View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PolyPublie

POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	VM processes state detection by hypervisor tracing
Auteurs: Authors:	Hani Nemati et Michel R. Dagenais
Date:	2018
Type:	Communication de conférence / Conference or workshop item
Référence: Citation:	Nemati, H. & Dagenais, M. R. (2018, avril). *VM processes state detection by hypervisor tracing*. Communication écrite présentée à Annual IEEE International Systems Conference (SysCon 2018), Vancouver, Canada (8 pages). doi:10.1109/syscon.2018.8369612

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/4204/
Version:	Version finale avant publication / Accepted version Révisé par les pairs / Refereed
Conditions d'utilisation: Terms of Use:	Tous droits réservés / All rights reserved

Document publié chez l'éditeur officiel
Document issued by the official publisher

Nom de la conférence: Conference Name:	Annual IEEE International Systems Conference (SysCon 2018)
Date et lieu: Date and Location:	23-26 avril 2018, Vancouver, Canada
Maison d'édition: Publisher:	IEEE
URL officiel: Official URL:	https://doi.org/10.1109/syscon.2018.8369612
Mention légale: Legal notice:	© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

**Ce fichier a été téléchargé à partir de PolyPublie,
le dépôt institutionnel de Polytechnique Montréal**
This file has been downloaded from PolyPublie, the
institutional repository of Polytechnique Montréal

http://publications.polymtl.ca

VM Processes State Detection by Hypervisor Tracing

Hani Nemati*, and Michel R. Dagenais†
Department of Computer and Software Engineering,
Polytechnique Montreal, Quebec, Canada
*Email: {*hani.nemati,†michel.dagenais}@polymtl.ca*

***Abstract*—The diagnosis of performance issues in cloud environments is a challenging problem, due to the different levels of virtualization, the diversity of applications and their interactions on the same physical host. Moreover, because of privacy, security, ease of deployment and execution overhead, an agent-less method, which limits its data collection to the physical host level, is often the only acceptable solution.**

In this paper, a precise host-based method, to recover wait state for the processes inside a given Virtual Machine (VM), is proposed. The virtual Process State Detection (vPSD) algorithm computes the state of processes through host kernel tracing. The state of a virtual Process (vProcess) is displayed in an interactive trace viewer (Trace Compass) for further inspection. Our proposed VM trace analysis algorithm has been open-sourced for further enhancements and for the benefit of other developers. Experimental evaluations were conducted using a mix of workload types (CPU, Disk, and Network), with different applications like Hadoop, MySQL, and Apache. vPSD, being based on host hypervisor tracing, brings a lower overhead (around 0.03%) as compared to other approaches.

***Keywords*-Wait Analysis, Virtual Process, Virtual Machine, Cloud, Performance Analysis**

I. Introduction

Cloud computing has been a vector for innovation, providing much better flexibility and greater resource utilization effectiveness, as compared to traditional computer systems organizations. It reduces hardware capital expenses by sharing existing resources among co-located VMs. The modularity of VMs also increases reliability by symplifying backup, redundancy and disaster recovery. It provides good performance by more easily upgrading to the latest generation of hardware[1]. However, cloud users may experience difficult to diagnose performance degradations. The reasons for application performance degradation, when running on a VM, can be categorized as follows.

1) Heavy load of an application inside the VM.
2) Contention with other applications inside the VM.
3) Contention with other co-located VMs.
4) Cloud platform failures.

The first reason is when a process brings a heavy load but the VM does not have enough resources. It increases the wait time for those resources. If it only impacts the processes inside the VM, we categorize this as a local impact. Conservative resource booking is a technique to mitigate the impact of heavy load in a VM, but it could waste resources if they remain unused. As result, it increases the costs for the cloud provider.

The second reason is when two or more processes within a VM compete for resources. This leads to processes slowdowns inside the VM but may not have a global impact on other co-located VM. In these first two cases, the owner of the VM should ask for more resources or ask for an elastic VM. An elastic VM enables the VM to increase or decrease its capacity (e.g. number of cores, memory) based on its needs, in order to optimize the response time and minimize the cost. On the other hand, having an elastic VM is more challenging and complex to manage.

The third case is when two or more VMs share the same resources. Cloud users may experience a performance degradation due to resource contention and interference between VMs. Interferences between VMs manifest themselves as latencies in response time of processes inside VMs.

The fourth case is when there is a hardware failure in the a physical machine. It impacts all the VMs at the same time and causes performance degradation on all VMs using the same hardware.

The first two cases could be managed by the owner of the VM, since he has access to the VM and knows the behaviour of the processes. The other cases need to involve the infrastructure provider. The problem is that each user does not know anything about the virtualization layer, and many issues, even for the first two cases, could happen because of the virtualization layer. Thus, there is a need for a technique to detect all four cases from the host side and to find out whether the problem has a global or local impact.

The best way to anticipate the needed resources is to investigate the significant processes inside the VM and analyze their needs. The significant processes are the ones which use the most resources, compared to other processes. Then, the IaaS provider could detect when and why a process is waiting.

