SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **00**:1–41
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

EMP: Execution Time Measurement Protocol for Compute-Bound Programs

Young-Kyoon Suh[1†], Richard T. Snodgrass[1], John D. Kececioglu[1],
Peter J. Downey[1], Robert S. Maier[1], and Cheng Yi[1*]

[1] *University of Arizona, Tucson, AZ 85721*

SUMMARY

Measuring execution time is one of the most used performance evaluation techniques in computer science research. Inaccurate measurements cannot be used for a fair performance comparison between programs. Despite the prevalence of its use, the intrinsic variability in the time measurement makes it hard to obtain repeatable and accurate timing results of a program running on an operating system. We propose EMP (Execution Time Measurement Protocol) for measuring the execution time of a compute-bound program on Linux, while minimizing that measurement's variability. During the development of EMP, we identified several factors that disturb execution time measurement. We introduce successive refinements to the protocol by addressing each of these factors, in concert, reducing variability by more than an order of magnitude. We also introduce a new visualization technique, what we term "dual-execution scatter plot" that highlights infrequent, long-running daemons, differentiating them from frequent and/or short-running daemons. Our empirical results show that the proposed protocol successfully achieves three major aspects—precision, accuracy, and scalability—in execution time measurement that can work for open-source and proprietary software. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: execution time; measurement; protocol; compute-bound programs

1. INTRODUCTION

Measuring program execution time is one of the most often used performance evaluation techniques in computer science research. The resulting timings are typically used to compare the performance of different programs. Despite the significance of accurately and precisely measuring execution time, the presence of considerable variability in the measured time has not been adequately addressed.

There are two basic definitions of program execution time: *elapsed time* and *process time*. Elapsed time represents the end-to-end time of a program (or process). It is a common way of measuring execution time. Process time is the execution time taken only by the process of interest, often calculated as the sum of the user time and system time of the process. The process time thus does not include the time spent in other processes, many of which are operating system daemon processes. The elapsed time measures what a user will experience if running their program on an identical system, though of course that time includes the vagaries of the various daemons.

[*] Currently at Google Inc. Mountain View, CA 94043
[†] Correspondence to: KISTI, 245 Daehak-ro, Yuseong-gu, Daejeon, 34141, Rep. of KOREA (current address). E-mail: yksuh@kisti.re.kr

Copyright © 2016 John Wiley & Sons, Ltd.

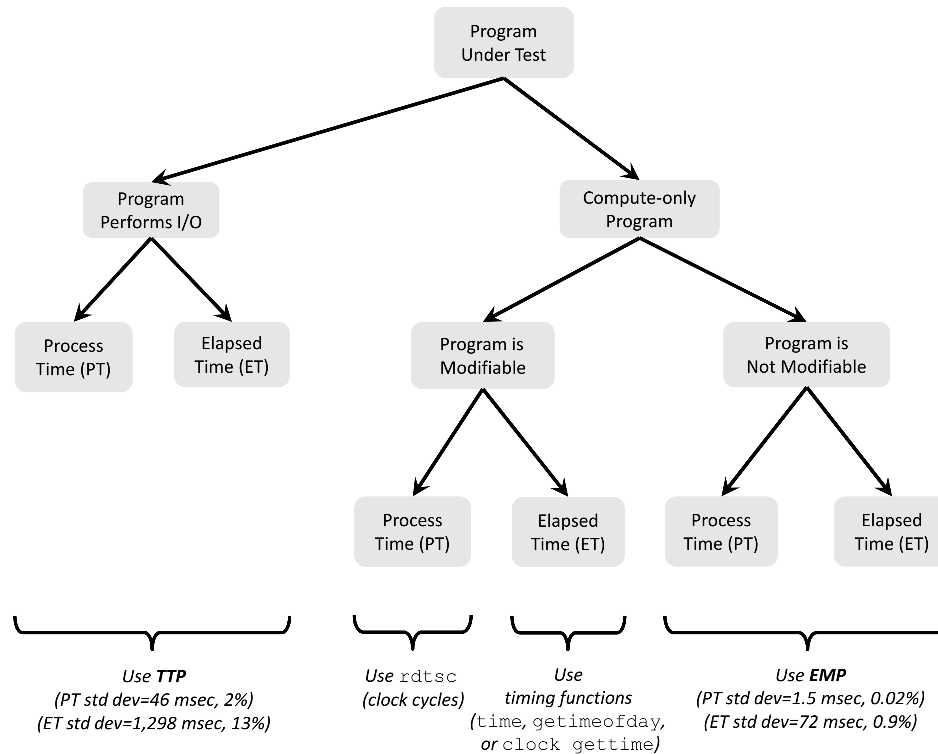


Figure 1. Taxonomy of Execution Time Measurements

As shown in Figure 1, different approaches and resulting accuracies apply to various kinds of programs. Programs that perform I/O (on the left, at the second level) are *much* harder to measure, because the I/O requests to a shared storage device (e.g., hard disk) from other programs running at the same time affect various measures in complex ways. Elsewhere we have provided the *Tucson Timing Protocol* (TTP) which figures out those interactions to reduce the variability of elapsed time of greater than 13% to about 2% for process time [1]. (The results for TTP given in the figure concern measuring execution time of SQL queries, taking from milliseconds to a substantial fraction of an hour. The queries were executed on a variety of proprietary and open-source DBMSes running on a Linux system enabling a single core with hyper-threading disabled.)

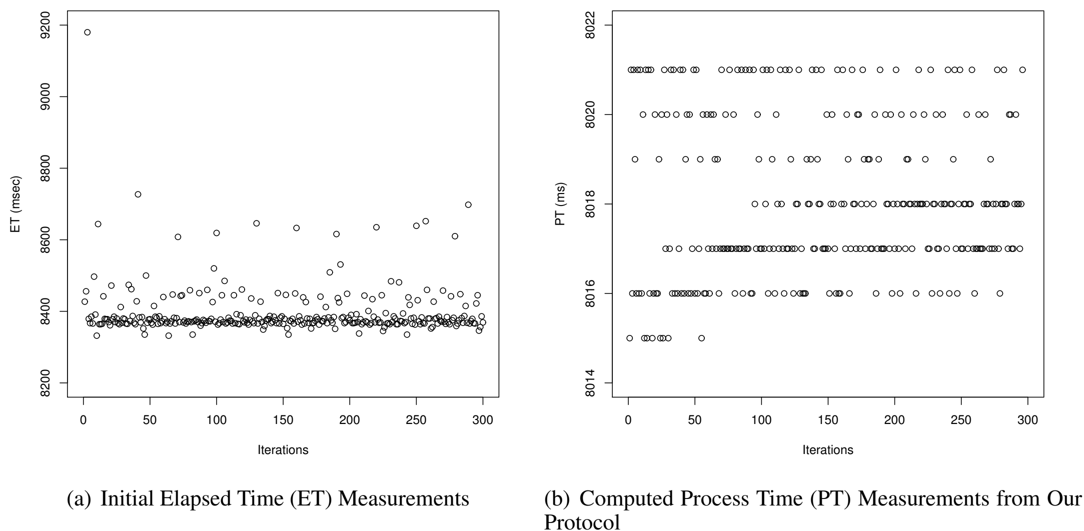
This article concerns the somewhat simpler task of measuring the program and elapsed times of programs that do not perform I/O, that is, are compute-only (on the right on the second level in the figure). (Why a compute-bound program? Because we realized that there still exist many unknown extraneous factors in timing programs that do not perform significant I/O in pure-computation mode. In this paper we will successively uncover these factors and compare the timing results before and after handling each such factor.)

If we can modify the program, we can use the `rdtsc` machine instruction to get the number of clock cycles required by the program (a clock cycle on modern machines is a fraction of a nanosecond), which is quite repeatable. If we want elapsed time (the three at the bottom right in the figure), there are a variety of timing methods available on Linux, to be examined shortly.

If the program is not modifiable, such as if one is measuring a proprietary program, elapsed time can be measured by an obvious timing tool such as `time` or Java's popular timing API (`System.currentTimeMillis`()). Using elapsed time, however, turns out to yield variability of about 0.9%, which is not much better than what the sophisticated TTP achieves on programs *with* I/O. (The results given here and in the figure for EMP concern INC8, which as we will describe shortly is a compute-only program running for about 8 seconds.)

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe



(a) Initial Elapsed Time (ET) Measurements

(b) Computed Process Time (PT) Measurements from Our Protocol

Figure 2. Execution Time Measurements of an 8-Second Compute-Bound Process

The purpose of this article is to show how to reduce this variability by over an order of magnitude, to under 2 msec for INC8, or 0.02%. As we discuss in Section 8, more accurate measurements can produce better prediction models for execution time and can also reveal previously-undetected phenomena within the operating system.

Relying on elapsed time measurement methods, therefore, may not be appropriate in circumstances in which it is important to know exactly how much actual time was spent *only* for the process. What is needed is a comprehensive timing protocol that provides both high resolution and low overhead, while eliminating extraneous factors.

Figure 2 shows the measured execution times of a program in pure-computation mode. In each plot, we ran the program under test 300 times, and so each plot records 300 points. This program, to be given in detail in Figure 7, repeatedly incremented a counter. For each iteration, we used `System.currentTimeMillis()`. We expected that the measured data for the many timings would be almost identical, as the program performs a very simple computation without I/O. Surprisingly, that expectation was untrue.

Figure 2(a) plots the elapsed times (ET) before we applied the refinements of our protocol, to be presented later, while Fig. 2(b) shows the process times (PT) computed by our measurement protocol. Note that the vertical axis of Fig. 2(b) (with times between 8014 and 8021 msec, a range of 7 msec, corresponding to the horizontal lines of PT measurements) is greatly expanded as compared with Fig. 2(a) (with times between about 8300 and 9200 msec, a range of about one second).

Figure 2(a) demonstrates how the measured ETs of such a simple compute-bound program could vary considerably. Here, we observed several layers in measured times. The thickest (main) band in the figure was formed around 8397 msec, or the mean value. Another band, less compact than the main band, was observed about 8450 msec. The other band was seen around 8600 msec. Also, one abnormally high sample (i.e. outlier) was seen around 9200 msec.

We wondered what sources such substantial variation of ET originated from, and how to minimize this variation. In other words, we would like to answer two fundamental questions: what *factors* might affect the timing of a program running on Linux, and what *intervention* one could apply to obtain more accurate timing measurements?

To answer these two questions, we have developed a sophisticated timing measurement protocol called EMP (Execution Time Measurement Protocol) for a compute-bound process running on Linux. EMP yielded very flat timing results, as illustrated in Fig. 2(b). EMP eliminates some

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

and minimizes other factors that affect the timing of a process running on Linux. The protocol also mitigates the variability of memory access time, as confirmed later in our experiments with real-world workloads.

Our contributions can be summarized as follows.

- We provide various timing options for a compute-bound program on Linux.
- We uncover several unknown factors that can seriously impact the timing results for that program.
- We introduce a sophisticated measurement protocol, termed EMP, which addresses these factors to provide more accurate and more precise measurements.
- This protocol uses the novel device of a dual-execution scatter plot to highlight what we term L-samples, to identify within those samples infrequent, long-running daemons, and thus to determine in a disciplined way cutoffs to remove samples with such daemon executions.
- We examine how this protocol applies to proprietary as well as open-source programs.
- We evaluate the performance of EMP by rigorous experiments, starting from a simple program in pure-computation mode to a popular CPU-bound benchmark suite, the SPEC benchmark [2].
- Our empirical results strongly support the effectiveness and scalability of EMP.

The following section discusses accuracy and precision in execution time and then describes the timing mechanisms in Linux and their limitations. In Section 3 we introduce our Execution Time Measurement Protocol (EMP). Section 4 explicates factors and presents experimental results detailing the successive refinement of this protocol. The evaluation of the protocol continues with more realistic scenarios. We then review existing literature over the last thirty-plus years related to execution time measurement. We conclude by discussing future work, including an intriguing phenomenon that our refined protocol uncovered.

2. BACKGROUND

This section describes an overall background of timing a program on Linux. Specifically, we clarify accuracy and precision in timing and discuss the Linux timing mechanism and some limitations.

2.1. Accuracy and Precision in Timing

The concepts of accuracy and precision in time measurement should be carefully differentiated. The *accuracy* of any measurement is the “closeness of agreement between a measured quantity value and a true quantity value of a measurand” while the *precision* of that measurement is the “closeness of agreement between ... measured quantity values obtained by replicate measurements on the same or similar objects under specified conditions” [3]. (In some contexts, accuracy is termed *external validity* and precision, *repeatability*.) Sometimes the precision means the unit a measured value can be appropriately expressed, such as milliseconds, microseconds, or nanoseconds. But in this paper that particular meaning is ascribed to the alternative term “resolution.”

A challenge in determining the accuracy is that we do not know the “true” value associated with a timed sample. That is, we do not have ground truth. But at least we can utilize protocol aspects that should improve the accuracy of the timing, such as, for example, by eliminating as many external factors as possible. The present paper identifies these factors by which external variability may be controlled to some extent by our protocol to yield high-quality measurements.

Determining the precision is easier, as it is equivalent to examining how far timed samples are from each other, which is then translated to the standard deviation among the samples.

A good timing protocol should be able to yield more accurate (that is, with appropriately reduced noise) and more precise (that is, low standard deviation and relative error) measurements.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

2.2. *Elapsed versus Process Time*

As mentioned in the introduction, elapsed time is the end-to-end time of a process' execution and *process time* is the execution time devoted to that process. The distinction arises from the fact that modern operating systems context-switch between processes for better performance.

These two times are illustrated in Figure 3. A process executing is depicted there as a horizontal line (time advances to the right), with context switches represented as vertical arrows. Here, a (as well as b, c, and d) is equal to the sum of time spent in user and system mode of a given program while e corresponds to the wall-clock time difference between t_s (the program's start time) and t_e (the program's end time).

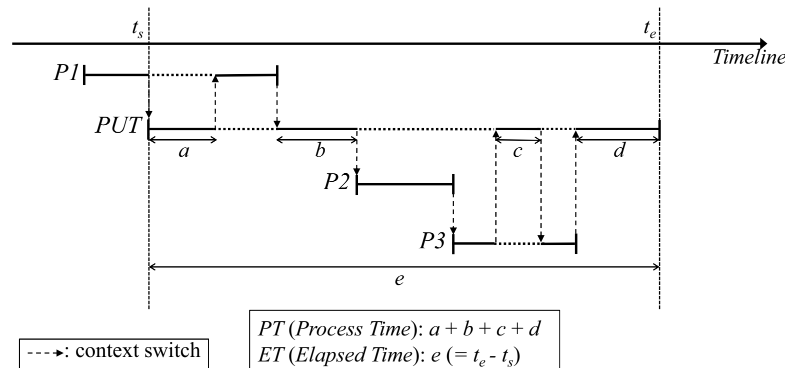


Figure 3. Process Time versus Elapsed Time

2.3. *Linux Timing Mechanism*

There is a spectrum of granularities with regard to Linux timing mechanisms, as illustrated in Figure 4. (Our protocol is currently restricted to this widely-used operating system.) The first decision is whether the operating system is running. When the operating system is down (depicted on the left of the second level), the *Real Time Clock* (RTC) (leftmost on the third level), a hardware clock device equipped in the system, can still tick thanks to power supplied by a small battery. When the system boots up (rightmost on the second level), the RTC sets the kernel clock (also known as *system* or *software clock*) (shown to the rightmost of the third level) during kernel initialization, and that kernel clock begins to keep track of current time on the system.

There are two basic schemes by which the kernel can record the passage of the time. One scheme is based on a counter incremented every CPU clock cycle (termed a *cycle counter*) (shown on the left of the fourth level). A representative cycle counter is *Time Stamp Counter* (TSC) (on the leftmost of the fifth level). TSC is a 64-bit register, incremented by 1 at each CPU cycle. If the CPU frequency is 1 GHz, the TSC is increased once every nanosecond. The `rdtsc` assembly instruction (on the leftmost of the sixth level) can be used to retrieve the current CPU cycle count from the TSC register. The `rdtsc` instruction simply reads the current CPU cycle count from the register. Hence, the overhead is very low. `rdtsc` has the highest resolution since the CPU frequency, read from the `/proc/cpuinfo` file, is usually the highest among the frequencies of all clock devices in a computer system.

The other scheme is based on a low frequency timer in the hardware periodically interrupting the CPU (*interval counter*) (on the right of the third level). The interval counter is incremented by timer interrupts issued by a hardware timer of the computer system. There are several hardware timers. One is *Programmable Interval Timer* (PIT) (the second box of the fifth level), which can issue timer interrupts at a fixed frequency. The Linux kernel can catch these interrupts, so that it can repeatedly perform certain tasks once every time interval. Other common timer devices include *High Precision Event Timer* (HPET) (the third box of the fifth level), and *Advanced Configuration and Power Interface Power Management Timer* (ACPI_PM) (the rightmost box of the fifth level).

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

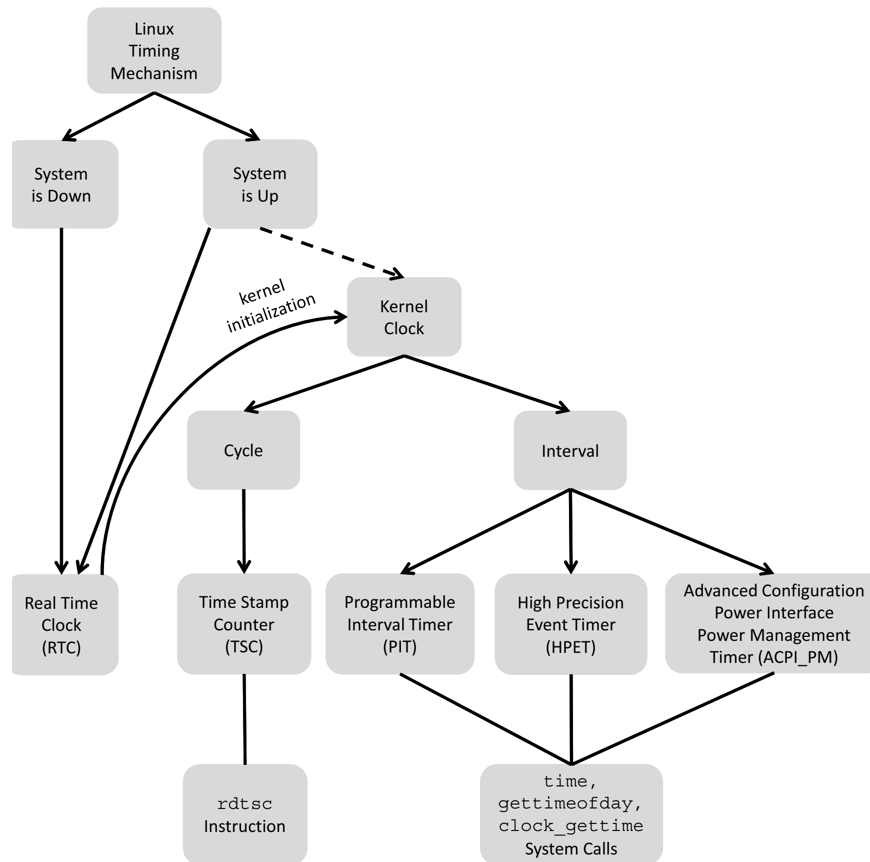


Figure 4. Diagram of Linux Timing Mechanism

Almost all timing system calls, including `time` (in secs), `gettimeofday` (in μsecs), and `clock_gettime` (with resolution in secs) (on the right of the sixth level), provided by the Linux kernel rely on a global variable, called `xtime`, which is maintained by the kernel. More details about the timing mechanism are covered in the Appendix A. The elapsed time mentioned in Section 2.2 can be provided by these system calls. We will consider ways to measure process time in Section 3.

