LETTER

SEDONA: A Novel Protocol for Identifying Infrequent, Long-Running Daemons on a Linux System

Young-Kyoon SUH[†a)], ***Member***

SUMMARY Measuring program execution time a much-used technique for performance evaluation in computer science. Without a proper care, however, timed results may vary a lot, thus making it hard to trust their validity. In this paper we propose a novel timing protocol to significantly reduce such variability.

key words: *Infrequent Long-running Daemon, Execution-Time Measurement*

1. Introduction

Measuring program execution time is a much-used technique for performance evaluation in computer science. Despite the importance of accurate and precise execution-time measurement, how to achieve *better* timing has not been well addressed. Surprisingly, there is considerable variability in the measured time. The goal of this paper is to propose a better timing protocol that can significantly reduce such variability and thus enable better timing results without a doubt of their validity.

Consider a simple compute-bound program, what we term *PUT* (program-under-test), as shown in Fig. 1. This program runs a nested for-loop with a specified task length (tl) (in seconds). The tl value is used to compute the number of repetitions (t) for which that for-loop is performed to reach the specified task length.

```
Algorithm PerformManyIncrements($tl$):
$t = tl$ * CONSTANT
for $k = 1$ to $t$ by 1 do
    for $i = 1$ to UINT_MAX-1 by 1 do
        $j$ += 1
    end for
end for
```

Fig. 1 Computation by a Program-Under-Test (PUT)

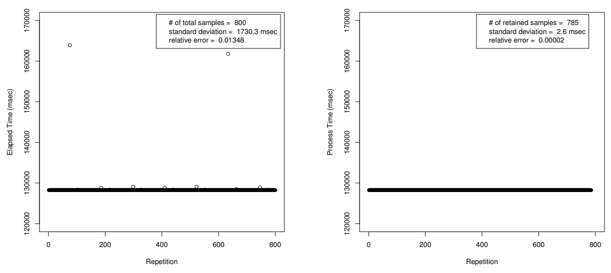
We used 128 seconds as the task length and ran the PUT program (termed PUT128) 800 times. Fig. 2 shows two histograms of the timing results on PUT128. Most of the 800 executions measured in elapsed time, via `System.currentTimeMillis()` in Java, are gathered in the tallest bar of 128,000–129,000 msec, with just a few executions outside of that bar, as can be seen in Fig. 2(a). But these outliers significantly impact the overall measurement quality: a standard deviation of 1,700 msec and a relative error of 0.03.

Manuscript received February 28, 2017.

[†]Supercomputing R&D Center, KISTI, 245 Daehangno, Yuseong-gu, Daejeon, 34141, Republic of Korea

a) E-mail: yksuh@kisti.re.kr

DOI: 10.1587/transinf.E0.D.1



(a) Initial Elapsed Time Measurement (b) Program Time After Our Protocol

Fig. 2 An Example of Measurement Results on a Compute-bound Program-Under-Test

In contrast, our scheme, which we propose shortly, significantly reduces such variability. The gist of the scheme is to (i) focus on *process time* (PT) rather than on elapsed time (ET) and (ii) remove some executions that involve *infrequent, long-running daemon processes*, which impact the process time of the process. Using PT is preferred, in that it considers the time taken for only a process of interest. PT can be calculated as the sum of ticks (where one tick is equal to 10 msec) in user and system mode, available via `taskstats` C struct, provided by the Linux NetLink facility [1]. This use of PT and selective elimination was able to improve measurement quality—in both standard deviation and relative error—up to by three orders of magnitude for PUT128, as illustrated in Fig. 2(b).

Related Work. McGeoch introduced two basic methods of measuring program time: elapsed time and CPU time [2]. Bryant and O'Hallaron [3] presented two timing schemes of using clock-cycle and interval counters. They proposed a measurement protocol, called *minimum-of-k*, that for observed elapsed ticks the minimum is chosen as the most accurate one. Odom et al.'s work [4] focused on timing long-running programs in a simulation framework via dynamic sampling of trace snippets during program execution. None of these prior works, however, takes into account the variability in timing and the influence of daemons that may significantly disturb the timing.

Commercial software tools measure execution time [5]–[7]. Since the tools' source code is not disclosed, there is no way of figuring out whether they can prevent such a daemon from timing.

