DOI: 10.14529/jsfi160405

Analysis of CPU Usage Data Properties and their possible impact on Performance Monitoring

Konstantin S. Stefanov[1], *Alexey A. Gradskov*[1]

© The Authors 2016. This paper is published with open access at SuperFri.org

CPU usage data (CPU user, system, iowait etc. load levels) are often the basic data used for performance monitoring. The source of these data is the operating system. In this paper we analyze some properties of CPU usage data provided by the Linux kernel. We examine the kernel source code and provide test results to find which level of accuracy and precision one may expect when using CPU load level data.

Keywords: performance monitoring, sensor properties, sampling rate, CPU usage, CPU load level.

Introduction

Today supercomputers show very impressive performance for their peak values and with some benchmarks like LINPACK [1]. Other benchmarks like HPCG [2] as well as real-world applications produce much worse results, often not reaching even 10% of peak performance.

Performance monitoring is one of the methods used to evaluate applications while they are running and determine the obstacles to higher sustained performance of applications. The idea of performance monitoring is to collect metrics which describe the state of the application being run. These data are collected for all compute nodes running the given application. Some performance monitoring approaches [3] try to analyze metrics obtained from components which are shared for the whole system and correlate those data to specific applications.

The source of metrics data, which we call sensors, may be hardware like performance counters in modern CPU, or software, like various data provided by the operating system such as CPU usage data, load average, memory usage etc. Some sensors may be somewhere on the border between software and hardware like InfiniBand interface counters, which are maintained by InfiniBand card firmware.

There are questions about the properties of such sensors and their suitability for performance monitoring in different modes. Of course all those sensors are widely used for a long time for performance monitoring and give useful data which lead to useful results in application analysis. But as performance monitoring systems evolve we may encounter some limitations of such sensors which may lead to their unsuitability for new approaches or new modes of usage. For example SuperMon [4] can achieve up to 6000 Hz sampling rate while reading Linux kernel data from `/proc` [5] filesystem. But do we need such sampling rate and are the results obtained at such a high rate reliable? Will such high rate affect the precision of the data?

This question is not widely discussed for every type of data used for performance monitoring. When performance counters were introduced in processors, hardware performance counters were analyzed [6–9] from the point of accuracy, predictability, reproducibility and so on. Paper [10] compares two modes of using performance counters and compares the results obtained in these modes. A discussion about Load Average data in Linux kernel aroused on mailing lists [11]. This discussion resulted in some patches on kernel source code to make Load Average results more accurate, but it is not clear if today kernel load average data are accurate enough, and we found no such analysis for sensors other than performance counters and load average.

[1]M.V. Lomonosov Moscow State University, Moscow, Russia

CPU usage data are obtained from Linux kernel and are widely used for performance monitoring. In this paper we try to analyze Linux kernel source code and make some testing to evaluate CPU usage data properties which are vital for analyzing application behavior.

The paper is organized as follows. Section 1 describes how the CPU usage data in their conventional form are obtained from the kernel. Section 2 gives results of analysis of Linux kernel source code in parts which relate to CPU usage data. Section 3 provides results of testing supporting the results which were given in Section 2. The last section contains the conclusion.

1. How CPU usage data are obtained

The Linux kernel gives CPU usage data in `/proc/stat` file. For every active CPU in the system the kernel gives the amount of time, measured in 1/100 ths of a second, that the system spent in different modes of execution [5] since boot. These different modes are: user mode (running user processes), user mode with low priority (nice), system mode (running kernel), idle, iowait (idle while waiting for IO request to complete), irq (processing interrupts), softirq (processing software interrupts) and a few other modes related to virtualization. To obtain CPU usage in the form we are accustomed to with `top` or other utilities (we call it CPU load level hereafter), one should take the difference in one mode values between two successive measurements and divide it by the sum of such differences for all modes. If T_m^i is the time spent in m-th mode at time moment i (these values are given in `/proc/stat`), than the CPU load level L_m for the mode m is

$$L_m = \frac{T_m^i - T_m^{i-1}}{\sum\limits_m (T_m^i - T_m^{i-1})}$$

One can also try to get CPU load level by dividing $T_m^i - T_m^{i-1}$ by the time difference between i and $i-1$ time moments, but as obtaining precise time is quite an expensive operation involving system call, this method is usually not used.

