Host-Based Virtual Machine Workload Characterization Using Hypervisor Trace Mining

HANI NEMATI, Polytechnique Montreal
SEYED VAHID AZHARI, Iran University of Science and Technology, and Polytechnique Montreal
MAHSA SHAKERI and MICHEL DAGENAIS, Polytechnique Montreal

Cloud computing is a fast-growing technology that provides on-demand access to a pool of shared resources. This type of distributed and complex environment requires advanced resource management solutions that could model virtual machine (VM) behavior. Different workload measurements, such as CPU, memory, disk, and network usage, are usually derived from each VM to model resource utilization and group similar VMs. However, these course workload metrics require internal access to each VM with the available performance analysis toolkit, which is not feasible with many cloud environments privacy policies.

In this article, we propose a non-intrusive host-based virtual machine workload characterization using hypervisor tracing. VM blockings duration, along with virtual interrupt injection rates, are derived as features to reveal multiple levels of resource intensiveness. In addition, the VM exit reason is considered, as well as the resource contention rate due to the host and other VMs. Moreover, the processes and threads preemption rates in each VM are extracted using the collected tracing logs. Our proposed approach further improves the selected features by exploiting a page ranking based algorithm to filter non-important processes running on each VM. Once the metric features are defined, a two-stage VM clustering technique is employed to perform both coarse- and fine-grain workload characterization. The inter-cluster and intra-cluster similarity metrics of the silhouette score is used to reveal distinct VM workload groups, as well as the ones with significant overlap. The proposed framework can provide a detailed vision of the underlying behavior of the running VMs. This can assist infrastructure administrators in efficient resource management, as well as root cause analysis.

CCS Concepts: • **Information systems** → **Computing platforms**; *Clustering*; Data cleaning; • **Software and its engineering** → *Software design techniques;*

Additional Key Words and Phrases: VM clustering, workload characterization, performance analysis, tracing, virtual interrupts, vCPU states, K-Means, PageRank, time series, machine learning

This word was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, Google, and EffciOS.

Authors' addresses: H. Nemati, M. Shakeri, and M. Dagenais, Polytechnique Montreal, 900 Edouard Montpetit Boulevard, Montreal, Quebec, H3T 1J4; emails: {hani.nemati, mahsa.shakeri, michel.dagenais}@polymtl.ca; S. V. Azhari, Iran University of Science and Technology, and Polytechnique Montreal, 900 Edouard Montpetit Boulevard, Montreal, Quebec, H3T 1J4; emails: azharivs@iust.ac.ir, seyed-vahid.azhari@polymtl.ca.

2376-3639/2021/06-ART4 $15.00
https://doi.org/10.1145/3460197

ACM Reference format:
Hani Nemati, Seyed Vahid Azhari, Mahsa Shakeri, and Michel Dagenais. 2021. Host-Based Virtual Machine Workload Characterization Using Hypervisor Trace Mining. *ACM Trans. Sen. Netw.* 6, 1, Article 4 (June 2021), 25 pages.
https://doi.org/10.1145/3460197

1 INTRODUCTION

Cloud computing technology has revolutionized software development and deployment at enterprises and organizations. The cloud environment provides on-demand access to shared resources, without the requirement of direct active management by the users. Despite the flexibility brought by this infrastructure, the complexity of such environments makes them prone to performance anomalies. For instance, an **Infrastructure as a Service (Iaas)** provider could include hundreds of hosts and thousands of **Virtual Machines (VMs)**, which requires a complex distributed resource management. In such an environment, an anomaly in one VM might cause performance degradations in other VMs, which could lead to significant financial penalties. In this regard, effective automatic monitoring of resource workloads could aid the service administrators to determine the VMs' behavior and thus avoid likely failures.

We argue that an effective approach to reduce the complexity of VM monitoring is to leverage a VM clustering technique. In this way, VMs are grouped into separate clusters, where VMs with similar behavior are categorized together. Therefore, the service administrator deals with a group of clusters rather than thousands of VMs, which facilitates the resource management procedure. For instance, consider the case where there is an excessive load on a specific system resource. The clustering technique can group all VMs suffering from contention in the same cluster. As a result, the administrator can easily identify this anomalous cluster and resolve this issue by adding more resources or migrating some VMs to another host.

To perform the clustering technique in such an environment, monitoring metrics must be gathered from each VM. This is challenging since the infrastructure provider usually does not have internal access to each VM because of security issues. In this regard, developing an agent-less monitoring technique is imperative to enable extracting metrics and features, without violating the VM's privacy policy.

In this article, we propose an unsupervised clustering approach based on an agent-less feature extraction technique. No prior knowledge is considered about the type of applications and processes running on the VMs. The designed clustering approach enables categorizing the VMs based on their similarity in terms of execution workload. The features are extracted based on hypervisor tracing, using the existing static kernel and **Kernel Virtual Machine (KVM)** tracepoints, along with an additional new dynamic tracepoint. Each **virtual CPU (vCPU)** state is analyzed, as well as the virtual interrupt injection rate. In addition, the VM exit reason is considered, along with the resource contention rate caused by the host and other VMs. In this work, we propose two approaches for feature extraction. In the first approach, features are extracted from the vCPUs of the VMs. In this case, we cluster VMs based on the behavior of all of their processes. The second approach uses the **PageRank (PR)** algorithm to model the importance of processes running in each VM. Then, features are extracted only from top processes instead of the vCPU. The performance metrics of the top important processes are then fed into a two-phase clustering scheme based on K-Means. Inter-clustering and intra-clustering similarity metrics of the silhouette score are used to depict the distinct workload groups. Our proposed framework can be used by an IaaS administrator as a distinguishing feature to improve scalability, resource management, and VM root cause analysis. That would also reduce the cost of resource usage by utilizing them.

The main contributions of this work are the following. First, the feature extraction is performed in an agent-less manner, in which tracing and analysis are performed without internal access to the VMs. This is a critical point, since access to the VMs is usually restricted in a public cloud. Second, we propose a two-phase feature selection and clustering technique that enables both coarse- and fine-grain workload characterization. Third, we go even further into the analysis by identifying the top important processes running on each VM, using a page ranking algorithm. This puts the irrelevant processes aside and thus provides a more distinct VM workload grouping. Fourth, we evaluated our proposed approach on a real dataset, including different software applications (e.g., mySQL, Apache) and by considering various scenarios such as overcommitment of resources and other possible problems. Our database is available online to be used in future studies and to the benefit of other developers [5]. In addition, we implemented different graphical views as follows: (1) a VM similarity plot that depicts the level of similarity between VMs; (2) a vCPU graphical view that presents a timeline for each vCPU and depicts different states of a VM for further analysis, after VM workload grouping; and (3) a process connectivity graph, which shows the communication between significant processes.

The rest of this article is organized as follows. Section 2 presents previous approaches on VM feature extraction and clustering. Section 3 introduces some background about virtualization technology and presents the different states of the processes running inside the VMs and their requirements. In addition, it describes the algorithm used to detect different states of the vCPUs, along with our feature extraction approach. Section 4 explains the details of the VM workload clustering model. Section 5 presents our experimental results along with a discussion on the essential behavior of different clusters. In Sections 6 through 8, we compare these approaches in terms of overhead, ease of use, and limitations. Section 9 concludes the article with possible future directions.

2 RELATED WORK

In this section, we present the available techniques for extracting workload-related metrics and survey the existing VM performance monitoring approaches in cloud infrastructures.

The VM workload metrics could be collected in two ways: agent based and agent-less. The agent-based techniques execute a daemon inside VMs to gather and send metrics to an external module for detailed analysis. Many available approaches [17, 35] monitor running applications by injecting an agent into a VM. The agent usually relies on common Linux APIs or simply employs the existing monitoring tools such as `iostat` [2] and `vmstat` [6].