Limitations of System Calls: There are some drawbacks of resorting to the system calls when they are used for execution time measurement. First, each system call comes at a cost (on average 50$\sim$100 μsec). In addition, the measured time via these calls cannot exclude system noise that could occur during the measurement. So the final time measure may not be entirely accurate or precise. Also, even though the system calls can be used for open-source software (by being embedded into and then recompiled with the code), the system calls cannot be added within proprietary software, which is the focus of our paper. While the `time` [4] command may be considered in that case, the measured time by `time` could still include system noise, as mentioned above.

As an alternative, consider `rdtsc`. That instruction does not actually measure the execution time of an operation. Rather, it only counts the CPU cycles consumed for the operation. Additionally, due to cycle scaling in modern processors that change the clock frequency to save power: the approach of dividing cycle counts by clock frequency will only approximate elapsed time. Therefore, `rdtsc` may not be the best choice for timing.

As we will see, on the contrary, EMP can produce as accurate and precise measurements as these system calls can, by extracting the time only taken by a program. In addition, EMP applies to any type of software, regardless of whether it is proprietary or not.

Copyright © 2016 John Wiley & Sons, Ltd.

3. MEASURING EXECUTION TIME

We now elaborate on measuring the execution time of a program (a Linux process) in pure-computation mode. (Previous work [1] details timing a process in mixed mode in which computation is accompanied by significant I/O.)

3.1. `Taskstats` *Interface*

The basis of our measurement protocol is the `taskstats` C struct, provided by the Linux NetLink facility [5]. `Taskstats` is an API that reports per-task (thread) and per-process statistics from the kernel to user space [6] through the NetLink [5] interface. `Taskstats` presents programmers with placeholders for many useful statistics of a task during its lifetime and upon termination. These statistics include a variety of measures for the task, such as time-related metrics like user and system times, IO metrics like number of bytes and characters written or read, and CPU-metrics like number of involuntary and voluntary context switches. That said, statistics other than user time and system time are not needed to measure the execution time of a compute-intensive program.

3.2. Time Measurement Resolution

As mentioned in Section 1, there are two types of execution time for a Linux program: elapsed time (ET) and process time (PT). Different resolutions can be considered for these two time measurements. For the PT measurement of a process in pure-computation mode, the highest resolution is microsecond, as the `taskstats` facility can provide user and system time in microseconds. Some might claim that the `/proc` system [7] could be used instead of the `taskstats` C struct, as it can also provide the measures of the user and system time for the process as well. But the kernel increments the user and system time in `/proc` by one *tick*, whose size is determined the value of the kernel constant `HZ` (as 1 `HZ` is equivalent to one hundredth of a second, the conventional value of this constant of 10 results in a tick being 10 msec). In other words, the resolution of `/proc` (10 msec) is less precise than that of `taskstats` (microseconds). For the PT measurement, therefore, we use the `taskstats` C struct instead of the `/proc` filesystem, which is detailed in Appendix B.

As far as the ET measurement of the pure-computation process is concerned, the highest resolution we can use is nanosecond, as we can exploit a Java API, called `System.nanoTime`(), which returns the current time in nanoseconds. (The Java API will internally invoke the system call of the bottom right of Figure 4.) However, in this paper we simply use millisecond-resolution via `System.currentTimeMillis`() for the ET measurement, as the millisecond resolution is sufficient, given our accuracy and precision capabilities.

3.3. Detailed Measurement of Execution Time of a Given Program

We give a detailed description of measuring the execution time of a given program. (Note that this measurement approach can be applied to any arbitrary program.)

The steps of the measurement approach are detailed in Figure 5. Consider a user program, called *userProgram*. To obtain the measurements of PT and ET of the *userProgram*, we collect a number of samples by repeatedly executing the *userProgram*. The repetition count (`NUM_REPETITIONS`) is predetermined. (As will be seen later, our protocol suggests ten for `NUM_REPETITIONS`.) For each sampling we do the following, as depicted in Figure 6. `getProcInfo`() asks the kernel to report the present per-process statistics in the `taskstats` container through the NetLink socket, recording the reported statistics of each process (`beforeImage`). `getTime`() then obtains the current timestamp (`startTime`). After that, we execute the *userProgram* via *executeProgram*(). Once the *userProgram* finishes its computation, then `getTime`() again requests the current time (stored as `endTime`), and `getProcInfo`() requests the kernel to send the current per-process statistics (`afterImage`). We then compute ET by `endTime-startTime` and calculate the difference (`procDiff`) between `beforeImage` and `afterImage`. Finally, we record ET and *procDiff* (as strings) into the database.

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

Algorithm measureExecutionTime(*userProgram*, `NUM_REPETITIONS`):
for k = 1 to `NUM_REPETITIONS` by 1 **do**
  beforeImage = *getProcInfo*()
  startTime = *getTime*()
  executeProgram(*userProgram*)
  endTime = *getTime*()
  afterImage = *getProcInfo*()
  elapsedTime = *endTime* - *startTime*
  procDiff = *getProcDiff*(*beforeImage*, *afterImage*)
  recordImageDiff(k, *elapsedTime*, *procDiff*)
end for

Figure 5. Measuring the Execution Time of a Given Run of a Program (e.g. *userProgram*)

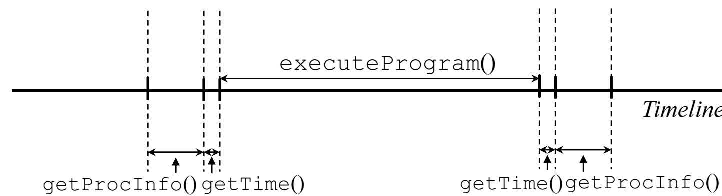


Figure 6. Our Measurement Approach Diagram

Later, for each execution of the run, we retrieve the respective *procDiff* from the database and extract per-process measures from the *procDiff*. We then calculate PT of the *userProgram* as the sum of user and system ticks in the extracted measures associated with the *userProgram*.

We now use this underling measurement approach as the basis for a succession of refinements to our protocol.

4. EXECUTION TIME MEASUREMENT PROTOCOL

We propose a novel protocol, termed the Execution time Measurement Protocol (EMP), for precisely and accurately timing a compute-bound program on Linux. EMP identifies and tackles various external factors of timing, leading to high-quality measurements. This protocol also considers the variability of execution time by memory references. The consideration is twofold. One is to use *warm cache*. Specifically, the very first measurement is thrown out due to code and input read, and the rest reflecting cache effects (misses or hits) is counted for measurements. Another is to utilize "PT," more robust (less variable) than ET to system noise.

The timing protocol has been established and incrementally refined through our rigorous experiments, running and collecting execution times and relevant timing measures on, a very simple program, or what we call INC, which we describe now.

Program-under-Test: The program-under-test that we use initially is INC, which is a simple C program composed of a nested for-loop. INC can be straightforwardly configured so that its running time completes in a specified task length (tl, in seconds). In the inner loop, INC increments an unsigned integer variable (k) by one for as many times as the maximum value (i.e., `UINT_MAX`) defined in `<limits.h>`. The outer loop runs the inner loop for a certain number (c) of iterations translated from the task length. For instance, using our test machine configured with Intel Core *i*7 CPU 870 at 2.93GHz we empirically found the corresponding number of iterations for a one-second task was c = 1.46 billion. For a longer task we simply derive its corresponding proportional constant.

In our experiments we increased the task length of INC from 1 second to 8 seconds (and later up to 16,384 seconds) by successive factors of two.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

```
Algorithm runNestedLoop(tl):
c = TranslateToConstant(tl)
for i = 1 to c by 1 do
   for j = 1 to UINT_MAX-1 by 1 do
      k += 1
   end for
end for
```

Figure 7. Computation Performed by INC

We emphasize that INC is only run internally to our timing protocol, and should not be confused with the user program whose running time our protocol is ultimately measuring. INC is designed to be the simplest-possible compute-bound program that performs no I/O, while the user program that our timing protocol is measuring is unrestricted, and can for instance perform arbitrary accesses to the machine's memory hierarchy.

Environment Settings: Our experimental machine was configured with an Intel Core i7-870 Lynnfield 2.93GHz quad-core processor on a LGA 1156 95W motherboard with 4GB of DDR3 1333 dual-channel memory and Western Digital Caviar Black 1TB 7200rpm SATA hard drive. As seen before in Section 1, we initially installed Red Hat Enterprise Linux (RHEL) [8] 6.1 with a kernel of 2.6.28 on the machine. Later, the machine's OS was upgraded to RHEL 6.4 with a kernel of 2.6.32. We specify which RHEL version was used for a specific protocol.

Our experiments were conducted on an experimental platform [9], which was originally designed for running large-scale experiments with thousands of queries on multiple DBMSes. We redesigned the platform's data collection schema to store the data (ET and process snapshot images) collected from the run of a given program (e.g. INC). We then applied the same experiment methodology used in the original platform. Specifically, we ran a centralized database server for the data collection and a process-monitoring program (called ProcMonitor) communicating with the kernel and supplying the process images. After that, we executed measureExecutionTime shown in Figure 5. Again, for each execution we recorded the data into the database.

Protocol Summary: Table I exhibits a succession of the refinements of EMP in a chronological fashion. For each protocol evolution, we list the primary refinement it identified. The rest of this section discusses each of six refinements in sequence and considers its influence on timing by comparing the measured data before and after the refinement. In Section 5.2, the performance of the most-refined EMP is evaluated on realistic scenarios.

Table I. Protocol Evolution

Protocol (Version)	Description
EMPv1	Deactivate non-critical daemon processes and use process time.
EMPv2	Activate the NTP daemon process.
EMPv3	Switch off Turbo mode and the SpeedStep feature.
EMPv4	Install an up-to-date Linux kernel.
EMPv5	Remove outliers in a disciplined manner.
EMPv6	Determine a suitably small sample size.

Figure 8 provides a high-level view of the proposed timing protocol. The protocol consists of four steps, each established from a specific refinement of EMP. Step 1 configures a timing environment using the refinements of EMPv1—EMPv4. As specified in Figure 6, Step 2 runs a given program and measures its execution time. We use the number of repetitions suggested by the refinement of EMPv6. Step 3 removes some timing results using the refinement of EMPv5. At the end, Step 4 calculates the execution time of the program: a single value *averaged* among the retained results. (Note that with regard to the number of repetitions of Step 2, we follow the suggestion of EMPv6, later identified after EMPv5.) Later in this paper, we evaluate the performance of this timing protocol on realistic workloads as well as the running program-under-test, INC.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Algorithm ExecutionTimeMeasurementProtocol():

Step 1 - Set Up the System for Measurement (along with EMPv1—EMPv4).
Step 2 - Run and Time a User's Program with Ten Repetitions (along with EMPv6).
Step 3 - Remove Some Timing Results by Well-Motivated Criteria (along with EMPv5).
Step 4 - Calculate the Execution Time of the Program Using the Average.

Figure 8. High-level View of the Proposed Timing Protocol

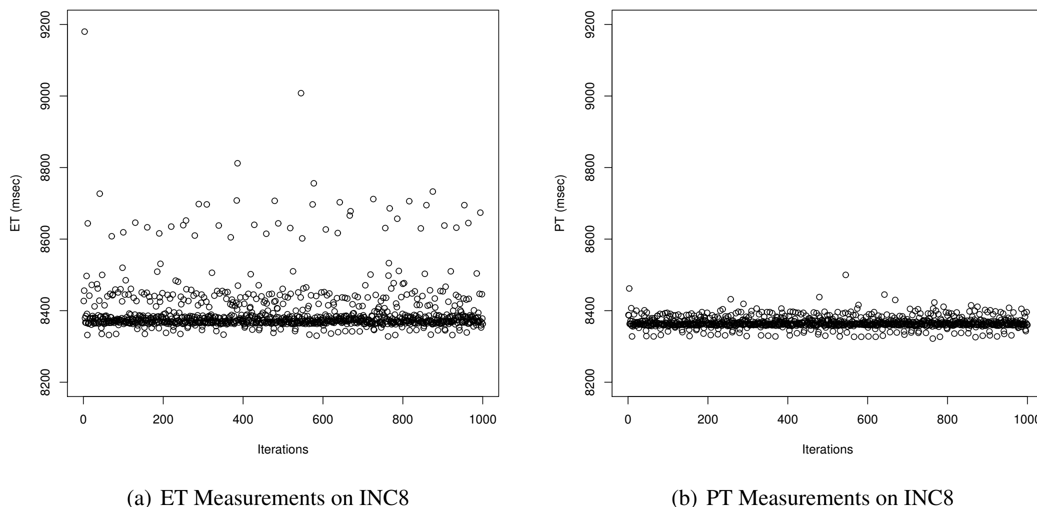
4.1. EMP Version 1: Deactivate Non-Critical Daemons and Use Process Time.

The basis of EMP is to disable as many system daemons as possible and to utilize PT against ET.

On Linux there may be many daemon processes running for system maintenance. These processes in general influence the execution time of INC. Some of the daemon processes could be inevitably necessary, but others may not need to be running.

Appendix C lists the daemons that are non-critical and so can be turned off for the purpose of measurement studies. With these daemons deactivated, our experiments were conducted along with the underlying measurement approach in Figure 5, to see the benefit that accrues from doing so.

Figure 9 illustrates the execution time measurement results for INC. We assigned the task length of 8 seconds to INC, termed INC8, which we ran 1,000 times. Let's first take a look at Figure 9(a) exhibiting the ET measurements of INC8. Interestingly we can see the main band formed around 8,328 msec on INC8. Above the main band there is another band weakly formed between 8,600 msec and 8,800 msec. The highest ET was measured at 9,180 msec, and the second highest one at 9,004 msec. Although many daemon processes were turned off, we found that these long-running ET instances appeared, because of the influence by live daemon processes captured during INC8 execution, that ran longer than usual.



(a) ET Measurements on INC8 (b) PT Measurements on INC8

Figure 9. ET and PT Measurements on INC8 by EMPv1

We now take a look at the PT measurements of INC8 as shown in Figure 9(b). Most of the PT measurements are gathered between 8,322 and 8,400 msec. We can also see a couple of outliers above the thick main band. The highest PT point was 8,500 msec. Compared with Figure 9(a), we can easily notice that 1) the number of outliers was significantly reduced, and 2) there was little variation in measurement. This implies that PT is relatively less influenced by daemon processes. Therefore, using PT for execution time measurement can significantly reduce the variation compared to ET.

Table II exhibits the overall statistics of ET measurements on not only INC8 but also INC1, INC2, and INC4. As a given task length increases, the variation in the ET measurements get increasingly

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table II. ET Statistics on INC1, INC2, INC4 and INC8

	Minimum (msec)	Maximum (msec)	Average (msec)	Std. Dev. (msec)
INC1	1,040	1,377	1,051	23.6
INC2	2,081	2,584	2,102	38.0
INC4	4,163	4,782	4,196	43.5
INC8	8,328	9,180	8,397	72.4

Table III. PT Statistics on INC1, INC2, INC4 and INC8 by EMPv1

	Minimum (msec)	Maximum (msec)	Average (msec)	Std. Dev. (msec)
INC1	1,040	1,076	1,047	3.6
INC2	2,080	2,146	2,093	6.4
INC4	4,159	4,256	4,183	9.9
INC8	8,322	8,500	8,367	15.4

higher. The difference between the maximum and the minimum in INC1's ET measurements was 337 msec. This difference was 503 msec, 619 msec, and 852 msec in INC2, INC4, and INC8, respectively. These results demonstrate that the measurement using ET do not scale with increasing task lengths.

Table III exhibits the overall statistics of the PT measurements on the same INC as shown in Table II. Compared with Table II, the quality of overall PT measurements was improved. For instance, an average of measured times was decreased by 4 msec to 30 msec, indicating less vulnerability to the live daemon processes and thus proving improved accuracy. (The maximum PT measurements also were significantly reduced, by up to 22%.) In addition, the standard deviations of each INC's PT measurements were significantly improved. The standard deviation of ET on INC8 was 72 msec while that of PT was 15 msec, a reduction of almost a factor of five. The degree of improvement ranged from 3x to 7x across different INCs. As a task length increased, PT also scaled better than did ET. These results reveal that PT measurements show better accuracy, precision, and scalability than ET measurements.

We now consider a sequence of refinements on the protocol, starting with the *Network Time Protocol* (NTP) [10].