One consequence of this calculation method is that CPU usage values are discrete in nature. The number of different values they can take depends on sampling interval. The more time passes between successive samples, the more levels CPU usage value can take. `top` utility gives us CPU usage percent with precision of 1 decimal place (1000 possible different values in range from 0 to 100%). The data supplied by the kernel which are used for calculations are measured in 1/100 ths of a second; to have real precision of 1/10 th of percent for CPU load level value we should take the measurements more rarely than once in 10 seconds. With more frequent sampling the precision of CPU load level will be less than 1 decimal place.

2. Source code analysis

We examined the Linux kernel source code to find how these values (time spent in different modes) are calculated.

These per-CPU values (and per-process values for time spent in user and system mode, too) are updated during timer interrupt processing. Timer interrupt frequency is a parameter set during kernel compilation (it is named `HZ`). The most common values for this parameter are 1000 and 250 (timer interrupt is raised 1000 or 250 times per second, respectively). When executing the timer interrupt handler (generally it is executed on every CPU with some exceptions described later), the kernel finds the mode in which the given CPU was before switching to interrupt

handler and which process was running. The whole tick is accounted to the mode and to the process which was active before the CPU received timer interrupt. The modes which were active in period between timer interrupts are not accounted in any way.

Internally, CPU usage times are calculated in kernel as numbers of 1/`HZ` second time intervals. When the results are returned to the user, they are scaled to 1/100 ths of a second. Such rescaling may introduce some rounding errors but they are not expected to be high.

When `NO_HZ` [12] kernel compilation parameter is active (true by default for modern SMP kernels), timer interrupts are not delivered to idle CPUs. When CPU usage values are requested, idle and iowait times are calculated at the moment of the request by finding the time since the given CPU became idle. When the CPU comes out of idle state, the values for idle and iowait are calculated and saved to accounting data structures.

The outcome of this calculation method is that the CPU usage times for user, system, and nice modes are updated only on timer interrupts and that some frequent changes from running to idle or between other modes may pass unnoticed by the accounting code.

3. Experiments

3.1. Measuring interval between CPU usage data changes

Our first experiment was designed to prove that CPU usage times for user, system, and nice modes are updated only on some periodic events. To check this we performed a test which was constantly requesting CPU usage time and calculated time which passed between successive CPU usage changes. The results are presented in fig. 1. Time interval in milliseconds between