This paper focuses on the wait analysis of processes inside the VMs, without accessing the VMs internally. The Virtual Process State Detection (vPSD) algorithm is being introduced and can detect the different states of a process inside a VM. Our technique can inspect the buried information in the vCPU thread and convert it to meaningful knowledge about the processes running on the vCPU. It can detect when and why a process inside the VM is waiting for a resource. It also can find out the sensitivity of processes inside the VM to a specific resource. We benchmarked our technique to find

978-1-5386-3664-0/18/$31.00 ©2018 IEEE

different issues for well-known applications (e.g., Hadoop, MySQL, and Apache Web Server) with different resource usage signatures.

Our main contributions in this paper are: **First**, we propose a fine-grained VM processes state analysis based on host tracing. All the tracing and analysis phase is hidden to the VMs. Therefore, there is no need for internal access within VMs, which is not allowed in most situations because of security reasons. **Secondly**, we use our technique to identify different issues for well-known applications like Hadoop TeraSort, MySQL, and Apache web server. **Thirdly**, we implemented a graphical view for processes inside VMs. Our graphical view presents a timeline for each process, with different states along with their interactions with the Virtual Machine Monitor (VMM).

The remainder of this paper is structured as follows. Section II reviews the related work. Section III explains different available states for the VM processes. In section IV, we present the algorithm used to detect different states of processes inside the VM. Section V demonstrates our experimental results. We also compare our method with other available methods in terms of overhead in sub-section V-E. Section VI concludes the paper with directions for future investigations.

II. Related Work

In this section, we review the available techniques for monitoring and debugging VMs.

Virtual Machine Introspection (VMI) is a technique to analyze the memory of a given VM to detect the state of the VM from outside. It has mostly been used for security thread analysis or simple VM monitoring. LibVMI[2] is an open-source library for analyzing the VM memory space. Other VMI tools like in [3][4] have been used to detect security threads. Analyzing memory space is time-consuming, especially if the memory space of the VM is large. On the other hand, none of the VMI tools target performance analysis in terms of resource usage and resource contention.

Yasuhiko in [5] proposed a performance monitoring technique for on-line applications. They sniff network packets to estimate the application response time. This technique works for the VMs which have network intensive applications. It could not be applied for other types of VM workloads.

J. Taheri et al. in [6] introduced an approach to predict the performance of virtual machine applications based on hypervisor metrics. They use a sensitivity analysis of VMs to cloud resources to predict their throughput. Their method is based on monitoring the whole VM and does not provide information about each process inside the VM. Furthermore, it also could not show exactly when and why the VM is waiting for a resource.

A technique to detect global fault and local fault has been proposed in [7]. In this paper, resource over-commitment could be detected by tracing each and every VM using

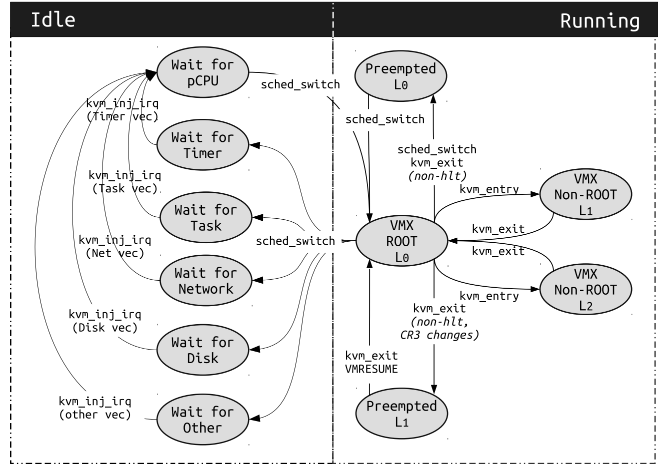

Figure 1. vProcess states

LTTng. As it is shown in sub-section V-E the overhead for tracing each VMs is much more than with our proposed method.

An adaptive SLA-based elasticity method has been proposed in [8]. In their method, the IMS network Call Setup Delay is being used as an end-to-end metric to scale up and scale down the number of CPU cores and their frequency. Their method is based on CPU scaling and does not provide any information about the disks or the network.

The work closest to ours, which motivated the current research, is presented in [9]. In this paper, they present a technique to find the active path for the threads along different machines. They could distinguish different states like wait for disk, network, timer, and task. Their approach could be applied for VMs but it would need to trace each VM. Then, they have to synchronize the traces from each VM to search through all threads and find the active path.

To the best of our knowledge, there is no pre-existing technique to analyze the state of the processes inside the VM. Our technique can uncover many issues inside VMs without internal access. Moreover, compared to other solutions, our method brings less overhead, and simplifies deployment in terms of tracing, since it limits its data collection to the host hypervisor level.

III. vProcess States

Intel-VT (and similarly AMD-V) supports two operating modes: Root mode and non-Root mode. Non-privileged instructions of VMs are executed as non-Root mode, and privileged instructions are executed as root mode (at a higher privilege level). The transitions between root mode and non-root mode are called Virtual Machine Extensions (VMX) transitions. In each VMX transition, the environment specifications of the VMs and the hypervisor are stored in an in-memory Virtual Machine Control Structure (VMCS)[10]. By analyzing VMCS structures, a lot of information about running VMs can be recovered.