However, the agent-less techniques gather metrics without the requirement of having an internal access to the VMs. One example of such technique is the VM Introspection approach [4], which analyzes the VM memory space. Some studies (e.g., [22, 23]) have used the VM Introspection technique for analyzing the performance of the VMs. Another approach to collect metrics in an agent-less manner has been proposed elsewhere [26–28], where hypervisor-level tracing analysis was employed.

Canali and Lancellotti et al. [13] exploited the correlation between the usage of multiple resources to determine which VMs were following the same behavioral pattern. Then, the K-Means approach was applied to cluster together similar VMs in an IaaS cloud data center. Their method sampled resource utilization metrics, such as CPU, memory, disk, and network usage, with an assumption of no knowledge on the processes running on the VMs.

In other work by Canali and Lancellotti [14, 15], the Smoothing Histogram based approach was proposed to group VMs showing similar behavior in a cloud environment. The monitoring metrics were periodically collected from the VM resource usage using the hypervisor APIs. The Bhattacharyya distance has been applied to measures the similarity between two VMs based on the probability distributions of resource usage. A spectral clustering based approach was employed

to categorize VMs into distinct groups. Their work focused on monitoring scalability by selecting a few representative VMs as the VMs closest to the clusters centroids. Further, Canali and Lancellotti [16] employed a similar strategy to propose a VM placement technique, namely class-based placement. The class-based method leverages the knowledge of VM classes to select a few representatives and reduce the size of the problem to be solved. The solution obtained was then replicated as a building block to solve the global VM placement problem. In the work of Rodrigues et al. [30], a placement algorithm is presented to reduce communication cost. In this work, network metrics are being studied and are being used to place the VM. In addition, in the work of Yao et al. [33], a method based on page ranking is proposed to maximize the utilization of a physical machine. None of these works use a method to collect data from real VM to be able to cluster and find the best fit for the VM.

In another work [9], a K-Means clustering based approach was proposed to monitor the cloud security against various anomalies, like intensive resource usage and malware attacks. System-level metrics, such as CPU, Memory, Disk, and network usage were collected for every monitored VM. Abdelsalam et al. [8] included the fine-grained information of each process in a VM for malware detection in cloud infrastructures. The proposed method trained a 2D Convolutional Neural Network model on performance metrics gathered from processes running in the VMs. A workload prediction-based multi-VM provisioning mechanism is presented by Li et al. [31] based on an ARIMA workload predictor with dynamic error compensation (ARIMA-DEC) and a time-based billing-aware multi-VM provisioning algorithm (TBAMP).

Zhang et al. [35] proposed a density-based clustering method (DBSCAN) for abnormal behavior detection in large-scale clouds. The performance-related metrics, such as CPU usage, memory utilization, and disk usage, were collected by BOSH agents. An entropy-based feature selection technique was used to reduce the data dimensionality.

All of the studies examined so far extracted system-level metrics, from process monitoring metrics (e.g., thread pool size) to resource utilization ones (e.g., CPU, memory, disk, or network usage). Other approaches have gone further by using lightweight tracing tools to gather detailed information on the underlying kernel and user-space executions.

Zhang et al. [34] collected kernel event traces of all active processes in a cloud infrastructure and then pushed them into a central log store. An event sketch modeling module converted the raw event traces to a group of kernel events having causality relationships. A set of statistical and data mining analysis tools were offered to summarize the event sketches and perform cloud performance diagnosis.

Tracing VMs was used by Abusitta et al. [10] to detect Denial of Service attacks in cloud infrastructures. A trace abstractor created high-level features from the statistics of the low-level events, such as CPU usage and the number of HTTP connections. The Support Vector Machine algorithm was then applied to detect changes in VM behaviors. Another work by Dean et al. [17] used kernel-level tracing for anomaly detection in cloud services. The proposed method, namely PerfCompass, was a statistical approach for finding external and internal faults in an online IaaS. PerfCompass was able to identify the top affected system calls and provide useful diagnostic hints for detailed performance debugging.

As mentioned in this section, none of the available cloud diagnosis approaches applied an agent-less technique with a lightweight tracing strategy. We believe that using detailed low-level information gathered by a lightweight tracer, in an agent-less manner, and clustering VMs with similar behavior could improve VM workload analysis. This improvement would be in terms of both security (agent-less technique) and overhead (lightweight kernel-level tracing).

Early results of this work were presented by Nemati et al. [21], which introduces an agent-less performance analysis approach to group VMs based on their workloads. In the current work, this

is expanded, notably by exploiting the PR algorithm (initially proposed for website ranking) for VM workload analysis. This enables us to do performance analysis by considering only the most significant processes running on the VMs. Our currently proposed strategy reveals more detailed information regarding the VMs resource utilization and provides more distinct VM clusters. In addition, our proposed method requires collecting data at the hypervisor level, without an internal access to the VMs. Therefore, our approach needs a simpler tracing infrastructure as compared to other solutions and adds less virtualization overhead.

3 VM METRICS AND FEATURES EXTRACTION

This section first presents some common definitions and terminology in the world of VMs, along with some background information about virtualization. Then, the **Wakeup Reason Analysis (WRA)** algorithm and **Process Ranking Algorithm (PRA)** are defined. The WRA algorithm is designed to detect different vCPU states in VMs and collect metrics for VM analysis. The PRA is developed to find the significant processes inside a VM by adopting a page ranking algorithm.

3.1 VMX Operation

Virtualization is supported by both Intel and AMD processors as the **Virtual Machine Extensions (VMX)** and Secure Virtual Machine instructions sets, respectively. Two types of VMX operation exist: VMX root and VMX non-root. Non-privileged instructions are executed as non-root, whereas privileged instructions are executed as root mode. Therefore, when a privileged instruction comes, the control is passed to the **Virtual Machine Monitor (VMM)** to execute the instruction in a safe environment. After handling the privileged instruction, the VM resumes to VMX non-root mode. The transition between VMX root to VMX non-root is called a *VM entry*, whereas the transition between VMX non-root to VMX root is called a *VM exit*. The VMM monitors these transitions by updating an in-memory **Virtual Machine Control Structure (VMCS)**. The VMM creates different VMCSs for each vCPU and uses `VMREAD`, `VMWRITE`, and `VMCLEAR` instructions to update VM information. Each VM exit comes with an exit reason number, which provides useful information about applications running inside a VM. The VMCS contains several important fields that are used in this article, including CR3, which points to the page directory of a process in VM, and SP, which points to the stack of the thread inside the VM. Therefore, CR3 and SP can uniquely identify a process or a thread, respectively.

3.2 Virtual Interrupt Injection

VMX instructions on x86 processors support virtual interrupts (`virq`) injection into VMs by filling the VM-entry interrupt-information field in the VMCS. Interrupt injection includes software interrupts, internal interrupts, and external interrupts. Then, the processor uses the interrupt information field to deliver a virtual interrupt through the VM Interrupt Descriptor Table. Usually, the `virq` will be injected into the guest during the next VM exit by emulating the `LAPIC` register. Once the VM resumes, the pending interrupt is executed by looking up the VM Interrupt Descriptor Table. After handling the interrupt, during the next VM exit, the processor performs an End of Interrupt virtualization to update the guest interrupt status. Looking at the injected interrupt reveals useful information about the application running inside the VM. We elaborate more on the extracted metrics from the injected interrupt `virq` in the next section.

