Study of CPU Usage Data Properties for Using in Performance Monitoring[*]

Konstantin Stefanov[1], Alexey Gradskov[2]

Research Computing Center Lomonosov Moscow State Unversity[1],
Computational Mathematics and Cybernetics faculty of
Lomonosov Moscow State University[2]

CPU usage data (CPU user, system, iowait etc. load level) are often used for performance monitoring. These data are provided by OS. In this paper we analyze different aspects of CPU usage data given by Linux kernel. We examine 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.

1. Introduction

Today supercomputers show very impressive performance for their peak values and with some benchmarks like LINPACK [1]. Other benchmarks like HPCG [2] or the benchmark used for ranking in Graph500 list [3] 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 to evaluate applications while they are running and determine the obstacles to higher sustained performance of the applications. The idea of performance monitoring is to collect metrics which describe the state of the running application. These data are collected for all compute nodes running the given application. Some performance monitoring approaches [4] 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 operating system like CPU usage data, Load Average, memory usage etc. Some sensors may be somewhere on border between software and hardware like InfiniBand interface counters, which are maintained by InfiniBand card firmware.

There's a question on the properties of those sensors and their suitability for performance monitoring in different modes. Of course all those sensors are long and widely used for performance monitoring and give useful data which lead to useful results in application analysis. But as performance monitoring systems evolve we may meet some limitations of those sensors which may lead to their unsuitability for new approaches or new modes of usage. For example SuperMon [5] can achieve up to 6000Hz sampling rate while reading Linux kernel data from /proc 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, papers [6–9] analyzed hardware performance counters from the point of accuracy, predictability, reproducibility etc. Paper [10] compares two modes of using performance counters and compares their results. 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.

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

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 2 describes how the CPU usage data in their conventional form are obtained from the kernel. Section 3 gives the results of analysis of Linux kernel source code in parts which relate to CPU usage data. Section 4 gives results of testing supporting the results which were given in section 3. Section 5 contains the conclusion.

2. How CPU usage data are obtained

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 1/100ths of a second, that the system spent in different modes of execution [12] since boot. These different modes are: user mode (running user processes), user mode with low priority (nice), system mode (running kernel), idle, iowait (waiting for IO to complete), irq (processing interrupts), softirq (processing softirqs) and some other modes related to virtualization. To obtain CPU usage as 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 on 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 CPU usage L_m for mode *m* is

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

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

One consequence of such method of calculations 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 1/100th of a second; to have real precision of 1/10th of percent for CPU load level value we should take the measurements more rare than once in 10 seconds. With more frequent sampling the precision of CPU load level will be less than 1 decimal place.

3. Source code analysis results

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

These per-CPU values (and per-process values, 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 timer interrupt handler (generally it is executed on every CPU with some exceptions described later), the kernel finds a 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 1/100ths of a second. Such rescaling may introduce some rounding errors but it is not expected to be high.

When NO_HZ kernel compilation parameter is active (true for modern 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 such method of calculating is that the CPU usage times 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.

4. Experiments

Our first experiment was designed to prove that CPU usage times 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. Relative deviation of period between CPU usage increments is on X-axis, and number of increments that occurred after these periods shown on Y-axis. We see an obvious peak which indicates that there is some fixed time period between successive increments.