Table I
EVENTS AND THEIR PAYLOAD BASED ON HOST KERNEL TRACING
$(vec_0 = Disk, vec_1 = Task, vec_2 = Net, vec_3 = Timer)$

#	Events	#	Events
1	sched_out(*vCPU0*)	17	inj_virq(vec_2)
2	sched_out(*vCPU1*)	18	enter_quest(*CR3 P#4*)
3	inj_virq(vec_0)	19	inj_virq(vec_0)
4	sched_in(*vCPU0*)	20	inj_virq(vec_0)
5	inj_virq(vec_0)	21	enter_quest(*CR3 P#3*)
6	enter_guest(*CR3 P#3*)	22	inj_virq(vec_1)
7	sched_out(*vCPU0*)	23	enter_quest(*CR3 P#2*)
8	sched_in(*vCPU1*)	24	sched_out(*vCPU1*)
9	inj_virq(vec_1)	25	sched_in(*vCPU1*)
10	enter_quest(*CR3 P#1*)	26	inj_virq(vec_1)
11	sched_out(*vCPU1*)	27	enter_quest(*CR3 P#1*)
12	inj_virq(vec_2)	28	sched_out(*vCPU0*)
13	sched_in(*vCPU0*)	29	inj_virq(vec_3)
14	inj_virq(vec_2)	30	enter_quest(*CR3 P#4*)
15	enter_quest(*CR3 P#5*)	31	sched_out(*vCPU1*)
16	sched_in(*vCPU1*)		

Figure 1 shows different states of a vProcess and the conditions to reach those states. In [11][12][13], we proposed several techniques to understand the different states of a vProcess when it is running at any virtualization level. In this paper, our focus is to discover the reason for being Idle, for each process.

A vProcess can be woken up for the following reasons. **First**, a process inside the VM sets a timer and the timeout occurred (Timer Interrupt). **Second**, a process inside the VM is awakened by another process (IPI Interrupt). **Third**, a process inside the VM is woken up by a remote task over a socket (Network Interrupt). **Fourth**, a process inside the VM is waiting for the disk (Disk Interrupt). The interrupt later injected into the VM reveals the reason for the Idle state.

IV. Virtual Process State Detection Algorithm (vPSD)

In this section, we propose an algorithm to understand the wait states for the vProcesses using the host trace. Before introducing the Virtual Process State Detection Algorithm (vPSD), we explain a very simple algorithm to detect the running state for vCPUs of a VM. The VMs vCPUs are like a normal userspace process, from the host perspective. Like other processes, the `sched_out` event shows that the thread related to a vCPU is scheduled out from the physical CPU (pCPU) and goes into the Idle State. The next `sched_in` event shows that the thread related to the vCPU is being scheduled in and the state changes from Idle to Running. This algorithm is simplistic and cannot detect the reason for the Idle state for a specific VM. Figure 2 (top) shows an example of this algorithm using the events from Table I. Much information is buried within the vCPUs thread. This information can be revealed by analyzing the interaction between the host hypervisor and VM OS. Our technique leverages existing static tracepoints inside the host hypervisor, along with our new added tracepoints, to convert the tracing information into meaningful visualization.

The vPSD algorithm can reveal the exact reason for the Idle State for each process and show the significant processes inside the VM. vPSD is illustrated with an example in Figure 2. When receiving the `sched_out` event, the vProcess sate goes to *Wait for Reason*. The key idea is that the event indicating the cause for the Idle state is unknown a priori. The state of vProcess can be changed to the exact reason later, when the interrupt is injected into the VM. Event #3 shows that the Disk thread injects a disk interrupt into the VM. Then, the vCPU is scheduled in (Event #4) and injects the received interrupt into the VM (Event #5). Then, `enter_guest` depicts that the VM is going to non-Root mode to run vProcess #CR3. The CR3 value (page table address) is being used as unique identifier for each vProcess in the VM. Moreover, the reason for waiting of vProcess #CR3 can be updated to *disk*, since vec_0 is the irq vector for the disk interrupt. When receiving Event #9, the process #1 *Wait for Reason* state changes to Task (vec_1 is the irq vector for IPI interrupts). Other states are built by examining the injected interrupt, when execution resumes *(running* state) for each vCPU. Surprisingly, vCPU0 was running without any `sched_out` event from Event #13 to #28. At the beginning, Process #5 was running and then Process #2 was scheduled in. This happens when the VM scheduler has many processes ready to run in its queue.

The pseudocode for the vPSD algorithm is depicted in Algorithm 1. The vPSD algorithm receives a sequence of events as input and updates the vProcess state for each VM. Event `sched_in` shows when a vCPU is running on a pCPU. When a vCPU is scheduled in, it saves the start time to find out later which vProcess is being scheduled in 2. By contrast, when a vCPU is scheduled out, it sets vProcess #CR3 Status as *Waiting for Reason* (Line 4) which will be updated later with the waiting cause. The event `enter_guest` depicts which vProcess is running on the vCPU and updates the state of vProcess to *running* (Line 8). In case the event is `inj_virq`, the vec field in this event is compared with the Task, Timer, Disk, and Network interrupt number. The unknown state is updated based on the *vec* number.

V. Performance Evaluation and UsesCases

In this section, we evaluate our wait analysis technique for analyzing the behavior of VMs and detecting related issues. First, we formulate the wait for resources for each process and VM, and then we use it to detect significant processes and their needs.

Wait for disk is shown in Equation 1. Here $f_{i,j}^{Disk}$ and $T_{i,j}^{Disk}$ are the time fraction and average wait time that *Process #j* of *VM #i* waits for disk, respectively. $T_{i,j}^{Total}$ is the total execution time for process #j in VM #i.

