Sampling-based Steal Time Accounting under Hardware Virtualization

Peter Hofer
Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems
Johannes Kepler University Linz, Austria
peter.hofer@jku.at

Florian Hörschläger
Johannes Kepler University Linz, Austria
florian.hoerschlaeger@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Virtualization enables the efficient sharing of hardware resources among multiple virtual machines (VMs). Because the physical resources are limited, the scheduler must often suspend one VM to allow some other VM to run. The operating system in a VM is typically unaware of the suspension and accounts periods of suspension as CPU time to the executing application thread. This misrepresentation of resource usage makes it difficult to tell whether a performance problem is caused by an actual bottleneck in the application or by the virtualization infrastructure.

We present a novel approach to compute to what degree the threads of an application in a virtual machine are affected by suspension. Our approach does not require any modifications to the operating system or to the virtualization software. It periodically samples the system-wide amount of "steal time" that is reported by the virtualization infrastructure, and divides it among the monitored threads according to their CPU usage. With a prototype implementation, we demonstrate that our approach accounts accurate amounts of steal time to application threads, that it can be used to compute the true resource usage of an application, and that it incurs only negligible performance overhead.

1. INTRODUCTION

Hardware virtualization is commonly used to efficiently utilize and share hardware resources. The resources of a physical machine (the *host*) are shared between several virtual machines (the *guests*). The creation and execution of virtual machines (VMs) is managed by a *hypervisor*. Each VM has its own operating system that runs in an isolated execution domain, which provides reliability and security. VMs can also be deployed and moved between sites with less effort than physical hardware.

ICPE '15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3248-4/15/01 ...$15.00.
http://dx.doi.org/10.1145/2668930.2695524.

The number of virtual CPUs of the guests can exceed the number of physical CPUs of the host. Therefore, the hypervisor cannot schedule all virtual CPUs on the available physical CPUs and must temporarily suspend some of them. Time periods when a virtual CPU is ready to execute, but is suspended, are referred to as *steal time.*

The amount of steal time is an important indicator for whether the assigned virtual resources are overprovisioned or whether the physical resources are insufficient. However, the operating system in the VM is often not aware of steal time or does not incorporate it into resource usage accounting because the concept of steal time does not exist with physical hardware. When a virtual CPU is suspended, the steal time is considered active CPU time and counted toward the resource usage of the currently executing thread. Therefore, it is difficult for a performance engineer to spot whether a performance problem is caused by an actual bottleneck in the application or by virtualization.

In this paper, we present an approach for how a performance analysis tool in an affected VM can estimate to what degree steal time affects individual application threads. Our approach relies on the hypervisor to provide the steal time for the entire VM. We periodically sample this steal time as well as the CPU usage of threads and then assign fractions of the steal time to them.

The main contributions of this work-in-progress paper are:

1. We describe a new technique to attribute VM-wide steal time to individual application threads. It does not require any changes to the hypervisor or to the operating system.

2. We present a preliminary evaluation that demonstrates that we are able to reliably separate hypervisor steal time from the actual CPU usage of application threads at negligible performance costs.

2. APPROACH

Hypervisors commonly provide an interface that allows the operating system or specific guest software in the VM to detect that they are running in a VM and to interact with the hypervisor, which can benefit the performance of the VM. A hypervisor interface typically also exposes steal time information. However, since the hypervisor has no knowledge of the processes and threads running within the VM, it accounts steal time only to virtual CPUs as a whole. On this level, steal time is only useful to performance analysis tools as an indicator for how the entire system is affected by virtual CPU suspension.

Our approach works from within the VM and breaks down the steal time provided by the hypervisor to the monitored application threads. We base our approach on the fact that those threads which use the most CPU time (or, to which the most CPU time is attributed) are also the ones that are most affected by the suspension of virtual CPUs. Hence, we divide the steal time among the threads in proportion to the CPU time they have consumed.