4.2. EMP Version 2: EMPv1, plus Activate the NTP Daemon.

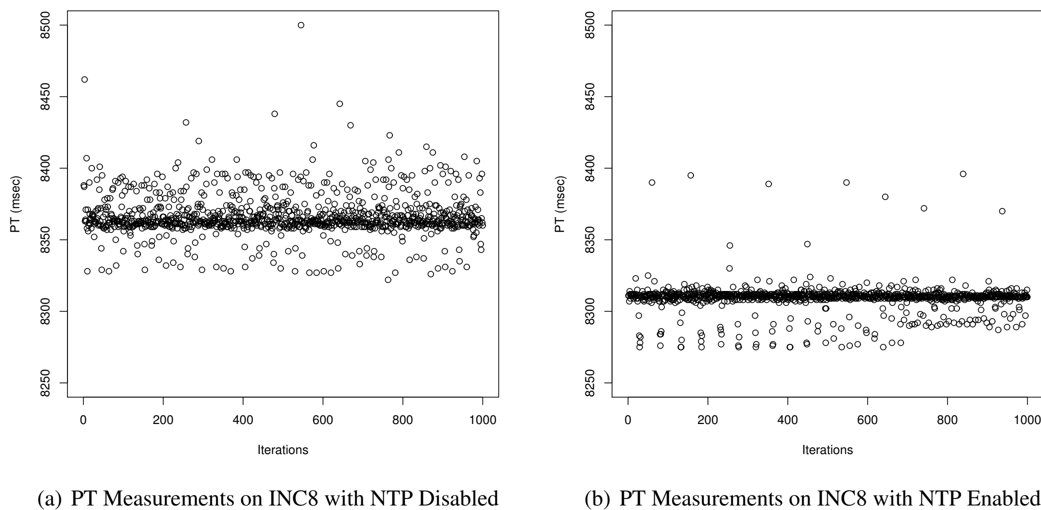
NTP is used to synchronize the local system time to a server in the network [10, 11]. The server is connected to other servers in a hierarchical structure, with an authoritative clock, such as the atomic clock maintained by the US government at the root. More details about NTP are provided in Appendix D.

When the NTP daemon was enabled, the time adjustments by NTP worked out by minimizing the variation. Figure 10 compares the PT measurements when the NTP daemon was deactivated and activated. Note that Figure 10(a) exhibits the same data used in Figure 9(b) in a zoom-in view, and thus, the y-axis of Figure 10(a) is different than that of Figure 9(b). As shown in Figure 10(a), when the NTP daemon was disabled, many values formed the thick black band around 8,366 msec while there appeared a lot of values spread above and below the band. In contrast, when the NTP daemon was enabled many values also formed the main band around 8,309 msec, but there were fewer upper and lower values than the average represented by the band (Figure 10(b)). In other words, the central band was thicker, and much fewer outliers were observed under the NTP daemon's activity. Activating the NTP daemon resulted in reducing the variation in measurement.

Table IV shows the overall statistics of the PT measurements on INCs with the same set of task lengths. Compared with Table III the average PT on INC1, INC2, INC4, and INC8 was reduced by 8 msec, 15 msec, 28 msec, and 58 msec, respectively, with the corresponding maximum measured values further lowered, improving the measurement accuracy. The standard deviations across INC dropped by about 30% on average, contributing to better precision and scalability.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe



(a) PT Measurements on INC8 with NTP Disabled (b) PT Measurements on INC8 with NTP Enabled

Figure 10. PT Measurements on INC8

Table IV. PT Statistics on INC1, INC2, INC4 and INC8 by EMPv2

	Minimum (msec)	Maximum (msec)	Average (msec)	Std. Dev. (msec)
INC1	1,034	1,094	1,039	2.4
INC2	2,068	2,157	2,078	4.3
INC4	4,136	4,242	4,155	7.5
INC8	8,275	8,396	8,309	10.7

4.3. EMP Version 3: EMPv2, plus Switch off Turbo Mode and the SpeedStep Feature.

The Intel processor in our machine includes a feature named “Intel® Turbo Boost Technology” [12]. This feature allows the processor to increase the base frequency and voltage at times through dynamic control of the CPU’s clock rate. This feature is supported by some Intel processors such as Core *i*5 and *i*7. The Turbo mode of our processor (Core *i*7) is auto-enabled by default.

Furthermore, our machine’s processor supports another feature, called “Enhanced Intel SpeedStep® Technology” [13], which can *reduce* power usage and heat based on the load of the processor. SpeedStep first scales frequency and then drops or raises voltage if necessary to match the frequency. In other words, it optimizes the voltage on the selected frequency. Therefore, with Turbo mode and SpeedStep enabled, the CPU clock speed can go faster or slower. To see if these features were the actual cause of some of the variance, we disabled both features in this test, thereby stopping frequency and voltage scaling together.

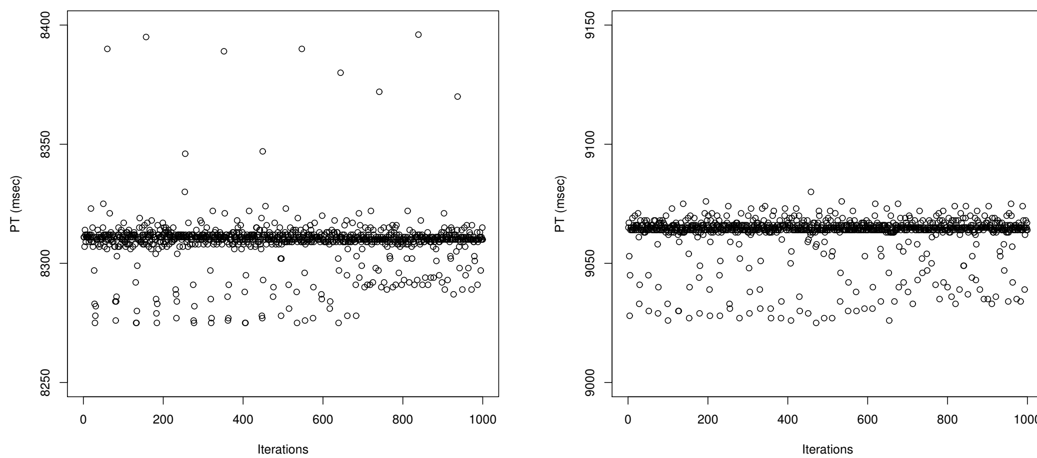
Figure 11 compares the measurement results on INC8 before and after Turbo mode and SpeedStep features were disabled. Note that Figure 11(a) was copied from Figure 10(b), but the range of the y-axis was shrunk to keep the same height as that of Figure 11(b). Check that the y-axis values are higher in Figure 11(b) than those in Figure 10(b).

In Figure 11(a) the NTP daemon was enabled, and the Turbo mode and SpeedStep features were switched on for the measurement. As mentioned before, some periodic lower values were observed. For instance, we can see the PT values at around 45th, 90th, 135th, 180th, and many later executions faster than the average (the thick black band). There were a couple of outliers much lower than those in the main band. We observed a similar tendency in the execution results of INC1, INC2, and INC4 presented in Table IV.

Figure 11(b) shows our measurement results before and after turning off the two features. After eliminating the influence of the features, overall PT measurement values were slightly higher due to

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe



(a) PT Measurements on INC8 with Turbo Mode and SpeedStep Enabled

(b) Per-Execution PT on INC8 with Turbo Mode and SpeedStep Disabled

Figure 11. PT Measurements on INC8 before and after Turbo Mode and SpeedStep Features were Disabled

Table V. PT Statistics on INC1, INC2, INC4 and INC8 by EMPv3

	Minimum (msec)	Maximum (msec)	Average (msec)	Std. Dev. (msec)
INC1	1,128	1,146	1,133	1.7
INC2	2,255	2,279	2,266	2.8
INC4	4,513	4,630	4,532	5.8
INC8	9,025	9,080	9,063	8.7

the disabled features. That said, we significantly reduces the measurement variation. The range was reduced from 121 msec to 55 msec, a difference of more than two.

Table V shows the overall statistics of the PT measurements. Compared with Table IV, the standard deviation was improved by about 0.7 msec (from 2.4 msec to 1.7 msec) (29%), 1.5 msec (35%), 1.7 msec (23%), and 2 msec (19%).

The measurements shown in Figure 11(b) still retained the lower values. The next version of our protocol, called EMPv4, eliminates those.

4.4. EMP Version 4: EMPv3, plus Use an Up-to-Date Linux Version.

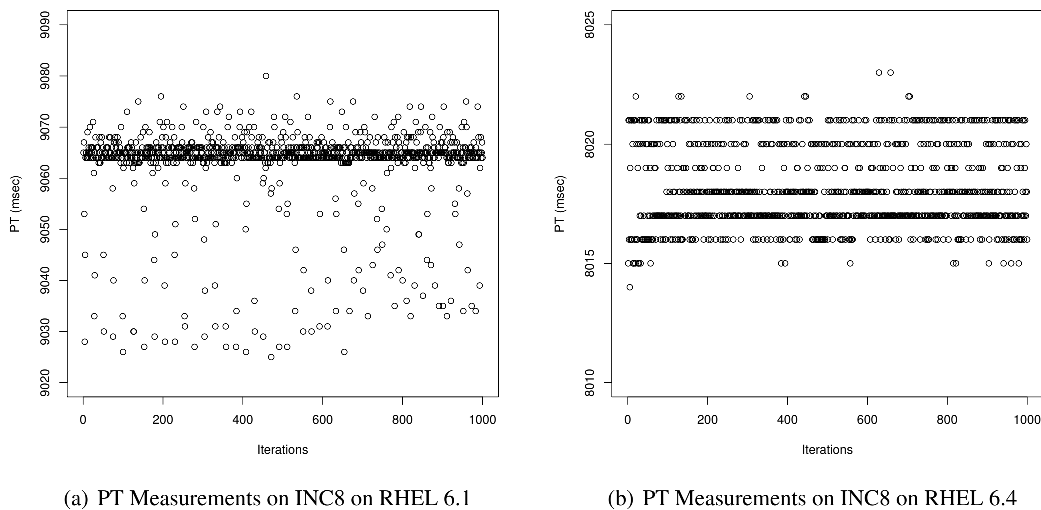
We determined that NTP, Turbo mode, and SpeedStep could significantly contribute to the variation in PT measurements. Recall that EMPv3 switched NTP on but Turbo mode and SpeedStep off before timing the program. However, EMPv3 still showed the values below the central band.

After much investigation, we found out that the lower values were induced by an out-of-date Linux version. The next version of the protocol, named EMPv4, used the most up-to-date Linux version at the time of conducting this experiment (on October 17, 2013).

Figure 12 represents the timing results of INC8 running on different RHEL versions. Figure 12(a), copied from Figure 11(b), shows the results measured on RHEL 6.1, and Figure 12(b) illustrates the results measured on RHEL 6.4. As demonstrated in Figure 12, our measurements on RHEL 6.4 were strikingly cleaner. Unlike Figure 12(a), the less-frequent loser values were eliminated. Most of the points in Figure 12(b) aligned along with the horizontal lines starting from only ten consecutive y-axis values. This results in an exceedingly low standard deviation, about 1.81 msec as shown in Table VI.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe



(a) PT Measurements on INC8 on RHEL 6.1 (b) PT Measurements on INC8 on RHEL 6.4

Figure 12. PT Measurements on INC8 running on RHEL 6.1 and 6.4

Red Hat had announced that one bug related to NTP was fixed in RHEL 6.4 [14]. The bug was that the `ntpd` daemon could terminate unexpectedly when one system network interface had an IPv6 address and the network service was stopped or started. Considering that 1) no version upgrade of NTP was made, and 2) no other bug fixes were provided between two Linux versions, it appears empirically that the bug fix helped minimizing the variability in timing.

Table VI presents the statistics of PT measurements via EMPv4. Here we extended the task lengths of INC to 4,096 seconds first and then up to 16,384 seconds in steps of 2x for further scalability assessment. Note that the number of repetitions was 1,000 when a given task length was fewer than or equal to 64 seconds. However, for the task lengths greater than 64 seconds, we estimated that it would take more than ten months to finish the extended tasks from 128 seconds to 16,384 seconds. For these extended task lengths we thus reduced the number of repetitions: 300 repetitions up to 4,096 seconds and thereafter, 40 repetitions (up to 16,384 seconds).

We decided on 300 iterations by examining the previous empirical data on shorter executions times, discovering that the best trade-off between total run time (by task length) and data quality (that is, standard deviation of PT) occurred in the first 300 samples (out of 1,000). For consistency, we used the same sample size (300) on the shorter tasks by taking the first 300 iterations. Indeed, there was little difference of the measurement quality between the original and reduced sample size for the shorter tasks.

As exhibited in Table VI, overall EMPv4 produced much cleaner data than EMPv3. In comparing Tables V and VI, the standard deviation reduced by more than a factor of two.

4.5. EMP Version 5: EMPv4, plus Remove Outliers in a Disciplined Manner.

Examining Figure 12(b), we see that the vast majority of the executions have times of 8,015–8,022 msec, with just three measurements outside of that, at 8,014 and 8,023 msec. This suggests removing the extreme outliers, while retaining most of the measures.

The next version of the protocol, termed EMPv5, eliminates such an outlier based on two criteria to be covered in this section. EMPv5 first identifies *infrequent, long-running daemons* and determines an (ET) cutoff for each such daemon. It then drops samples containing such a daemon's execution time exceeding that cutoff. A second sanity check drops samples whose corresponding PT measurements are higher or lower than two standard deviations from the average of PT measurements in the retained samples (those remaining after applying the first sanity check).

Copyright © 2016 John Wiley & Sons, Ltd.

Table VI. PT Statistics by EMPv4

	Minimum (msec)	Maximum (msec)	Average (msec)	Std. Dev. (msec)
INC1	999	1,005	1,002	0.7
INC2	1,996	2,007	2,005	1.4
INC4	4,004	4,012	4,009	1.6
INC8	8,014	8,022	8,018	1.8
INC16	16,029	16,039	16,034	1.9
INC32	32,065	32,079	32,068	1.9
INC64	64,128	64,145	64,135	2.3
INC128	128,244	128,260	128,251	2.3
INC256	256,494	256,523	256,502	3.3
INC512	512,995	513,152	513,005	9.4
INC1024	1,025,996	1,026,141	1,026,011	11.4
INC2048	2,051,981	2,052,156	2,052,012	11.2
INC4096	4,105,451	4,105,629	4,105,526	26.0
INC8192	8,207,870	8,207,967	8,207,918	21.0
INC16384	16,415,757	16,415,964	16,415,810	40.4

In the following, we first enumerate the steps to (a) identify infrequent, long-running daemons, using a single run of many samples of INC128, (b) refine the list using a single run of many samples of INC16384, and then (c) use the data from those two runs to determine the cutoff for each so-identified daemon. We can then apply these cutoffs to remove outliers from subsequent runs of any INC.

4.5.1. Protocol Summary Here is the series of steps that EMPv5 takes, building upon the steps taken for EMPv1–EMPv4.

1. Perform a single INC run (specifically, INC128) for many samples (specifically, 800). (Why INC128? Because it is long enough to perhaps experience an infrequent daemon. Why 800? To capture infrequent daemons that may only run every few hours or even only once a day, as these samples will require over 28 cumulative hours.)
2. Consider each pair of ET measurements to be a *dual-execution* measurement (that is, equivalent to 400 samples of dual-INC256, to be termed, where each dual-INC256 sample contains a pair of consecutive INC128 samples). Examine a scatter-plot to see if it it displays an L-shape. (It is this shape that helps us identify infrequent daemons. We will give an example in the next section.)
3. Zoom into the central cluster to ensure that it is symmetric (roughly circular). We term this this symmetric collection of samples the "central cluster," differentiated from the other samples, termed the "L-samples." (We emphasize that we use this sequence of scatter plots to *visualize* and then differentiate these two flavors of samples. This is admittedly a heuristic. We do this only to differentiate infrequent, long-running daemons from the other daemons, to be so identified in the subsequent steps.)
4. Compute the maximum and standard deviation of the process time for each daemon encountered within the central cluster samples. (Why? These are daemon executions that we *do not* regard as "long-running," as we will explain in detail in the next section.)
5. Identify for each L-sample the infrequent long-running daemon executions: those whose process time is over two standard deviations above the maximum from that computed in the previous step. Intuitively, it is those execution(s) that have moved each such sample from the central cluster to one of the arms of the "L". (Note: we do not assert that this step will have identified *all* infrequent, long-running daemons. But it will detect a good number of them, allowing us to target them in the remainder of the protocol.)

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

6. Gather those daemon executions that weren't in the central cluster but were in the L-samples. Determine which might be periodic. Compute for each daemon, the minimum process time from those executions identified.
7. Perform Steps 1–6 above for a single run consisting of a small number of executions (40) of INC16384. (Why INC16384? Because it is much longer, about 4.5 hours, helping us to identify very infrequent daemons. Why 40? Because we need a good number of samples, but not too many, as that would take long. This run requires about 180 hours, or around eight days) Use this data to see if any of the infrequent daemons just identified using INC128 are actually frequent when the execution time is much longer.
8. For each of those infrequent daemons so identified, compute a cutoff that differentiates the normal case. For each such daemon, take the midpoint (as the computed cutoff) between the average for that daemon in the samples of the central cluster (or 0, for those daemons not present at all in the central cluster) and the average for those daemon executions within the L-samples that have been identified as long-running daemon executions. Do so for both the INC128 and INC16384 samples. Take 5% of the observed periodicity (if there is one) to define two cases. (Why 5%? To ensure that such daemons only impact a small number of the shorter INCs, while be accounted for in longer INCs.) Also include daemons that were identified as infrequent and long-running from INC128 but not in the INC16384 L-samples but were in the INC16384 central cluster. For each case, use the maximum of the two cutoffs for the final cutoff.

The above eight steps compute a cutoff PT for each infrequent, long-running daemon (perhaps two cutoffs for periodic daemons). For any subsequent experiment we can use the cutoffs just determined.

For a run of an arbitrary INC, first discard those few samples of the original set of samples that have one or more of those daemons previously identified associated with a program time above their previously-determined cutoff. This should remove most if not all of the samples with long-running daemons.

Additionally, discard from the remaining samples those few samples that exhibit a PT measurement for the Program Under Test that is higher or lower than two standard deviations from the average PT across that run. This will remove some of the samples at the boundary of the central cluster.

We now elaborate the above protocol with an example, to explain and justify each step.

4.5.2. A Running Example: Table VII exhibits the run-statistics of INC128 with 800 samples and Figure 13 plots those 800 ET measurements. (This is the first step of EMPv5 given above.) We see three rows in the plot: a solid row of many samples, perhaps six or more samples that are just above that solid row, and two samples that are way above the solid row. We will now drill down into these samples, and even to specific daemon executions within these samples, to show how to reliably eliminate the indirect influence of some "infrequent, long-running daemons" on the PT of INC.

