these variables, and thus also the determinism of the observations. [7]

4.2 Multithreaded Uniprocessor Real-Time Systems
A multithreaded program running on a uniprocessor system has multiple tasks executing with the illusion of concurrency. This is possible via things like context switches, resource sharing, synchronization, message passing, etc. This scenario confronts more problems than a sequential program, which include: competition for resources among threads, slowdown due to synchronization, context switching overheads, conflicting considerations in scheduling, tradeoffs in task distribution, non-overlapped I/Os and high levels of interactions between these factors. [4] This can give rise to non-deterministic behavior under even the most docile conditions. Consequently, the probe effect must be taken into consideration and be addressed head of time (by design) for effective performance monitoring. Otherwise, the results could be useless or worse yet, misdirect the analysis.

4.3 Distributed Real-Time Systems
Distributed real-time systems are a collection of networked, intercommunicating, independent computing systems with their own local memory and processing resources. These systems and their applications are subject to the timing constraints inherent to a real-time system, and their distributed nature leads to extremely dynamic and complex behavior. Moreover, this is a complex environment from the perspectives of software design, development, testing, and operation. A key requirement for real-time system monitors is low overhead, but even more important is predictable overhead [3]; unpredictable overhead will compound the issues of non-determinism making performance monitoring even more difficult.

When monitoring a DRTS there are some fundamental questions that must be answered [7]:
- How can one extract enough (useable) information from the system?
- How can one eliminate the perturbations that the observations cause?
- How can one correlate the observations, i.e., how to define a consistent global time state?
- How can one reproduce the observations?
Monitoring of sequential software is straightforward, but for distributed real-time systems it is more complicated, since race conditions with respect to order of access to shared resources occurs naturally. Any intrusive observation or probing of the distributed real-time system affects the timing, and consequently the outcome of the races. Therefore, when trying to assess the performance characteristics through observation and or measurement, it is important to do so in a way that is the least intrusive.

5. Approaches to Nonintrusive Monitoring

When attempting to monitor the performance of a software-based real-time system, two fundamental issues must be acknowledged:

- How do I monitor useful information about my system, and
- How do I monitor without perturbing my system?

As straightforward as it may seem to the casual observer, performance monitoring is not a simple as it looks. The act of measurement itself can change the actual behavior of the monitored system (vs. the unmonitored version), a prospect that is compounded when looking at the intercommunicating distributed multiprocessor, or multithreaded uniprocessor real-time systems. Various approaches to circumvent this dilemma include using non perturbing hardware monitors, or minimally intrusive software probes. Limiting the data collected to only that which is useful or desirable can also mitigate the degree of perturbation.

5.1 Hardware Probes

Hardware probes are physically coupled to the system under observation, and can detect and record the values in memory and the time they occurred. This is useful because it provides the desired information with little to no perturbation on the system. “Since the monitoring hardware is interfaced to the target system’s hardware via the CPU socket (emulator) or via the data and address busses, it can observe the target system without interfering with its execution, and thus not introduce any probe-effects. Passive hardware monitors can provide detailed, low-level information about a system, such as communication activities, memory accesses, and I/O patterns with little interference to the monitored system. [3]”

Hardware monitors, however, cost money, and often are not sufficiently generic enough to be applied to more than one system. Consequently, a hardware monitor often has to be custom designed, and can be very expensive and overkill for some “seat-of-pants” monitoring of certain applications [7]. Moreover, hardware monitors do not debug. It is also difficult to use hardware monitors in computer systems which include such complexities as on-chip cache memories, coprocessor, and parallel or distributed processors. [3]

5.2 Software Probes

An alternative to hardware probes for performance monitoring is to use software probes. Software probes exist in software itself to implement the monitors, and these probes are included as part of or an attachment to the system. However, the act of monitoring with a software probe requires that computations be done and resources be used to execute the behavior of the probes, and therefore they do have an associated probe effect.

In general there are four categories of software probes: kernel probes, inline probes, probe tasks, and probe nodes [7]. Kernel probes are part of the system and monitor runtime entities. “These types of probes are typically not programmable by the application designer, but rather given as an infrastructure by the real-time kernel. In order to avoid the probe effect, these types of probes should be left permanently in the kernel; their contributing overhead must also be predictable, and minimized. [7]” An example of a system with kernel probes would be the tornado IDE which has Windview and Browser allowing the operator to graphically monitor the activity of the threads, message queues, and other runtime entities and behaviors associated with the operation of a multitasking real-time system.