When a VM has multiple virtual CPUs, we would ideally determine which threads were executed on each virtual CPU and divide that CPU's steal time among these threads. Unfortunately, common operating systems do not make scheduling information available. However, bare-metal hypervisors often employ a form of co-scheduling in which all virtual CPUs of a VM are scheduled to run on physical CPUs at the same time, which avoids problems such as when one virtual CPU waits for some other virtual CPU that is currently suspended by the hypervisor [1]. With such scheduling, all virtual CPUs of a VM are similarly affected by hypervisor steal time. Other hypervisors simply rely on the host operating system's scheduler, and a fair scheduler should ensure that one ready virtual CPU of a VM is not suspended significantly longer than others. Therefore, we sum up the steal time from all virtual CPUs of the VM and divide it among all threads that consumed CPU time.

The extent of hypervisor suspension in a VM also depends on the load in other VMs and on the host and thus varies over time. To account for this fact, we use a sampling approach and periodically read or compute the following values to account the steal time since the last sample:

$\Delta t_{steal,total}$: The total steal time of all virtual CPUs since the previous sample.

$\Delta t_{cpu,total}$: The total apparent CPU time (including steal time) that was consumed by the VM since the previous sample.

$\Delta t_{cpu}(T)$: The apparent CPU time (including steal time) that was consumed by thread T since the last sample.

The hypervisor and the operating system typically make steal times and CPU times available as total times since the start of the VM or since the start of the thread. Computing the deltas between samples thus requires storing the values that were read in the previous sample. We then use the deltas between samples to divide the steal time among the monitored threads using the following equation for each thread T:

$$\Delta t_{steal}(T) = \Delta t_{steal,total} \frac{\Delta t_{cpu}(T)}{\Delta t_{cpu,total}}$$

The steal time of a process can be computed the same way when the operating system provides a per-process CPU usage that includes all threads. Alternatively, the steal time of each thread of the process can be computed individually and added up, but this can lead to inaccuracies when threads exit between samples.

Figure 1 shows an example of how our approach works in a schedule with three threads $T_{1,2,3}$ executing on three virtual processors $VCPU_{1,2,3}$. Initially, $VCPU_1$ is executing thread T_1, $VCPU_2$ is executing T_2, and $VCPU_3$ is idle. At $t_1 = 12$, we take a first sample. Because we do not have a previous sample, we cannot compute deltas at this point and do not account steal time to the threads. At $t = 14$, the hypervisor suspends both $VCPU_1$ and $VCPU_2$ until it resumes them at $t = 19$. At $t = 21$, the operating system schedules out T_2 on $VCPU_2$ and executes T_3. Finally, at $t_2 = 22$, we take a second sample and compute the deltas to the first sample as follows:

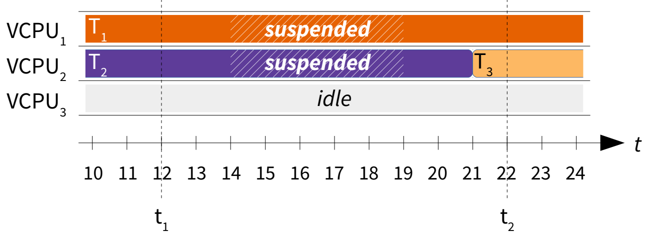

Figure 1: Schedule with three threads on three virtual CPUs

$$\Delta t_{cpu}(T_1) = 10 \qquad \Delta t_{cpu}(T_2) = 9 \qquad \Delta t_{cpu}(T_3) = 1$$
$$\Delta t_{cpu,total} = 10 + 9 + 1 = 20$$
$$\Delta t_{steal,total} = 5 + 5 = 10$$

The time during which the virtual CPUs were suspended is accounted as CPU time to the scheduled threads because the operating system does not consider steal time in its CPU time accounting. Note that $VCPU_3$ was idle, so it does not contribute any steal time, although it was technically suspended. Our approach now computes the following steal times for the individual threads:

$$\Delta t_{steal}(T_1) = 10\frac{10}{20} = 5 \qquad \Delta t_{steal}(T_2) = 10\frac{9}{20} = 4.5$$
$$\Delta t_{steal}(T_3) = 10\frac{1}{20} = 0.5$$

Therefore, T_1 is assigned the correct amount of steal time. Although T_3 was not affected by suspension, it is assigned a small portion of the steal time because it was scheduled briefly at the end of the sampling period. For the same reason, T_2 is assigned slightly less than the actual amount of suffered steal time. However, we expect that such deviations become insignificant with frequent samples.