ET/PT	Minimum (msec)	Maximum (msec)	Average (msec)	Standard Deviation (msec)	Relative Error
ET	128,245	163,913	128,343	1,730.3	1.3×10^{-2}
PT	128,244	128,493	128,251	10.9	8.5×10^{-5}

Table VII. Statistics of INC128 with 800 Samples by EMPv4

To identify such daemons, we use a novel scatterplot: those of pairs of successive samples. So samples 1 and 2 form the first pair, samples 3 and 4 form the second pair. We then plot each pair as a single circle with the ET of the first half of the pair on the x-axis and the ET of the second pair on the y-axis. As we'll see, such dual-execution scatterplots can be very effective at highlighting infrequent, long-running daemons.

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

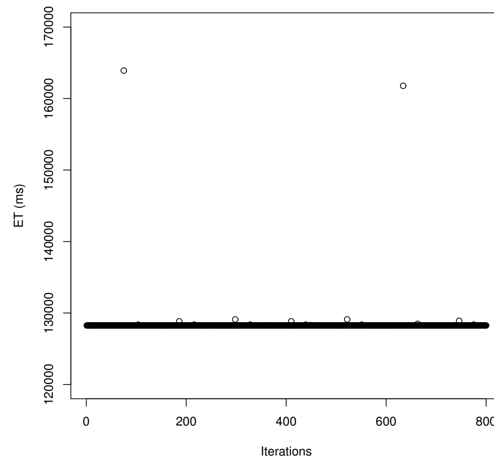


Figure 13. INC128 with 800 Samples (Step 1 of EMPv4)

Figure 14 exhibits a series of zooms on a scatter plot of 400 samples (fromed consecutive pair) constructed from a run of 800 INC128 samples, which we term dual-INC256. (This is Step 2 of the protocol.) Figure 14(a) presents all of the ET measurements. We see two quite obvious outliers, corresponding to sample # 75 and # 634, with ETs of 163,913 msec (rightmost), and 161,785 msec (uppermost), respectively.

Here is the central intuition of the EMPv5 protocol: a *frequent*, long-running daemon (a) will often not occur in either sample of a pair of INC128 samples, (b) may sometime occur in the first of a pair of INC128 samples, (c) may sometime occur in the second of a pair of INC128 samples, and (d) will occasionally occur in *both* samples in a pair of INC128 samples. But an *infrequent*, long-running daemon will *not* appear in set (d).

The cluster at the bottom left of this scatter plot is mainly those pairs in set (a). The single sample in the top left indicates a pair in set (b). There is probably one or more long-running daemons present specifically in the *first* sample of that pair.

The single sample at the bottom right indicates a pair in set (c); it indicates that here is probably one or more long-running daemons present specifically in the *second* sample of that pair.

Importantly, there are no samples in the top right of the cluster, indicating there are no pairs in set (d). This implies that no *frequent* long-running daemons are high-lighted in Figure 14(a); at least they are not readily apparent.

We informally term this phenomenon of a scatter plot of a dual-execution run an “L-shape,” and ascribe it as evidence for the presence of infrequent long-running daemons. Again, samples containing such daemons (termed the L-samples) will appear along the left y-axis or along the bottom x-axis, but will not occur in the top right of the scatter plot of the dual-execution.

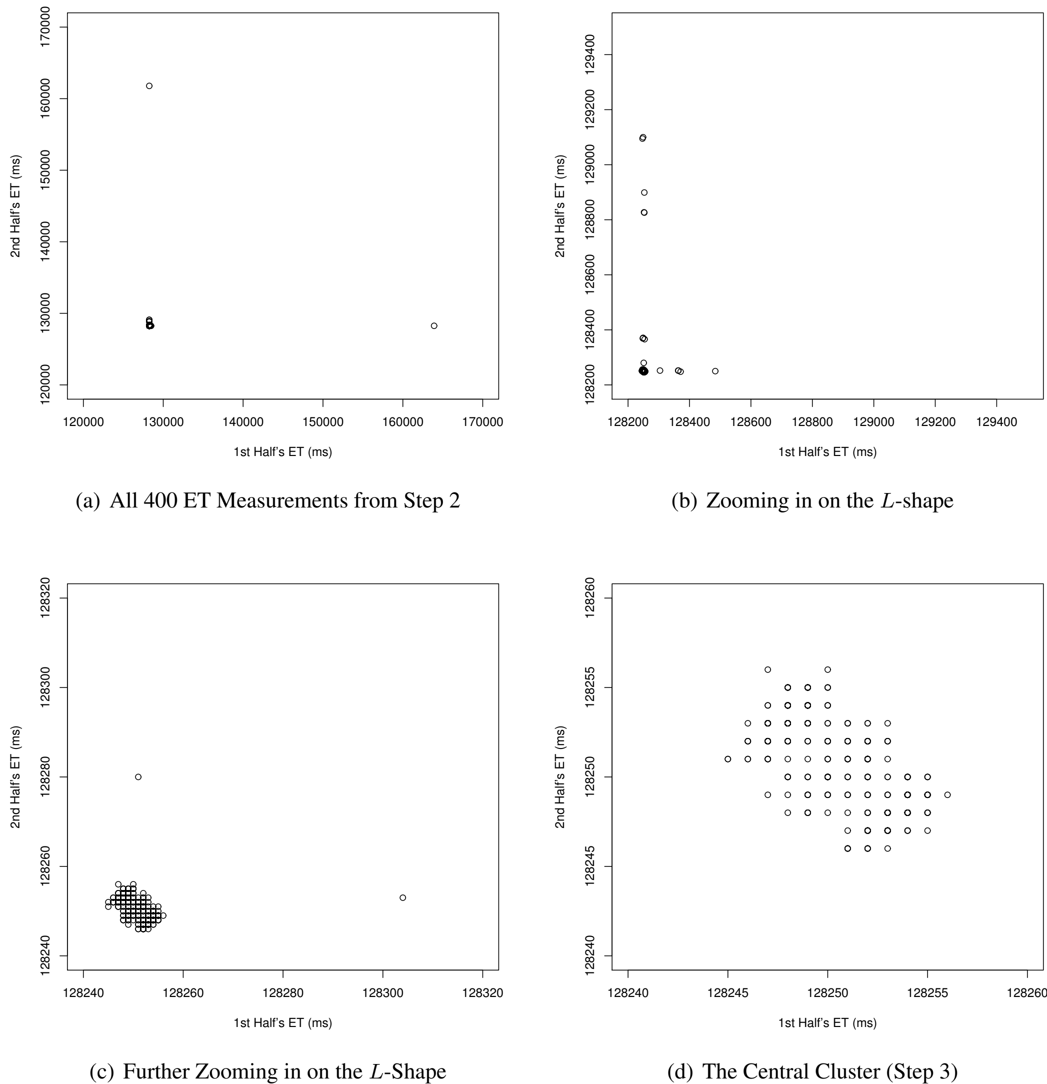
Figure 14(b) zooms into the lower left region, focusing on the tight cluster of samples. Interestingly, this plot continues to exhibit an L-shape, with perhaps a dozen or more L-samples in the left and bottom arms of the “L,” and again no samples in the upper right portion of the scatter plot.

We further zoom in the scatter plot to arrive at Figure 14(c), which exhibits two L-samples, which can also be seen in Figure 14(b).

We continue zooming until we get to Figure 14(d), which shows a central cluster (this is Step 3). We confirm the symmetry of the ET measurements in the central cluster: there is no L-shape, and thus no L-samples, and thus no obvious infrequent long-running daemons. (We emphasize that these samples may have lots of frequent daemons, as well as infrequent, *short-running* daemons.) There were 384 dual-INC256 samples in this central cluster, implying that 16 of the INC128 samples were L-samples and the 784 remaining of the original 800 INC128 samples had no infrequent, long-running daemons, which motivates that term.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe



(a) All 400 ET Measurements from Step 2

(b) Zooming in on the L-shape

(c) Further Zooming in on the L-Shape

(d) The Central Cluster (Step 3)

Figure 14. Successive Zooms of a Scatter plot of a Dual-INC256 (drawn from INC128 with 800 samples), in Elapsed Time (Steps 2 and 3)

We then perform Step 4, which collects in Table VIII the daemon processes observed in the central cluster samples (those in Figure 14(d)) and provides the statistics on the PT of those daemon processes.

Step 5 focuses on the daemons occurring in the INC128 L-samples, those in one of the arms of the L-shape. (Again, the dual-INC256 samples in the top left of the scatter plots indicate that the former of the pair of INC128 samples is an L-sample; symmetrically, for the dual-INC256 sample in the lower left of the scatter plot, the latter of the pair is an INC128 L-sample.) Table IX lists the daemon processes captured in the 16 INC128 L-samples in Figures 14(a), 14(b), and 14(c), and provides their PT measurements.

We then identify, for each daemon in the L-samples, those that are actual long-running daemon executions. We define such executions as those whose PT (process time, note the switch in emphasis from execution time to process time) is over two standard deviations above the maximum PT for that daemon in the central cluster samples, shown in Table X. (Note that these also appear in the cluster samples, cf. Table VIII.)

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Daemon Process Name	Maximum PT (msec)	Standard Deviation of PT (msec)
cifsd	1	0
flush-9:0	1	0
java	6	0.8
jbd2/md0-8	2	0.2
kblockd/0	1	0
khugepaged	1	0
md0_raid1	4	0.4
ntpd	1	0
proc_monitor	204	1.1

Table VIII. PT Measurements of Daemon Processes Observed in the 784 Central Cluster Samples (Step 4)

Sample #	Daemon Process Name: PT
75	flush-9:0: 126 msec, java: 3 msec, jbd2/md0-8: 31 msec, kblockd/0: 1 msec, md0_raid1: 78 msec, proc_monitor: 202 msec, rhn_check: 35,176 msec, rhnsd: 6 msec
104	java: 3 msec, proc_monitor: 202 msec, rhsmcertd-worke: 115 msec
186	java: 3 msec, proc_monitor: 200 msec, rhn_check: 562 msec, rhnsd: 4 msec
216	java: 3 msec, java: 1 msec, md0_raid1: 1 msec, proc_monitor: 204 msec, rhsmcertd-worke: 114 msec
298	java: 3 msec, proc_monitor: 202 msec, rhn_check: 832 msec, rhnsd: 5 msec
328	java: 3 msec, proc_monitor: 202 msec, rhsmcertd-worke: 115 msec
366	bash: 2 msec, cifsd: 1 msec, grep: 1 msec, java: 3 msec, proc_monitor: 203 msec, sshd: 15 msec, sshd: 3 msec,
410	java: 3 msec, md0_raid1: 1 msec, proc_monitor: 202 msec, rhn_check: 571 msec, rhnsd: 4 msec
439	java: 3 msec, java: 1 msec, md0_raid1: 1 msec, proc_monitor: 200 msec, rhsmcertd-worke: 114 msec
451	bash: 1 msec, grep: 6 msec, grep: 5 msec, grep: 1 msec, grep: 1 msec, java: 3 msec, proc_monitor: 202 msec, sshd: 13 msec, sshd: 12 msec, sshd: 3 msec, sshd: 3 msec,
522	java: 3 msec, proc_monitor: 204 msec, rhn_check: 833 msec, rhnsd: 6 msec
551	java: 3 msec, proc_monitor: 201 msec, rhsmcertd-worke: 114 msec
634	flush-9:0: 127 msec, java: 3 msec, jbd2/md0-8: 6 msec, md0_raid1: 65 msec, proc_monitor: 202 msec, rhn_check: 33155 msec, rhnsd: 3 msec
663	java: 3 msec, proc_monitor: 200 msec, rhsmcertd: 1 msec, rhsmcertd-worke: 114 msec, rhsmcertd-worke: 114 msec,
746	java: 3 msec, jbd2/md0-8: 1 msec, md0_raid1: 1 msec, proc_monitor: 203 msec, rhn_check: 629 msec, rhnsd: 5 msec
775	java: 3 msec, proc_monitor: 200 msec, rhsmcertd-worke: 116 msec

Table IX. Daemon Processes Observed in the Sixteen L-Samples (Step 5)

We also identify "extra" infrequent daemons: those found only in the L-samples (those in Table X but not in Table VIII), along with those identified as infrequent and long-running from the central cluster, listed in Table X, to arrive at Table XI. (This is Step 6.)

We use a heuristic to determine the last column, the daemon's periodicity: the daemon must occur regularly in a sequence of samples. So the `rhn_check` daemon appears in samples 75, 186, 298, 410, 522, 634, and 746, or roughly every 112 samples (which corresponds to very close to every four hours). The `rhnsd` daemon appears in the same samples. Similarly, the `rhsmcertd-worke` daemon appearing in samples 104, 216, 328, 439, 551, 663, and 775, with the same periodicity. Note in all three cases, the entire 800 samples are covered. Four others (`flush-9:0`, `jbd2/md0-8`,

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Daemon Name	Sample #	PT
flush-9:0	75	126 msec
	634	127 msec
jbd2/md0-8	75	31 msec
	634	6 msec
md0_raid1	75	78 msec
	634	65 msec

Table X. Infrequent, Long-Running Daemon Executions Identified from the Sixteen L-Samples (Step 5)

Daemon Name	Maximum PT from central cluster	Minimum PT from L-samples	Periodicity from L-samples
bash	—	1 msec	—
flush-9:0	1 msec	126 msec	20 hours (?)
grep	—	1 msec	—
jbd2/md0-8	2 msec	6 msec	20 hours (?)
md0_raid1	—	65 msec	20 hours (?)
rhn_check	—	562 msec	4 hours
rhnsd	—	3 msec	4 hours
rhsmcertd	—	1 msec	—
rhsmcertd-worke	—	114 msec	4 hours
sshd	—	3 msec	—

Table XI. Collected Infrequent, Some of which are Long-running, Daemons from Dual-INC256 (Step 6)

md0_raid1, and rhn_check) all occur together (in samples 75 and 634) and have a periodicity of perhaps 559 samples (that is, five times longer, or just about 20 hours). We say "perhaps" because only two instances do not convincingly imply a periodicity. It is important only which samples a given daemon shows up in; we don't care in this heuristic whether daemons are co-occurring.

We can compute for each so-identified infrequent, long-running daemon its minimum time in the L-samples, shown in the third column of Table XI (this is Step 6). This table provides a rough, initial distinction of a "long-running" daemon. Consider bash. That daemon occurs only in (three) L-samples, and so is infrequent, but is very fast, taking only 1 msec. Contrast this behavior with that of jdb2/md0-8, which runs very quickly, hence, retaining that sample in the central cluster. But every 20 hours or so, it runs for a much longer time, at least 126 msec, moving that sample into one of the legs of the "L". The valley between the maximum PT from the central cluster and the minimum PT from the L-samples differentiated "short-running" from "long-running" executions of the daemon. For those daemons that never appear in the central cluster, for example, grep, we would from this initial analysis conclude only that they are infrequent.

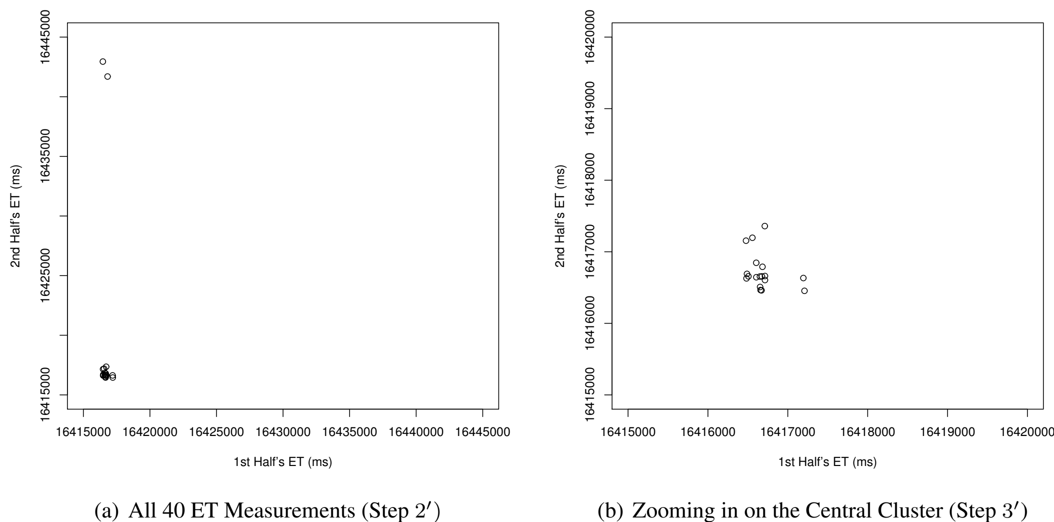
We then repeat steps 1–6, but instead with the much-longer running INC16384 instead (4.5 hours per sample versus 2 minutes), to see if any of our identified infrequent daemons are actually frequent at that much longer INC execution time.

Figure 15 illustrates a zoom of a scatter plot of 20 samples (each a pair) drawn from a run of 40 INC16384 samples, which we term dual-INC32768. As seen in Figure 15(a), there are two extreme outliers on the y-axis (Step 2′). If we zoom in the central cluster (Figure 15(b), Step 3′), there seems to be no ET measurements outside the central cluster, though it is not as neat as Figure 14(d).

Table XII gathers the daemon processes from the central cluster (Step $4'$). Compared with Table VIII, we see that there are some frequent daemon processes that appear in both the dual-INC256 and dual-INC32768 central clusters: flush-9.0, java, jbd2/md0-8, kblock/0, md0_raid1, ntpd, and proc_monitor. That said, the central cluster also contains other processes not seen in the dual-INC256 central cluster: grep, rhn_check, rhnsd, rhsmcertd, rhsmcertd-worke, and sshd. But these daemons were categorized in the dual-INC256 analysis (see Table XI) as *infrequent*, several having periodicities estimated at four or twenty hours.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe



(a) All 40 ET Measurements (Step 2′)

(b) Zooming in on the Central Cluster (Step 3′)

Figure 15. Successive Zooms of a Scatter plot of a Dual-INC32768 (drawn from INC16384 with 40 samples) [cf. Figure 14]

Daemon Process Name	Maximum PT (msec)	Standard Deviation of PT (msec)
flush-9:0	7	1.5
grep	8	2.1
java	3	0.8
jbd2/md0-8	7	1.4
kblockd/0	4	1
md0_raid1	26	4
ntpd	1	0
proc_monitor	206	1.3
rhn_check	714	93
rhnsd	9	1.6
rhsmcertd	1	0
rhsmcertd-worke	117	1
sshd	14	4.7

Table XII. Process Time Measurements of Daemon Processes Observed in the 39 Central Cluster Samples on Dual-INC32768 (Step 4′) [cf. Table VIII]

It is easy to understand what has just happened: when INC had a “short” program time (in this case, two minutes), daemons with a periodicity of hours are infrequent. But with an INC with a “long” program time (in this case, 4.5 hours), some of those daemons are now frequent, and appear in the central cluster. We handle this in the protocol by using the observed periodicity in our categorization of a daemon as “infrequent”, later in Step 8.