Inline probes are line of code that express points of occurrence or locations in code and report some form of data from the module that they are instrumenting, and are not part of the functionality of that module [7]. An example of an inline probe would be a debugging printf statement used to show that a certain point in the code was reached.
Probe tasks are tasks dedicated to reaping information from the kernel probes. They use system runtime resources and need to be accounted for in the design with regard to timing and resource usage considerations if they are to be part of the system. Probe nodes are dedicated nodes that collect data from probe tasks. These types of probes are “usually self contained computing elements, and can be regarded as passively observing hardware monitoring elements” [7], but may still need to be taken into account for bandwidth considerations.

Because of the issues with hardware and other alternatives to monitoring, there is an evident push towards trying to find a general approach to software performance monitoring using the software itself, and to be able to do this during runtime without excessive (or any) system perturbation. However, there are many issues associated with this approach, which is one of the motivations for performing this investigation. Consequently, discussion of these problems will be deferred to section 6: Difficulties of Real-time Software Performance Monitoring.

5.3 Record and Replay

Another way to work around this issue of performance monitoring and the probe effect is to look at a system in operation and record its behavior. Observing system performance by monitoring replay of a recording is useful because the invariant nature of the recording is immune to the probe effect (if the recording is probed and accounted for properly). Unfortunately, this approach to recording does have its own set of issues. The recording process can produce a tremendous amount of data, and it can often be mostly useless or uninteresting data [2][7][5]. However, recording allows for a closer scrutiny of the performance of the software resulting in more interesting and “high-resolution” observations [7]. Unfortunately, it does not respond well to observing the effects of any changes, because the recorded run is effectively static.

Additionally, while recording of software behavior may be fairly straightforward for single threaded tasks, in multithreaded and especially distributed real-time systems; this approach hits an issue of maintaining the timing relationships between events. For example, on DRTS, one cannot always know when an event on one node provoked a response on another node without having some timing and ordering information; in order to fully understand dependencies and races for system resources correctly, one must know the order of events. This can be difficult without establishing a global time clock [7], and implementing this global clock can be a challenge in itself. System and communication resources on each node must be dedicated to maintaining clock synchronization, such that each local clock is within a certain tolerance or “drift” amount from the global clock. Consequently, record and replay as a solution to performance monitoring has its own set of engineering tradeoffs.

5.4 Simulation

Another possible approach to performance monitoring is by means of a system simulation; the performance can then be monitored under an infrastructure of great control. For this reason, simulators have been used as flexible software debugging tools. However, for performance monitoring purposes, it is particularly difficult to simulate a complex distributed real-time system, or validate the correctness of simulations. Also, simulated execution is also much slower than execution replay [3].

5.5 Example Monitoring Approaches

In the paper, “Monitoring and Debugging Distributed Real-time Programs,” the authors introduce a novel approach to DRTS performance monitoring using the Hexagonal Architecture Mesh Distributed RTS (HARTS) as a case study. They explain the idea of decoupled external monitor nodes with deterministic interference. The interference is regarded as deterministic as a consequence of timing and scheduling of monitoring behavior and can be accounted for, implementing a “transparent” approach to performance monitoring [3].

In the paper, “Performance Measurements for Multithreaded Programs,” the authors use a thread monitoring tool called Tmon to execute a case study of performance improvement on an existing file server system. The system in question was a multithreaded application with low performance relative to the limits of the hardware on which it existed. Using Tmon (a single, decoupled monitoring system with a client-server relationship to the system in
question), the authors were able to use observations of behavior (as opposed to measurements of performance) to identify various bottlenecks and inefficiencies. Concepts of semi-busy-wait states of threads are introduced as a performance bottleneck. Authors were able to improvement the performance of the file server in four different cases with non-quantitative performance monitoring analyses using Tmon to identify the offending bottlenecks (as revealed by long waits in the “ready” queue) [4].

FERRET is an example of a performance monitor for a special Linux system that uses kernel level software probes made available to the user space. The authors describe an approach to runtime performance monitoring using these mechanisms. Because the probes exist as part of the system, and are within the kernel, they do not contribute a probe effect; as described in the next section it is the act of removing the probes that causes the system to behave differently than when it is being monitored [6].