3.3 VM Machine States Analysis

Different states of a vCPU on a VM and the conditions to reach them are shown in Figure 1. Each vCPU, similar to a **physical CPU (pCPU)**, could be either in a running or idle state. As mentioned in previous sections, a VM exits to the VMX root mode to execute a privileged instruction and then

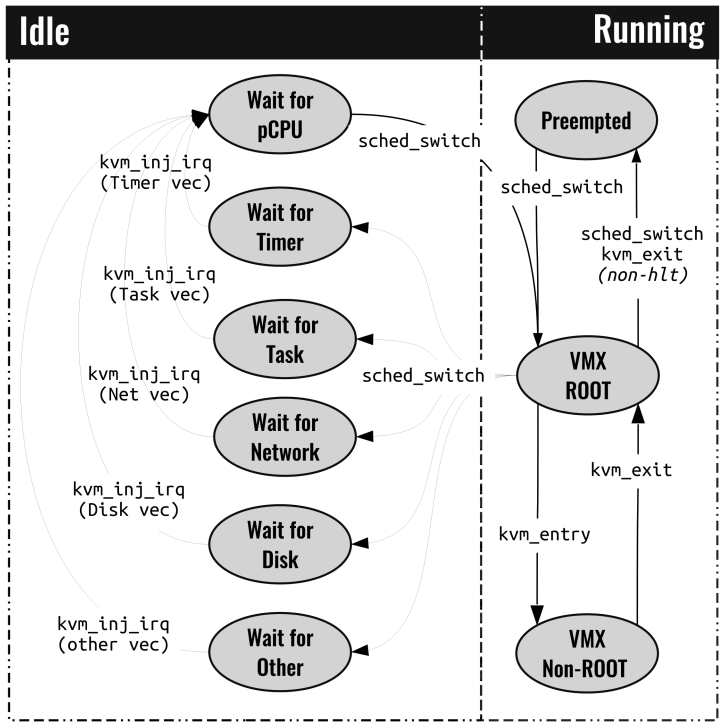


Fig. 1. VM's vCPU state transition.

resumes to the VM non-root mode with VM entry. From a VM point of view, the whole resources are allocated to that VM, whereas in reality, each VM shares the resources with other VMs and the host. This causes resource contention and leads to a preemption state. The preemption state occurs when the host scheduler takes the pCPU and gives it to another VM or process. In this state, the executing VM is halted for a while, without being notified. As a result, the VM process is in the running state inside the VM but does not execute any instruction. The preemption state is one of the states that cannot not be shown from inside a VM. The idle state occurs when a vCPU is being scheduled out voluntarily by sending a `hlt` signal, which is a privileged instruction. This instruction causes an exit to the VMX root mode and notifies the host scheduler to schedule the vCPU thread out of the pCPU. In this state, the vCPU waits for a wakeup signal from the host scheduler. Generally, the following main reasons cause the host scheduler to wake up the vCPU thread: (1) Timer interrupt, in which a process inside the VM sets a timer and the timer is fired; (2) IPI interrupt, in which inter-processor communication wakes up a process; (3) Network interrupt, in which a process inside the VM waits for an incoming packet from a remote process; and (4) Disk interrupt, in which a process inside the VM waits for Disk. In addition to the Timer, IPI, Network, and Disk interrupts, the VM could wait for another device with a different `virq` number. We label this state as "wait for other." Thus, the wait for X states as well as the running state will be used in our feature extraction technique to collect information about VMs.

We propose the WRA algorithm to detect the reason for waking up. Algorithm 1 shows the pseudocode for the WRA algorithm. The input to the algorithm is a sequence of events and the output is the detected wait for vCPU reason of a given VM. Whenever a VM has something to execute, it asks for the pCPU from the host scheduler. Receiving a `sched_wakeup` event shows

that the VM asked for a pCPU, and thus the state of the vCPU is updated to waiting for pCPU. Event `sched_in` shows that a vCPU is scheduled on a pCPU, but it is still in a waiting state. We label this state as "wait for vCPU."Before running the vCPU on a pCPU, the hypervisor injects interrupts into the VM. These interrupts are the key in our analysis. The event related to interrupt injection is `vm_inj_virq`. The vec field in this event is compared with the Task, Timer, Disk, and Network interrupt number. The reason for waiting is updated based on the *vec* number. The different interrupt numbers are retrieved by following the pattern of using devices. This pattern is different between hypervisors. The `vcpu_enter_guest` event changes the vCPU state to running (line 20). We also store the CR3 value in the payload of this event. CR3 points to the page directory of a process, in any level of virtualization, and can be used as an unique identifier for a process. Note that in our algorithms, we used hash map data structure to store last exit reason (used by getLastExit($vCPU_j^{id}$)) and as well as last CR3 (used by ($vCPU_j^{id}$)). As mentioned before, a running vCPU can be preempted either in host level (L0) or guest level (L1). If the last exit is `hlt`, which means that the VM runs the idle thread, the state will be modified as "wait for a reason."In the other case of exit number, the state will be changed to preempted (line 5). The preemption of L1 happens when the process inside the VM is preempted by the guest scheduler. In this case, the CR3 will be changed without running the `hlt` instruction (line 19).

3.4 Process Ranking Algorithm

To find the significant processes inside a VM, we adopt the PR algorithm [32], which has been used by the Google search engine to select significant web pages. The PR algorithm counts the number of links to a web page to evaluate the importance of that page.

Let p be a process and Bp a set of processes that point to process p. Then, let w be a process and Nw the number of links from w. Finally, let c be a normalization factor for the total rank of all processes. Then the rank (R) of p is computed as

$$R(p) = c \sum_{w \in B_p} \frac{R(w)}{N_w}. \tag{1}$$

The pseudocode for the PRA is presented in Algorithm 2. It mainly uses the `sched_ttwu` event to identify the callee (Wakee) and caller (Waker) processes (line 5). Then, in cases where the injected interrupt is `IPI` (line 9), it creates a link between the two processes (line 13) in our process connectivity graph. Using `sched_ttwu` event follows with finding injected interrupt IPI helps us find out interconnected processes. The same approach is being used in the work of Nemati et al. [29] to find critical path of a thread inside a VM using host trace. Next, the PRA computes the rank of each process using process connectivity graph and our adopted page ranking approach (line 15).

Then, the weakly connected sub-graphs are extracted by ignoring edge direction. Note that weakly connected components in a directed graph are equivalent to connected components in undirected graphs if the direction of edges is ignored. A traversal algorithm, either depth-first-search or breadth-first-search, can be used to find the weakly connected components starting from every unvisited vertex. Here, the breadth-first-search method is used as a default traversal algorithm. However, the use of the depth-first-search method would produce the same output. The resulting connected components represent processes that are grouped together if they wake up each other at least once.

The de-noise function in line 17 receives the connected groups of processes along with their rank and removes the groups that do not contain the top five processes. In this work, based on our experiment, we aggregate the metrics of the groups of processes that contain the top five top

ALGORITHM 1: Wakeup Reason Analysis (WRA) Algorithm

```
1  if event == sched_in then
2      vCpu_j^state = root;
3      lastExit = GetLastExit(vCpu_j^tid);
4      if last_exit != hlt then
5          vCpu_k^state = preempted_L0
6      else
7          vCpu_k^state = waiting_for_reason
8  else if event == vm_exit then
9      pCr3 = GetLastCr3(vCpu_k^tid);
10     PutLastExit(exit_number);
11     vCpu_j^state = root;
12     pCr3^state = root;
13 else if event == sched_wakeup then
14     vCpu_j^state = pCpu_waiting;
15 else if event == vm_entery then
16     lastCr3 = GetLastCr3(vCpu_j^tid);
17     if lastCr3 != cr3 then
18         vCpu_j^internal = preempted_L1;
19         lastCr3^state = preempted_L1;
20     vCpu_j^state = non-root;
21     cr3^state = non-root;
22     PutLastCr3(cr3);
23 else if event == sched_out then
24     vCpu_j^state = idle;
25     pCr3^state = idle;
26 else if event == vm_inj_virq then
27     pCr3 = GetLastCr3(vCpu_k^tid);
28     if vec == TaskIrq then
29         vCpu_j^state = task;
30         pCr3^state = task;
31     else if vec == TimerIrq then
32         vCpu_j^state = timer;
33         pCr3^state = timer;
34     else if vec == DiskIrq then
35         vCpu_j^state = disk;
36         pCr3^state = disk;
37     else if vec == NetworkIrq then
38         vCpu_j^state = network;
39         pCr3^state = network;
40     else
41         vCpu_j^state = other;
42         pCr3^state = other;
```