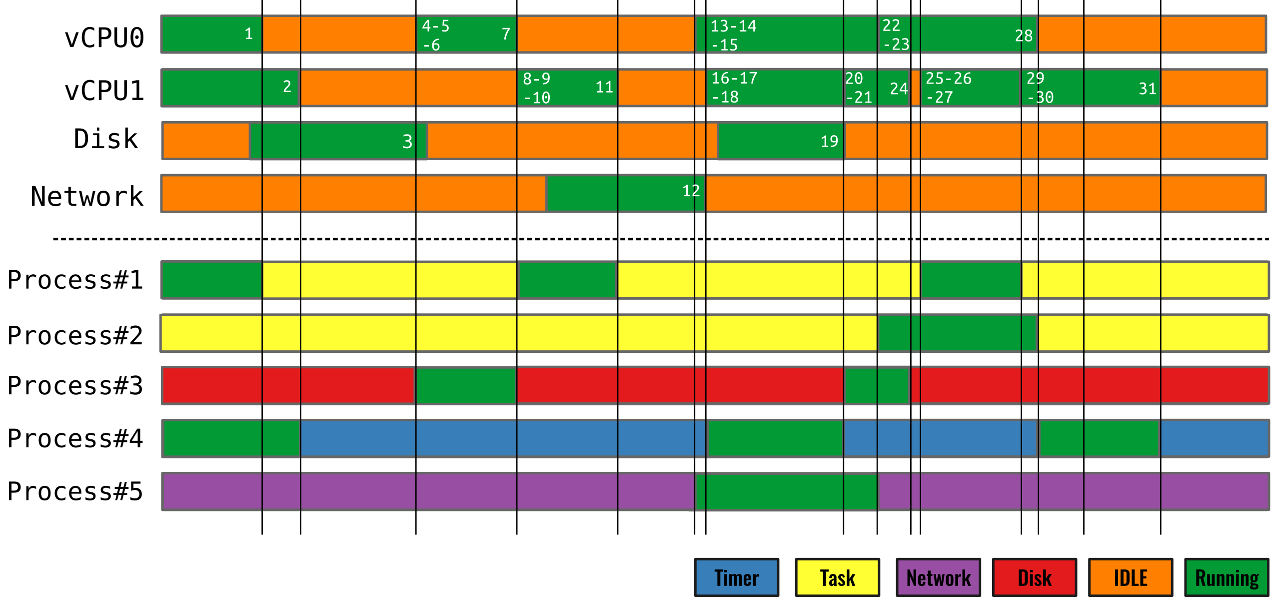

Figure 2. vProcess state detection using vPSD algorithm

Algorithm 1: Virtual Process State Detection (vPSD) Algorithm

1 **if** *event* == *sched_in* **then**
2   Save Start Time for the vCPU;
3 **else if** *event* == *sched_out* **then**
4   Query previous stored CR3 for vCPU;
5   Save End Time for Process #CR3;
6   Set Process #CR3 Status as Waiting for Reason;
7 **else if** *event* == *enter_quest* **then**
8   Save running process #CR3 for vCPU;
9   Query Start Time for the vCPU;
10   Set Process #CR3 Status as Running;
11 **else if** *event* == *inj_virq* **then**
12   Query End Time for Process#CR3
13   **if** *vec* == *Task Interrupt* **then**
14   Update Wating for Reason State for Process #CR3 as Task;
15   **else if** *vec* == *Timer Interrupt* **then**
16   Update Wating for Reason State for Process #CR3 as Timer;
17   **else if** *vec* == *Disk Interrupt* **then**
18   Update Wating for Reason State for Process #CR3 as Disk;
19   **else if** *vec* == *Network Interrupt* **then**
20   Update Wating for Reason State for Process #CR3 as Network;
21   **else if** *vec* == *Unknown* **then**
22   Update Wating for Reason State for Process #CR3 as Unknown;

$$W_{i,j}^{Disk} = \frac{f_{i,j}^{Disk} T_{i,j}^{Disk}}{T_{i,j}^{Total}} \times 100 \tag{1}$$

Sometimes *Process #j* waits to receive a packet. This could be because of a slow network or a failure in the network stack. Equation 2 shows the wait for network. Here $f_{i,j}^{Net}$ and $T_{i,j}^{Net}$ are the fraction of time and average wait time that *Process #j* of *VM #i* waits for network, respectively.

$$W_{i,j}^{Net} = \frac{f_{i,j}^{Net} T_{i,j}^{Net}}{T_{i,j}^{Total}} \times 100 \tag{2}$$

Sometimes a process communicates with another one in order to take advantage of the task parallelism. Although it can increase the performance of applications, it can be surprisingly difficult to get good performance. Equation 3 represents the time that a *Process #j* of *VM #i* is idle because it was waiting for another task to finish. $f_{i,j}^{Task}$ and $T_{i,j}^{Task}$ are the fraction of time and average wait time that *Process #j* of *VM #i* waits for another task to finish something, respectively.

$$W_{i,j}^{Task} = \frac{f_{i,j}^{Task} T_{i,j}^{Task}}{T_{i,j}^{Total}} \times 100 \tag{3}$$