We now locate which are infrequent and long-running (Step 5′). As exhibited in Table XIV, we additionally identify one more infrequent daemon, rhn_check. But no extra processes were found in the L-samples of dual-INC32768 (as opposed to the case of INC128).

Step 6′ takes the infrequent daemons found only in the L-Samples (in Table XIV) that are not in the central cluster (Table XIII), in this case, there are no such daemons, adding those identified as infrequent and long-running from the central cluster, listed in Table XIV, to arrive at Table XV. From Table XIII, we see that all four of these processes were in samples 10 and 16, for an estimated periodicity of 6 samples, or about 20 hours, consistent with what was previously estimated. However, such a periodicity would imply that these daemons would appear in sample 4

Copyright © 2016 John Wiley & Sons, Ltd.

Softw. Pract. Exper. (2016)
DOI: 10.1002/spe

Sample #	Daemon Process Name: PT
10	`flush-9:0`: 89 msec, `java`: 3 msec, `java`: 1 msec, `jbd2/md0-8`: 14 msec, `kblockd/0`: 1 msec, `md0_raid1`: 76 msec, `proc_monitor`: 204 msec, `rhn_check`: 24,942 msec, `rhn_check`: 523 msec, `rhnsd`: 9 msec, `rhsmcertd-worke`: 116 msec
16	`flush-9:0`: 91 msec, `java`: 3 msec, `java`: 1 msec, `jbd2/md0-8`: 21 msec, `kblockd/0`: 2 msec, `md0_raid1`: 78 msec, `proc_monitor`: 204 msec, `rhn_check`: 26,667 msec, `rhnsd`: 3 msec, `rhsmcertd-worke`: 115 msec

Table XIII. Daemon Processes Observed in the Two L-Samples (Step $5'$) [cf. Table IX]

Daemon Name	Sample #	PT
`flush-9:0`	10	89 msec
	16	91 msec
`jbd2/md0-8`	10	14 msec
	16	21 msec
`md0_raid1`	10	76 msec
	16	78 msec
`rhn_check`	10	24,942 msec
	16	26,667 msec

Table XIV. Infrequent, Long-Running Daemon Executions Observed in the Two L-Samples in the Dual-INC32768 run (Step $5'$) [cf. Table X]

Process Name	Maximum PT for central cluster	Minimum PT for L-Samples	Periodicity for L-Samples
`flush-9:0`	7 msec	89 msec	20 hours
`jbd2/md0-8`	7 msec	14 msec	20 hours
`md0_raid1`	26 msec	76 msec	20 hours
`rhn_check`	714 msec	24,942 msec	4 hours

Table XV. Collected Infrequent (Some of which are Long-running) Daemons in Dual-INC32768 Run (Step $6'$) [cf. Table XI]

as well. And in fact they did: `flush-9:0` did show up in that sample, but with a PT of only 5 msec, under the maximum PT from the central cluster. We require only that the periodicity detected for the dual-INC256 samples be consistent with those L-samples in dual-INC32768 run. Thus, the `flush-9:0`, `jdb2/md0-8`, and `md0_raid1` daemons all retain that periodicity. Finally, for `rhn_check`, a periodicity of 4 hours is consistent with it appearing in these two L-samples. As before, we don't include processes (e.g., `java`) whose maximum central time is greater than the min L-Sample time. This concludes Step 7.

We then perform Step 8. We compute the cutoff for each of those infrequent, long-running daemons so identified, based on INC128 (Table XI) and the INC16384 (Table XV) as collected in Table XVI.

1. For the cutoff of such a daemon with INC128, we take the midpoint between the maximum of that daemon's PTs in the central cluster (or 0, if absent) for those in Table XI, and the minimum of those in the L-samples, shown in the second column of Table XVI.
2. For the cutoff of such a daemon with INC16384, we do the same from Table XV, shown in the third column of Table XVI.
3. We compute a "task time" as 5% of the inferred periodicity. (As mentioned before, 5% ensures that such infrequent daemons will impact only a small percentage of the shorter INCs, while presumably being associated with much larger cutoffs for the very long INCs.)

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Daemon Process Name	Cutoff PT on INC128	Cutoff PT on INC16384	Task Time	**Final Cutoff PT**
`bash`	1 msec	—	—	**1 msec**
`flush-9:0`	64 msec	—	$<$ 1 hour	**64 msec**
	—	48 msec	$\geq$ 1 hour	**48 msec**
`grep`	1 msec	12 msec	—	**12 msec**
`jbd2/md0-8`	4 msec	—	$<$ 1 hour	**4 msec**
	—	11 msec	$\geq$ 1 hour	**11 msec**
`md0_raid1`	35 msec	—	$<$ 1 hour	**35 msec**
	—	51 msec	$\geq$ 1 hour	**51 msec**
`rhn_check`	281 msec	—	$<$ 12 min	**281 msec**
	—	12,828 msec	$\geq$ 12 min	**12,828 msec**
`rhnsd`	2 msec	—	$<$ 12 min	**2 msec**
	—	12 msec	$\geq$ 12 min	**12 msec**
`rhsmcertd`	1 msec	1 msec	—	**1 msec**
`rhsmcertd-worke`	57 msec	—	$<$ 12 min	**57 msec**
	—	119 msec	$\geq$ 12 min	**119 msec**
`sshd`	2 msec	23 msec	—	**23 msec**

Table XVI. Collected Infrequent, Long-running Daemons and Their Final Cutoff Process Time (Step 8)

4. We also include daemons that (a) were identified as infrequent and long-running from INC128 and (b) were not identified as so in INC16384 L-samples, but may have in the dual-INC32768 central cluster (in which case we add two standard deviations): `bash`, `grep`, `rhnsd`, `rhsmcertd`, `rhsmcertd-worke`, and `sshd`.
5. We then take the *maximum* of the two cutoffs for the final cutoff PT, resulting in the last column of Table XVI.

For any subsequent run of an arbitrary INC, we apply two sanity checks on the samples of that run.

First, we discard any sample containing an infrequent, long-running daemon execution over the corresponding cutoff. We thus remove the following fifteen of the 800 INC128 samples: 75, 104, 186, 216, 298, 328, 366, 410, 439, 522, 551, 634, 663, 746, and 775 and only two of the forty INC16384 samples: 10 and 16, as exhibited in Table XVII.

	# of Samples	Samples
INC128	800	75, 104, 186, 216, 298, 328, 366, 410, 439, 522, 551, 634, 663, 746, 775
INC16384	40	10, 16

Table XVII. Samples Removed by Infrequent Daemons' Cutoffs (First sanity check)

We additionally remove those samples in which the PT measurements are greater or less two standard deviations from the average of the combined samples. In this particular case, no additional samples were removed.

(For an arbitrary INC with *x*-second, that is, the set of outliers by the second sanity check is defined as being outside ± 2 standard deviations:

$$\{P_i \in P \mid P_i > \bar{P} + 2\sigma(P) \text{ or } P_i < \bar{P} - 2\sigma(P)\},$$

where P denotes a set of process time (PT) measurements for a certain INC with an *x*-second task.)

Finally, we calculate the execution time of INC128 as the average PT measurement in Table XVIII: 128,250 milliseconds (rounded up to the nearest millisecond, because PT is measured in msec). Table XVIII shows the statistics of the PT and ET measurements. (In the third column, the number of samples dropped by the first sanity check (15 for ET) and by the second sanity check (1) are indicated.) Our original goal was to determine using a carefully-motivated protocol the program time (PT) for this program under test (INC). We see that the standard deviation and relative error

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

	ET/PT	# of Outliers (=I+II) / # of Samples	Minimum (msec)	Maximum (msec)	Average (msec)	Stdev. (msec)	Relative Error
INC128	ET	16 (=15+1) / 800	128,245	128,256	128,250	2.5	2.0×10^{-5}
	PT	20 (=15+5) / 800	128,245	128,255	128,250	2.6	2.0×10^{-5}
INC16384	ET	6 (=2+4) / 40	16,416,454	16,417,155	16,416,637	138.6	8.4×10^{-6}
	PT	2 (=2+0) / 40	16,415,757	16,415,855	16,415,804	27	1.6×10^{-6}

Table XVIII. Statistics of INC128 (above) and INC16384 (below) by EMPv5

Table XIX. PT Statistics by EMPv5

	Num. of Outl.	Minimum (msec)	Maximum (msec)	Average (msec)	Std. Dev. (msec)
INC1	57	1,001	1,003	1,002	0.6
INC2	42	2,002	2,007	2,005	1.0
INC4	28	4,006	4,011	4,009	1.5
INC8	11	8,015	8,021	8,018	1.7
INC16	12	16,031	16,038	16,034	1.7
INC32	16	32,065	32,071	32,068	1.5
INC64	13	64,131	64,139	64,135	1.9
INC128	9	128,247	128,255	128,251	2.1
INC256	24	256,497	256,506	256,502	2.3
INC512	31	512,998	513,009	513,004	2.6
INC1024	15	1,026,001	1,026,019	1,026,011	4.1
INC2048	23	2,051,997	2,052,156	2,052,012	6.4
INC4096	33	4,105,477	4,105,572	4,105,526	20.9
INC8192	2	8,207,877	8,207,955	8,207,918	18.4
INC16384	2	16,415,757	16,415,855	16,415,804	26.7

have both been considerably reduced by this principled elimination of samples containing identified infrequent, long-running daemons.

Figure 16 compares the EMPv4 and EMPv5 data over INC8, INC64, and INC4096. Figures 16(a), 16(c), and 16(e) show the EMPv4 data with outliers, each enclosed by a square and a triangle (indicated by the first and second sanity checks). Figure 16(b), 16(d), and 16(f) exhibit the EMPv5 data with the outliers eliminated. As can be easily seen, EMPv5 produces much cleaner data.

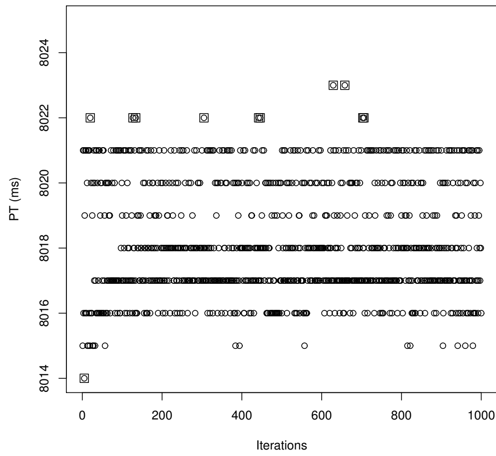
Table XIX provides the numerical comparison: number of outliers removed and results (min, max, average, stdev) without the outliers. EMPv5 obtains a standard deviation reduction of 65% (at INC1024), while the outliers were reduced by at most 3.7% (found in INC8).

The remaining questions that we had in measuring the execution time of INC were: *Do we even need the 300-sample size? Can we further reduce this sample size? How can we reach a desired (perhaps minimal) sample size?* The next version of the protocol addresses these questions.

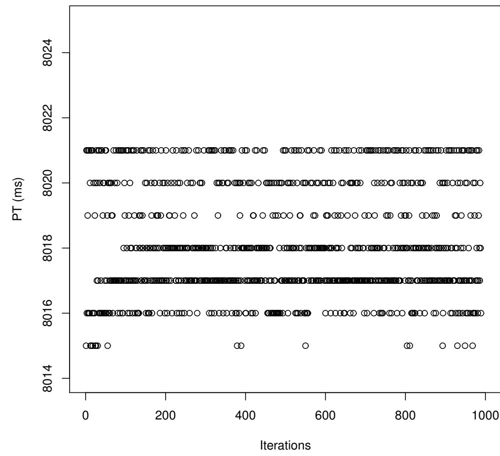
4.6. EMP Version 6: EMPv5, plus Use a Suitably Small Number of Sample Size.

We originally collected a total of 1,000 samples for INC assigned with a task length shorter than 64 seconds, and for the longer tasks up to 4,096 seconds, we lowered the total sample size to 300. (INC4096 takes over an hour per sample.) Indeed, we felt that collecting even 300 samples takes too long for the longer tasks. Furthermore, the standard deviations of the 300 samples by EMPv5 were sufficiently small. For these two reasons, we further investigated how the sample size relates to the number of outliers, to choose a smaller sample that retained the structure of the data.

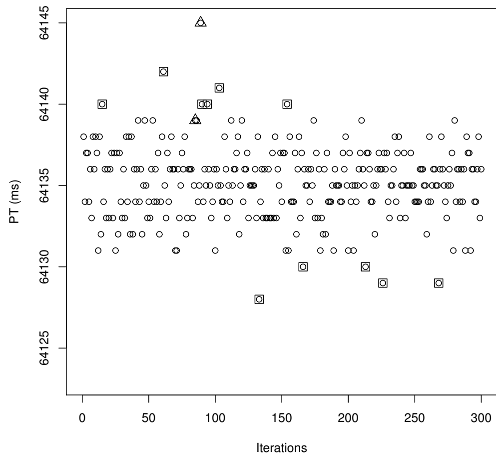
Copyright © 2016 John Wiley & Sons, Ltd.



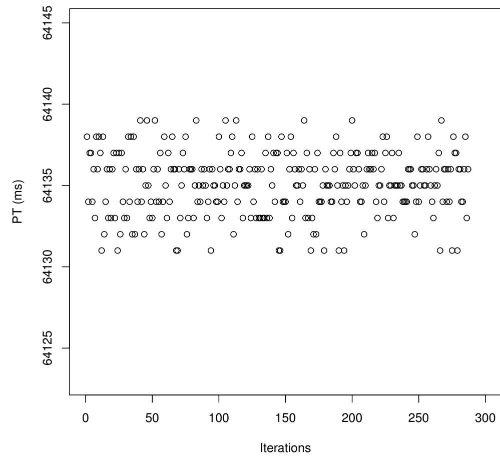
(a) PT Measurements on INC8 by EMPv4



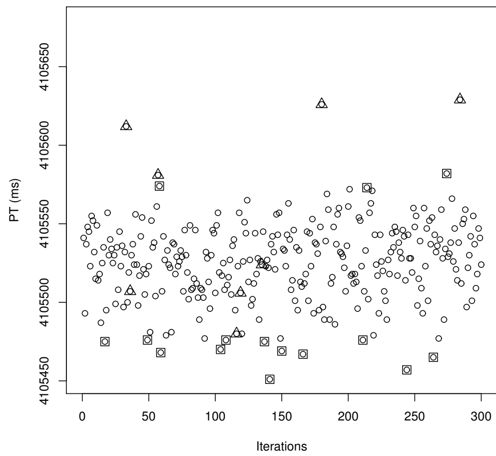
(b) PT Measurements on INC8 by EMPv5



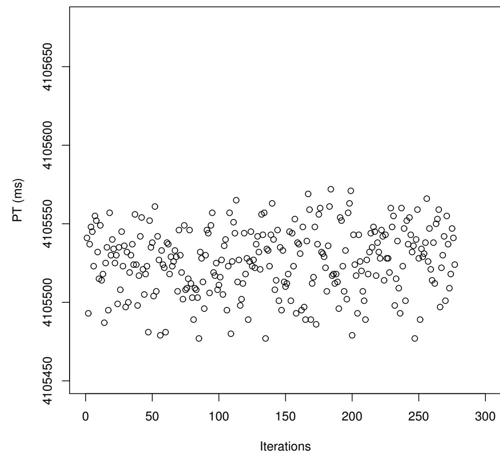
(c) PT Measurements on INC64 by EMPv4



(d) PT Measurements on INC64 by EMPv5



(e) PT Measurements on INC4096 by EMPv4



(f) PT Measurements on INC4096 by EMPv5

Figure 16. Comparison of PT Measurements by EMPv4 and EMPv5

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

4.6.1. Description Here is a series of steps to determine what is a proper number of samples for EMPv6.

1. We recursively divide the whole size to be half (or less) until a sample size is sufficiently minimal.
2. We compute the number of outliers and the standard deviation of PT measurements for each sample size.
3. We then choose a proper sample size, considering the similarity of the number of outliers and the standard deviation of each subset to those of the whole set.
4. To confirm the proper size, run INC1—INC4096 and INC8192—INC16384 with the respective sizes of 300 and 40 samples, repeat Steps 1—2, and then compare the number of outliers and the standard deviation of each subset and to those of the whole set.

4.6.2. A Running Example In this running example, we divide the whole set of 800 samples on INC128 to a total of seven subsets, each having (the first) 400, 200, 100, 40, 30, 20, and 10 samples, respectively. (This is Step 1.) For each of the seven subsets, we examine the number of outliers with ET (PT) measurements, greater or lower than two standard deviations from the average ET (PT) measurements computed in each subset. (This is Step 2.) As a result, Table XX illustrates the number of outliers identified in each sample size.

ET/PT	400	200	100	40	30	20	**10**	**800** (samples)
ET	9 (=7+2)	5 (=4+1)	3 (=2+1)	1 (=1+0)	1 (=1+0)	0	**0**	**16 (=15+1)**
PT	13 (=7+6)	7 (=4+3)	3 (=2+1)	0	1 (=1+0)	2 (=1+1)	**0**	**20 (=15+5)**

Table XX. Comparison of the Number of Outliers (= I+II) for Different Sample Sizes on INC128. () indicates the breakdown of the total number of outliers: 1) how many samples are discarded by the cutoff and 2) by the average ± two standard deviations.

Table XXI exhibits the standard deviation associated with each sample size.

ET/PT	400	200	100	40	30	20	**10**	**800** (samples)
ET	2.5	2.5	2.5	2.7	2.7	2.4	**2.7**	**2.5**
PT	2.6	2.5	2.7	2.7	2.9	2.4	**2.6**	**2.6**

Table XXI. Comparison of Standard Deviations in PT for Different Sample Sizes on INC128

All things considered, we see that 10 is the most appropriate number of samples, based on the similarity between the standard deviation and the number of outliers of each subset and those of the whole set.