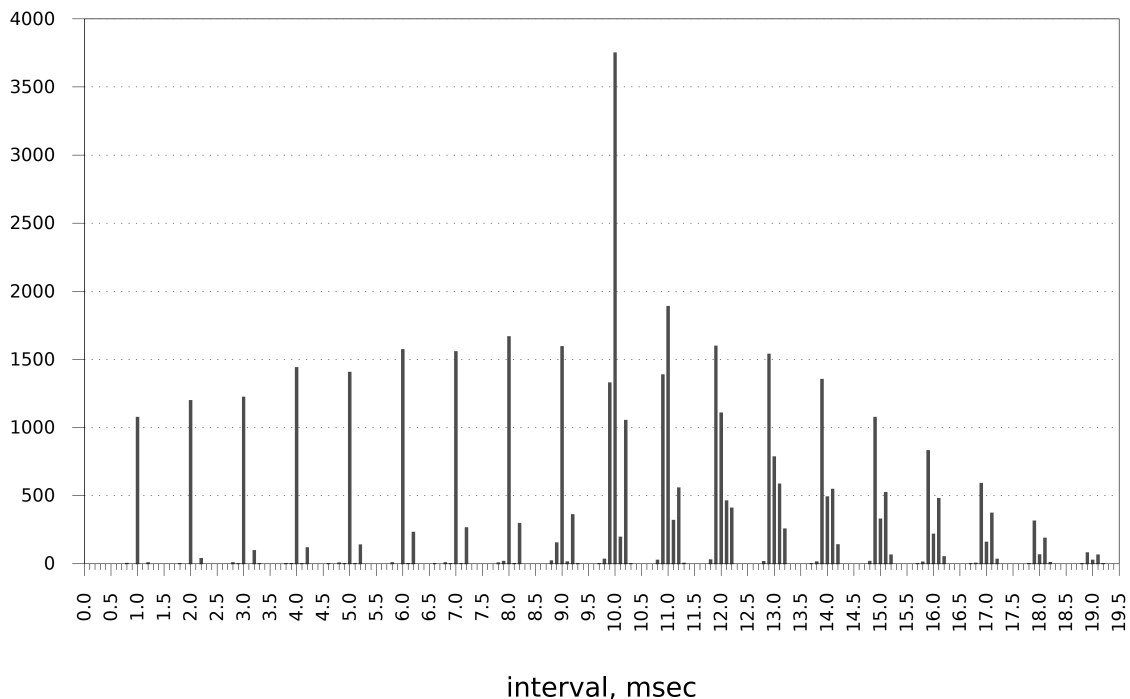

Figure 1. Distribution of time between CPU usage values increments for user, system, and nice modes

CPU usage increments is on X-axis, and number of increments that occurred at these intervals is shown on Y-axis. We see that most of CPU usage values updates happen at intervals that are

multiples of 1 millisecond, which is 1/`HZ` second (`HZ`=1000 for OS installed on our test machine). The peak at 10.0 milliseconds (1/100 th of a second) is caused by the fact that the data exported to user is rescaled to units of 1/100 ths of a second.

The load for the test was produced by two threads both bound to the same CPU. One thread was mostly iterating in an empty loop and sometimes performed `nanosleep` [13] system call with incorrect input parameters, thus creating a bit of system mode load. `nanosleep` system call delays the calling process for a time measured in nanoseconds, but in fact `nanosleep` resolution is rougher than nanosecond. Argument to `nanosleep` is a structure with separate members for seconds and nanoseconds to sleep, so when the nanoseconds member is set to a value higher than 10^9, the value is incorrect and the call returns immediately, quickly switching to system mode and back. The second thread was mostly performing `nanosleep` call with incorrect parameter thus creating mostly system mode load and sometimes making some iterations in an empty loop. This mix of user mode and system mode load on single CPU made system update values for user, and system modes happen frequently.

When we changed the test to look for intervals between increments of CPU usage data for all modes (thus including idle, iowait, irq, softirq and virtualization-related modes into consideration), the results changed, see fig. 2. CPU usage data increments happen mostly at intervals that are multiples of 0.2 millisecond, which definitely can't happen only on timer interrupts as they are known to happen at intervals of 1 millisecond.