Table 1. Events and Their Payload Based on Host Kernel Tracing
($virq\#1 = Timer, virq\#2 = Task, virq\#3 = Disk, virq\#4 = Network$)

#	Events	#	Events
1	sched_in_vcpu(vcpu2, virq#1, P#4)	22	sched_in_vcpu(vcpu1, virq#1, P#5)
2	sched_out_vcpu(vcpu2, reason = hlt, P#4)	23	sched_out_vcpu(vcpu1, reason = hlt, P#5)
3	sched_in_vcpu(vcpu1, virq#1, P#5)	24	sched_out_vcpu(vcpu0, reason = hlt, P#1)
4	sched_out_vcpu(vcpu0, reason = hlt, P#1)	25	sched_in_vcpu(vcpu1, virq#4, P#3)
5	sched_out_vcpu(vcpu1, reason = hlt, P#5)	26	sched_in_vcpu(vcpu2, virq#1, P#4)
6	sched_in_vcpu(vcpu1, virq#3, P#2)	27	sched_in_vcpu(vcpu0, virq#1, P#5)
7	sched_in_vcpu(vcpu2, virq#1, P#4)	28	sched_out_vcpu(vcpu2, reason = hlt, P#4)
8	sched_in_vcpu(vcpu0, virq#1, P#5)	29	sched_out_vcpu(vcpu0, reason = hlt, P#5)
9	sched_out_vcpu(vcpu2, reason = hlt, P#4)	30	sched_in_vcpu(vcpu0, virq#1, P#5)
10	sched_out_vcpu(vcpu0, reason = hlt, P#5)	31	sched_out_vcpu(vcpu0, reason = hlt, P#5)
11	sched_ttwu(vcpu1, P#1)	32	sched_in_vcpu(vcpu2, virq#1, P#4)
12	sched_out_vcpu(vcpu1, reason = hlt, P#2)	33	sched_out_vcpu(vcpu2, reason = hlt, P#4)
13	sched_in_vcpu(vcpu2, virq#1, P#4)	34	sched_in_vcpu(vcpu0, virq#1, P#5)
14	sched_in_vcpu(vcpu0, virq#1, P#1)	35	sched_out_vcpu(vcpu0, reason = hlt, P#5)
15	sched_in_vcpu(vcpu1, reason = hlt, P#5)	36	sched_ttwu(vcpu1, P#1)
16	sched_out_vcpu(vcpu1, reason = hlt, P#5)	37	sched_out_vcpu(vcpu1, reason = hlt, P#3)
17	sched_out_vcpu(vcpu2, reason = hlt, P#4)	38	sched_in_vcpu(vcpu2, virq#1, P#4)
18	sched_in_vcpu(vcpu1, virq#1, P#5)	39	sched_out_vcpu(vcpu2, reason = hlt, P#4)
19	sched_out_vcpu(vcpu1, reason = hlt, P#5)	40	sched_in_vcpu(vcpu0, virq#1, P#1)
20	sched_in_vcpu(vcpu2, virq#1, P#4)	41	sched_in_vcpu(vcpu1, virq#1, P#5)
21	sched_out_vcpu(vcpu2, reason = hlt, P#4)	42	sched_out_vcpu(vcpu1, reason = hlt, P#5)

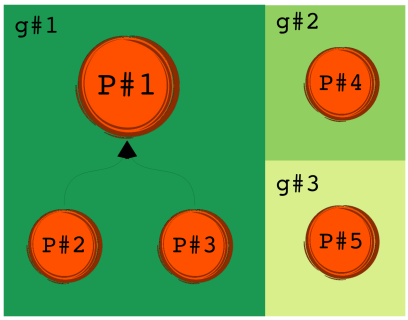


Fig. 2. Connectivity graphs for different groups of processes. Here, three groups of g#1, g#2, and g#3 are shown.

processes. In the case of choosing more processes, we might have more noises in our aggregated metrics. Our de-noise function improves the clustering since most of the works in the VMs are some services for the outside world. For example, assume a process running upon a VM sets a timer and does some job without interacting with the outside. In that case, it means the work is not critical and could be delayed.

For example, using events from Table 1, for $c = 1$ the page rank for different processes is computed as R(P#1)= 2, R(P#2)= 1, R(P#3)= 1, R(P#4)= 1, and R(P#5)= 1. As a result, P#1 is considered as a very important process. Then, we use the breadth-first-search algorithm [11] to find the connected processes. Figure 2 depicts different groups of processes in our example. As shown, there are three groups and the g#1 group contains the significant process P#1.

To better understand the relationships between detecting significant processes and the PR algorithm, we provide an example. Here, we consider a simple algorithm to detect the vCPU execution period that uses `sched_in` and `sched_out`. For simplicity, we grouped a sequence of events into one event in Table 1. `sched_in_vCPU(vcpu, vec, cr3)` represents the sequence of `sched_in(vcpu)`, `vm_inj_virq(vec)`, and `vm_entry(cr3)` events. Furthermore,

ALGORITHM 2: Process Ranking Algorithm (PRA)

```
if event == sched_ttwu then
    vCpu_k^tid = GetVmVcpu(waker_tid);
    wakerCr3 = GetLastCr3(vCpu_k^tid);
    j = GetVmVcpu(wakee_tid);
    UpdateWaker(vCpu_j^tid, wakerCr3);
else if event == vm_inj_virq then
    j = GetVmVcpu(tid);
    pCr3 = GetVmVcpu(vCPU_j^tid);
    if vec == IpiIrq then
        wakerCr3 = QueryWaker(vCPU_j^tid);
        graph.AddNode(wakerCr3);
        graph.AddNode(pCr3);
        graph.Link(wakerCr3, pCr3);
for all process cr3 ∈ graph.nodes do
    R(cr3);
subGraphs = BreadthFirstSearch(graph);
customizedGraph = denoise(subGraphs, R(cr3));
```

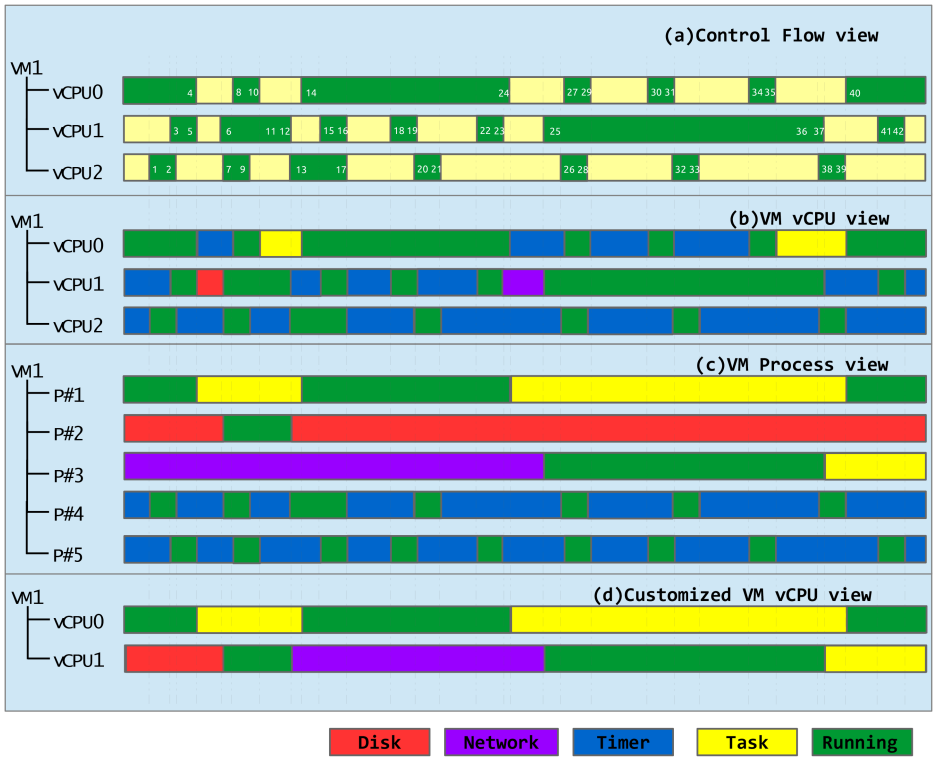


Fig. 3. Example.