3. IMPLEMENTATION

We implemented a prototype of our approach that computes the steal time of all threads of a Java application running in a Linux guest under a Linux host with the Kernel-based Virtual Machine (KVM, [7]) as a hypervisor. To integrate our prototype with the Java virtual machine (JVM), we implemented it as an agent that uses the Java VM Tool Interface (JVMTI, [9]).

Linux exposes information about the resource usage of the system, its processes, and its threads via the *procfs* pseudo-filesystem, which is typically mapped to */proc*. The

/proc/stat file provides the resource usage of the entire system, which is broken down into CPU time spent executing application code, CPU time spent in the operating system (mostly on behalf of an application), steal time, and idle time. These times are given in clock ticks since the start of the operating system. Unlike for threads, the system-wide CPU times exclude the steal time. We therefore compute $t_{cpu,total}$ as the sum of all times except the idle time, and use the provided steal time as $t_{steal,total}$. We then use the stored values from the last sample to compute $\Delta t_{cpu,total}$ and $\Delta t_{steal,total}$.

The CPU time of an individual process is available in its */proc/[pid]/stat* file, where *[pid]* is the numeric process identifier. The CPU time of a specific thread of a process is available in */proc/[pid]/task/[tid]*, where *[tid]* is the thread identifier. The times are given in clock ticks since the start of the thread and split up into "regular" user-space ticks and in system ticks. System ticks are time spent in the operating system on behalf of the application. In our implementation, we do not distinguish between user-space ticks and system ticks. Whenever we take a sample, we take their sum and compare it to the stored sum from the last sample to get $\Delta t_{cpu,total}(x)$.

Our JVMTI agent is loaded by the JVM at startup time. During initialization, we create data structures to store the tick counts from /proc that are required to compute the deltas, and to store the accounted steal time for each thread. Our agent registers for JVMTI events so that it is notified when application threads start and end. This enables us to create a record for a thread when it starts and to stop monitoring a thread (but keep its record) when it ends.

When the application is launched, our agent starts a separate thread with a sampling loop. In this loop, we periodically retrieve the system-wide tick counts as well as the tick counts for all existing application threads. We then compute the deltas using the stored values from our records, account the new steal time to the threads, and update our records. At the end of an iteration, the sampling loop pauses for the sampling interval before it takes another sample and attributes the new steal time.

4. EVALUATION

We used our prototype implementation to evaluate the accuracy and the overhead of our approach. We set up KVM virtualization under openSUSE Linux 13.1 on a computer with a quad-core Intel Core i7-4770 processor with 16 GB of memory. To get more stable results, we disabled the hyperthreading, turbo boost and dynamic frequency scaling features. With the exception of essential system services, no other processes were running during our measurements.

We relied on the DaCapo suite [2] for our tests, which consists of non-trivial benchmarks with different multi-threading characteristics[1]. The workload of the benchmarks is constant, so the consumed CPU times are similar between executions. Therefore, we test whether subtracting the accounted steal time results in a similar CPU time as when the benchmarks are not subjected to hypervisor suspension. However, because most benchmarks use thread pools to divide work between threads, the CPU time of individual benchmark threads can vary between executions. Therefore, we compared the total CPU time of all benchmark threads (but excluding threads of the JVM and the sampling thread).