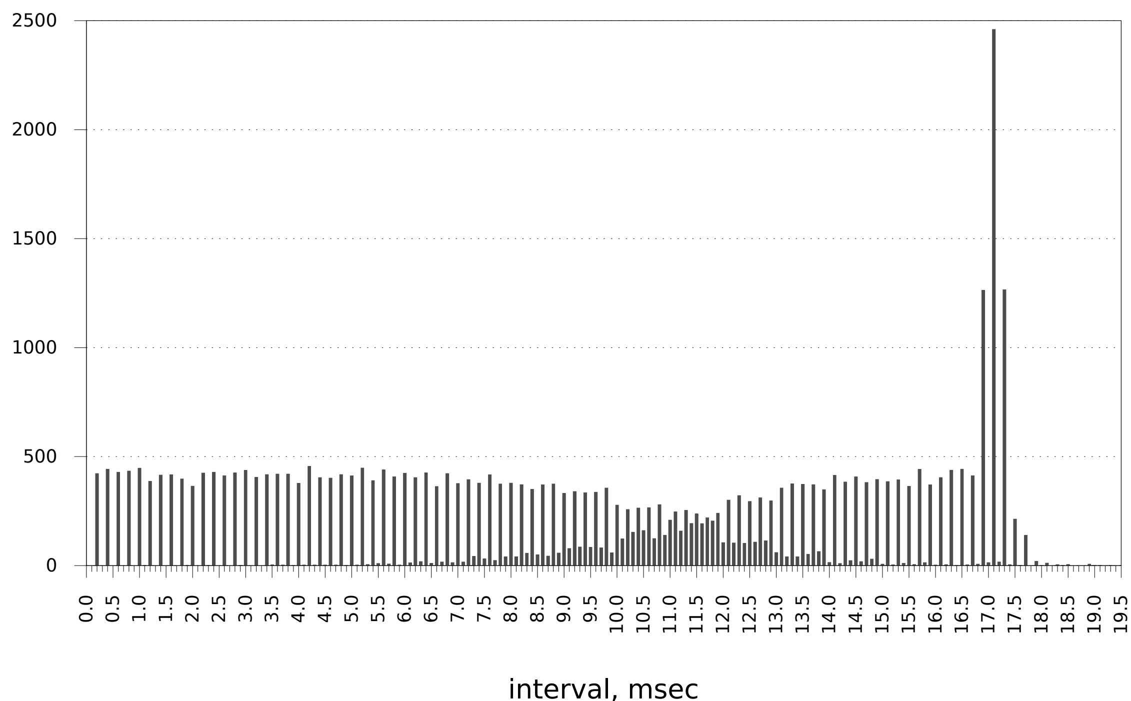

Figure 2. Distribution of time between CPU usage values increments for all modes

The load for this test was produced by a thread with mixed user mode load and idle state. The thread performed an empty loop (for user mode load) and made `nanosleep` call to introduce some idle time for the CPU to allow idle CPU usage values to be incremented.

We can't explain why the intervals between CPU usage value increments are the multiples of 0.2 milliseconds. We propose that this value is somehow connected with hardware timer resolution, but this requires further research.

3.2. Estimating the accuracy of CPU usage data

Our second experiment tries to estimate the error introduced by the fact that CPU state (for user and system mode) is examined only when timer interrupt occurs and no changes between interrupts are accounted for. The test pseudocode is shown in fig. 3

```
for (int j = 0; j < 10000000; ++j)
  nanosleep(&delay, NULL);
```

Figure 3. Test pseudocode

The test is just a `nanosleep` call in a loop. We use different values for the delay. To have zero-length delay we use an incorrect value so `nanosleep` returns immediately. Thus we can measure the time needed for performing all the work except sleep itself. The results are shown in fig. 4 and fig. 5, and the data with some uninteresting points omitted are given in tab. 1.

Run time for different delays is shown in fig. 4. The ‘Real’ line is the total run time for the test measured with the `time` [14] Linux utility. The ‘Estimated’ line is the requested delay length multiplied by the number of iterations. Both lines are parallel for delays greater than 200 microseconds, so we may assume that the real delay introduced by `nanosleep` is quite accurate for such delays. For delays less than 200 microseconds the real delay seems to be less accurate, but still it is approximately equal to the requested delay.

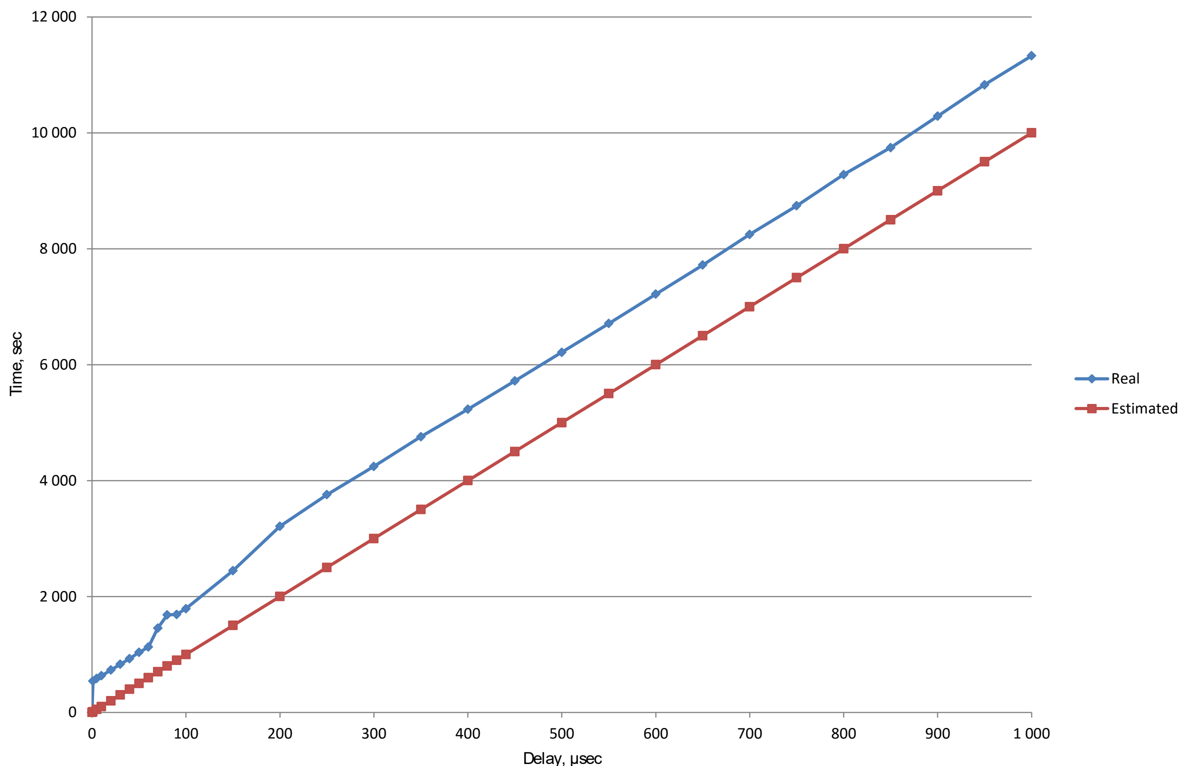

Figure 4. Real and estimated runtime for the delay test

But the values measured for the times the test program spent in user and system mode during execution are not so accurate (see fig. 5). We expected beforehand that the values for user and system time will be the same as in a no-delay run (0.29 and 0.32 seconds, respectively), as user mode and system mode work done by the processor seem to be independent of the delay

value. But the results are not so trivial. For the delays greater than 200 microseconds user time is approximately constant, but for the lesser delays the values are a bit chaotic. For system time the results are even more strange. System time is approximately constant for the delays in range from 200 to 700 microseconds, but has unpredictable values outside that range.

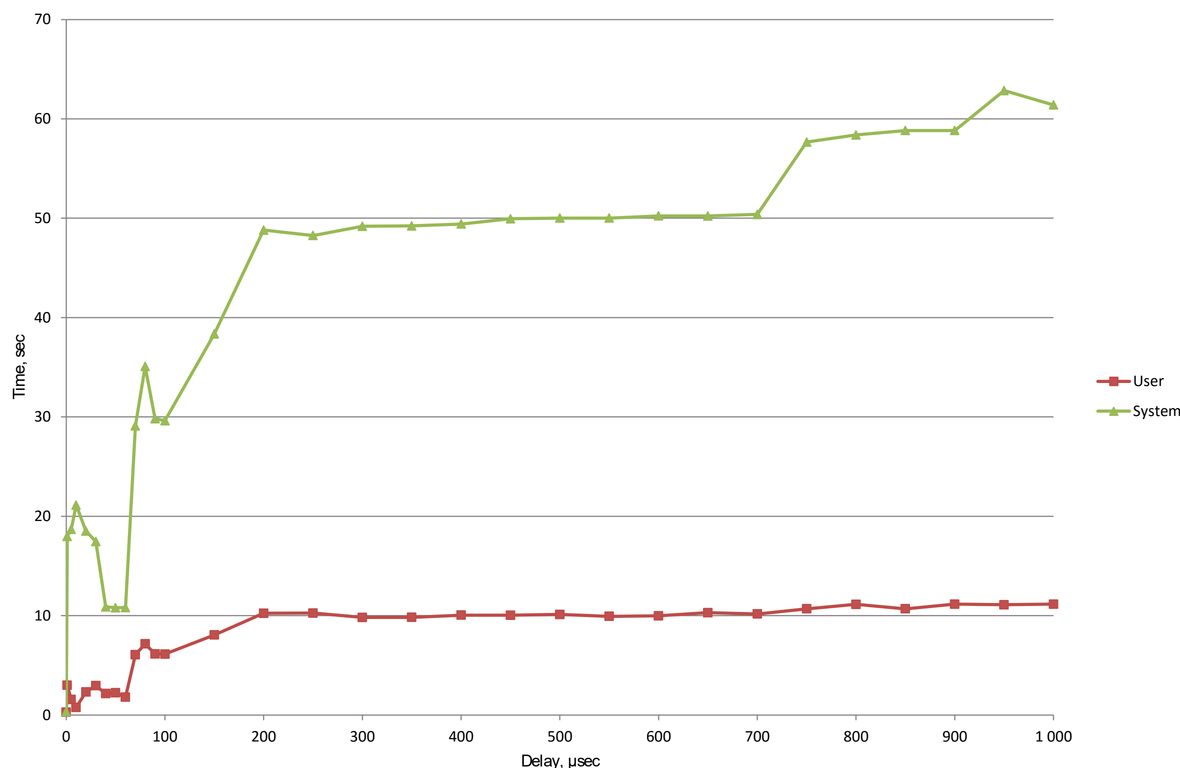

Figure 5. User and system mode times for the delay test

The only explanation we see is that some of the delays are somehow synchronized with timer interrupts, and as check for the CPU mode is done on timer interrupts, the results for user and system time depend on the fraction of delays which overlap with timer interrupts.

For example, if we compare a no delay run and a run with 90-microseconds delay, the same loop with the same system call is accounted 30 times more for user time when inserting a real sleep, more than 7 times for system time. For this example we chose a delay value to have the overaccounting effect high. But how high is the probability that such synchronization occurs in real world applications? Of course this question demands further research, but at least scheduling events are done in timer interrupt handler, and it is known that in networks seemingly unconnected independent events tend to synchronize [15]. As HPC applications use network for communication, we may expect similar effect as well.

Conclusion and Future Work

We analyzed CPU usage data provided by the Linux kernel and how CPU load level is calculated based on these data. The result is that to have the precision of CPU load level percentage of 1 decimal place (when CPU load level is measured in percent of full load) one should sample CPU usage data no more frequently than once every 10 seconds. CPU usage data are not continuously updated, they are updated on timer interrupt which occurs HZ (common values are 250 or 1000) times per second. When calculating accounting data for user, system,

Table 1. The results of the delay test

Delay, μsec	Run time, sec	User time, sec	System time, sec
0	0.62	0.29	0.32
1	541.05	2.99	17.98
5	586.24	1.55	18.69
10	631.19	0.77	21.13
20	730.32	2.31	18.51
30	831.06	2.96	17.46
40	927.86	2.15	10.88
50	1036.01	2.23	10.8
60	1128.32	1.8	10.81
70	1454.09	6.07	29.1
80	1681.41	7.16	35.08
90	1690.68	6.15	29.82
100	1789.82	6.13	29.62
150	2446.33	8.05	38.35
200	3211.62	10.24	48.81
700	8246.95	10.17	50.38
750	8741.05	10.69	57.66
1000	11329.95	11.16	61.4

and nice modes, only the state of the system at the moment of the interrupt is examined, no changes in between the interrupts are accounted. Our experiments show that the same amount of work may be measured very differently when delays are introduced between work periods.

Our future task is to run more elaborate tests and find real application examples when delays between periods of work (calculations) affect the accuracy of CPU load level measurements. We think that it will be especially interesting if the delays are done by waiting for communications which is quite a common case for HPC applications.

The reported study was supported by the RFBR research project No. 16-07-01121.

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. Dongarra JJ, Moler CB, Bunch JR, Stewart GW. LINPACK User's guide. Society for Industrial and Applied Mathematics; 1979. Available from: `http://epubs.siam.org/doi/book/10.1137/1.9781611971811`.

2. Dongarra J, Heroux MA, Luszczek P. HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems. Knoxville, Tennessee: lectrical Engineering and Computer Sciente Department, Knoxville, Tennessee; 2015. Available from: `http://www.eecs.utk.edu/resources/library/file/1047/ut-eecs-15-736.pdf`.

3. Kluge M, Hartung M. Mapping of RAID Controller Performance Data to the Job History on Large Computing Systems. In: 2014 International Workshop on Data Intensive Scalable Computing Systems. New Orleans, Louisiana, USA; 2014. p. 73–80. Available from: http://conferences.computer.org/discs/2014/papers/7038a073.pdf.

4. Sottile MJ, Minnich RG. Supermon: a high-speed cluster monitoring system. In: Proceedings. IEEE International Conference on Cluster Computing. IEEE Comput. Soc; 2002. p. 39–46. Available from: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1137727.

5. proc(5) - process information pseudo-file system;. Available from: http://linux.die.net/man/5/proc.

6. Korn W, Teller PJ, Castillo G. Just how accurate are performance counters? In: Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (Cat. No.01CH37210). IEEE; 2001. p. 303–310. Available from: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=918667.

7. Weaver VM, McKee SA. Can hardware performance counters be trusted? In: 2008 IEEE International Symposium on Workload Characterization, IISWC'08. vol. 08; 2008. p. 141–150.

8. Weaver V, Dongarra J. Can hardware performance counters produce expected, deterministic results. Proceedings of Third Workshop on Functionality of Hardware Performance Monitoring. 2010;Available from: http://icl.cs.utk.edu/news_pub/submissions/fhpm2010_weaver.pdf.

9. Weaver VM, Terpstra D, Moore S. Nondeterminism and Overcount in Hardware Counter Implementations. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, TX: IEEE; 2013. p. 215–224. Available from: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6557172.

10. Moore SV. A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware. In: Computational Science ICCS 2002. Springer Berlin Heidelberg; 2002. p. 904–912. Available from: http://link.springer.com/10.1007/3-540-46080-2_95.

11. Smythies D. Linux reported load averages, for example from top and uptime commands, can be incorrect; 2012. Available from: http://www.smythies.com/~doug/network/load_average/.

12. NO_HZ: Reducing Scheduling-Clock Ticks;. Available from: https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.

13. nanosleep(2): high-resolution sleep;. Available from: https://linux.die.net/man/2/nanosleep.

14. time(1) - time a simple command or give resource usage;. Available from: https://linux.die.net/man/1/time.

15. Floyd S, Jacobson V. The synchronization of periodic routing messages. IEEE/ACM Transactions on Networking. 1994 apr;2(2):122–136. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=298431.