A VM process could be awakened by a timer interrupt, which indicates that a timer expired inside the VM. It could be because of scheduler ticks, which means that the VM does not have anything to run. It also could be because of a timer that an application sets. This could make a vCPU idle for a long time. Equation 4 represents the time that a vCPU is idle because it was waiting for a timer to be fired. $f_{i,j}^{Timer}$ and $T_{i,j}^{Timer}$ are the fraction of time and average wait time that *Process #j* of *VM #i* waits for a timer, respectively.

$$W_{i,j}^{Timer} = \frac{f_{i,j}^{Timer} T_{i,j}^{Timer}}{T_{i,j}^{Total}} \times 100 \quad (4)$$

Total percentage of waiting time for *VM #i* with process `p` is computed in equation 5.

$$T_i^{wait} = \sum_{j=1}^{p} W_{i,j}^{Disk} + W_{i,j}^{Net} + W_{i,j}^{Task} + W_{i,j}^{Timer} \quad (5)$$

When waiting for a task or timer, the VM does not wait for a resource. As a result, the vCPU is unused and can be allocated to another VM.

Total fraction of running *Process #j* of *VM #i* is calculated in Equation 6. More scheduling activity in a vCPU causes more virtualization overhead, as the VM must read and update the VMCS structure from the pCPU.

$$f_{i,j}^{Total} = f_{i,j}^{Task} + f_{i,j}^{Timer} + f_{i,j}^{Disk} + f_{i,j}^{Network} \quad (6)$$

A. Analysis Architecture

We have chosen the Kernel-based Virtual Machine (KVM), under the control of OpenStack, as experimental setup. KVM is the most commonly used hypervisor for Openstack[14]. For the userspace part of the hypervisor, we installed QEMU to execute the OS support for the VM. Our architecture is shown in Figure 3. As we can see, events are gathered by our tracer (LTTng) from the host hypervisor first, and then the events are sent to the trace analyzer (TraceCompass). For the prototype implementation on Linux, we used LTTng as the tracer. The host kernel and KVM module are instrumented by different static tracepoints. LTTng gathers the events from the KVM module and kernel space, and sends them to the analyzer. Trace Compass is an open source software for analyzing traces and logs. It has some pre-built views, especially for LTTng [15]. We prototyped our wait analysis tool as a separate view in Trace Compass.

For our analysis, three events should be enabled in LTTng: `sched_switch` to find out the running states, `kvm_inj_virq` to update the wait states and `vcpu_enter_guest` to find out which process is running on the vCPU.

The process identifier (PID) and process name of each thread inside the guest is not directly accessible from host tracing. The only information which can be uncovered by host tracing about the threads inside the VMs, is written in CR3. Indeed, CR3 points to the page directory of a process in the virtual machine. In each VMX transition, we retrieve the CR3 value of a VM using a new tracepoint. We added the new tracepoint, `vcpu_enter_guest`, to extract the CR3 register from the guest area of the VMCS. To have more information about processes inside the VMs, we need to map CR3 to the PID and process name. This is not strictly necessary, since CR3 is a unique identifier for processes, but

Table II
EXPERIMENTAL ENVIRONMENT OF HOST AND GUEST

	Host Environment	Guest Environment
CPU	Intel(R) i7-4790 CPU @ 3.60GHz	Three vCPUs
Memory	Kingston DDR3-1600 MHz, 32GB	3 GB
OS	Ubuntu 15.10 (Kernel 4.2.0-27)	Ubuntu 15.10 (Kernel 4.2.0-27)
Qemu	v2.5	v2.5
LTTng	v2.8	v2.8

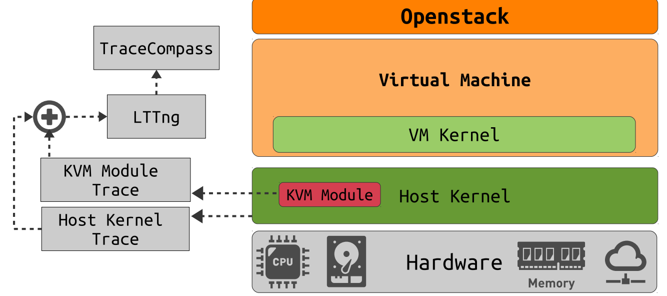

Figure 3. Architecture of our implementation

it is more convenient and human readable if we can map the process information inside the guest with the information we get from the `vcpu_enter_guest` tracepoint.

Our experimental setup is described in Table II. The analysis is performed with the following steps: 1) Start LTTng on the host. 2) Run the VMs of interest. 3) Stop LTTng. 4) Run the analyses. 5) Display the results in Trace Compass.

B. UseCase 1 - Disk Issue

Predicting VM workloads is a big challenge. Sometimes, cloud users set the VM resource cap too low, which causes inadequate resource allocation (local impact on applications). In another case, the cloud administrator over-commits the resources and shares among too many VMs (global impact on VM). In both cases, these problems cause latency for the VMs. Most Cloud providers like Amazon limit the resource usage for each VM in order to prevent important performance reductions due to sharing resources. For example, the IO size is capped at 256 KiB for SSD volumes and 1024 KiB for HDD volumes in Amazon EC2 instances, with a certain number of I/O operations per second allowed [16]. There is a trade-off between setting resource limits too low and too high. When the resource limit is too low, the VM waits for the resource most of the time.

