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 is a much-used technique for performance evaluation in computer science. Without 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.

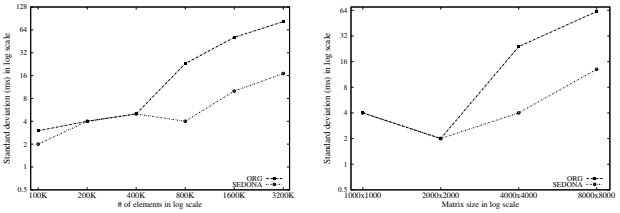
key words: *Infrequent Long-running Daemon, Execution-Time Measurement*

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.

Fig. 1 compared the performance of the existing timing scheme, termed *ORG* using end-to-end (elapsed) time and our timing protocol, termed *SEDONA* (Selective Elimination through Detection of infrequent, lOng-ruNning dAemons), to be proposed shortly. In this comparison we used two kinds of real-world programs such as insertion sort, termed SORT and matrix multiplication (in column major), termed MM. In this comparison SEDONA outperformed ORG in terms of measurement quality gauged via standard deviation. To be more specific, the performance gap between the two schemes over growing workload were increasingly widened, reaching up to by 6x.

Characteristics. There are several important specifics of our protocol. First, we utilize *process time* (PT) rather than on elapsed time (ET). The use of PT is preferred, as it takes into account the time taken for only a "process" (or program) of interest. PT is defined as the sum of ticks (where one tick is equal to 10 msec) in user and system mode. These tick measures are obtainable via `taskstats` C struct, provided by the Linux NetLink facility [1]. Second, the protocol throws out some executions that involve *infrequent, long-running daemon processes*. We observe that such daemon processes substantially impact the execution time of the process. By eliminating those executions, the protocol greatly improves the overall measurement quality, which would otherwise be biased by them.



(a) SORT - Standard Deviation (b) MM - Standard Deviation

Fig. 1 Performance Comparison on Real-world Programs

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. But none of these prior works considers the variability in timing and the impact of daemons that may disturb the timing a lot.

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.

Contribution. Our contributions are following.

- We show empirical evidence that measuring execution time can be seriously affected by daemons.
- We present an algorithm to identify and eliminate such daemons that are infrequent, long-running and thus impact the timing of a given program.
- We propose a novel timing protocol that can considerably reduce variance via the elimination.
- The evaluation results on real-world programs show a support for the effectiveness of the protocol.

The rest is organized as follows. We next elaborate on the proposed timing protocol. In turn we evaluate the performance of the scheme using real workloads.

Manuscript received April 7, 2017.

[†]Dept. of Scientific Platform Development, Korea Inst. of Sci. and Tech. Info. (KISTI), Daejeon, 34141, Korea

*This work was in part supported by the EDISON program (NRF-2011-0020576). The author acknowledges Prof. Rick Snodgrass for his suggestions to improve this article.

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

DOI: 10.1587/transinf.E0.D.1

2. Proposed Scheme

In this section we propose our SEDONA timing protocol. SEDONA (1) utilizes PT (process time) to avoid absorbing timing noise from other co-running processes and (2) identifies and eliminates executions including daemon processes that are infrequent and long-running via a *cutoff* measure.

Our SEDONA protocol consists of a total of ten steps, as described in Fig. 2. Note that this protocol (algorithm) is applicable to any arbitrary Linux system.

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. 2 Summary of the SEDONA Timing Protocol

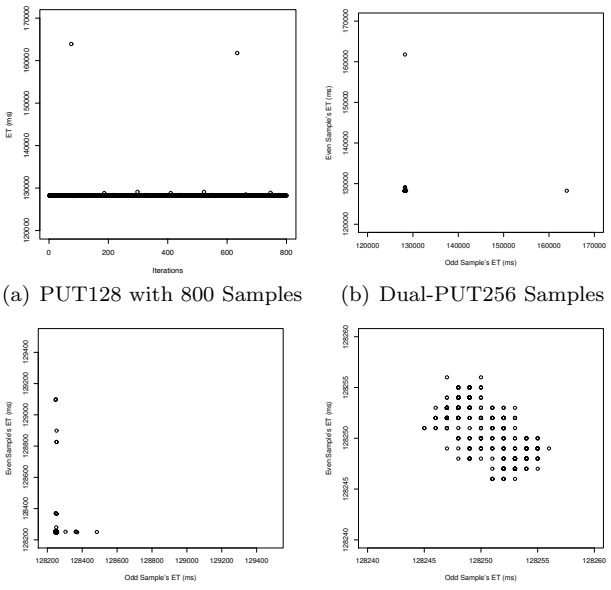
A Running Example. In Step 1 we configure the same timing environment as used in our prior work [8], to eliminate known timing factors. The environment requires (i) deactivating non-critical daemons, (ii) switching on the Network Timing Protocol daemon, (iii) turning off particular CPU features [9], [10], and (iv) getting an up-to-date kernel installed.