`sched_out_vCPU(vcpu, reason)` shows the sequence of `vm_exit(reason)`, `sched_out(vcpu)` events.

Figure 3(a) presents the vCPU detection algorithm using a sequence of events shown in Table 1. The first event, `sched_in_vCPU` event, shows that the vCPU2 is being scheduled in on pCPU and the vCPU state goes from the blocking to the running state. Then, the second event (`sched_out_vCPU` event) shows that the pCPU is yielded and the state changes to blocking. This algorithm is simplistic and just represents the running and blocking states. It cannot detect the reason of the blocking state. WRA is used to detect the reason for blocking.

Figure 3(b) presents a vCPU view using the events in Table 1. The key idea is that the event indicating the cause of the blocking state is unknown *a priori*. The WRA algorithm uses the virtual injected interrupt to reveal the blocking state. For example, event #6 depicts that the vCPU1 is being scheduled in and the reason for blocking was waiting for Disk.

Figure 3(c) presents a process view using the WRA algorithm. The WRA algorithm uses CR3, which is the base pointer to the page table of a process, to identify the running process on the vCPU. For example, event #25 shows that vCPU1 is scheduled in and the reason for blocking was Network. It also depicts that P#3 is the process that was waiting for the packet to receive from the network.

Figure 3(d) shows the customized vCPU view, using the significant processes detection algorithm. The PRA computes the process rank and filters the processes that are not significant. Then, it aggregates their features and builds a new VM feature. For this example, we consider the group that contains the first significant process. In this case, the customized VM vCPU view shows the aggregated states for P#1, P#2, and P#3.

3.5 Feature Extraction and Feature Selection

In this section, the metrics and features extracted from the VMs are explained. Here, the events of each VM are monitored by our agent-less tracing mechanism. Since each minute of tracing data could include thousands of events, we need to apply a feature extraction strategy to have a more compact data representation. Thus, based on our domain knowledge, we define the feature extraction approach with several components, as shown in Figure 4. These features provide a simple representation of the complex original data that could lead to a more suitable input for the learning algorithm. In addition, we apply a feature selection strategy to feed a subset of the initially selected features to the clustering phase. The advantage of feature selection is when the original features are very diverse and might mislead the learning algorithm.

3.5.1 Trace Data Collection. Many tracers are available for Linux kernel that could be used for our agent-less monitoring agent. In this project, the Linux Trace Toolkit Next Generation (LTTng) [18] is used to monitor the workload on each VM. LTTng is a lightweight and open source tracing tool on Linux that provides detailed information on the interactions between the kernel and user space. In addition, LTTng is compatible with the KVM [3] as our hypervisor-based virtualization.

In our work, the collected tracing logs are fed to the Trace Compass [7] open source tool, which is designed to view and analyze traces using pre-built modules. Since Trace Compass does not support agent-less VM analysis, we have implemented the WRA algorithm as a module (available at GitHub [1]) in TraceCompass. Our designed WRA module collects useful information from streaming trace logs and stores it into a database called **State History Tree (SHT)**. The SHT data structure uses a disk-based format to store large interval data on permanent storage with a logarithmic access time [25]. Figure 5 shows the structure of the SHT designed in our analysis. For instance, each VM with its specific ID has different vCPUs, named by their CPU ID. Each vCPU

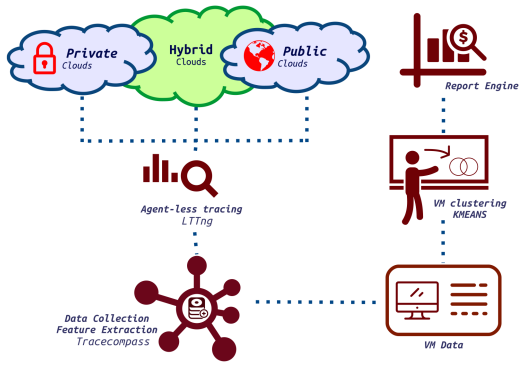


Fig. 4. Architecture of our implementation - Our Agent-less tracing mechanism takes the data from VMs and at the end our VM clustering method clusters VMs based on their workload

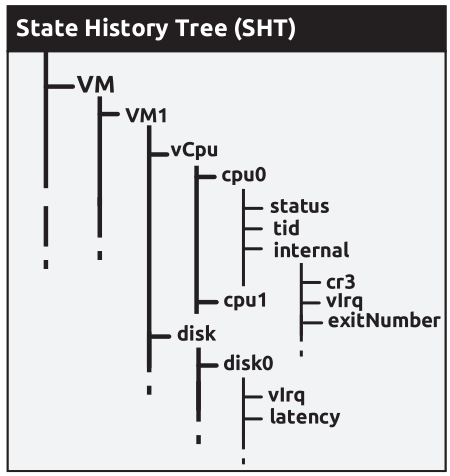


Fig. 5. SHT used to store different information of the VM.

can be in various states, as shown in Figure 1. In addition, the same information is stored for each process.

Once the WRA algorithm constructs the SHT structure, useful metrics are extracted by querying the SHT attributes. KVM threads on the host are shown as a normal host threads with some information related to their resource usage. Using such high-level metrics (resource usage) would help us to only classify the VMs based on the four important classes as CPU, Disk, Network, or compounds. But using tracing would reveal more data about behavior of internal VM threads as we are able to not only find out the CPU-intensive VMs but also classify them into Multithreaded VMs, CPU hungry VMs. In total, 25 metrics are considered per VM. Table 2 provides a complete list of the metrics, along with a short description. The W_X metric, where $X \in \{Disk, Net, Timer, Task\}$, is the average time that a vCPU waits for a signal to be woken up. The E_X with $X \in \{Root, non-Root\}$ metrics are the average time that a VM is running in a VMX root or VMX non-root mode. The f_X with $X \in \{Disk, Net, Timer, Task\}$ metrics show the waking up frequency of a vCPU by different signals. W_X, E_X, and f_X are calculated based on the status attribute of each VM in the SHT. The

Table 2. Complete List of Metrics Extracted from Streaming of Trace Data

Term	Features Collected by Tracing
W_{Disk}	Wait for Disk average time
W_{Net}	Wait for Net average time
W_{Timer}	Wait for Timer average time
W_{Task}	Wait for Task average time
E_{Root}	Root mode average time
$E_{non-Root}$	Non-root mode average time
f_{Disk}	The frequency of wait for Disk
f_{Net}	The frequency of wait for Net
f_{Timer}	The frequency of wait for Timer
f_{Task}	The frequency of wait for Task
I_{Disk}	Virtual disk irq injection rate
I_{Net}	Virtual network irq injection rate
I_{Timer}	Virtual timer irq injection rate
I_{Task}	Virtual task irq injection rate
FP_{VMVM}	The frequency of two VMs preempt each other
FP_{HostVM}	The frequency of host preempts a VM
FP_{VMProc}	The frequency of VM processes preempt each other
$FP_{VMThread}$	The frequency of VM threads preempt each other
N_{Exit}	The frequency of different VM exit reason
f_{Read}	The frequency of read from Disk
f_{Write}	The frequency of write to Disk
B_{Read}	Total Block Disk Read
L_{Read}	Total Latency Disk Read
B_{Write}	Total Block Disk Write
L_{Write}	Total Latency Disk Write

I_X, where $X \in \{Disk, Net, Timer, Task\}$, metrics are the frequency of injecting virtual interrupts into a VM.