6. Difficulties of Real-time Software Performance Monitoring

Henrik Thane has explained that “it is more difficult to observe embedded [real-time] computer systems, simply because they are embedded, and that they thus have very few interfaces to the outside world, and the actual act of observing a real-time systems or distributed real-time system can change their behavior [7].” This phenomenon, referred to as the “Heisenberg Uncertainty Principle for Software [6],” is described next.

6.1 The Probe Effect, Revisited

As Fred Brooks once described in his famous paper “No Silver Bullet,” one of the difficulties of software engineering is that software is invisible: to get insight and feedback on the internals of a piece of code, it needs to be probed. In software performance monitoring, the act of monitoring the system often results in probes that can (and likely will in some way) alter the behavior of the system. Consequently, the observations will reflect the performance of the perturbed system, not the actual, unperturbed system. This paradox is referred to as the probe effect.

The problem with software probes is that they require either inline code or a separate concurrently running monitor of the variables, memory, execution time, etc., (and for RTS things like synchronization variable and resource usage, queues, context switches, etc.). The use of inline software monitors alter the behavior and sensitive timing issues of software when monitoring, and also uses system resources to do the monitoring, therefore corrupting the actual performance. Much effort is being done to find a way that allows for use of non-intrusive software probes that are free of the confounding influence of the probe effect.

One elegant approach is to simply design the system with the monitors in place and constantly running, and never take them out. Thus the monitors are part of the system, and therefore will not give corrupted values of the systems behavior. While this seems like a brilliant and simple solution, this can be a tremendous waste of resources when the monitors are no longer in need, and also add as a source of complexity for possible fault sources. Additionally, this simply may not be feasible for small, simple and/or resource constrained systems [7].

Moreover, with regard to safety critical software, the prospect of leaving the probe in place may also not be an option. There is this concept of “dead code” in software, (i.e. code that does not get executed, but has been left in as a relic of the development process). Accidental execution of dead code (or “code islands”) can lead to unpredictable and unexpected behavior that is regarded as a risk or threat to safety. This leads to a difficult situation with regard to monitoring performance of safety-critical real-time software; the probe effect can perturb the system in ways that will corrupt the results of monitored performance (as compared to actual performance of the unperturbed system) [7]. Therefore, in this situation, leaving the probe code in place as part of the defined system is not an option. Consequently, side-stepping the probe effect paradox may require more expensive, passive hardware probes as described in 5.1.

6.2 Quantization Errors

Performance monitoring often involves measurements that utilize a system clock. The system clock will only be able
to take measurements or timestamps at integer multiples of its own clock period. When an event being timed happens to occur in between one of the clock ticks, it cannot be determined whether it will be recorded as \( T \) ticks, or \( T+1 \) ticks. This uncertainty is therefore a random source of error, referred to as quantization errors.

The quantization error becomes important when juxtaposed with the probe effect. Higher sampling and polling resolution (lower clock tick periods) is desired for higher resolution measurements, but the more frequently a system is polled, the more perturbation is introduced into its behavior. Alternatively, the less frequently it is polled, the lower the resolution the measurements are, and yielding higher uncertainty. Consequently there is a balance that must be struck when choosing the sampling frequency. In some cases, there are approaches that will help dictate the optimal polling frequency (signals and communication sampling theory), but the point is that the more the system is polled and the higher the resolution one seeks in their measurements, the tradeoff is more perturbation to the system. This of course, increases uncertainty and lowers confidence in the measured results.

6.3 Determinism and Perturbations

For sequential or single-tasked/single-threaded programs, the behavior in some cases is simple enough to be deterministic (i.e. repeatable outputs for same inputs and initial internal states). This determinism allows for deterministic monitors to be introduced, and accounted for in ways that will make analytical corrections for the perturbations they introduce. This is one approach to effectively transparent software performance monitoring. However, for multithreaded RTS and DRTS, the complexity and dynamics of the system are too non-deterministic to make this correction a trivial task. The perturbations can non-deterministically affect the outcome of race conditions and resource contentions, and therefore do not lend well to the method discussed for sequential programs.

6.4 Large Amounts of Data

One other problem with performance monitoring is that in some cases it can produce huge amounts of data, and data produced can be inflated with a tremendous amount of useless or uninteresting information. This data requires its own set of analysis which can be very cumbersome. Some ways around this problem are ensuring that only salient and uncorrelated values or behaviors are monitored. Another approach is data reduction (described more in the next section), which actively condenses the monitored data, providing only useful and summary results of performance monitored.