First, we created a VM with four virtual CPUs and installed openSUSE Linux 13.1 and the Oracle Java SDK 8u20 in it. We then executed the DaCapo benchmarks with our JVMTI agent, which we configured to take samples every 50 milliseconds. To reduce the impact of the startup phase of the JVM, we executed 10 successive *iterations* of every benchmark in a single JVM instance. We recorded the total CPU times of all benchmark threads over all iterations. We further executed multiple *rounds* of each benchmark (with 10 iterations each) and took the median total CPU time.

Next, we created a second, identical VM which only executes a tool that continuously generates artificial CPU load by computing checksums in four threads. Therefore, both VMs continuously compete with each other for physical CPU time, and the hypervisor is forced to suspend virtual CPUs. We started this tool in the benchmarking VM as well, so that the suspension of its virtual CPUs affects not only the benchmark threads. Instead, depending on the multi-threading characteristics of each benchmark and on scheduling, suspension affects the benchmark threads and the tool's threads to different degrees over time, which is more likely to expose inaccuracies in our steal time accounting approach. Using this setup, we repeated our measurements.

4.1 Corrected CPU Times

Figure 2 shows the corrected and uncorrected median total CPU times of all application threads for each benchmark. The values of each benchmark are normalized to its median CPU time without extra load. The error bars indicate the 5th and 95th percentiles. The figure demonstrates that the corrected CPU times under load are commonly within a few percent of the CPU times in an idle environment, whereas the uncorrected CPU times under load are typically around twice as large as the CPU times in the idle environment. In some cases, our prototype accounts too much steal time, such as for *avrora* or *tradebeans.* For other benchmarks, the steal time is underestimated, such as for *tomcat, tradesoap* and *lusearch.* The reasons seem to be more complex than just differences between the multi-threading characteristics of the benchmarks, because *tradebeans* and *tradesoap* are identical in that regard, but show opposite results. We are investigating these deviations in our ongoing work.

4.2 Overhead

We measured the overhead of our approach by comparing the wall-clock execution times of each benchmark when our agent is enabled and when the agent is not enabled. We performed these measurements in an otherwise idle VM, with no other VMs running. Figure 3 shows the normalized median overheads and the 5th and 95th percentiles for all benchmarks. *fop, lusearch* and *sunflow* show a slight median overhead between 0.5% and 1.5%. For *avrora, h2* and *tradebeans,* the total execution times with the agent enabled are even 1% to 5% smaller than without the agent, which probably comes from effects that the agent's sampling thread has on scheduling. The other benchmarks did not show any distinctive changes in execution time. The geometric mean of the execution times with steal time accounting is 99.2% of that without steal time accounting. Therefore, we consider the overhead of our approach negligible or even non-existent.

[1]We did not use the DaCapo suite's *batik* and *eclipse* benchmarks because they do not run on OpenJDK 8.

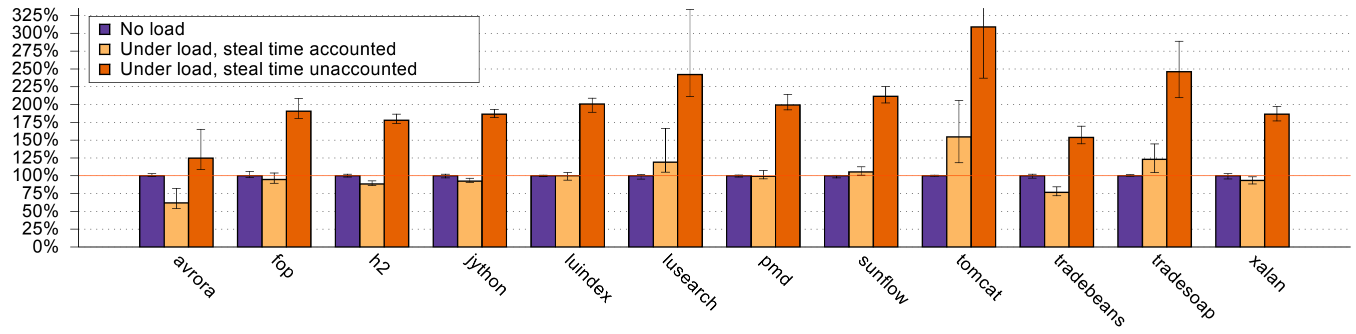

Figure 2: Normalized total CPU time of all application threads with and without accounted steal time