4 VM WORKLOAD CLUSTERING MODEL

Since no labeled training data is available, we propose to apply the K-Means unsupervised algorithm [24] to cluster VMs into distinct groups. Unlike supervised classification approaches, which require a training dataset to assign a predefined label to a VM, a clustering approach groups VM workloads based on the actual workload distribution. Therefore, it potentially discovers interesting distinct VMs clusters, where each one could represent one specific workload type. For this purpose, each data sample x_i is defined as follows:

$$x_i = (f_{i1}, f_{i2}, \ldots, f_{ij}, \ldots, f_{iM}), \tag{2}$$

where M is the maximum number of workload-related metrics that are considered as features in our analysis. To enable meaningful comparisons, sample vectors are individually normalized using the $L2$ norm such that they become of equal magnitude [19]—that is,

$$\sum_{m=1}^{M} f_{im}^2 = 1, \ \forall i = 1..N. \tag{3}$$

The K-Means clustering algorithm finds the clusters $C_1, \ldots, C_K$ over the data matrix $X_{N,M}$, where N is the total number of VMs and M is the number of features. K-Means is a representative-based algorithm in which the clustering is performed based on a group of centroids (partitioning representatives). Centroids are hypothetical sample points placed at the center of each cluster. Given a certain number of centroids, K, a distance function can be used to assign the data points to their closest representatives. In the K-Means algorithm, the sum of squares of the Euclidean distances of data points to their closest centroids is used to quantify the objective function of the clustering as

$$\min_{C} \sum_{j=1}^{K} \sum_{x_i \in c_j} \|x_i - \mu_j\|^2, \tag{4}$$

where the set C is $\{c_1, \ldots, c_k\}$. The term c_j shows the set of points that belong to cluster j and μ_j is the mean of points in c_j.

An advantage of using the K-Means algorithm in our VM analysis approach is that each centroid will represent the workload of its cluster. This provides a condensed useful information about the behavior of many VMs.

Once the VM workload clustering is determined, it is important to evaluate its quality. As we know, clustering validation is often difficult in real datasets, since the problem is defined in an unsupervised way. Here, we use the silhouette score as an unsupervised validation criteria to interpret and evaluate the consistency of the clustering results. The silhouette coefficient measures how similar a sample VM x_i is to its own cluster (cohesion) compared to other clusters (separation). Let in_i be the average distance of x_i to other VM data points within its cluster and out_i be the minimum average distance of x_i to points in other clusters. The silhouette score s_i, specific to the ith sample, is computed as

$$s_i = \frac{out_i - in_i}{max(in_i, out_i)}. \tag{5}$$

The silhouette score ranges from −1 to +1, where large positive values indicate highly separated clustering and negative values represent some level of data points mixing from different clusters. Having in_i as close as possible to zero indicates a very cohesive cluster, and a large out_i value represents a large separation from the closest neighboring cluster. A negative score is undesirable, since it corresponds to the case where the average distance to in-cluster samples exceeds the minimum average distance to out-cluster data points—that is, $(in_i > out_i)$. It should be noted that, while the quality of a clustering algorithm plays an important role in finding good clusters, having cohesive and well separated clusters also largely depends on the nature of the data and the distance metric. Reporting the average silhouette coefficient over all points of a single cluster indicates how densely the points of the cluster are grouped. The average silhouette score over all data points of the entire dataset represents how appropriately all sample points are clustered.

A two-step clustering approach is employed in our analysis, where first all VMs are grouped into a set of clusters. Then, the second-stage clustering is applied to each discovered cluster in the first stage. Here, each VM would be characterized by a tuple (Ci, Cj), indicating the cluster it belongs to, in the first and second stage, respectively. This maintains a better grouping of VM workloads by first performing a coarse-grain characterization of VMs, followed by a more detailed clustering. Therefore, we believe that the idea of the two-stage clustering provides a good intuitive feel of both coarse- and fine-grain workload characterization. It should be noted that the number of clusters selection criteria is always guided by the total silhouette score. In other words, at any stage we select the clustering with maximum total silhouette score as the desired solution.

Table 3. List of Applications for Workload Generation

Application Behavior	Application Name	Application Description
CPU	Sysbench, Stress, Burn, Chess, Compress, encode, interBench, openSSL, smallpt, infinitLoop	Bunch of well-known benchmarking tools for Linux to stress CPU including computing prime number, compressing files, video and audio encoding, mathematic operations.
Disk	Sysbench, Compress, dbench, dd, interBench, hdparam, fsMarker, aio, ioZone, pgbench-disk, pqlight, stream, tioBench	Include different testing toolkits for Disk I/O. These toolkits provide realistic scenarios for reading/writing with different options to evaluate Disk I/O.
Net	Wget, iperf, netperf, netping, netSpeedTest, scp, ab, httperf	Include real network applications in Linux. It covers web server, file transfer server, downloading file.

5 EXPERIMENTAL EVALUATION

In this section, we present the experiments on our proposed two-stage clustering approach. Different types of workloads are generated using well-known Linux applications (e.g., MySQL, Apache, dd, Sysbench, netSpeed, wget). The list of applications along with their description is shown in Table 3. We broke down these applications into three categories based on their behavior. Then, we mixed them to generate more than 60 different workloads and scenarios. Our database is available online at GitHub [5]. As mentioned in Section 3, once feature metrics are collected, a de-noising PR-based algorithm can be used to filter out non-important processes. Most of the works in the VMs are service. The service has lots of communications with other processes and with the outside world. Assume a process running upon a VM sets a timer and does some job without interacting with the outside. In that case, it means the work is not critical and could be delayed. Here, we present the results of the VM clustering method with and without the de-noising strategy.

In the first experiment, we use the VM execution time in VMX root and VMX non-root states along with the virtual interrupt injection rate for the whole VM as our feature vector. As we show in the following section, the injection interrupt rate could expose the behavior of VMs since it shows the communication between processes and host devices. Note that our feature vectors are normalized, and their absolute values are of no significance.

At the first stage of the clustering, we realize that applying K-Means to detect three coarse clusters yields the best total silhouette score of 0.42, and per cluster silhouette scores of $C0 = 0.38, C1 = 0.35, C2 = 0.48$. Interestingly, this matches the reality since our generated workloads are divided into three types (CPU, Disk, and Network).

Figure 6 depicts the centroids of the first-stage clustering. The prototype usage pattern in Figure 6 shows a high CPU usage and high timer interrupt rate for the first cluster (C0), which could represent a CPU-intensive cluster. As mentioned before, the VMM executes the privileged instructions in a safe environment, VMX root mode. Then the VM resumes to VMX non-root mode after handling the privileged instruction. In Linux, the KVM module clears the CPU preemption flag before going to the VMX non-root mode. As a result, no one can preempt the VM in VMX non-root mode. To be able to yield the pCPU, the Linux scheduler kicks the VM by injecting external interrupts into the vCPU. This causes an exit to VMX root mode (i.e., exit number one) to handle the exit reason. In this state, the preemption flag is set to authorize the host scheduler to preempt

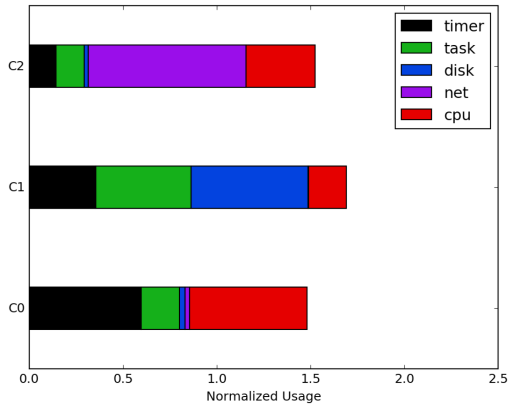


Fig. 6. Characteristic of workloads discovered in the first stage of clustering.

the VM. As a result, the CPU-intensive VMs have a high rate of timer interrupt, along with high CPU usage in VMX non-root mode.