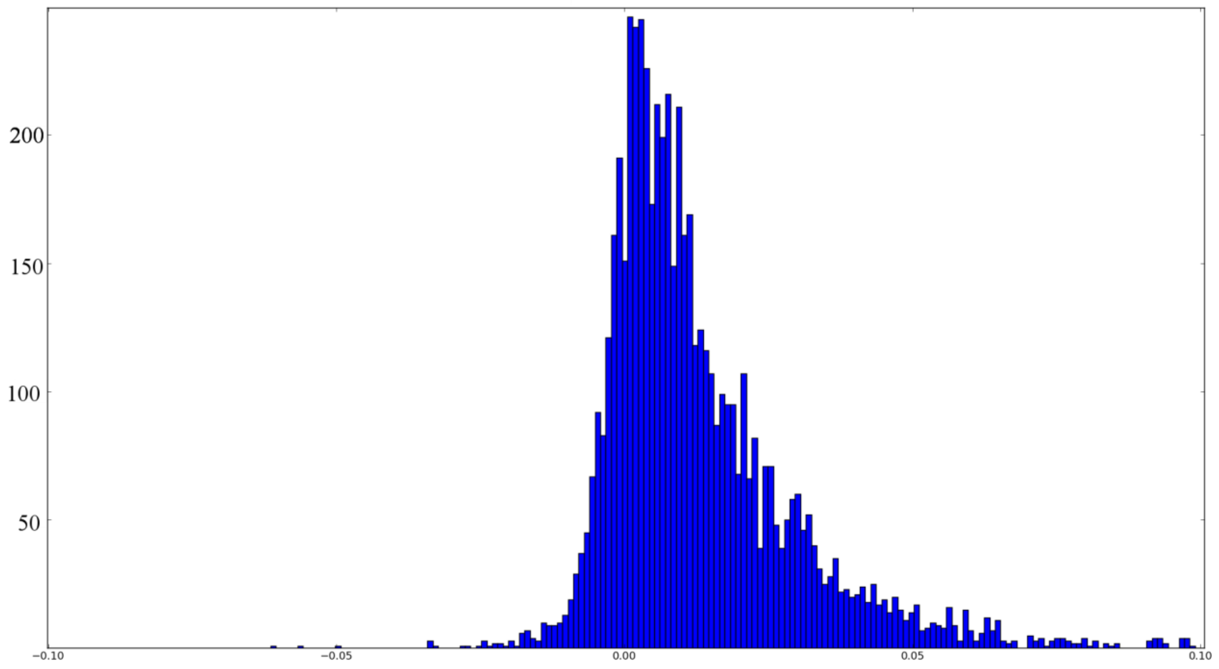

Fig. 1. Distribution of time between CPU usage increments

Our second experiment tries to estimate the error introduced by the fact that the states of CPUs are only examined when timer interrupt occurs and no changes between interrupts are accounted. The test pseudocode is shown in fig. 2

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

Fig. 2. Test pseudocode

In fact the test is just a nanosleep call in a loop. Nanosleep system call delays the calling process for a time measured in nanoseconds, but in fact nanosleep resolution is rougher than single nanoseconds. We use two runs with different values for the delay. The first value is an incorrect value (argument to nanosleep is a structure with separate members for seconds and nanoseconds to sleep, so when nanoseconds member is set to a values higher than 10^9, the value is incorrect). When nanosleep receives an incorrect argument, it returns immediately. We thus may measure the time needed for performing all the work except sleep itself. The second run sets the delay to 90 000 nanoseconds. With ideal nanosecond-resolution timer the program should execute a little longer than 900 seconds. But in real life nanosleep round the delay up and synchronizes wakeups with timer interrupts. The testing was performed on a compute node with no other running tasks to eliminate the influence of other tasks, cache pollution etc. The results are given in Table 1.

Table 1. The results of nanosleep testing

	No sleep (incorrect delay)	**Sleep 90 microseconds**
Overall run time	0.6 s	1623 s
User time	0.2 s	6.3 s
System time	0.4 s	30.2 s

As we see, 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. The most feasible explanation we see is that the wakeup moment is synchronized with the timer interrupt so when the timer interrupt occurs, the process is much more likely to be in running state than it should be according to time it really spends running.

We choose delay value to have the over accounting effect high. But how high is the probability that such synchronization occurs in real world applications? Of course this demand 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 [13]. As HPC applications use network for communication we may expect similar effect as well.

So our experiments indicate that CPU usage values are indeed update once every 1/HZ seconds and that the values accounted may vary for the same amount of work.

5. Conclusion and Future Work

We analyzed CPU usage data provided by Linux kernel and how CPU load level is calculated based on those data. The result is that to have the precision of CPU load level of 1 decimal place (when CPU load level is measured in percent of full load) one should sample CPU usage data not 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, 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 accounted very differently when delays are introduced between work periods.

Our future work is to make more elaborate test and find the 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.

References

1. Dongarra J.J. et al. LINPACK User’s guide. Society for Industrial and Applied Mathematics, 1979. 344 p.
2. Dongarra J., Heroux M.A., Luszczek P. HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems. Knoxville, Tennessee, 2015.
3. Ang J.A. et al. Introducing the graph 500. 2010.
4. Kluge M., Hartung M. Mapping of RAID Controller Performance Data to the Job History on Large Computing Systems // 2014 International Workshop on Data Intensive Scalable Computing Systems. New Orleans, Louisiana, USA, 2014. P. 73–80.
5. Sottile M.J., Minnich R.G. Supermon: a high-speed cluster monitoring system // Proceedings. IEEE International Conference on Cluster Computing. IEEE Comput. Soc, 2002. P. 39–46.
6. Weaver V.M., McKee S.A. Can hardware performance counters be trusted? // 2008 IEEE International Symposium on Workload Characterization, IISWC’08. 2008. Vol. 08. P. 141–150.
7. Weaver V., Dongarra J. Can hardware performance counters produce expected, deterministic results // … on Functionality of Hardware Performance …. 2010.
8. Weaver V.M., Terpstra D., Moore S. Nondeterminism and Overcount in Hardware Counter

Implementations // 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, TX: IEEE, 2013. P. 215–224.

9. Korn W., Teller P.J., Castillo G. Just how accurate are performance counters? // Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (Cat. No.01CH37210). IEEE, 2001. P. 303–310.
10. Moore S. V. A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware // Computational Science — ICCS 2002. Springer Berlin Heidelberg, 2002. P. 904–912.
11. Smythies D. Linux reported load averages, for example from top and uptime commands, can be incorrect [Electronic resource]. 2012. URL: http://www.smythies.com/~doug/network/load_average/ (accessed: 14.06.2016).
12. proc(5) - Linux man page [Electronic resource]. URL: http://linux.die.net/man/5/proc.
13. Floyd S., Jacobson V. The synchronization of periodic routing messages // IEEE/ACM Transactions on Networking. IEEE, 1994. Vol. 2, № 2. P. 122–136.