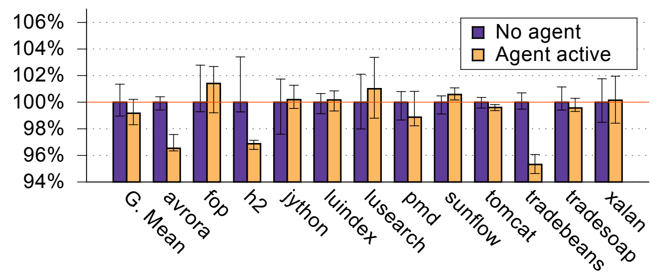

Figure 3: Overhead of the agent

5. RELATED WORK

Timing issues in VMs have been investigated before. Lampe et al. measured the impact of inaccurate timing in publicly available cloud computing services [8]. Johnson et al. describe extensions to the Performance API (PAPI) project which provide more accurate timers and shared resource usage statistics in VMs [5]. However, these extensions only provide system-wide steal time information, and the paper emphasizes the need for per-process steal time accounting. Chen et al. describe a modification to the Xen hypervisor that delivers hypervisor scheduling events to a modified guest operating system which then uses them to correctly account steal time to processes and threads [3]. M. Holzheu proposed a patch for the Linux kernel that detects increases of the per-CPU steal time and accounts them to the scheduled processes and threads [4]. Both approaches require modifications to the operating system or hypervisor, while our approach does not.

6. CONCLUSIONS AND FUTURE WORK

We described a novel approach for determining to what degree application threads in a virtual machine are affected when the hypervisor suspends virtual CPUs. Our approach does not require any modifications to the hypervisor or to the operating system. We implemented our approach for Java applications in a Linux guest with KVM as the hypervisor. An evaluation with the DaCapo benchmarks demonstrated that our approach accurately accounts the system-wide steal time to the application threads at negligible overhead, and that the accounted steal time can be used to compute the application's true execution time on a physical CPU.

As next steps, we plan to extend our prototype with the capability to track *transactions* that execute within a thread, such as individual web requests. Measuring the duration and resource usage of transactions is a central feature of Application Performance Monitoring (APM) systems such as that from our industry partner Compuware [10]. Suspension can considerably distort these measurements because of the typically short execution time of transactions.

Furthermore, we plan to develop a test suite that generates different deterministic load patterns that we can use to validate our approach in more detail. We also plan to examine the effect of the sampling rate on the accuracy, particularly with different load scenarios. We further consider experimenting with scheduling traces from the Linux perf subsystem [6] for validation.

7. ACKNOWLEDGEMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Compuware Austria GmbH.

8. REFERENCES

[1] The CPU scheduler in VMware vSphere 5.1. VMware Inc., 2013.

[2] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. OOPSLA '06, pages 169–190. ACM, Oct. 2006.

[3] H. Chen et al. XenHVMAcct: Accurate CPU Time Accounting for Hardware-Assisted Virtual Machine. *2010 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 191–198, Dec. 2010.

[4] M. Holzheu. [RFC][PATCH v2 4/7] taskstats: Add per task steal time accounting. `https://lkml.org/lkml/2010/11/11/271`, 2010.

[5] M. Johnson et al. PAPI-V: Performance monitoring for virtual machines. In *ICPP Workshops*, pages 194–199, 2012.

[6] kernel.org. perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/`.

[7] A. Kivity et al. KVM: the Linux virtual machine monitor. In *Linux Symposium*, volume 1, pages 225–230, 2007.

[8] U. Lampe et al. The Virtual Margin of Error. *Proc. of CLOSER*, 2012, 2012.

[9] Oracle. JVM^{TM}Tool Interface version 1.2.1. `http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html`.

[10] A. Reitbauer et al. *Java Enterprise Performance*. 2012.