The second cluster (C1) represents the Disk-intensive VMs. The prototype usage pattern of Disk-intensive VMs includes a large disk interrupt rate along with a high task interrupt rate. This is an indication of Disk-intensive VMs, waiting more frequently for some task to finish I/O. The third cluster (C2) is found to include Network-intensive VMs. The Network-intensive VMs have high network interrupt rates. The processes in Network-intensive VMs wait more for processes on other machines compared to Disk-intensive ones that wait more for processes in the same machine.

The result of the first stage can be used by the cloud infrastructure administrator to optimize resource usage. A host with too many VMs from the same cluster could encounter resource contention. For instance, assigning too many CPU-intensive VMs to one host machine causes preemption on pCPUs, which leads to latency. Thus, using our clustering method, the administrator can balance the host resource utilization and reduce resource contention.

In the second stage, we apply K-Means to each of the first-stage clusters independently and choose the clustering with the largest silhouette score. For our particular set of workloads, the first-stage cluster C0 turns out to be best characterized as five distinct groups, with a total silhouette score of 0.59. Figure 7 shows the similarity plot for the five clusters found in the second stage. Each point on the X and Y axes corresponds to a particular VM, which is labeled once on both axes. The second-stage cluster index corresponding to this VM is enclosed in brackets.

Each point on the similarity plot presents the normalized Euclidean distance, defined as in (6), between VM feature vectors intersecting at that point. Note that d_{ij} is the actual Euclidean distance between VM i and j, whereas d_{max} and d_{min} correspond to the maximum and minimum distances, respectively. The similarity score sim between any two VMs ranges from zero to one. Interestingly, sorting VMs according to their cluster index results in a similarity plot that resembles a block diagonal shape, for cohesive and well-separated clusters.

$$sim_{ij} = 1 - \frac{d_{ij} - d_{min}}{d_{max} - d_{min}} \tag{6}$$

Figure 7 illustrates such a block diagonal pattern, revealing how good the clustering is. This is also reflected in Figure 8(a) as the computed silhouette scores for each cluster show positive and mostly large values. Figure 9(a) depicts the characteristics of workloads in five different types of workloads. These groups mainly differ in the amount of CPU usage as well as timer and task interrupts. For example, there are two groups, (C0, C2) and (C0, C3), which have high timer in-

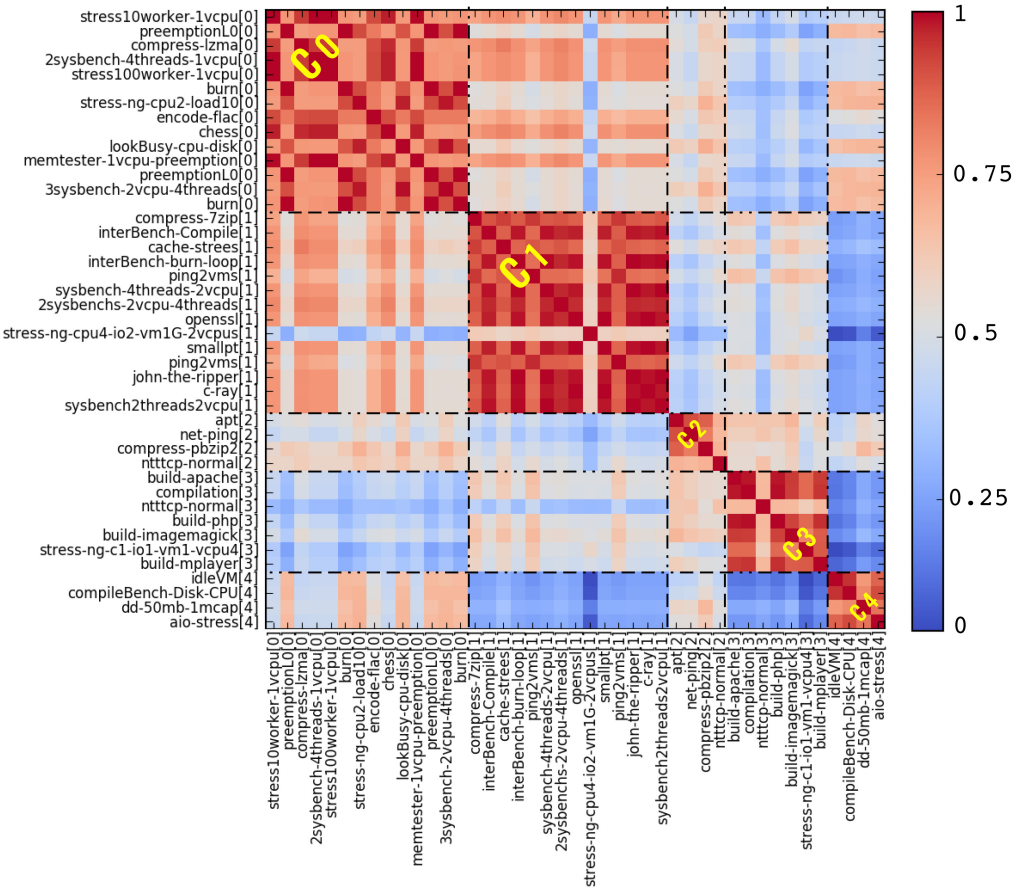
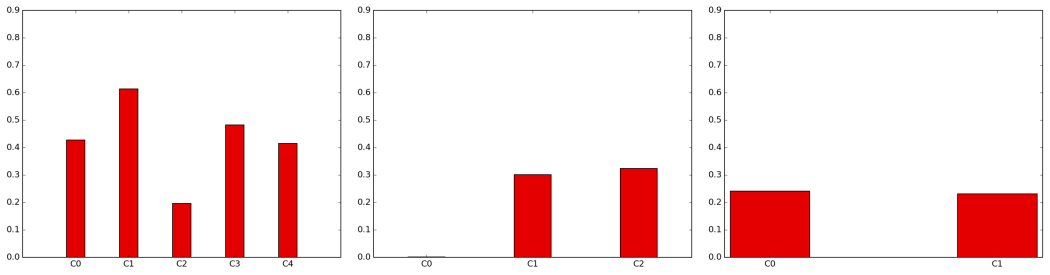


Fig. 7. Similarity plot for the five workload clusters discovered in the first group (i.e., (C0, Cj), $j \in [0, 4]$ (Silhouette = 0.59)).

terrupts, reflecting multiple process dependencies. Interestingly, (C0, C4) is almost entirely timer intensive, with minimal CPU usage, indicating that the VMs in this group represent mostly idle workloads. Instances in this group do not cause any contention and could be deployed in any machine.

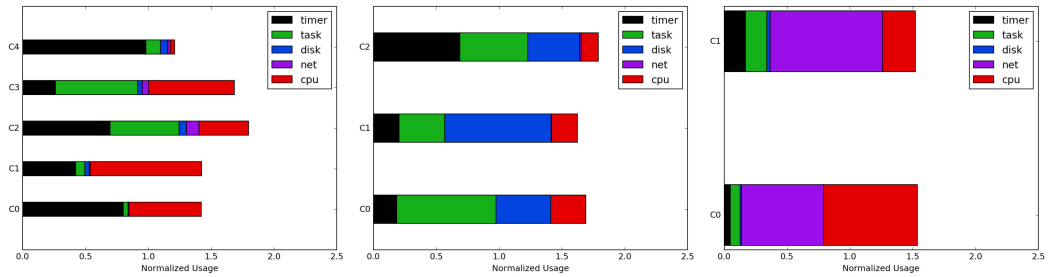
Let us elaborate on a few interesting observations from the similarity plot for C0. The first interesting observation is regarding VMs in clusters (C0, C3) that have a high task interrupt rate. The processes inside these VMs are either dependent on other processes or dependent on their own threads. Interestingly, all VMs that build an application are in this group, since making a file depends on compiling other files. Figure 10 shows an example of this group. As shown, processes wait for other processes to finish their job.