In our first experiment, we use *Hadoop TeraSort* to sort a huge amount of randomly generated data. Hadoop TeraSort tasks are a combination of CPU-intensive job and IO intensive job. In this test, each row is 100 bytes; thus the total amount of data written to disk is 100 times the number

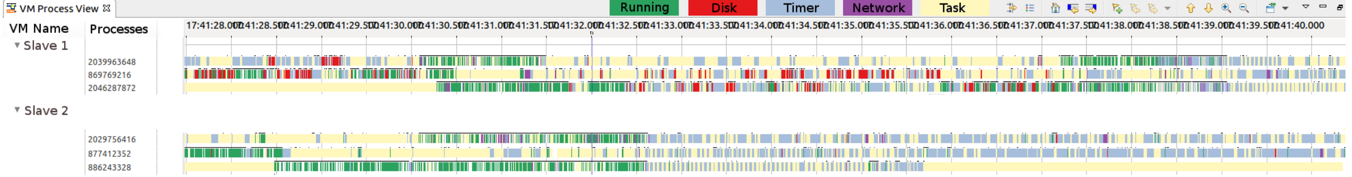

Figure 4. Two slaves running Hadoop TeraSort 5GB- VM-Slave1 response is late because of wait for Disk

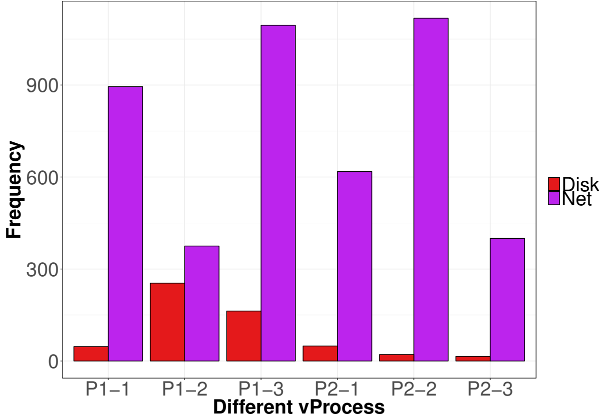

Figure 5. Frequency of wait for disk for Slave 1 and Slave 2 - P1-1: 2039963648, P1-2: 869769216, P1-3: 2046287872, P2-1: 2029756416, P2-2: 877412352, P2-3: 886243328

of rows (i,e., to write 100MB of data, 1 million rows should be used). The 5GB of random data is being generated by Hadoop TeraGen and is being sorted by Hadoop TeraSort. We conducted our experiment on a cluster where each VM had 3 vCPUs with 3 GB of memory. One VM is designated as master and 2 VMs are configured as Slaves. The Hadoop version was 2.8.1 and Java version was 8. We used the YARN framework for job scheduling and resource management. We realized that sometimes the execution time for sorting 5GB of data is larger than expected. We investigated further and found out that node VM-Slave 1 finishes its associated task after other nodes. The same job is submitted again and the host is traced with LTTng. In our investigation with LTTng and vPSD (shown in Figure 4), we found that node VM-Slave 1 is waiting more for disk, compared to other VM-slaves. VM-Slave 1 is compared with VM-Slave 2 by looking at their processes. We found that both VMs have three significant processes and, since VM-Slave 1 waits more for disk, it responds later to the Hadoop Master. Table III represents the percentage of wait for different resources ($W_{i,j}^{Disk}$, $W_{i,j}^{Net}$, $W_{i,j}^{Task}$, and $W_{i,j}^{Timer}$). As shown, wait for disk ($W_{i,j}^{Disk}$) for Slave-1 is larger than for Slave-2. Moreover, Figure 5 shows the frequency of wait for Disk and Network, for Slave-1 and Slave-2. Slave-1 waits more for disk as compared to Slave-2.

The reason could be putting the Disk cap too low, or co-locating with other IO intensive VMs in the same resource group. In this case, the issue was a limitation on disk usage.

In order to show how our technique could find out

Table III
WAIT ANALYSIS OF HADOOP TERASORT

Processes	Root	non-Root	Task	Timer	Disk	Net
2039963648	0.004	1.499	54.040	43.512	0.198	0.217
869769216	0.001	0.979	39.938	56.978	1.251	0.700
2046287872	0.004	3.125	57.357	38.022	0.766	0.283
2029756416	0.002	1.898	36.508	60.883	0.098	0.340
877412352	0.001	0.970	24.947	72.767	0.029	1.130
886243328	0.005	8.350	85.240	5.588	0.003	0.258

contention between VMs on the same resource, the Linux `dd` command with *sync* has been used to write a random file to the disk. In this experiment, the VM is on a SSD and there is no cap on disk usage. We found out that writing takes more time than expected.

Using vPSD, we found out that the two VMs are waiting for disk and there is contention between the two VMs in order to use the shared disk. Figure 6 represents the contention between the two VMs. VM1 has 6 significant processes and VM2 has 7 significant processes. As it is shown, two VMs request disk access at the same time and wait to complete the request.

C. UseCase 2 - Network Issue

In the second experiment, a Remote Procedure Call (RPC) server and client are written to experiment the effect of waiting for network for a RPC server and client. The RPC client is put in VM2 and it sends commands to the RPC server to execute every 10 ms. The command that we send is a program to calculate 1000 Fibonacci numbers. We use the `tc` traffic shaper to manipulate the traffic control settings. The traffic shaper is applied to the virtual network interface to increase the network latency by 25 ms for the RPC client and by 30 ms for the RPC server. Figure 7 depicts the graphical representation of the vProcess state. The green intervals are the running state, the purple intervals are wait for network, and light blue intervals are wait for timer. As shown, the significant process in these two VMs waits for network most of the time.