Now let's examine whether the 10-sample size still holds for the running INCs with 1—16384 seconds. (This is Step 3.)

Let's start with PT (process time) measures. Based on the measured data by EMPv5, we divided up the whole 300 samples in a binary manner; namely, each subset was given 150, 75, 40, 20, 10, and 5 samples, respectively. We applied EMPv5 to each subset of the EMPv4 data and identified outliers in the respective subset, computing the average and standard deviation within each subset and determining the outliers in that subset, as shown in Table XXII.

The numbers in bold are from the 10-sample size and from the original 300-sample size (in the left-to-right order). As the subset size got smaller, the number of outliers decreased. In particular, the subsets containing fewer than 20 samples had no more than two outliers across INCs. If the number of outliers were the only criteria of determining the minimal sample size, we would have recommended collecting only five samples, as 1) very few (at most 2) outliers were found in that subset, and 2) the size of the samples was also smallest. However, the fact that a certain subset had no outliers was not enough to conclude that the size of the subset is the best minimal size. We also had to take into account the standard deviation.

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

Table XXII. Comparison of the Number of Outliers for Different Sample Sizes by EMPv5

	150	75	40	20	**10**	5	**300** (samples)
INC1	9	3	0	0	**0**	0	**12**
INC2	12	6	3	2	**1**	0	**17**
INC4	0	0	0	0	**0**	0	**0**
INC8	0	0	0	1	**0**	0	**4**
INC16	3	5	1	0	**0**	0	**12**
INC32	8	3	2	1	**1**	0	**16**
INC64	8	3	1	1	**0**	0	**13**
INC128	4	4	0	1	**1**	0	**9**
INC256	7	6	5	0	**0**	0	**24**
INC512	13	8	4	2	**0**	2	**31**
INC1024	7	3	2	1	**1**	0	**15**
INC2048	7	3	2	2	**0**	0	**14**
INC4096	11	3	2	3	**0**	0	**33**
				25	**10**	5	**40** (samples)
INC8192				2	**0**	0	**2**
INC16384				2	**0**	0	**2**

Table XXIII. Comparison of the Standard Deviations of PT for Different Sample Sizes

	150	75	40	20	**10**	5	**300** (samples)
INC1	0.6 ms	0.7 msec	0.4 msec	0.5 msec	**0.4 msec**	0.5 msec	**0.6 msec**
INC2	0.6 msec	1.0 msec	0.6 msec	0.5 msec	**1.0 msec**	0 msec	**0.7 msec**
INC4	1.5 msec	1.5 msec	1.6 msec	1.9 msec	**1.7 msec**	0.5 msec	**1.5 msec**
INC8	2.1 msec	1.5 msec	1.5 msec	1.0 msec	**1.5 msec**	1.5 msec	**1.8 msec**
INC16	1.8 msec	1.4 msec	1.6 msec	2.1 msec	**2.2 msec**	2.2 msec	**1.7 msec**
INC32	1.4 msec	1.4 msec	1.7 msec	1.6 msec	**2.1 msec**	1.6 msec	**1.5 msec**
INC64	2.0 msec	2.0 msec	1.5 msec	1.8 msec	**2.4 msec**	1.9 msec	**1.9 msec**
INC128	2.3 msec	1.8 msec	2.2 msec	1.6 msec	**0.7 msec**	2.5 msec	**2.1 msec**
INC256	2.4 msec	2.4 msec	1.7 msec	2.3 msec	**3.3 msec**	4.2 msec	**2.3 msec**
INC512	2.8 msec	2.3 msec	2.5 msec	2.1 msec	**2.9 msec**	5.2 msec	**2.6 msec**
INC1024	4.0 msec	4.7 msec	4.1 msec	3.0 msec	**2.3 msec**	3.1 msec	**4.0 msec**
INC2048	6.1 msec	6.7 msec	6.6 msec	4.6 msec	**6.0 msec**	6.5 msec	**6.4 msec**
INC4096	21.4 msec	23.6 msec	19.2 msec	17.7 msec	**22 msec**	15.9 msec	**20.9 msec**
				25	**10**	5	**40** (samples)
INC8192				16.6 msec	**20.6 msec**	25.9 msec	**18.4 msec**
INC16384				25.1 msec	**28.9 msec**	18.6 msec	**26.7 msec**

After removing these outliers (that is, in the EMPv5 data), we further calculated the average and standard deviation, exhibited in Table XXIII. For comparison purpose, the last columns in the table are added to show the standard deviation of PT values measured in an arbitrary INC over the full sample size.

In examining these results, it is hard to tell which sample size is best. The smallest sample, 5 executions, exhibited a standard deviation for INC512 of 5.2, which was somewhat different than that for 300 executions. This inconsistency was observed across the other sample sizes.

The standard deviation results of the 10-sample subsets were fairly close to non-decreasingly monotonic over increasing task length (even the 300-sample wasn't entirely monotonic). In addition, the 10-sample collections' outliers much fewer than those of the bigger subset although the outliers were slightly more than those of the smaller collections, as shown in Table XXII. In other words, the smaller subsets did not drop as many samples as the bigger subset, while showing very close overall

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table XXIV. Comparison of the Standard Deviations of ET for Different Sample Sizes

	150	75	40	20	10	5	**300** (samples)
INC1	0.6 msec	0.6 msec	0.6 msec	0.5 msec	**0.5 msec**	0.4 msec	**0.6 msec**
INC2	0.8 msec	1.1 msec	0.8 msec	0.8 msec	**1.5 msec**	0 msec	**1.1 msec**
INC4	1.7 msec	1.8 msec	1.8 msec	2.2 msec	**1.6 msec**	1.1 msec	**1.8 msec**
INC8	2.1 msec	1.3 msec	1.7 msec	1.6 msec	**1.5 msec**	1.3 msec	**1.8 msec**
INC16	1.8 msec	1.6 msec	1.6 msec	2.3 msec	**4.1 msec**	2.3 msec	**1.7 msec**
INC32	1.6 msec	1.8 msec	1.9 msec	1.7 msec	**1.7 msec**	1.1 msec	**1.6 msec**
INC64	2.4 msec	1.8 msec	1.3 msec	1.6 msec	**2.8 msec**	2.7 msec	**2.0 msec**
INC128	2.3 msec	2.0 msec	1.9 msec	1.5 msec	**0.5 msec**	1.6 msec	**2.1 msec**
INC256	2.3 msec	2.6 msec	2.1 msec	2.2 msec	**3.2 msec**	3.0 msec	**2.5 msec**
INC512	3.1 msec	2.8 msec	3.0 msec	2.0 msec	**2.9 msec**	2.9 msec	**2.9 msec**
INC1024	4.6 msec	5.2 msec	4.8 msec	4.2 msec	**5.2 msec**	247.7 msec	**4.9 msec**
INC2048	73.8 msec	109.4 msec	109.1 msec	91.5 msec	**130.1 msec**	7.4 msec	**103.5 msec**
INC4096	366.7 msec	319.5 msec	366.0 msec	312.2 msec	**303.0 msec**	404.8	**355.7 msec**
				25	**10**	5	**40** (samples)
INC8192				364.3 msec	**297.0 msec**	353.9 msec	**339.6 msec**
INC16384				121.7 msec	**197.8 msec**	94.1 msec	**138.6 msec**

statistics compared to those of the whole collection. As seen in Table XX, EMPv6 thus utilizes 10 samples, which are empirically sufficient to represent the overall statistics on the PT measurements of INC.

We now turn to ET (elapsed time) measures. As before, we computed the standard deviations of the measurements over decreasing sample size, as exhibited in Table XXIV. We affirmed that the sample size of 10 yielded statistics broadly similar to those the original sample size, also in the measured ET data. Therefore, we reaffirmed that 10 samples are sufficient to represent a suitably small sample size for measuring execution time.

Our recommendation for measuring the running time (PT or ET) of a program is to run it once for 300 samples and prepare a table such as Table XXIII to independently confirm that a sample size of 10 is reasonable (as one would expect). (A smaller sample size is probably not possible, as the previous step may drop a few samples.) The goal is to choose the smallest sample size that approximates that of the 300 samples. Once the sample size has been determined, then it can be used for the entirety of the study, reporting that sample size.

4.7. *Comparison of Protocol Quality*

Each of the EMP versions has its own sequence of steps:

- EMPv1: deactivate non-essential daemons (p. 10)
- EMPv2: activate the NTP daemon (p. 11)
- EMPv3: switch off Turbo mode and the SpeedStep feature (p. 12)
- EMPv4: use an up-to-date Linux version (p. 13)
- EMPv5: remove outliers using protocol on pp.15-16
- EMPv6: determine the proper number of sample using steps on pp. 24-26.

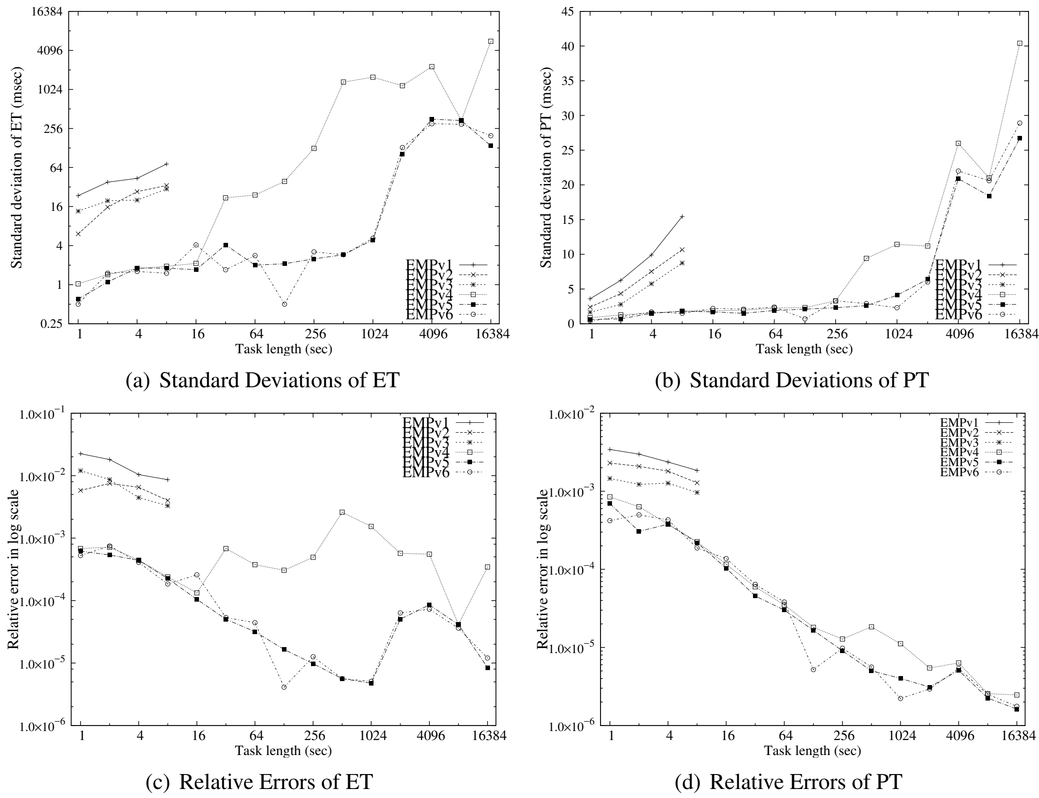
We now compare the measurement quality of EMP with incremental refinements. Figure 17 shows the standard deviations and relative errors of the measured ET and PT data by the incrementally-refined EMP over increasing task lengths.

Recall that prior to EMPv4 the maximum task length of INC was 8 seconds; we extended the length up to 16,384 seconds.

We can learn several things from Figures 17(a) and 17(b). First, the standard deviation in milliseconds increases as task length increases. Note that both the x and y axes are logarithmic in the ET figure but the y axis is linear in the PT figure. Second, for successive versions of EMP, the standard deviation generally improves. At INC8, EMPv6 improves on EMPv1 by about 36x, from 72 msec to about 2 msec (for ET), and by over 7x, from 15 msec to 2 msec, for PT, respectively.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe



(a) Standard Deviations of ET

(b) Standard Deviations of PT

(c) Relative Errors of ET

(d) Relative Errors of PT

Figure 17. Measurement Quality Comparison among the EMP versions

Third, the standard deviation of the PT data was much lower than the ET data; PT outperformed ET by about an order of magnitude for the longest task length (at 16,384 seconds), at 29 msec versus 198 msec. Fourth, the largest effect was observed for EMPv4; apparently, the bug in RHEL 6.1 added a factor of 4 to the variability of the times. Fifth, there is something fishy happening with INC512 and INC4096 in EMPv4 as well as INC4096 in EMPv5. We reran EMPv6, and the anomaly at INC4096 disappeared (the standard deviation was 8 msec and the relative error was 2.0×10^{-6}), showing quite smooth behavior over the wide range of program run times.

Figures 17(c) and 17(d) show *relative* errors over the ET and PT data of each INC measured by the EMP versions. Here the y axes are both logarithmic. PT outperforms ET by an order of magnitude in any of the EMP versions and that the later versions of EMP starting with EMPv4 smooth out, though with the same bumps as before (as expected).

5. FURTHER EVALUATIONS

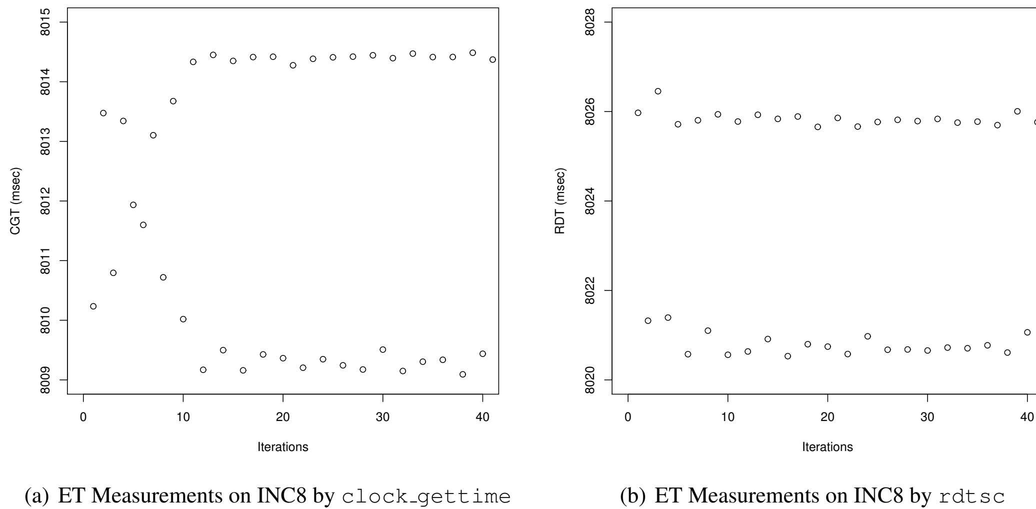
In this section we further evaluate the performance of EMP, comparing the the performance of timing system calls and EMP, and applying EMP to more realistic programs.

5.1. Comparison with Existing System Calls

One question might arise: *how close or even better the measurement quality of EMP is compared to that of timing-relevant system calls*? To respond to this question, we chose `clock_gettime` and `rdtsc` as examples of interval-based and cycle-based methods, respectively. INC8, motivated in Figure 2, was chosen for the comparison.

Figure 18 exhibits 40 ET measurements on INC8 using the two system calls. The measured values (CGT) from `clock_gettime` (Figure 18(a)) alternate up and down, but the overall CGT values vary only a little, over a range of 8,009 to 8,015 msec. Likewise, the measured values (RDT)

Copyright © 2016 John Wiley & Sons, Ltd.



(a) ET Measurements on INC8 by `clock_gettime` (b) ET Measurements on INC8 by `rdtsc`

Figure 18. ET Measurements on INC8 by `clock_gettime` and `rdtsc`

Table XXV. Statistics of ET Measurements on INC8 by `clock_gettime` and `rdtsc`

	Minimum (msec)	Maximum (msec)	Average (msec)	Std. Dev. (msec)	Rel. Err.
`clock_gettime`	8,009	8,014	8,012	2.4	0.0003
`rdtsc`	8,020	8,027	8,023	2.5	0.0003
EMP	8,014	8,023	8,018	1.6	0.0002

from `rdtsc` (Figure 18(b)) are almost constant (e.g., about 8,021 or 8,026 msec) while the RDT values bounce up and down in an alternative fashion. We don't know exactly why this phenomenon occurs. Perhaps the RDT measured may have a correlation with the processor's cycles, though more investigation is needed.

Table XXV compares the statistics for CGT, RDT, and PT. `clock_gettime` revealed the lowest gap (5 msec) between the minimum and the maximum, and thus it might realize slightly better accuracy than the other two. (That said, there is no ground truth on the measured time.) EMP yielded the lowest standard deviation and relative error among the three methods. The overall performance of `rdtsc` was not as good as that of the other two, despite its higher resolution.

Recall that `clock_gettime` cannot be used with proprietary software. Additionally, its measured time is typically contaminated by system noises, hurting precision and accuracy. As discussed before, `rdtsc` is also not appropriate. Furthermore, its quality seems behind the other two's quality.

In sum, EMP can be considered an alternative to system calls, especially when timing proprietary software with compute-bound workloads.