7. Analysis Techniques and Metrics

For performance monitoring, typical analysis techniques may include the basic tools of the statistician. Taking measurements, looking for central tendency and dispersion indices, and using these values to characterize the performance of the system is a generally accepted approach. Statistical analysis allows for comparison of systems, and can give insight into the degree of interaction that the different observed factors have on each other. Beyond this, and what is of interest to this investigation, are some of the more exotic or novel approaches to analysis. The next section will talk about a particularly interesting analysis approach called Statistical Performance Monitoring, and give examples of some interesting performance metrics when monitoring software performance.

7.1 Statistical Performance Monitoring

An idea referred to as “Statistical Performance Monitoring,” takes the approach that “most of the time, users are not interested in knowing the system state in great detail. Some basic statistics about system metrics, such as mean, peak value, and standard deviation could well be enough [2].” Moreover, “not all metrics are equally interesting. A first step in reducing the amount of data collected and transmitted is to stop collecting uninteresting metrics or to collect them infrequently [2].” The approach of Statistical Performance Monitoring focuses on compiling statistics of metrics for a large interval, rather than retaining metrics at the sample interval level [2].

An important issue in designing a performance tool is deciding on what measures it should use for evaluating performance [4]. In many cases, an observer is only interested in behaviors that are unusual or exceptional in some way as compared to their “neighbors.” Statistical Performance Monitoring introduces the concept of the “Boring
Range,” which is the average value for a given window plus or minus some range value. The idea is that by performing statistical analysis on-the-fly before recording the results, much of the useless or boring information in this range can be filtered out [2].

Figure 1 and Figure 2 two give an example of data collected to highlight the use of Statistical Performance Monitoring and the Boring Range as a filter. Figure 1 shows a record of 90 sampled values of a CPU’s utilization for a particular system measured over a 15 minute interval. Much of the data is scattered and confusing, and does not appear to provide much insight. Figure 2 shows how, when values within the boring range are replaced with the current value of the moving average, a much cleaner picture emerges. Moreover, as stated by the proponents of Statistical Performance Monitoring, “often we are only interested in large values. Since, in general, half the values that differ from the mean by a large amount are small values, significant additional savings can occur by only storing large values that exceed the threshold” [2]. It is shown that with statistical analysis, looking at only the peaks off of the average (those above the Boring Range), there is a 99% confidence interval that the value is off by at most 19%. While 19% is a lot of margin, it does give some insight into a general picture of “what’s going on” with 99% certainty.

This approach also discusses the alternative mechanisms with which one can compute a mean value on which to base the Boring Range. Favored is a moving average approach, in which the average value is not a global average, but one for a window that moves along with the current value. In some cases, better estimations can be obtained by using weighted averages in the moving window. Weighted averages give more weighting to more recent values obtained; the older a value is, the less influence it has on determining the current average. One example used is the exponential moving average, in which the weighting given to a value considered in the current mean decreases exponentially with its age. Also, discussed also is an approach using incremental updates to the moving average, and the ways to calculate the corresponding updates to the variance.

Finally, another interesting aspect of Statistical Performance Monitoring is an elaboration on the idea of choosing interesting and useful metrics. In many cases, some metrics chosen can have high correlation and by monitoring both, there is waste. By checking for the correlation of the metrics chosen, one can reduce waste by only choosing one or the other.

### 7.2 Example Performance Metrics

Statistical Performance Monitoring is offered as an interesting approach to managing and analyzing the data collected from some aspects of performance monitoring. However, this analysis must be based off data acquired for various metrics. In general, a performance metric is either a count of events, a measure of the duration of events, or size of some parameter [5]. It is desirable that each metric produces data that can be seen as linear, reliable,
repeatable, easy, consistent, and independent (uncorrelated to other data collected). Listed below are some
interesting examples of possible metrics that have been seen during the course of this investigation:

- Examples of basic and derived metrics
  - clock rate (measuring the number of cycles per unit time)
  - execution time (amount of time taken to perform a task)
  - response time (amount of time between a request and the response)
  - throughput/bandwidth (the amount of data or transferred per some unit time measure)
  - event count (the number of times an event of interest occurs in a given interval)