In another experiment, the RPC server and RPC client are executed when there is a natural network latency. Figure 8 shows the state of vProcess for both VMs. Since both VMs are in the same host, the delay between the two VMs is less than 1 ms. As shown, the server immediately responds to incoming requests from the RPC client. Table IV represents the percentage of wait for RPC server and client. The wait

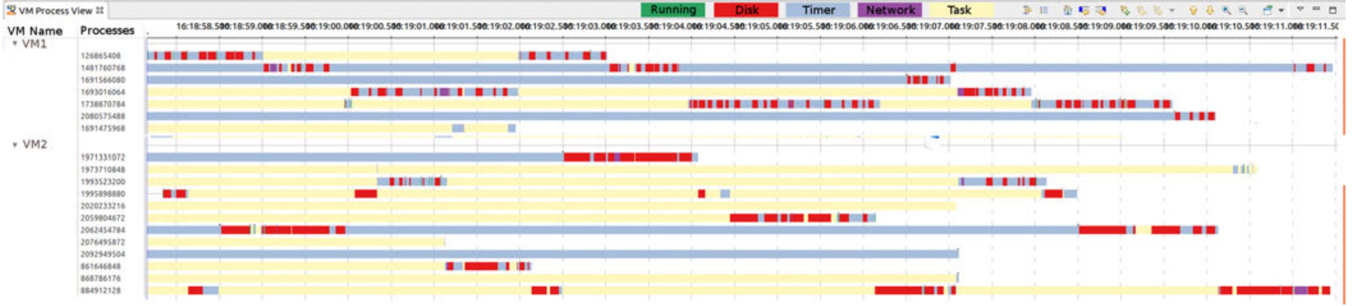

Figure 6. Wait analysis of vProcess when there is resource contention between two VMs

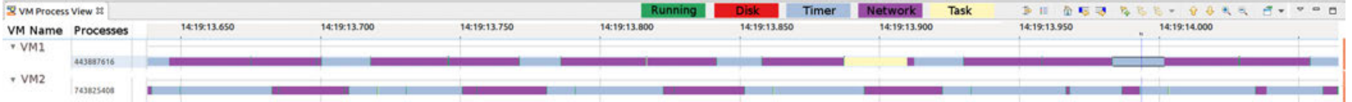

Figure 7. Wait analysis of vProcess with network issue

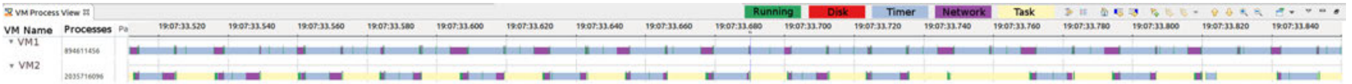

Figure 8. Wait analysis of vProcess without network issue

Table IV
WAIT ANALYSIS OF RPC SERVER AND CLIENT

Processes	Root	non-Root	Task	Timer	Disk	Net
443887616	0.001	1.140	49.246	12.783	0.0	36.751
743825408	0.001	1.246	48.023	27.978	0.0	22.643
894611456	0.006	1.743	49.639	37.677	0.0	10.298
2035716096	0.003	1.109	57.443	33.142	0.0	7.975

for network increases when there is a delay between the RPC server and the RPC client.

D. UseCase 3 - Sensitivity Analysis

Cloud providers now host millions of VMs to fulfill large scale applications and cloud services. Co-locating several VMs in the same host leads to performance reduction for sensitive VMs, which could decrease tremendously their QoS. It is important to understand the behavior of application inside the VM and to quantify their sensitivity to a specific resource. We use tracing and the vPSD algorithm to identify the sensitivity of a VM to its allocated resources.

In order to show how our algorithm could find the sensitivity of a VM to its resource, Apache2 and MySQL were used. To benchmark the sensitivity of Apache to the network performance, the `tc` command is being used to add a delay to the virtual interface of a VM. Moreover, to benchmark the sensitivity of MySQL to disk, the VM is being limited to read and write 512 KB/s. Table V shows the total duration of each state, in percentage, for the Apache and MySQL lifecycle. As shown, Apache is highly sensitive to the network performance and MySQL is not very sensitive to disk. The proportion of waiting for network for Apache-0 (Apache with natural network latency) is 1.5 % as compare to 15% for Apache-50 (Apache with 50 ms network latency),22% for and Apache-100 (Apache with 100 ms network latency). Another interseting result is depicted in Table VI which shows the frequency of waiting for a resource. Apache-100 waits 16 times more than Apache-0. The result shows that the Apache web server is highly sensitive to network latency.