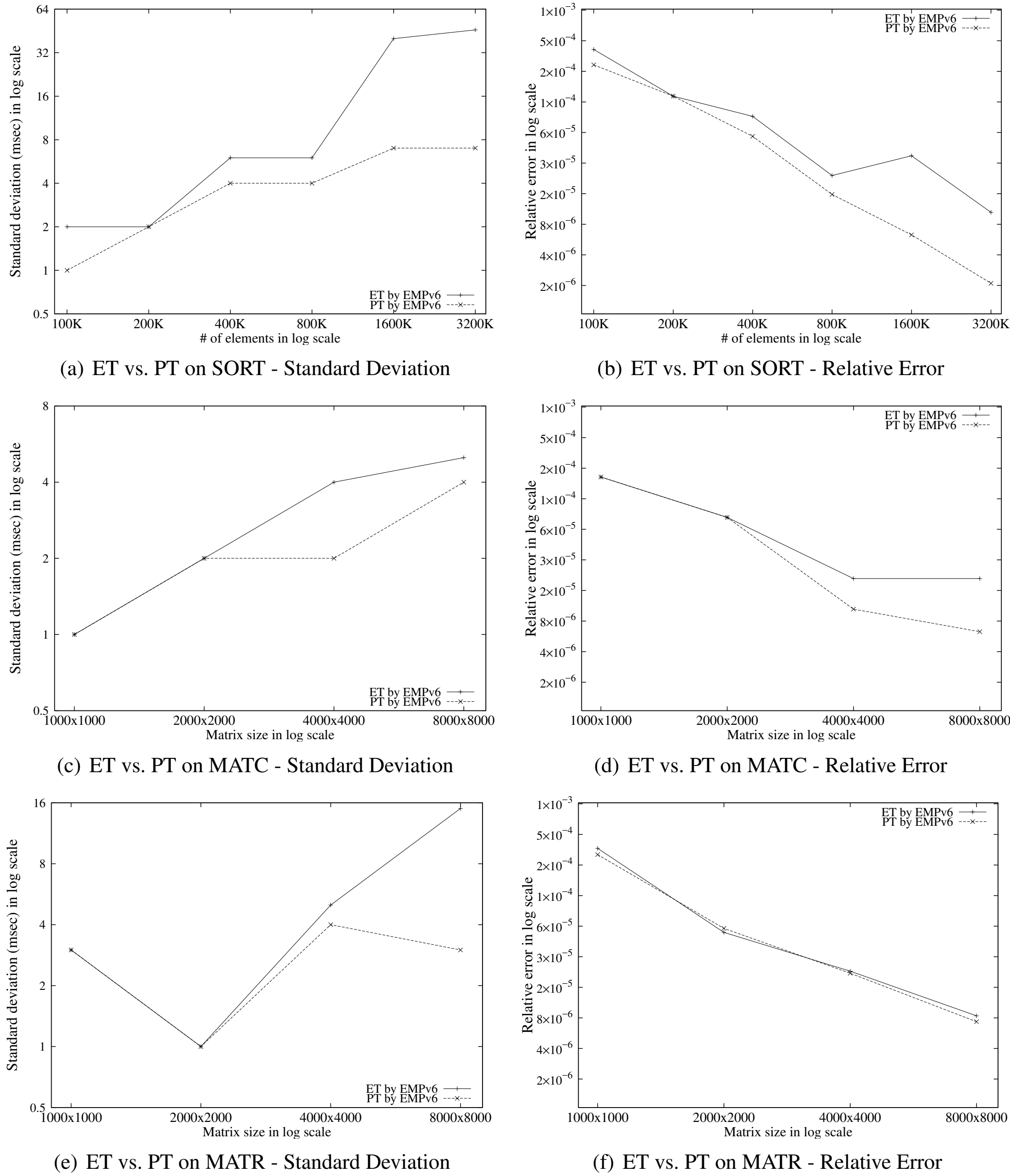
We now proceed to more realistic workloads.

5.2. Experiments with Real Programs

Another question may arise: *does EMP really work for more realistic workloads than a simple program like Figure 7?* Some may claim that the nested-loop of INC measured above is too straightforward to evaluate EMP's efficacy.

To address this concern, we tackle more challenging cases: (i) some real-world programs with memory accesses and greater computation and (ii) a well-known CPU-bound benchmark suite.

Copyright © 2016 John Wiley & Sons, Ltd.



(a) ET vs. PT on SORT - Standard Deviation

(b) ET vs. PT on SORT - Relative Error

(c) ET vs. PT on MATC - Standard Deviation

(d) ET vs. PT on MATC - Relative Error

(e) ET vs. PT on MATR - Standard Deviation

(f) ET vs. PT on MATR - Relative Error

Figure 19. Performance Evaluation on Real-world Programs

Real-world Programs: We apply our measurement methodology to the following three types of programs: 1) an insertion sort program (termed SORT), 2) a square-matrix-multiplication program in column-major order (MATC), and 3) the same program but in row-major order (MATR). SORT performs an insertion sort on an array filled with random integers. That array grows in size from 100K elements to 800K elements. MATC (MATR) performs in column (row) major the multiplication of two given matrices filled with random integers and then stores the results into a third matrix. The size of each matrix increases from 1K-by-1K to 8K-by-8K.

Figure 19 exhibits the evaluation results of EMP on the above real-world programs. Both ET and PT exhibited increasing standard deviation, but PT's was less, with the relative standard deviation falling faster with PT over increasing array size (Figure 19(a)). In particular, the gap between ET and PT became somewhat wider, up to over 5x for the array with 1600K and 3200K elements. The relative error of PT was always lower than that of ET even though there was an overlap observed at 200K (Figure 19(b)).

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

For MATC (MATR), PT also exhibited somewhat slower increasing standard deviation and faster decreasing relative error compared with ET as the matrix size increased. For that biggest matrix, PT dominated ET by 2x.

Concerning the standard deviations of ET on the real-world programs, in particular, there was a significant rise on the y-axis at the longest task length on the x-axis at all times. The rise was mainly attributed to a strong correlation with the run times of daemon processes. In other words, the longer time taken to complete a computation task by SORT or MATC(R), (i) the more likely some existing daemon processes (e.g., `kslowd000` and `kslowd001`) ran longer than usual, and (ii) the more likely some unusual daemon processes (e.g., `flush-9:127`) suddenly showed up and ran for some time, thereby hurting the ET (and PT). (This empirical evidence has been demonstrated with INC throughout this paper.) In contrast, PT was relatively less vulnerable to these daemons' influence. That said, PT was not totally independent of the daemon processes' activities. Further investigation is needed to examine why such infection occurs to PT.

In sum, the performance of EMP (advocating PT) still holds valid for real-world workloads.

We now turn to an evaluation with real-world CPU-bound applications.

SPEC CPU2006: There is an industry-standard benchmark, SPEC CPU2006, designed to evaluate with CPU-intensive workloads the performance of different computer systems. This benchmark reflects various compute-bound real applications. The SPEC benchmark can provide a basis for measuring a system's performance by stressing different components including the processor, the memory subsystem, and the compiler. One of its use cases is to measure and compare the latency of completing a single task. Thus, employing the SPEC benchmark as a real-world workload is a reasonable approach to evaluate EMP.

Table XXVI shows the evaluation results of EMP on the SPEC CPU2006 benchmarks. Note that in the figure there are two empty bins associated with the `481` and `483` benchmarks. For `481`, we could not obtain its measurements, as there happened an unknown runtime error (seemingly caused by provided binary input data). For `483`, we excluded that benchmark in that I/O was involved. Other than these two benchmarks, we had successful runs of a total of 29 (out of 31) benchmarks.

As exhibited in Table XXVI, EMP was effective for the real workloads. Overall, PT outperformed ET on the standard deviation and relative error across the SPEC benchmarks. Most of the benchmarks revealed a smaller standard deviation of PT than that of ET. In addition, regarding the relative error PT surpassed ET for slightly under an half (specifically, 12) of the benchmarks. In particular, a 10x margin between ET and PT was observed in a certain benchmark (`400`).

EMP also scaled well for these real CPU-bound benchmarks, in terms of growth of relative error as the execution time lengthened. For some lightweight benchmarks (e.g., `400`, `403`, `410`, `434`, `445`, and `999`) (taking under 100 sec), PT outperformed ET by about 4.5x, on average. PT continued its dominance against ET for the medium-weight ones (e.g., `447`, `456`, `470`, and `473`) (100$\sim$400 sec). For the heavyweight ones (e.g., `436` and `454`) ($>$ 900 sec), the relative error of PT was lower than that of ET, up to about 1.3x.

Lastly, we still observed a substantial standard deviation (and relative error) for some benchmarks (e.g., `436` and `462`). The high variance seemed mainly attributed to the activities of the `kslowd000` and `kslowd001` daemon processes, as also identified in the real-world programs. These two were not observed during the daemon-cutoff study discussed in Section 4.5. The daemons cannot be switched off, as they involve kernel threads. More investigation is needed to hopefully better remove daemon executions interfering with timing a program.

In summary, these results empirically demonstrate that measurement quality—accuracy, precision, and scalability—of EMP is also realized with real CPU-bound benchmarks.

6. RELATED WORK

Surprisingly, there is little literature regarding program execution time measurement. We found several Unix programming books that address the measurement of running time. Stevens introduces

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

Table XXVI. Performance Evaluation on the SPEC CPU2006 Benchmarks

Benchmarks	Standard Deviation (msec)		Relative Error	
	ET	PT	ET	PT
400.perlbench	21	2	5×10^{-2}	4×10^{-3}
401.bzip2	991	931	2×10^{-3}	2×10^{-3}
403.gcc	106	89	4×10^{-3}	3×10^{-3}
410.bwaves	50	22	6×10^{-3}	3×10^{-3}
416.gamess	1,077	1,087	1×10^{-3}	1×10^{-3}
429.mcf	523	482	2×10^{-3}	2×10^{-3}
433.milc	1,100	1,106	2×10^{-3}	2×10^{-3}
434.zeusmp	73	11	4×10^{-3}	7×10^{-4}
435.gromacs	1,137	1,232	1×10^{-3}	1×10^{-3}
436.cactusADM	4,798	4,536	4×10^{-3}	4×10^{-3}
437.leslie3d	1,462	1,000	3×10^{-3}	2×10^{-3}
444.namd	251	164	4×10^{-4}	3×10^{-4}
445.gobmk	109	23	1×10^{-3}	3×10^{-4}
447.dealII	291	109	6×10^{-4}	2×10^{-4}
450.soplex	109	65	3×10^{-4}	2×10^{-4}
453.povray	646	591	2×10^{-3}	2×10^{-3}
454.calculix	662	560	4×10^{-4}	3×10^{-4}
456.hmmer	108	49	3×10^{-4}	1×10^{-4}
458.sjeng	547	537	9×10^{-4}	9×10^{-4}
459.GemsFDTD	2,135	2,094	3×10^{-3}	3×10^{-3}
462.libquantum	5,265	5,178	9×10^{-3}	9×10^{-3}
464.h264ref	677	611	1×10^{-3}	9×10^{-4}
465.tonto	1,115	1,074	1×10^{-3}	1×10^{-3}
470.lbm	73	35	2×10^{-4}	1×10^{-4}
471.omnetpp	339	289	9×10^{-4}	8×10^{-4}
473.astar	331	320	9×10^{-4}	9×10^{-4}
481.wrf	Runtime Error			
482.sphinx3	1,740	1,787	3×10^{-3}	3×10^{-3}
483.xalancbmk	Out-of-Scope			
998.specrand	0.6	0.6	5×10^{-3}	5×10^{-3}
999.specrand	0.6	0.7	5×10^{-3}	6×10^{-3}

a variety of different library functions [15]. He also discusses measuring performance and presents user-implemented timing methods [16].

Profilers [17, 18] are discussed as tools for checking code bottleneck based on function execution time, and the timers as tools for measuring the amount of time spent executing code segments.

McGeoch in her book presents two basic methods of measuring process time, the time taken for an actively running program [19]. One is to calculate real time, wall clock time or elapsed time derived by comparing the two timestamps of the clock register taken before and after the process' execution. The other is to use interval timing to report computation or CPU time. As a general rule, she claims that CPU time is preferred to algorithm researchers, as the time taken by other processes can be ignored in CPU time, compared to the wall clock time.

The closest work to EMP is the measurement schemes suggested in Bryant and O'Hallaron [20]. The authors introduce two basic mechanisms used by computers for recording the passage of time, one based on a counter incremented every clock cycle, and one based on a low frequency timer periodically interrupting CPU. On top of these two mechanisms, the book presents the two timing schemes by cycle counters (the total number of cycles spent during program execution), and by interval counters (the total elapsed clock ticks during program execution). The two authors found that the most accurate value for the observed elapsed clock ticks was the minimum one. Hence, they devise an approach, called a *minimum-of-k* protocol.

None of these existing approaches takes into consideration variability in these measurements. The longer a Linux program takes, the more variance in the program execution time is observed.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Odom et al.'s work [21] also focuses on measuring long running programs. Their study was conducted on a simulation framework. They used dynamic sampling of trace snippets while execution an application. Our measurement technique is much simpler than their approach, and we don't require an execution trace of an application.

There is a rich body of literature [22, 23, 24, 25] discussing the worst case execution time (WCET) of a process. As addressed here, the papers also point out that measuring worst-case execution time at least needs some support from the underlying operating system. However, the papers focus on measuring the WCET of a deadline-sensitive process on real-time embedded systems. We don't consider real-time constraints.

Several library packages provide access to hardware performance monitor information [26, 27, 28, 29, 30, 31, 32]. Among the packages, the Performance Application Programming Interface (PAPI) [29] seems capable of measuring the execution time of a process. A call to the function `PAPI_flops()` returns four parameters, two of which concerns measured time: *rtime* and *ptime*. According to the PAPI's website [29], rtime indicates total real time in seconds since the first `PAPI_flops()` call, and ptime is total process time in seconds since the first `PAPI_flops()` call. But ptime's resolution is in seconds. Thus, ptime may not be appropriate for a program ending earlier than a second.

Note that a timer should be chosen based on its resolution, taking into account how long the program runs. The timer for a very short code block might be different than that for a long-executing code block.

In Windows, there is an infrastructure, called Windows Management Instrumentation (WMI), for collecting management data and operations on a Windows-based OS [33].

There are also commercial software tools for measuring execution time [34, 35, 36]. These software analyzers offer a timing trace to show exactly what process is executing at what time, similar to open-source tools such as `prof` and `gprof`.

Measuring execution time concerns benchmarking and resource management. A component of BenchExec [37], `runexec`, can measure the CPU time and wall-clock time of a program, as can EMP. There are several differences between EMP and BenchExec. First, EMP identifies a variety of sources of variability in execution time and eliminates the identified sources to minimize the variability before actually measuring the execution time of INC. Second, EMP captures all concurrently-running daemon processes around the main program (INC) and uses them for post analysis. Third, EMP can remove significant outliers greatly contributing to variance in the execution times of INC and then produces the refined timing results with limited variability.

A variety of works take an empirical approach to different aspects of software. Some perform an empirical study related to measuring software performance [38, 39]. Others empirically investigate the performance of an algorithm written in C [40] or in Java [41]. Our work is similar from a methodological perspective, though we (i) identify key sources of variation that have been overlooked and (ii) present a high-quality timing protocol that carefully controls sources of variation.

We mentioned in the introduction that we had earlier developed a protocol for measuring execution time for programs exhibiting I/O, in particular, query execution time in DBMSes [1]. That study identified a variety of Linux measures (e.g., user ticks, system ticks, IOWait ticks, etc.) relevant to measuring query time, presented a structural causal model of explaining the variance of query time, and introduced a timing protocol called TTP (Tucson Timing Protocol) for calculating the query time. That protocol is applicable for measuring execution time of any program on Linux.

7. CONCLUSION

In this paper, we first presented a taxonomy of execution time measurements, distinguishing process time from elapsed time, compute-bound programs from programs that perform I/O (the latter that the Tucson Timing Protocol (TTP) presented earlier [1] is applicable to), and observing that three cases, elapsed time for modifiable programs and program and elapsed time for non-modifiable programs, were still in need of a refined protocol. We introduced a variety of factors affecting measuring the

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

execution time of a compute-bound program on Linux. We then proposed a sequence of refinements that ultimate achieved a protocol (Execution Time Measurement Protocol, or EMP) that exhibits precision, accuracy and scalability. This protocol uses a novel visualization, a dual-execution scatterplot, and a carefully-motivated method of first identifying outliers in EMP (called L-samples) and then outliers (in PT) of infrequent, long-running daemons, and finally cuttoffs that distinguish those daemons from frequent and/or short-running daemons. Our experiments showed the validity and effectiveness of EMP, specifically, that the relative error was reduced by at least an order of magnitude and the observed behavior across widely varying process times (four orders of magnitude) was more regular. The effectiveness of EMP was observed in real-world applications as well.

EMP does not require any modifications to the program under test. It does not rely on hardware counters, and thus is not platform-specific, though at this point the protocol only works on Linux. But it does utilize other measurements provided by Linux to improve the time measurement quality.

8. FUTURE WORK

It would be useful to replicate EMPv6 to see if the strange behavior at INC1024 for ET persists. It would also be helpful to run EMPv6 at higher runtimes, just to see what happens to the relative error.

As stated in Section 4.2, we do not know the reason that EMPv2 (enabling on NTP) reduced the average time. It would be useful to look into this.

But more importantly, now that we have a highly accurate and repeatable means of measuring algorithm time, with relative errors at least an order of magnitude better than a simple protocol, it would be useful to understand the cause(s) of the behavior of the increasing standard deviation and decreasing relative error observed in Figure 17(b). One hypothesis is that there is variability in the actual time measurement; that was rejected by the increasing standard deviation. Another hypothesis is that there is variability on each context swap; that was rejected by the fact that the standard deviation is not linear with program execution time. We looked briefly into whether the number of context switches correlated with relative error and found no such connection, which also argues against that hypothesis. Thus, at this point the origin of the shape of these curves is entirely unknown. There must be some underlying mechanism that generates the observed (actual and relative) standard deviation, but this is the first time to our knowledge that this phenomenon has been observed.

Now that EMPv6 can now provide quite accurate program execution time measurements, even given the existence of the newly-seen mechanism just described, one can now consider whether repeated timing of an algorithm on input data of different sizes could infer the asymptotic complexity of that algorithm. If one wishes to make predictions about running times in the future of an actual program on different size inputs, it is necessary to fit a running time model. Variability of the timing of the algorithm increases the challenge of such an empirical characterization of running time, because multiple cost functions can fit empirical timing data. The more accurately one can measure times on the given inputs, the better the resulting model that is obtained from fitting. The more the timing data is constrained, the more alternative cost functions can be rejected.

9. ELECTRONIC APPENDIX

A separate electronic appendix provides (a) details on the Linux time-keeping mechanism, (b) details on the Linux `proc` file system, (c) a list of non-critical daemons, and (d) details on the NTP daemon.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls

10. ACKNOWLEDGMENTS

We thank Phil Kaslo, Tom Lowry, Tom Buchanan, and Bob Burton for constructing and maintaining our experimental instrument, a laboratory of seven machines and associated software. This research was supported in part by NSF grants IIS-0639106 and IIS-1016205.