We then run a simple program-under-test (called PUT) many times, as shown in Fig. 3 (Step 2). PUT runs a nested for-loop with a specified task length (tl) (in seconds). The tl value is used to compute the number of iterations (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. 3 Computation by a Program-Under-Test (PUT)

The run of Step 2 assigns a task length of 128 sec to PUT, termed *PUT128*, and repeats executing PUT128 800 times. 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.



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

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

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

Fig. 4(a) plots all the 800 elapsed times (ETs) 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 will now drill down into these outliers to show how to reliably eliminate the indirect influence of some "infrequent, long-running daemons" on PT (process time) of the PUT.

To identify such daemons, we use a novel scatter plot: those of *pairs of successive samples*. So samples #1 and #2 form the first pair and samples #3 and #4 form the second pair. Such a pair is termed "*dual-PUT*." A sample of dual-PUT256, for instance, consists of two consecutive (odd and even) samples of PUT128: sample #1 of dual-PUT256 is equivalent to samples #1 and #2 of PUT128.

Fig. 4(b) presents such a scatter plot of 400 samples of a run of dual-PUT256 constructed from a run of the 800 PUT128 samples (Step 3). The xy-plane corresponds to the dual-PUT samples. For example, sample #2 of a run of dual-PUT256 is plotted as a point with samples #3 and #4 of a run of PUT128 on the respective x and y axes. In that sense the x (y) axis is named as "Odd (Even) Sample's ET." There are two quite obvious outliers with ETs of 163,913 ms

(rightmost) and 161,785 ms (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 ET to PT) of the daemon processes (i.e. `flush-9:0`) 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: `bash`, `grep`, `rhn_check`, `rhnsd`, `rhsmcertd`, `rhsmcertd-worke`, and `sshd`, those found only in the L-samples but not in the central cluster.

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, `rhn_check` appears roughly every 112 samples (or almost every four hours). Four others (`flush-9:0`, `jbd2/md0-8`, `md0_raid1`, and `rhn_check`) all occur together 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, or 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 dual-PUT256 and dual-PUT32768, each consisting of pairs of two successive samples of PUT128 and PUT16384. 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 PUT128 had a "short" program time (or, two minutes), daemons with a periodicity of hours are infrequent. But with PUT16384 with a "long" program time (or, 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 runs of PUT128 and 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). (Note that a different distribution than RedHat Enterprise Linux may yield different cutoff data from those of Tab. 1, but this cutoff idea is applicable to any Linux distribution and arbitrary program.)

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

Based on Tab. 1, SEDONA discards any sample containing an infrequent, long-running daemon execution over the corresponding cutoff.

The following section presents the evaluation results of SEDONA via a popular compute-bound benchmark suite.

3. Evaluation

We now evaluate the performance of SEDONA compared to that of the original timing scheme, termed ORG, based on elapsed time. 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

The validity of our protocol was evaluated with SPEC CPU2006 benchmarks [11], providing various compute-bound real applications (as well as the real-world programs as shown in Fig. 1). 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 SEDONA outperformed ORG, on the standard deviation and relative error across the very different SPEC benchmarks. To be more specific, all the benchmarks revealed a smaller standard deviation from SEDONA as compared to that of ORG. Our timing protocol quite effectively filtered out infrequent daemon executions in the industrial workloads. The relative error of SEDONA was also lower than that of ORG for almost every benchmark. For example, about a 10x margin between the two resulted from `434`.

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 method 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`). Even for the long-running benchmarks (e.g. `416`, `436`, and `454`, both >1,000 sec), the relative error of SEDONA was still 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 programs than the ET-based existing method. The experimental results demonstrate the general applicability of SEDONA to timing any CPU-bound program.

4. Conclusion

We proposed a novel timing protocol called *SEDONA*, and performed an empirical evaluation to show that it is more precise and accurate than the extant method.

	Execution Time (ms)		Standard Deviation (ms)		Relative Error	
	ORG (in ET)	SED. (in PT)	ORG	SED.	ORG	SED.
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

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`, accessed April 3, 2017.

[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] J. Odom, J.K. Hollingsworth, L. DeRose, K. Ekanadham, and S. Sbaraglia, "Using Dynamic Tracing Sampling to Measure Long Running Programs," Proceedings of the ACM/IEEE Conference on Supercomputing (SC), p.59, IEEE, 2005.

[5] Intel, "VTune™ Amplifier 2017." `https://software.intel.com/en-us/intel-vtune-amplifier-xe`, accessed April 5, 2017.

[6] TimeSys Corporation, "Timesys LinuxLink." `http://www.timesys.com/embedded-linux/linuxlink`, accessed May 18, 2016.

[7] Wind River, "Wind River Workbench." `http://www.windriver.com/products/product-notes/workbench-product-note.pdf`, accessed February 27, 2016.

[8] S. Currim, R.T. Snodgrass, Y.K. Suh, and R. Zhang, "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 August 7, 2016.

[10] Intel, "Enhanced Intel SpeedStep® Technology." `http://intel.com/content/www/us/en/support/processors/000005723.html`, accessed September 4, 2016.

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