We previously developed a timing protocol called

TTP (Tucson Timing Protocol) for programs exhibiting I/O, in particular, query execution time in DBMSes [8]. Our study identified a variety of Linux measures (e.g., user ticks, system ticks, IOWait ticks, etc.) relevant to timing a single query and then presented a structural causal model of explicating the variance of query time. Based on this model, we proposed the protocol for calculating the query time. Our scheme we propose in this manuscript is applicable to the TTP protocol, to further improve the query time calculation.

Contribution. Our contributions are following.

- We show empirical evidence that measuring program time can be seriously affected by daemons.
- We propose a novel timing protocol that identifies infrequent, long-running daemons that impact the timing results for that program.
- We evaluate the performance of the protocol with rigorous experiments, starting from a simple program in pure-computation mode to a popular industrial benchmark suite.
- The experimental results show a support for the effectiveness of our scheme.

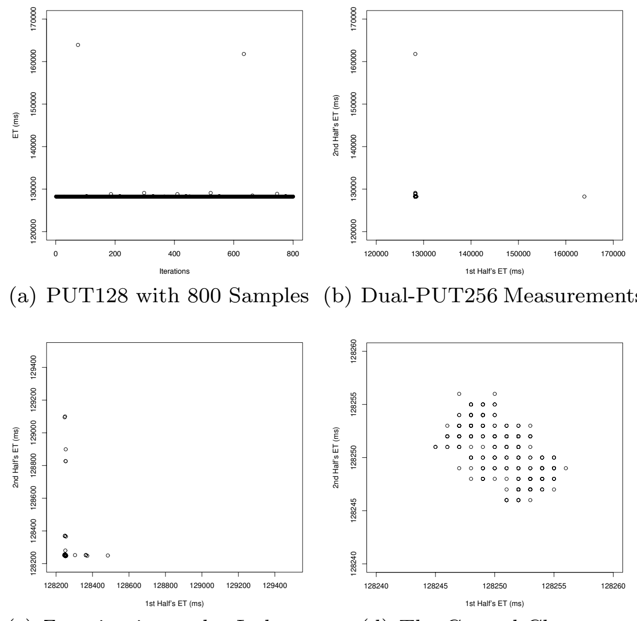
The rest of this letter is organized as follows. Section 2 elaborates on the proposed scheme. In the following section, we evaluate the performance of the scheme using real workloads.

2. Proposed Scheme

In this section we propose a novel execution-time measurement scheme, called *SEDONA* (Selective Elimination through Detection of infrequent, lOng-ruNning dAemons). Our scheme catches and eliminates executions including daemon processes that are infrequent and long-running via a *cutoff* measure, and significantly improves measurement quality. The SEDONA protocol consists of a total of ten steps, as described in Fig. 3.

Algorithm The SEDONA Timing Protocol:
Step 1. Set up the timing environment.
Step 2. Perform a single PUT run (specifically, PUT128) for many samples (specifically, 800).
Step 3. Consider each pair of elapsed time measurements to be a dual-PUT measurement and examine a scatter-plot to see if it it displays an L-shape.
Step 4. Zoom into the central cluster to ensure that it is symmetric (roughly circular).
Step 5. Compute the maximum and standard deviation of the process time for each daemon encountered within the central cluster samples.
Step 6. Identify for each sample in the L-shape infrequent, long-running daemon executions.
Step 7. Determine potentially periodic daemons based on the L-executions and for each daemon compute the minimum process time from those executions identified.
Step 8. Perform Steps 1–6 above for a single run consisting of a small number of executions (specifically, 40) of PUT16384.
Step 9. Compute the cutoffs for each identified daemon.
Step 10. Discard an execution including a daemon of which process time is greater than the respective cutoff time.

Fig. 3 Summary of the SEDONA Timing Protocol



(a) PUT128 with 800 Samples (b) Dual-PUT256 Measurements

(c) Zooming in on the L-shape (d) The Central Cluster

Fig. 4 Successive Scatter plots of a PUT128 with 800 samples (equivalent to a Dual-PUT256 with 400 samples) in Steps 2—4

A Running Example. As motivated by our prior work [8], we configure a timing environment by i) stopping non-critical daemons, ii) activating the Network Timing Protocol daemon, and iii) switching off particular CPU features[9], [10] if any (Step 1).

We then run a program-under-test (called PUT) of Fig. 1 with a task length of 128 sec, termed *PUT128*, 800 times (Step 2). (We render Fig. 2(a) using the 800 samples from this run.) We use 128 seconds because that is long enough to perhaps experience an infrequent daemon. We run it 800 times to capture infrequent daemons that perhaps run every few hours or even once a day. Note that we collect all daemon processes as well as the PUT and their measures through the Netlink interface from the kernel before and after each timing.

Fig. 4(a) plots all the 800 elapsed times of the run of PUT128. The plot clearly shows three rows; that is, the top and middle rows represent over a dozen of outliers far from the rest of the samples clustered in the bottom row. We'll now drill down into these outliers to show how to reliably eliminate the indirect influence of some "infrequent, long-running daemons" on the process time (PT) of the PUT.

To identify such daemons, we use a novel scatter plot: those of *pairs of successive samples.* Fig. 4(b) presents such a scatter plot of 400 samples of a dual-PUT256 constructed from a run of the 800 PUT128 samples (Step 3). There are two quite obvious outliers with ETs of 163,913 msec (rightmost) and 161,785 msec (uppermost), respectively.

We informally term this phenomenon of a scatter plot of a dual-PUT run an "L-shape," and attribute it to the presence of infrequent long-running daemons.

Figure 4(c) 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 continue zooming until we get to Figure 4(d), which shows a central cluster (Step 4). 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 then perform Step 5, which computes the maximum process time and standard deviation of PT (process time, note the switch in emphasis from elapsed time to process time) of the daemon processes (e.g. `cifsd`, `flush-9:0`, `jbd2/md0-8`, `kblockd/0`, `khugepaged`, `md0_raid1`, and `ntpd`) observed in the central cluster samples in Fig. 4(d).

In Step 6 we identify, for each daemon in the L-samples, those that are actual long-running daemon executions. We define such executions as those whose PT is over two standard deviations above the maximum PT for that daemon in the central cluster samples. In the running example `flush-9:0`, `jbd2/md0-8`, and `md0_raid1` are determined as infrequent, long-running. We also identify "extra" infrequent daemons: those found only in the L-samples but not in the central cluster. The running example reveals the following extra daemons: `bash`, `grep`, `rhn_check`, `rhnsd`, `rhsmcertd`, `rhsmcertd-worke`, and `sshd`.

For each of the infrequent daemons we use a heuristic to determine the daemon's periodicity: the daemon must occur regularly in a sequence of samples. For instance, the `rhn_check` daemon appears roughly every 112 samples (which corresponds to very close to every four hours). Four others (`flush-9:0`, `jbd2/md0-8`, `md0_raid1`, and `rhn_check`) all occur together (in those two outliers in Fig. 4(b)) and have a periodicity of about every 559 samples (5x longer, or just about 20 hours).

Next, we can compute for each so-identified infrequent, long-running daemon its minimum time in the L-samples (Step 7). This computation provides a rough, initial distinction of a "long-running" daemon, namely, the valley between the maximum PT from the central cluster and the minimum PT from the L-samples, to differentiate "short-running" from "long-running" executions of the daemon. For those daemons (i.e. `grep`) never appearing in the central cluster, this initial analysis concludes only that they are infrequent.

In Step 8 we repeat Steps 1–6, but instead with the much-longer running PUT16384 (4.5 hours per sample versus 2 minutes), to see if any of our identified infrequent daemons are actually frequent at that much longer PUT execution time. We find some frequent daemon processes appearing in both of the clusters of the dual-PUT256 and dual-PUT32768: `flush-9.0`, `jbd2/md0-8`, `kblock/0`, `md0_raid1`, and `ntpd`. That said, the central cluster also contains other processes not seen in the dual-PUT256 central cluster: `grep`, `rhn_check`, `rhnsd`, `rhsmcertd`, `rhsmcertd-worke`, and `sshd`. But these daemons were categorized in the dual-PUT256 analysis as *infrequent*, several having periodicities estimated at four or twenty hours. When the PUT had a "short" program time (in this case, two minutes), daemons with a periodicity of hours are infrequent. But with a PUT with a "long" program time (in this case, 4.5 hours), some of those daemons are now frequent, and appear in the central cluster.

In Step 9 we compute the cutoff for each of those infrequent, long-running daemons so identified, based on the PUT128 and the PUT16384 as collected in Tab. 1. Here is how to compute the cutoff. For the cutoff of such a daemon with PUT128, we take the midpoint between the maximum of that daemon's PTs in the central cluster (or 0, if absent) and the minimum of those in the L-samples. For the cutoff of such a daemon with PUT16384, we do the same. We then compute a "task time" as 5% of the inferred periodicity. This 5% ensures that such infrequent daemons will impact only a small percentage of the shorter PUTs, while presumably being associated with much larger cutoffs for the very long PUTs. We also include daemons that (a) were identified as infrequent and long-running from PUT128 and (b) were not identified as so in the PUT16384 L-samples, but may have in the dual-PUT32768 central cluster. We then take the *maximum* of the two cutoffs for the final cutoff PT (the last column of Tab. 1).

Process Name	Cutoff PT on PUT128	Cutoff PT on PUT16K	Task Time	**Final Cutoff PT**
`bash`	1 msec	—	—	**1 msec**
`flush-9:0`	64 msec	—	< 1 hour	**64 msec**
	—	48 msec	≥ 1 hour	**48 msec**
`grep`	1 msec	12 msec	—	**12 msec**
`jbd2/md0-8`	4 msec	—	< 1 hour	**4 msec**
	—	11 msec	≥ 1 hour	**11 msec**
`md0_raid1`	35 msec	—	< 1 hour	**35 msec**
	—	51 msec	≥ 1 hour	**51 msec**
`rhn_check`	281 msec	—	< 12 min	**281 msec**
	—	12,828 msec	≥ 12 min	**12,828 msec**
`rhnsd`	2 msec	—	< 12 min	**2 msec**
	—	12 msec	≥ 12 min	**12 msec**
`rhsmcertd`	1 msec	1 msec	—	**1 msec**
`rhsmcertd` `-worke`	57 msec	—	< 12 min	**57 msec**
	—	119 msec	≥ 12 min	**119 msec**
`sshd`	2 msec	23 msec	—	**23 msec**

Table 1 Collected Infrequent, Long-running Daemons and Their Final Cutoff Process Time (Step 9)

Based on Tab. 1, we discard any sample containing an infrequent, long-running daemon execution over the corresponding cutoff. We thus end up dropping just fifteen of the 800 PUT128 samples and only two of the forty PUT16384 samples. As a result, the overall measurement quality—the standard deviation and relative error —for PUT128 was improved by three orders of magnitude (as shown in Fig. 2) and by about two orders of magnitude for PUT16384.

3. Evaluation

We now evaluate the performance of the SEDONA protocol. Our experiments were conducted on a machine described in Table 2.

OS	Red Hat Ent. Linux (RHEL) 6.4 with a kernel of 2.6.32
CPU	Intel Core i7-870 Lynnfield 2.93GHz quad-core processor
RAM	4GB of DDR3 1333 dual-channel memory
HDD	Western Digital Caviar Black 1TB 7200rpm SATA Drive

Table 2 Machine Configurations

We evaluated the performance of SEDONA using SPEC CPU2006 benchmarks [11], providing various compute-bound real applications. The results are provided in Table 3. Note that in the table the results for `481` and `483` benchmarks are omitted because of some runtime error and incurred I/O, respectively.

Table 3 shows that the SEDONA protocol (which uses PT) significantly outperformed the original measurement technique (using ET), termed ORG, on the standard deviation and relative error across the different SPEC benchmarks. None of the benchmarks revealed a bigger standard deviation from the SEDONA protocol as compared to that of ORG. Our protocol quite effectively filtered out infrequent daemon executions in these real-world workloads. Furthermore, the relative error of SEDONA was lower than that of ORG for all the benchmarks. For instance, about a 10x margin between the two was yielded for `434.zeusmp`.

SEDONA also scaled well for the SPEC workloads, with regard to growth of relative error as the execution time lengthened. For the short benchmarks (e.g., `400`, `403`, `410`, `434`, `445`, and `999`: those taking under 100 sec), our scheme outperformed the ORG by about 3.5x, on average. The SEDONA protocol continued its dominance against the conventional technique for the medium-length benchmarks (e.g., `447`, `456`, `470`, and `473`). For the long-running benchmarks (e.g., `436` and `454`, both> 1,000 sec), the relative error of SEDONA was slightly lower than that of ORG.

To summarize, our proposed SEDONA protocol can achieve *better* accuracy, precision, and scalability in measuring the execution time of real compute-bound benchmarks than the conventional timing method.

	Execution Time (ms)		Standard Deviation (ms)		Relative Error	
	ORG (in ET)	SEDONA (in PT)	ORG	SEDONA	ORG	SEDONA
400	454	445	3	2	6×10^{-3}	3×10^{-3}
401	536,639	528,517	1,185	1,161	2×10^{-3}	2×10^{-3}
403	26,109	25,695	138	96	5×10^{-3}	4×10^{-3}
410	7,938	7,801	46	19	6×10^{-3}	2×10^{-3}
416	1,015,342	999,846	965	876	1×10^{-3}	9×10^{-4}
429	235,811	232,209	623	600	3×10^{-3}	3×10^{-3}
433	480,586	473,256	743	725	2×10^{-3}	2×10^{-3}
434	16,495	16,242	75	7	5×10^{-3}	5×10^{-4}
435	990,575	975,445	947	900	1×10^{-3}	9×10^{-4}
436	1,160,742	1,143,078	3,914	3,843	3×10^{-3}	3×10^{-3}
437	581,635	572,775	1,492	1,475	3×10^{-3}	3×10^{-3}
444	591,201	582,229	294	281	5×10^{-4}	5×10^{-4}
445	84,435	83,164	91	28	1×10^{-3}	3×10^{-4}
447	521,846	513,493	150	108	3×10^{-4}	2×10^{-4}
450	341,030	335,848	99	91	3×10^{-4}	2×10^{-4}
453	258,797	254,496	623	582	2×10^{-3}	2×10^{-3}
454	1,721,804	1,695,613	678	627	4×10^{-4}	4×10^{-4}
456	410,533	404,328	85	50	2×10^{-4}	1×10^{-4}
458	589,541	580,591	542	513	9×10^{-4}	9×10^{-4}
459	798,917	786,726	2,143	2,132	3×10^{-3}	3×10^{-3}
462	595,188	586,120	3,326	3,274	6×10^{-3}	6×10^{-3}
464	649,838	639,939	601	563	9×10^{-4}	9×10^{-4}
465	895,754	882,106	883	797	1×10^{-3}	1×10^{-3}
470	349,830	344,510	143	94	4×10^{-4}	3×10^{-4}
471	367,589	361,959	2,114	2,072	6×10^{-4}	6×10^{-4}
473	362,587	357,090	359	317	1×10^{-3}	9×10^{-4}
482	654,208	644,223	2,436	2,392	4×10^{-3}	4×10^{-3}
998	128	127	0.6	0.6	4×10^{-3}	4×10^{-3}
999	128	127	0.8	0.6	6×10^{-3}	5×10^{-3}
		Averages	852	815	3×10^{-3}	2×10^{-3}

Table 3 Performance Evaluation on the SPEC Benchmarks

4. Conclusion and Future Work

We presented a novel execution-time measurement scheme called *SEDONA*, which is more precise and accurate than the traditional method. Our plan is to integrate SEDONA into the query timing protocol [8].

References

[1] Linux Programmer's Manual, "Netlink - Communication between Kernel and User Space (`AF_NETLINK`)." `http://man7.org/linux/man-pages/man7/netlink.7.html`.

[2] C.C. Mcgeoch, **A Guide to Experimental Algorithmics**, Cambridge University Express, 2012.

[3] E.R. Bryant and D.R. O'Hallaron, **Computer Systems: A Programmers Perspective**, Addison Wesley, 2002.

[4] Odom, J. et al., "Using Dynamic Tracing Sampling to Measure Long Running Programs," Proc. of ACM/IEEE Int. Conf. on Supercomputing (SC), p.59, IEEE, 2005.

[5] Intel, "VTune^{TM} Amplifier XE 2013." `https://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[6] TimeSys Corporation, "Timesys LinuxLink." `http://www.timesys.com/embedded-linux/linuxlink`, accessed 2016.

[7] Wind River, "Wind River Workbench." `http://www.windriver.com/products/product-notes/workbench-product-note.pdf`, accessed 2016.

[8] Currim, S. et al., "DBMS Metrology: Measuring Query Time," ACM TODS, vol.42, no.1, pp.3:1–3:42, Nov. 2016.

[9] Intel, "Intel Turbo Boost Technology 2.0." `http://intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`, accessed 2016.

[10] Intel, "Enhanced Intel SpeedStep® Technology." `http://intel.com/content/www/us/en/support/processors/000005723.html`, accessed 2016.

[11] C.D. Spradling, "SPEC CPU2006 Benchmark Tools," SIGARCH Comp. Arch. News, vol.35, March 2007.