REFERENCES

1. Currim S, Snodgrass RT, Suh YK, Zhang R. DBMS Metrology: Measuring Query Time. *ACM Transactions on Database Systems (TODS)* November 2016; **42**(1):3:1–3:42.
2. Spradling CD. SPEC CPU2006 Benchmark Tools. *SIGARCH Computer Architecture News* March 2007; **35**.
3. WORKING GROUP 2 OF THE JOINT COMMITTEE FOR GUIDES IN METROLOGY (JCGM/WG 2). International Vocabulary of Metrology—Basic and General Concepts and Associated Terms 2008.
4. Kerrisk M. LINUX User's Manual: TIME(1). `http://man7.org/linux/man-pages/man1/time.1.html` , viewed on April 29, 2016.
5. Linux Programmer's Manual. Netlink - Communication between Kernel and User Space (`AF_NETLINK`). `http://man7.org/linux/man-pages/man7/netlink.7.html` , viewed on Apr 10, 2014.
6. The Linux Kernel Archives. Per-task Statistics Interface. `https://www.kernel.org/doc/Documentation/accounting/taskstats.txt` , viewed on April 10, 2014.
7. The Linux Kernel Archives. The `/proc` Filesystem. `https://www.kernel.org/doc/Documentation/filesystems/proc.txt` , viewed on April 10, 2016.
8. Red Hat. Red Hat Enterprise Linux. `http://www.redhat.com/en/technologies/linux-platforms/enterprise-linux` , viewed on April 10, 2014.
9. Suh YK, Snodgrass RT, Zhang R. AZDBLAB: A Lab Information System for Large-scale Empirical DBMS Studies. *Proceedings of the VLDB Endowment (PVLDB)* September 2014; **7**(13):1641–1644.
10. Mills DL. Internet Time Synchronization: The Network Time Protocol. *IEEE Transations on Communication* October 1991; **39**(10):1482–1493.
11. Mallada E, Meng X, Hack M, Zhang L, Tang A. Skewless Network Clock Synchronization Without Discontinuity: Convergence and Performance. *IEEE/ACM Transactions on Networking (TON)* October 2015; **23**(5):1619–1633.
12. Intel. Intel Turbo Boost Technology 2.0. `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html` , viewed on March 9, 2015.
13. Intel. Enhanced Intel SpeedStep® Technology. `http://www.intel.com/cd/channel/reseller/ASMO-NA/ENG/203838.htm` , viewed on March 9, 2015.
14. Red Hat. Red Hat Enterprise Linux 6.4 Technical Notes - Networking Timing Protocol. `https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/6.4_Technical_Notes/ntp.html` , viewed on Jan 17, 2014.
15. Stevens WR. ***UNIX Network Programming: Networking APIs, Second Edition, Vol. 1***. Prentice Hall, 2003.
16. Stevens WR, Rago SA. ***Advanced Programming in the UNIX® Environment (2nd Edition)***. Addison-Wesley Professional, 2005.
17. Graham SL, Kessler PB, McKusick MK. `gprof`: A Call Graph Execution Profiler. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*, ACM, 1982; 120–126.
18. CXperf H. HP CXperf Performance Analyzer. `http://www.hp.com/softwarereleases/releases-media2/proginfo/mail176/5971-4689.htm` , viewed on June 12, 2014.
19. Mcgeoch CC. ***A Guide to Experimental Algorithmics***. Cambridge University Express, 2012.
20. Bryant ER, O'Hallaron DR. ***Computer Systems: A Programmers Perspective (1st Edition)***. Addison Wesley, 2002.
21. Odom J, Hollingsworth JK, DeRose L, Ekanadham K, Sbaraglia S. Using Dynamic Tracing Sampling to Measure Long Running Programs. *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*, IEEE, 2005; 59.
22. Burns A, Wellings AJ. Measuring, Monitoring and Enforcing CPU Execution Time. *Ada Letters* Mar 1993; **XIII**(2):54–64.
23. Grund D, Reineke J, Gebhard G. Branch Target Buffers: WCET Analysis Framework and Timing Predictability. *Journal of Systems Architecture* June 2011; **57**(6):625–637.
24. Santos OMD, Wellings A. Measuring and Policing Blocking Times in Real-time Systems. *ACM Transactions on Embedded Computing Systems (TECS)* Aug 2010; **10**(2):1–29.
25. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, *et al.*. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)* May 2008; **7**(3):1–53.
26. Anderson JM, Berc LM, Dean J, Ghemawat S, Henzinger MR, Leung STA, Sites RL, Vandevoorde MT, Waldspurger CA, Weihl WE. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems* November 1997; **15**(4):357–390.
27. DeRose LA. The Hardware Performance Monitor Toolkit. *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing (Euro-Par '01)*, Springer-Verlag, 2001; 122–131.
28. OProfile. OProfile: A System-wide Profiler for Linux Systems. `http://oprofile.sourceforge.net` , viewed on July 15, 2014.
29. Browne S, Dongarra J, Garner N, London K, Mucci P. A Scalable Cross-platform Infrastructure for Application Performance Tuning Using Hardware Counters. *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC'00)*, 2000.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

30. Berrendorf R, Ziegler H, Mohr B. PCL - The Performance Counter Library. `http://berrendorf.inf.h-brs.de/PCL/PCL.html` , viewed on July 16, 2014.
31. Chaney C, DeWitt J, Ganesan K, Jagoda JS, Jones ST, Johnson M, Kacur J, Lenz IN, Levine F, Mu M, *et al.*. Performance Inspector. `http://perfinsp.sourceforge.net` , viewed on July 15, 2014.
32. Zagha M, Larson B, Turner S, Itzkowitz M. Performance Analysis Using the MIPS R10000 Performance Counters. *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC '96)*, IEEE, 1996.
33. Microsoft. Windows Management Instrumentation. `http://msdn.microsoft.com/en-us/library/aa394582(v=vs.85).aspx` , viewed on July 16, 2014.
34. Intel. VTune™ Amplifier XE 2013. `https://software.intel.com/en-us/intel-vtune-amplifier-xe` , viewed on July 15, 2014.
35. TimeSys Corporation. Timesys LinuxLink. `http://www.timesys.com/embedded-linux/linuxlink` viewed on July 15, 2014.
36. Wind River. Wind River Workbench. `http://www.windriver.com/products/product-notes/workbench-product-note.pdf` , viewed on July 16, 2014.
37. Beyer D, Löwe S, Wendler P. Benchmarking and Resource Measurement. *Proceedings of the 22nd International SPIN Symposium on Model Checking of Software (SPIN 2015)*, Springer, 2015; 160–178.
38. Welponer M, Abeni L, Marchetto G, Cigno RL. Measuring and Reducing the Impact of the Operating System Kernel on End-to-end Latencies in Synchronous Packet Switched Networks. *Softw. Pract. Exper.* November 2012; **42**(11):1315–1330.
39. Hashemian R, Krishnamurthy D, Arlitt M. Web Workload Generation Challenges - An Empirical Investigation. *Softw. Pract. Exper.* May 2012; **42**(5):629–647.
40. Nesmachnow S, Luna F, Alba E. An Empirical Time Analysis of Evolutionary Algorithms As C Programs. *Softw. Pract. Exper.* January 2015; **45**(1):111–142.
41. Collberg C, Myles G, Stepp M. An Empirical Study of Java Bytecode Programs. *Softw. Pract. Exper.* May 2007; **37**(6):581–641.

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

A. DETAILS OF LINUX TIME-KEEPING MECHANISM

We continue the discussion of the Linux timing mechanism in Section 2.3.

A.1. Clock Devices

All the clock devices (RTC, TSC, PIT, HPET, and ACPI_PM) have counters associated with them that are updated at each clock tick. Some clock devices have fixed tick frequencies, such as ACPI_PM which ticks exactly 3,579,545 times per second (as defined in `include/linux/acpi_pmtmr.h`). (This strange number is the same as the NTSC (*National Television System Committee*) color synchronization frequency.) There are also clock devices whose frequencies are machine-dependent. For instance, the frequency of TSC is equal to the CPU frequency. Since the CPU frequency varies from machine to machine, TSC's frequency can be determined only when the system is running. Each computer system has a collection of these clock devices. The available clock devices in our experiment machine are TSC, HPET, and ACPI_PM (as seen in `/sys/devices/system/clocksource/clocksource0/available_clocksource`, and the current device is set to TSC (as shown in `/sys/devices/system/clocksource/clocksource0/current_clocksource`).

A.2. Timer Interrupts

Timer interrupts are used in the Linux kernel not only for time keeping, but also other kernel activities such as *process scheduling*. Timer interrupts are issued by the system's hardware (usually PIT or HPET). The frequency of timer interrupt is defined by constant `HZ` in `asm/param.h`. At each timer interrupt, the kernel updates the wall time of the system and records important information such as the current clock cycle count since the last system reset.

A.2.1. `xtime` This variable is used to keep the system wall time. `xtime`'s value is set to the time of RTC during kernel initialization. After that, `xtime` is updated once every timer interrupt. It is necessary to hold the sequential lock `xtime_lock` before accessing `xtime`.

A constant time (the timer interrupt interval) should be added to `xtime` at each timer interrupt.

Almost all the timing functions provided by the Linux kernel are based on the value of `xtime`. However, `xtime` is sometimes too coarse to be used directly. Therefore, for functions like `gettimeofday` and `clock_gettime`, we will need to further process `xtime` in order to provide higher precision.

A.2.2. Clock Sources The Linux kernel uses a `clocksource` data structure to provide an abstraction for clock sources, so that various clock sources can be accessed in a uniform way. A clock source must be registered to the kernel before it can be used, and most clock sources are registered during kernel initialization. The kernel keeps a list of available clock sources, and chooses the one with the highest resolution to use. The kernel checks the clock source list at each timer interval. If there exists a new clock source which is better than the current one, the kernel will switch to that new one. The current clock source is recorded in the `clock` variable in `kernel/time/timekeeping.c`.

A.2.3. Timing Functions This section reviews time-related system calls in the Linux kernel.

- The `time()` system call returns the current time in seconds since midnight of January 1, 1970. It uses a variable named `xtime_cache` that can be considered as a copy of `xtime`. It is synchronized with `xtime` at every timer interrupt. Hence, the `time()` system call simply returns the `tv_sec` field, defined in `xtime_cache`, representing the current time in seconds.
- The `gettimeofday()` system call returns the current time in microseconds. The function also takes a parameter indicating the current time zone, which is simply set to `NULL`. Internally, the system call uses the current clock source used to determine the clock cycle

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

count elapsed since the last timer interrupt. The cycle count is changed into nanoseconds, which are added to the current time and then converted to microseconds and returned.

- The `clock_gettime()` system call is defined in POSIX[‡]. If the function succeeds, the current time in nanoseconds is returned. The function takes a parameter specifying which POSIX clock to use. The most commonly used clocks are `CLOCK_REALTIME` and `CLOCK_MONOTONIC`.
- `CLOCK_REALTIME` returns the current wall time, and it uses the same underlying functions as `gettimeofday`. Thus, their results should be identical except for precision.

Since `xtime` can be modified by the time setting system calls (e.g., `settimeofday`), which can be invoked by arbitrary programs with appropriate superuser privileges, the wall clock time may not be monotonic (i.e., `gettimeofday` or `CLOCK_REALTIME` may return a time value which is earlier than the result of a previous call). `CLOCK_MONOTONIC` solves this problem by always returning monotonic time since the system is up, which means the result would not be affected by modifications to the current wall time.

The Linux kernel maintains another variable `wall_to_monotonic`, which is initialized to be 0. If `xtime` is changed by δ by the set-time functions, then `wall_to_monotonic` would be changed by -δ, such that the sum of the two variables remains unchanged, keeping the result monotonic.

For instance, suppose that at `10:05 AM` someone sets the wall clock time to `11:00 AM` using the time setting system call. As a result, `CLOCK_REALTIME` becomes `11:00 AM`, `wall_to_monotonic` is -55 minutes, and `CLOCK_MONOTONIC` is still `10:05 AM`. Five minutes later (`CLOCK_REALTIME = 11:05 AM`, `CLOCK_MONOTONIC = 10:10 AM`), a user sets the wall clock time back to `10:05 AM`. Now `CLOCK_REALTIME` becomes `10:05 AM`, `wall_to_monotonic` is 5 minutes, and `CLOCK_MONOTONIC` is `10:10 AM`.

A.2.4. `rdtsc` It is worth mentioning that some modern CPUs support more than one frequency, and some Linux operating systems are able to change the CPU frequency on demand, i.e., increasing the CPU frequency while doing bulk computations and decreasing the CPU frequency when idle. This is good for the purpose of power saving but may affect the result of `rdtsc`. Therefore, it is recommended to use `cpufreq-selector` facility to set the CPU frequency to a fixed value before using `rdtsc` for time measurements.

B. `PROC` FILE SYSTEM

The `proc` file system [7] is a pseudo-file system which resides entirely in main memory. It is usually mounted at `/proc`. It provides the same interface as a file system to allow user level access to kernel data structures.

In the `proc` file system, there is a sub-directory for each process which is named by its *process id* (pid). The `/proc/pid/stat`[§] file provides the status information of the process with id *pid*, including user time and system time. Furthermore, the `/proc/pid/task/tid/stat` file provides even more detailed statistics on the threads (identified by *tid*) belonging to the process.

The kernel keeps track of the flow of time by means of timer interrupts. Namely, every time a configured clock device (e.g. `TSC`[¶] in our system) experiences a timer interrupt, the kernel increments the user or system time by one tick. While a process or a thread gets time to run, the accumulated ticks are charged to them. Accordingly, the `/proc/pid/task/tid/stat` gets updated.

[‡] *Portable Operating System Interface for Unix* (POSIX) is the collective name of a family of related standards specified by the IEEE to define the API, along with shell and utilities interfaces for software compatible with variants of the Unix operating system.
[§] The `ps` utility actually reads process information from the `/proc/pid/stat` file.
[¶] This can be confirmed through `/sys/devices/system/clocksource/clocksource0/current_clocksource`

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

C. NON-CRITICAL DAEMONS

We identified unnecessary daemon processes such as abrtd, acpid, atd, auditd, certmonger, crond, cups, haldaemon, iptables, ip6tables, postfix, vmware, vmware-USBArbitrator, and xinetd. The details about these daemons are exhibited in Table XXVII.

We then turned off the daemon processes before conducting actual execution time measurement on INC. After deactivating the daemons, we still observed other running daemons in the following: abrt-dump-oops, aio, async/mgr, ata, ata_aux, bdi-default, cgroup, crypto, dbus-daemon, events, ext4-dio-unwrit, flush-9:0, kacpid, kacpi_hotplug, kacpi_notify, kauditd, kblockd, khelper, khubd, khugepaged, khungtaskd, kintegrityd, kpsmoused, kseriod, kslowd000, kslowd001, ksmd, ksoftirqd, kstriped, ksuspend_usbd, kswapd0, kthrotld, kthreadd, hd-audio0, hd-audio1, ib_addr, ib_cm, ib_mcast, infiniband, init, ipoib, iw_cm_wq, jbd2/md0-8, mcelog, md, md_misc, md0_raid1, mingetty, migration, netns, ntpd, pciehpd, pm, portreserve, rdma_cm, rhnsd, rhsmcertd, rpcbind, rpc.statd, rpciod, rpc.idmapd, rpc.statd, rsyslogd, scsi_eh_0, scsi_eh_1, scsi_eh_2, scsi_eh_3, scsi_eh_4, scsi_eh_5, sshd, sync_supers, ttm_swap, udevd, usbhid_resumer, watchdog, and ypbind.

These live daemons either belong to the kernel or are managed by the *root* user. Due to their association with the kernel, it is necessary to not to turn them off, otherwise, doing so may jeopardize the proper function of the OS.

Table XXVII. List of Eliminated System Daemons

Daemon Name	**Description**
abrtd	Automated bug reporting tool's daemon. A daemon watching for application crashes.
acpid	Advanced Configuration and Power Interface event daemon. A daemon designed to notify user-space programs of ACPI events.
atd	atd runs jobs queued by at
auditd	The Linux Audit daemon. Responsible for writing audit records to the disk, as the user space component to the Linux Auditing System.
certmonger	A daemon that monitors certificates for impending expiration, and is capable of optionally refreshing soon-to-be-expired certificates with the help of a certificate authority (CA).
crond	The daemon to execute scheduled commands (ISC Cron V4.1)
cups	Common Unix Printing System
haldaemon	A Hardware Monitoring System. Auto-recognizes various kinds of hardware and mountable media.
ip6tables	IPv6 packet filter administration
iptables	Administration tool for IPv4 packet filtering and NAT
postfix	Postfix control program. Used to submit mail.
vmware	VMware SVGA video driver. An Xorg driver for VMware virtual video cards.
vmware-USBArbitrator	VMware USB Arbitration Service daemon. Allows USB devices plugged into the HOST to be usable by the guest.
xinetd	The extended Internet services daemon

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

D. TIME ADJUSTMENT BY NETWORK TIME PROTOCOL

Consider two alternative methods of adjusting the local system time. One is to directly modify `xtime` (using set-time functions such as `settimeofday`), so that the system time is corrected immediately. However, there are some problems with this method. If the clock device is itself inaccurate, the local system time will keep deviating from the correct time, and thus we may have to modify `xtime` repeatedly in order to keep the system time accurate. In addition, a large change in the system time may confuse programs like `make` whose functionality depends on the time stamps of files.

Another method is to adjust the number of nanoseconds to be added to `xtime` at each timer interrupt (using `adjtimex` system call). In this way we can try to make the system clock more accurate. This method takes effect slowly, but it can avoid the previous method's problems.

The `adjtimex` system call is used to adjust the value of variable `tick_length`, which is the interval that NTP thinks should be added to `xtime` at each timer interrupt. As we discussed in Section A.2.1, the actual number of nanoseconds added to `xtime` at each timer interrupt is determined by `clock->xtime_interval`. The difference between these two values is calculated and added to `clock->error` at each timer interrupt. If `clock->error` becomes larger than certain threshold, the value of `clock->xtime_interval` and `clock->mult` would be adjusted according to the NTP algorithm to correct the clock.

The NTP daemon (`ntpd`) program adopts both methods. NTP uses `settimeofday` for big adjustments and `adjtimex` for small adjustments. `settimeofday` would affect the result of `gettimeofday` and `CLOCK_REALTIME`, but would not affect the result of `CLOCK_MONOTONIC`, whereas `adjtimex` may affect the result of all subsequent get-time functions.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls