Execution Time Compensation for Cloud Applications by Subtracting Steal Time based on Host-Level Sampling

Masao Yamamoto, Kohta Nakashima
Computer Systems Laboratory
Fujitsu Laboratories Ltd., Kawasaki, Japan
{masao.yamamoto, nakashima.kouta}@jp.fujitsu.com

ABSTRACT

Accurate measurement of program execution time is indispensable to time-based charge systems and performance debugging in all computer systems. However, cloud application execution time cannot be measured properly because measurement in a virtual machine (VM) includes additional time called *steal time*. The steal time of each program in a VM is unrecognizable by existing standard operating system (OS) tools. Therefore, it is quite difficult for performance engineers to grasp the accurate execution time of each program in a VM.

In this ongoing work, we show the novel point of steal in the broad sense and describe how to compensate for function-level execution time in each program in a VM. Our novel approach works by subtracting steal time, which is based on the time-series data of host-level sampling in each function. We implement our approach as a host-level kernel module based on hardware performance counters and some user-level analysis programs. Therefore, our method requires no modification of user applications, guest OSes, a virtual machine monitor (VMM) or a host OS. Finally, our results demonstrate accurate execution time of a function-level guest program, with an overhead lower than 1% for practical use.

CCS Concepts

•**Software and its engineering** → **Virtual machines;** *Software performance;* •**General and reference** → *Measurement; Performance;*

Keywords

Virtual machines; Cloud; Execution Time; Steal time; Performance; Measurement; Sampling; Analysis

ICPE'16 Companion, March 12 - 18, 2016, Delft, Netherlands

ISBN 978-1-4503-4147-9/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2859889.2859899

1. INTRODUCTION

Accurate cloud application execution time is required for time-based meter rate charge systems and for performance debugging of guest programs. It provides opportunities for cloud service providers to charge based on actual CPU usage rather than instance hours, which are CPU elapsed time. Accordingly, the costs of cloud service usage can be reduced by paying precisely for the instance being used. Besides, accurate execution time of guest programs can also be used to determine which part has the cause of performance degradation; a proprietary program, other virtual machines (VMs) or a virtual machine monitor (VMM). Thus, accurate measurement of program execution time is indispensable in a virtualized environment, which is the underlying infrastructure in cloud computing.

However, the precise execution time of each program cannot be obtained in a virtualized environment because guest-inside measurement includes steal time as additional time. Steal time is a characteristic issue in a virtualized environment. It is the duration in which a guest is unable to run because physical CPUs cannot be assigned to the guest. That is, it is the time spent in involuntary wait by a virtual CPU (vCPU) while a VMM is servicing another vCPU. Steal is caused by the operation of a VMM. The cause of steal in a guest exists outside the guest. It is in other guests or a VMM. Moreover, although recent VMMs expose the steal time of a VM to the guest OS, the steal time of each program is still unrecognizable using existing standard OS tools, application programming interfaces and VMM interfaces. Guest OSes are unaware of being in virtualized environments. They are implicitly convinced that they occupy the underlying physical resources. Consequently, accurate execution time excluding steal time cannot be obtained in a virtualized environment, and it is quite difficult for performance engineers to measure the actual execution time of each program in a VM.

In this paper, we reveal steal time even in the case of no CPU oversubscription. Such steal time is unrecognizable, even through a VMM interface. We propose a solution that compensates for the function-level execution time of each program in a guest. Our solution subtracts steal time, which is included, from each function based on the time-series data of host-level sampling. This ongoing research draws on our earlier work [1, 2], *Unified Performance Profiling of an Entire Virtualized Environment*, for the host-level sampling. Finally, we implement our approach, and our results demonstrate accurate function-level execution time of a guest program, excluding steal time, with a low overhead

of 1% or less for practical use.

The main contributions of this paper are as follows:

1. We generalize steal in the broad sense and analyze unrecognizable steal, even through a VMM interface.
2. We present a novel method of attributing steal data sampled in the host to corresponding processes, threads and functions in the guest. It requires no modification of user applications, guest OSes, a VMM or a host OS.
3. We present a preliminary evaluation. We demonstrate accuracy and feasibility. Then, we show negligible overhead using real applications on a more recent platform than that used in previous work [1, 2].

2. STEAL IN THE BROAD SENSE

Some guest OSes can show the steal usage of an entire VM or vCPU using standard statistical tools, such as sysstat. This is because a VMM exposes the information of steal usage to guest OSes in each vCPU. In *the Linux manpages proc(5)* [3], steal time is defined as the time spent in other operating systems when running in a virtualized environment. However, the blank time in a VM includes not only the time spent in other VMs, but also the time spent in a VMM. Accordingly, from the user point of view, we propose referring to the former as "*steal in the narrow sense*" and both as "*steal in the broad sense*". In addition, the former is also referred to as type-1 and the latter type-2. We cannot capture type-2 steal with an existing standard tool, as demonstrated below.

Table 1: Summary of experimental environment

Host Environment	
CPU	Intel Xeon E5-2697 v3, 2.60GHz
Num of pCPU	14
Memory	128GB
OS	RHEL Server 7.1[1] 64bit (kernel 3.10.0-229.el7.x86_64)
VMM	KVM [4] (qemu-kvm-1.5.3-86.el7.x86_64 [5])
Guest Environment	
Num of vCPU	2
Memory	8GB
OS	RHEL Server 7.1[1] 64bit (kernel 3.10.0-229.el7.x86_64)

The experimental environment is summarized in Table 1. The host machine is a Fujitsu Primergy CX400 M1 server. We enabled only one socket and disabled the CPU Hyper-Threading, Enhanced SpeedStep and Turbo Boost features to obtain more stable results. We used PostgreSQL 9.2.7-1 [6] for evaluation of type-2 steal as unrecognizable steal. PostgreSQL is a widely used open-source relational database management system. It is a popular database for Web applications. Moreover, we used SysBench 0.4.12-12 [7] for measurement of PostgreSQL's OLTP (online transaction processing) performance. In this section, we used only one VM. No other VM was booted. We had two vCPUs available and pinned each vCPU to the specified physical CPU (pCPU). Furthermore, PostgreSQL processes were bonded to vCPU0, and a SysBench process was bonded to vCPU1. We simultaneously invoked 16 OLTP threads to make CPU usage almost 100%.

[1]Red Hat Enterprise Linux Server release 7.1

Table 2: Unified VM profiling

Total samples:29932

Samples	%Ratio	Function	Module
2366	7.90	[steal]	(outside)
1098	3.67	rb_iterate	postgres
855	2.86	ip6t_do_table	ip6_tables
655	2.19	hash_search_with_hash_value	postgres
575	1.92	SearchCatCache	postgres
532	1.78	PostgresMain	postgres
(snip)			
9	0.03	[idle]	(halt_exit)
(snip)			

The average results of the ***mpstat*** command on vCPU0 showed that **%usr was 53.27, %sys 23.33, %soft 23.36, %idle 0.03 and %steal 0.00**. In contrast, we found the blank time for type-2 %steal on vCPU0 to be 7.90 from the result of unified VM profiling [1, 2], as shown in Table 2. From the analysis of the VM-exit reason number, it can be seen that the causes of these steal cycles include the execution of the WRMSR instruction in the guest (exit reason number 32) with 90.25%, external interrupts (exit reason number 1) with 9.62%, EPT violation (exit reason number 48) with 0.08% and execution of I/O instruction in the guest (exit reason number 30) with 0.04%. In other data, we can also confirm other causes of type-2 steal, such as the execution of the CPUID instruction in the guest (exit reason number 10), the execution of the PAUSE instruction in the guest (exit reason number 40) and EPT misconfiguration (exit reason number 49). From these results, we found that the impact of steal could occur without other VMs running on the same host. In addition, we found that such type-2 steal is unrecognizable with existing standard OS tools, even with steal information from a VMM.

3. COMPENSATION OF EXECUTION TIME

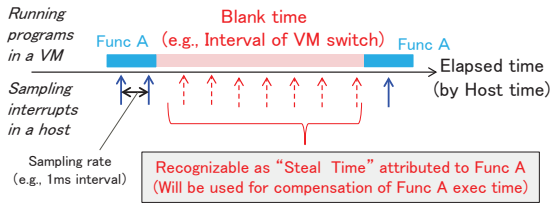


Figure 1: An executing function and a steal time section in a VM with time-series samplings in a host

Approach: In our approach, we assume two things. One is that sampling rate is less than an interval of VM switch, as shown in Figure 1. For instance, an interval of VM switch is 20 milliseconds, and a sampling rate is 1 millisecond. The other is that a running program context is the same one across a steal time section. Thereby, we can recognize such steal time is attributed to the context and use it for compensation of execution time of the running program. The context includes processes, threads or functions. For instance, in Figure 1, the running function is the same "Func A" before and after the steal time which is also known as blank time in a VM. Then, we can recognize the steal time as that attributed to "Func A", shown in Figure 1. Thus, the apparent

execution time of "Func A" becomes larger than the actual execution time. The guest OS is unaware of the occurrence of steal time, which is additional CPU time to "Func A" of the executing program. This is because the control of the VM is stolen involuntarily by a VMM from the guest OS when the steal occurs. Incidentally, when the context is different from each other across a steal time section, we don't use such steal time for compensation of execution time. This is because we cannot identify the context which such steal time is attributed to. Our techniques, based on these two assumptions, are that:

- Firstly, we extract steal samples attributed to the target context from guest-level virtual sampling data.
- Secondly, we sum up steal time of the target context using above-mentioned target steal sampling data.
- Finally, we get accurate execution time by subtracting the aggregation steal time of the target context from measured apparent execution time of the context.

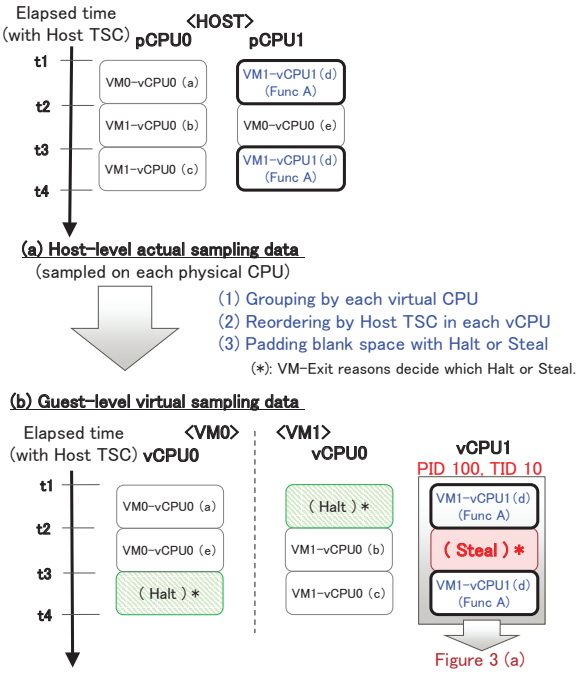


Figure 2: Converting host-level real sampling data to guest-level virtual sampling data based on host time

Generating guest-level sampling data from host-level sampling data: Each guest has a blank time in which it is unable to run because a physical CPU cannot be assigned to a guest. We believe that the blank time should be interpolated with host time as a single common time, and we chose the physical TSC value as host time. As a result, host-level sampling results reflect the actual time even to guests, including the blank time. To interpolate the blank time in each guest, the compensated sampling data of each guest should be generated. Figure 2 illustrates how to build the compensated virtual guest sampling data.

Figure 2 assumes an environment in which the host has two pCPUs and two VMs running: VM0 and VM1. VM0 has one vCPU, and the VM1 has two vCPUs. No vCPU is bound to a pCPU. The time axis represents elapsed time by host time. This figure shows sampling data from t1 to t4. The virtual guest sampling data is generated with the host time by the virtual CPU unit from real host-level sampling data. The above figure displays the sampling format held in the kernel buffer. We can convert this format into the format below as virtual guest sampling data. When doing so, the host time is used as the base common time, thereby correctly reflecting the blank time of each vCPU. Blank time is an interval that does not include valid periodic sampling data corresponding to the vCPU. Blank time can be classified into two states, halt or steal, by the VM-Exit reason. If the exit reason number is 12, blank time is recognizable as halt. Otherwise, blank time is handled as steal.

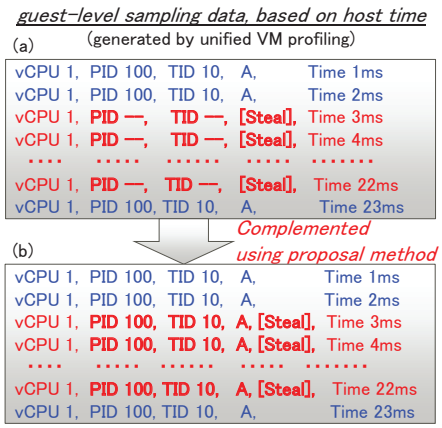


Figure 3: Attributing steal samples to the corresponding context

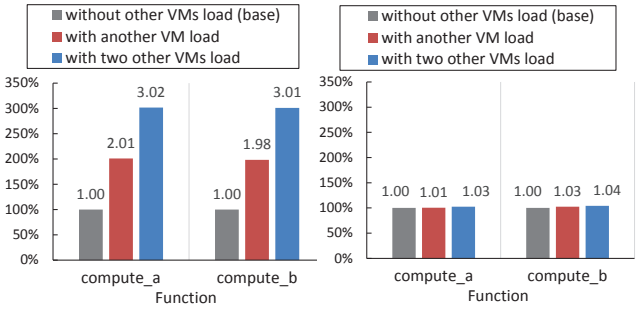
Extracting steal samples and attributing them to the corresponding processes, threads, and functions: In Figure 3, a line represents one sample data in the time-series sampling. Figure 3a illustrates our input data which are guest-level virtual sampling data obtained from the results of unified VM profiler. Those data are used for extracting the target context steal. Figure 3b represents complemented steal sampling data with the corresponding context information those are processes ID, thread ID and function name.

4. EVALUATION

4.1 Results

The experimental environment is the same as shown in Table 1 in §2. In this subsection, we boot three VMs and pinned all VM's vCPU0 to pCPU0. This way, each vCPU0 is overcommitted on pCPU0. Each VM's vCPU1 is pinned to a pCPU other than pCPU0. We verify the accuracy of our approach using a computation-intensive workload, which performs floating point arithmetic. It consists of two functions, compute_a and compute_b, which consume 80 and 20% of CPU cycles, respectively. Consequently, total CPU usage is almost 100% with this workload. In one VM, referred to as an execution time measurement VM, we ran this workload with a fixed number of iterations. In other VMs, as CPU load generator VMs, we continuously ran that. We mea-

sured the wall-clock execution time of each function in an execution time measurement VM. We simultaneously took samples every 1 millisecond in a host. A Hardware performance counter triggered the sampling through the counter overflow interrupt after a specified number of event counts. At every interrupt, sampling data were recorded into the host kernel buffer in turn without aggregation [1, 2].

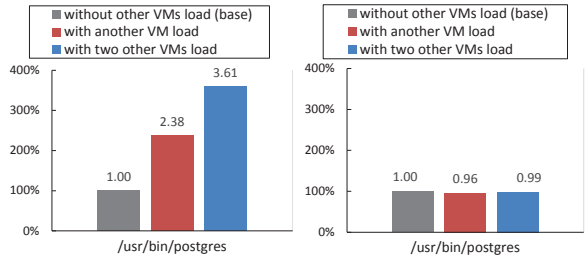


(a) apparent time including steal time

(b) compensated time excluding steal time

Figure 4: Comparison of normalized execution time

Figure 4a shows the results of apparent execution time of each function. They are found to be twice or three times than criteria of each function execution time. This indicates that the apparent execution time increases in proportion to the CPU load generated by other VMs. This is because the workload is CPU-bound. By contrast, Figure 4b shows the results of compensated execution time. These corrected execution time are found to be close to the criteria. From the results, the accuracy of our approach can be confirmed. Furthermore, the effect of excluding steal time can be confirmed by comparison between Figure 4a and 4b.



(a) apparent time including steal time

(b) compensated time excluding steal time

Figure 5: Comparison of normalized total execution time of multiple processes of PostgreSQL in a VM

In addition, we conducted preliminary study for applying our method to real applications. An environment is the same as explained in this section. We used the number of sampling of PostgreSQL for this study. First, we continuously ran the aforementioned workload in a load generator VM. Next, in an execution time measurement VM, we invoked 16 OLTP threads in the same way in §2. Then, we performed host-level sampling every 1 millisecond for 60 seconds during OLTP performance measurement. As a result, for example, we obtained 213.34 seconds as an apparent execution time and captured 35788 steal samples on vCPU0 for 60 seconds. This number of steal samples can be converted to 127.25 seconds for total period of the measurement. Corrected execution time becomes 86.09 seconds by subtracting steal time from apparent time. Reference execution time was 89.51 seconds without other VMs load. Thus, we obtained above results, as shown in Figure 5. It can be seen from Figure 5 that our approach can be used to compensate real application execution time. On the other hand, an average result of %steal by mpstat was 56.27%. Using this result, corrected execution time becomes 94.01 seconds. Our result is closer to the reference execution time. This difference between the result of our approach and that of mpstat is assumed to be caused by type-2 steal.

4.2 Overhead

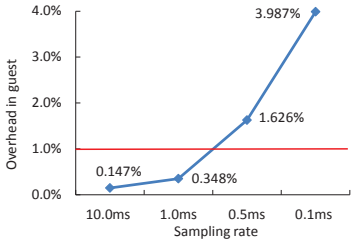


Figure 6: Overhead of our approach against execution time of PostgreSQL by 16 OLTP threads

We determined the overhead by comparing the execution time, with and without host-level sampling, of 16 OLTP threads invoked in the same way in §2. Figure 6 shows the overhead results (blue curve) and the allowable borderline (red line) of our sampling overhead for practical use even in performance-critical cases. From the results, we found that, with host-level sampling, we could use 1 millisecond as the practical sampling rate, even in virtualized environments.

5. RELATED WORK

The issue of steal time has been addressed in previous work. Weaver et al. extended PAPI to provide the support for running inside VMs [8]. This extension enables developers to use PAPI to carry out performance analysis in virtualized cloud environments. It relies on Linux and KVM/Xen, which support steal time reporting. It is a sort of paravirtualized function that provides information from the VMM on how often a VM was scheduled out. In PAPI 5, this function is used to implement compensation for steal time in the PAPI_get_virt_usec routine under KVM. However, this function only provides system-wide steal time values. PAPI only returns per-process results. Therefore, it is difficult for PAPI to automatically and completely adjust process time measurements in guests. Regarding this problem, Weaver et al. stated that *steal time is only an issue if a machine is oversubscribed with VMs; in most HPC situations, only one task is running per node, so this might not be a critical limitation.* On the other hand, in more general virtualized environments, this becomes a critical limitation. In most general cloud-service situations, VMs and vCPUs are overcommitted to physical resources. In addition, as was shown in §2, steal time occurs even if VMs and vCPUs are not overcommitted. Therefore, this work focuses on resolving the steal time issue.

Hofer et al. broke down the steal time provided by the VMM to the monitored application threads [9]. They base their analysis on the fact that threads that use the most CPU time (or to which the most CPU time is allocated) are the ones most affected by the steal of vCPUs. Hence, they divide the steal time among the threads in proportion to the CPU time they consumed. Actually, they implemented their approach in a Java VM. Therefore, their approach can handle only Java applications in a VM. Furthermore, their approach depends on the steal information provided from a VMM. Hence, their approach cannot include type-2 steal time.

6. CONCLUSIONS AND FUTURE WORK

Accurate execution time of cloud applications is required. However, the actual execution time of each program cannot be obtained in a virtualized environment because of steal time as additional time. This is a specific issue that must be tackled in virtualized environments. In this paper, we revealed steal time even when there is no overcommitted CPU and demonstrated such steal time is unrecognizable even through VMM interfaces. We then proposed a novel method for compensation of execution time of cloud applications by subtracting steal time based on host-level sampling. We developed prototype implementation and demonstrated that it works as well as expected. That is, its results reflected the actual execution time even in guest programs.

In subsequent steps, we will evaluate our method in each process and in each thread. We then plan to provide accuracy information more about real-world cloud applications, such as OLTP applications, Web applications and big data analyzers. Such experiments using real applications will help determine how our approach applies to meter rate charge systems. Furthermore, we plan to generalize this approach by extending our prototype to a general tool. A general tool is required for performance debugging of cloud applications as well as application performance management in the cloud. As a challenge for future work, we are considering applying this technique to anomaly detection in cloud environments, which have the characteristic steal issues of virtualized environments. For that purpose, enhancement with continuous sampling and automated online analysis is required.

7. REFERENCES

[1] M. Yamamoto, M. Ono, K. Nakashima, and A. Hirai. Unified performance profiling of an entire virtualized environment. In *CANDAR 2014 Second International Symposium on*, pp. 106–115, Dec 2014.

[2] Masao Yamamoto, Miyuki Ono, Kohta Nakashima, and Akira Hirai. Unified performance profiling of an entire virtualized environment. *International Journal of Networking and Computing*, Vol. 6, No. 1, pp. 124–147, 2016.

[3] Linux Programmer's Manual PROC(5). *http://man7.org/linux/man-pages/man5/proc.5.html*.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Linux Symposium*, 2007.

[5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pp. 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[6] PostgreSQL. *http://www.postgresql.org/*.

[7] SysBench. *https://github.com/akopytov/sysbench/*.

[8] V.M. Weaver, D. Terpstra, H. McCraw, M. Johnson, K. Kasichayanula, J. Ralph, J. Nelson, P. Mucci, T. Mohan, and S. Moore. Papi 5: Measuring power, energy, and the cloud. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 124–125, April 2013.

[9] Peter Hofer, Florian Hörschläger, and Hanspeter Mössenböck. Sampling-based steal time accounting under hardware virtualization. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pp. 87–90, New York, NY, USA, 2015. ACM.