Table V
WAIT ANALYSIS OF DIFFERENT APPLICATIONS, TO FIND OUT THE SENSITIVITY OF A VM TO A SPECIFIC RESOURCE

Processes	Root	non-Root	Task	Timer	Disk	Net
Apache-0	0.075	38.536	46.945	4.947	0.0	1.545
Apache-50	0.092	18.900	53.405	3.412	0.002	15.016
Apache-100	0.052	11.952	54.067	6.439	0.004	22.270
MySQL-0	0.257	85.393	11.395	2.908	0.0	0.045
MySQL-512	0.003	83.000	15.774	0.873	0.004	0.0

Table VI
FREQUENCY OF WAIT FOR DIFFERENT APPLICATIONS, TO FIND OUT THE SENSITIVITY OF A VM TO A SPECIFIC RESOURCE

Processes	f^{Task}	f^{Timer}	f^{Disk}	$f^{Network}$	f^{Total}
Apache-0	13161	108	0	586	13855
Apache-50	96667	4764	4	79654	181090
Apache-100	99078	10257	11	118307	227654
MySQL-0	694	62	0	3	759
MySQL-512	393	40	1	0	434

MySQL is also being studied. Surprisingly, MySQL is less sensitive to Disk than we expected. Wait for disk does not increase from MySQL-0 (no-limit on disk) to MySQL-512 (Limited to write or read no more than 512 KB/s).

As demonstrated, our technique could detect which application in the VM is sensitive to a specific resource.

Table VII
OVERHEAD ANALYSIS OF vPSD AS COMPARED TO CPA

Benchmark	Baseline	CPA	vPSD	Overhead	
				CPA	vPSD
File I/O (ms)	450.92	480.38	451.08	6.13%	0.03%
Memory (ms)	612.27	615.23	614.66	4.81%	0.01%
CPU (ms)	324.92	337.26	325.91	3.65%	0.30%

E. Analysis Cost

In this section, the overhead of vPSD is compared with the Critical Path Analysis (CPA), as proposed in [9]. In order to compare the two approaches, we enabled the tracepoints that were needed for the CPA approach, and we traced the VMs. Also, it is worth mentioning that our new vPSD algorithm needs only to trace the host. As shown in Table VII, the critical path analysis approach adds more overhead in all tests, since it needs to trace the VMs. We used the Sysbench benchmarks to reveal the overhead of both approaches, with sysbench configured for Memory, Disk I/O and CPU intensive evaluations. Our approach has negligible overhead for CPU and Memory intensive tasks at 0.3% or less.

VI. CONCLUSION

Virtualization in the Cloud leads to increased overall resource utilization, but raises concerns over resource sensitive VMs. Different approaches for debugging and troubleshooting the VMs were proposed. However, for the process wait analysis, in the context of VMs, the current monitoring and analysis tools do not provide enough information. We developed a novel host-based process state detection algorithm that can not only find the state of running processes but also recover the reason for being idle. Our approach is based exclusively on host hypervisor tracing, which adds less overhead as compared to other approaches. Our overhead analysis shows that the overhead of our approach is almost 10 times less than for other approaches. We conducted several experiments over well-known applications to find out different issues and discover sensitive applications in the VMs. As future work, our current technique can be enhanced to further investigate interferences between processes in VMs and predict the exact amount of desirable scale up or scale down for the resources.

REFERENCES

[1] "Top benefits of cloud computing, howpublished = https://azure.microsoft.com/en-ca/overview/what-is-cloud-computing/, note = Accessed: 2017-11-20."

[2] "Libvmi, virtual machine introspection library, howpublished = http://libvmi.com/, note = Accessed: 2017-08-03."

[3] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Vmm-based hidden process detection and identification using lycosid," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346269

[4] L. Litty and D. Lie, "Manitou: A layer-below approach to fighting malware," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ser. ASID '06. New York, NY, USA: ACM, 2006, pp. 6–11. [Online]. Available: http://doi.acm.org/10.1145/1181309.1181311

[5] Y. Kanemasa, S. Suzuki, A. Kubota, and J. Higuchi, "Single-view performance monitoring of on-line applications running on a cloud," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, June 2017, pp. 350–358.

[6] J. Taheri, A. Y. Zomaya, and A. Kassler, *vmBBThrPred: A Black-Box Throughput Predictor for Virtual Machines in Cloud Environments*. Cham: Springer International Publishing, 2016, pp. 18–33.

[7] D. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut, "Perfcompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2015.

[8] H. Nemati, A. Singhvi, N. Kara, and M. E. Barachi, "Adaptive sla-based elasticity management algorithms for a virtualized ip multimedia subsystem," in *2014 IEEE Globecom Workshops (GC Wkshps)*, Dec 2014, pp. 7–11.

[9] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, Aug 2016.

[10] I. Corporation, "Intel 64 and ia-32 architectures software developers manual," December 2015, pp. 1–3883.

[11] H. Nemati, S. D. Sharma, and M. R. Dagenais, "Fine-grained nested virtual machine performance analysis through first level hypervisor tracing," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 84–89. [Online]. Available: https://doi.org/10.1109/CCGRID.2017.20

[12] M. D. S Sharma, H Nemati, "Low overhead hardware assisted virtual machine analysis and profiling," in *IEEE Globecom Workshops (GC Workshops)*, 2016.

[13] H. Nemati and M. R. Dagenais, "Virtual cpu state detection and execution flow analysis by host tracing," in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, Oct 2016, pp. 7–14.

[14] "Open source software for creating clouds ," https://www.openstack.org/, accessed: 2017-11-16.

[15] "Trace compass," https://projects.eclipse.org/projects/tools.tracecompass, accessed: 2017-11-16.