- Other examples
  - Examining computation to bandwidth ratio as a measure of parallelism
  - Computing normalized processor time for each thread in a multithreaded application
  - Measuring contention for synchronization with wait graphs
  - Looking at waiting time of threads (total time threads are asleep)
  - Looking at semi-busy waiting time (total time threads are repeatedly notified to check a value and
    then put back to sleep)
  - Number of Messages sent over a DRTS network

8. Conclusion

This investigation looked at some of the obstacles associated with real-time software performance monitoring, while
focusing on the probe effect and its confounding influence. Discussed in this paper was the idea that performance is
a “nebulous” concept; an engineer must decide what is meant by “system performance” in order to maintain control
on the scope of his efforts. To investigate this performance, discussed was the concept of real-time software
performance monitoring, which includes both observing and measuring a software system’s behavior. Ideas like the
probe and quantization effect were explored, and the use of hardware and software probes to implement the
monitoring. Ideas like determinism versus non-determinism in the context of performance monitoring were
explained, and each was looked at in the context of single threaded, multithreaded, and DRTS systems. Examples
were provided to demonstrate the existence of active work towards finding a universal approach to transparent or
non-intrusive, runtime performance monitoring for real-time software systems. Finally, after all other factors were
taken into consideration, discussed were some approaches to analyzing the results of the performance monitored so
that they may be understood in a meaningful and useful way.

9. References


Boston. 2002.

Summary: Authors discuss the concept of using a technique of reducing the large amounts of information that
can be accumulated when doing low level data collection of software performance monitoring. Their approach
uses a separate computing system to collect the data and perform some amount of basic statistical calculations
as a filter for removing “boring” information, yielding leaner and more valuable results based on statistical
analysis. Paper discusses various statistical approaches and justifications of when to use each.

Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann

Summary: Paper discusses common issues (perturbation and probe effects) with monitoring the behavior RTS
and in particular distributed RTS. Introduces a novel approach using the Hexagonal Architecture Mesh
Distributed RTS (HARTS) as a case study. Explains idea of external monitors with deterministic interference as
a consequence of timing and scheduling of monitoring behavior.


Summary: Authors use a thread monitoring tool called Tmon to execute a case study of performance improvement on an existing file server system. Concepts of semi-busy-wait states of threads are introduced as a performance bottleneck. Authors were able to improvement the performance of the file server in four different cases with non-quantitative performance monitoring analyses; using Tmon to identify the offending bottlenecks, (as revealed by long waits in the “ready” queue)


Summary: Course text book on the discipline of computer and software performance monitoring and measurement. Emphasizes statistical concepts and approaches, and gives a nice refresher for statistical concepts, all done in the concept of performance analysis.

http://os.inf.tu-dresden.de/papers_ps/pohlack06runtime_monitoring.pdf

Summary: Discusses the concept of software sensors placed within a Linux kernel as means to monitor low level variable activity. Has a discussion on some of the common problems and issues identified with easy approaches to software performance monitoring, and why their approach is less intrusive and therefore better.


Summary: A broad-brush explanation of issues and concepts in performance monitoring, and different approaches to design for performance monitoring. Focuses on single task, multitask, and distributed approaches to computing, and proposes some minimum design requirements for reducing the intrusiveness of the performance monitoring. Discusses the concept of and difficulties associated with implementing a global clock for keeping track of relative events when recording distributed real time events for playback analysis.

10. Paper PSP

Table 1: Paper PSP

<table>
<thead>
<tr>
<th>Completion Date</th>
<th>Item</th>
<th>Estimated Time</th>
<th>Completed Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/27/07</td>
<td>Topic Selection</td>
<td>1 hour</td>
<td>1.5 hours</td>
</tr>
<tr>
<td>10/04/07</td>
<td>Research</td>
<td>3 hours</td>
<td>9.0 hours</td>
</tr>
<tr>
<td>11/01/07</td>
<td>Draft Paper</td>
<td>5 hours</td>
<td>5.5 hours</td>
</tr>
<tr>
<td>10/25/07</td>
<td>Presentation Prep</td>
<td>2 hours</td>
<td>3.0 hours</td>
</tr>
<tr>
<td>11/01/07</td>
<td>Final paper</td>
<td>3 hours</td>
<td>12.0</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td></td>
<td><strong>14 hours</strong></td>
<td><strong>31.0 hours</strong></td>
</tr>
</tbody>
</table>