Moreover, there is a moderate similarity among VMs in clusters (C0, C1) and (C0, C0). This is visible from the moderately warm areas at the intersection of these two clusters. Further inspection of the VMs reveals that these two groups are both CPU intensive, with negligible task dependency. It means that both groups have almost single-threaded processes. This is clearly observed from the centroids for C1 and C0 in Figure 9(a), further showing a sharp contrast in CPU and timer composition among the two clusters. VMs in cluster (C0, C0) have a special behavior. Figure 11



(a) (C0, Cj), $j \in [0, 4]$ (b) (C1, Cj), $j \in [0, 2]$ (c) (C2, Cj), $j \in [0, 1]$

Fig. 8. Silhouette score of VM clusters discovered in the second stage of clustering. Cj represents the second-stage cluster index.



(a) (C0, Cj), $j \in [0, 4]$ (b) (C1, Cj), $j \in [0, 2]$ (c) (C2, Cj), $j \in [0, 1]$

Fig. 9. Characteristic of workloads discovered in the second stage of clustering. Cj represents the second-stage cluster index.

depicts one instance of this cluster. As shown, it executes a small amount of code and then waits for a timer to be fired. Other VMs could be affected by this catastrophic behavior. Some instances of this group do not utilize the pCPU and preempt other vCPUs running on the pCPU. Thus, all instances with preemption at the host level are in this group. In contrast, VMs in cluster (C0, C1) are highly CPU intensive and utilize the pCPU all the time. Despite this overlap, Figure 8(a) shows that the silhouette score for (C0, C1) is significantly larger than (C0, C0), suggesting that C1 is a better cluster. This is because (C0, C1) is a more cohesive cluster compared with (C0, C0), as suggested by the very warm texture of (C0, C1). This implies that the VMs in this group are very similar and closely resemble the prototype centroid representing this cluster. This information can be useful for the cloud administrator when dealing with deployment of these very similar workloads. The VM deployment strategy should satisfy both IaaS providers and VM users. It should guarantee an efficient usage of host resources while avoiding resource contention between VMs.

Another interesting observation is that two particular VMs in (C0, C3) and (C0, C1) are quite dissimilar to their co-cluster VMs. These are the `ntttcp-normal` workloads, which moderately use the network but are put into these clusters due to making significant use of timer and task as well as CPU resources. Although it is debatable to put these workloads in the same group with non-network VMs, the similarity plot can easily visualize such peculiarities for the system administrator.

The second and third stage-one clusters are also further processed into three and two groups of workloads by our approach. Figures 12 and 13 provide similarity plots for C1 and C2, respectively. As for (C0, C3) and (C0, C1), (C2, C1) also includes a rather dissimilar VM, namely `host-slow-disk`,

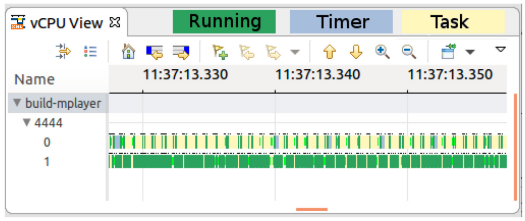


Fig. 10. vCPU view for the build-mplayer VM that has a high task dependency and high CPU utilization.

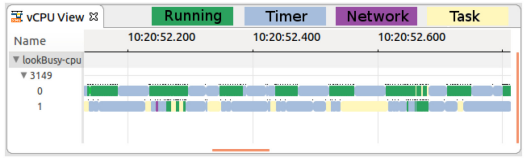


Fig. 11. One of the instances from cluster (C0, C0).

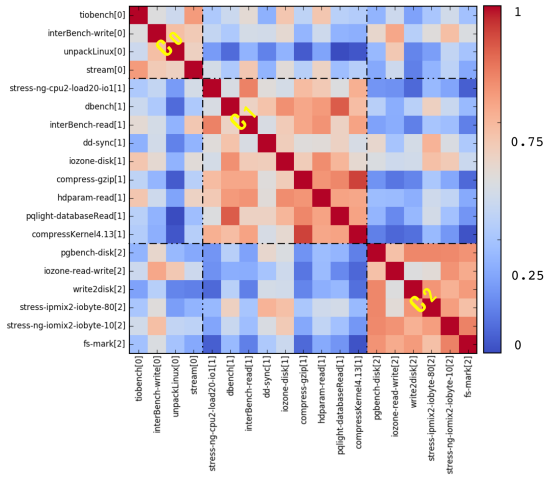


Fig. 12. Similarity plot for the three workload clusters discovered in the second group (i.e., (C1, Cj), $j \in [0, 2]$ (Silhouette = 0.40)).

which, although highly network intensive, also makes moderate disk usage. The `host-slow-disk` VM executes `scp` to copy a big file from one VM to the host. For this special VM, we put a cap on disk usage to slow down the data reading. This is realistic, since many IaaS providers like Amazon limit number of Block I/O accesses to reduce disk contention. As a result, we could call the VM more network intensive rather than Disk-intensive. Therefore, it lies somewhere between the Disk-intensive and Network-intensive groups, but closer to the latter one.

Another peculiar case is (C1, C0) with a cluster silhouette score of almost zero, as indicated in Figure 8(b). Inspecting the similarity plot of Figure 12 for C1 reveals a rather loose cluster in this case (i.e., C0). Although Figure 9(b) shows that the centroid for this cluster (C0) is highly task dependent, further inspection of the feature vectors extracted for the VMs in (C1, C0) reveals that they are rather diversely affected by timer interrupt and CPU usage. Such cases, with low cluster quality, are generally not of interest from a resource management point of view. However, they

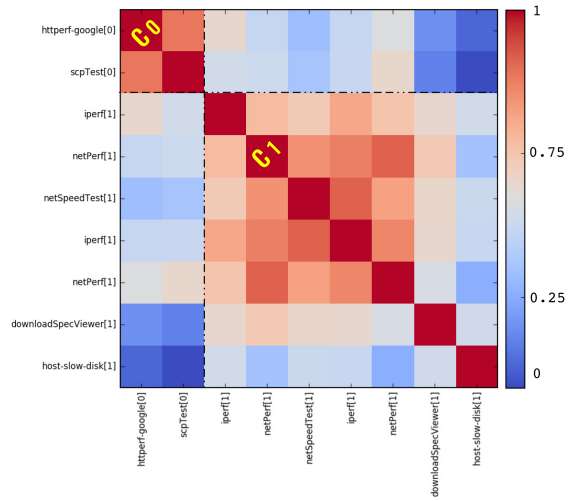


Fig. 13. Similarity plot for the two workload clusters discovered in the third group (i.e., (C2, Cj), $j \in [0, 1]$ (Silhouette = 0.46)).

can still be used to investigate the root cause of performance issues through their representative prototype workload.

VMs in cluster (C1, C2) read or write a large number of files at once and then wait for some time and then continue. In contrast, VMs in cluster (C1, C0) read small data chunks. Reading small files is extremely harmful for other VMs since it issues a number of disk requests to the host level and causes contention.

Let us now elaborate on the workload prototypes discovered in the second stage, which are shown in Figure 9(a), (b), and (c), corresponding to first-stage clusters C0, C1, and C2, respectively. Recall that C0 included CPU-intensive VMs, whereas C1 and C2 were discovered to be disk and Network-intensive, respectively. We start from the Network-intensive group C2, which was found to have only two clusters of workloads: (C2, C1), which is highly Network-intensive, and (C2, C0) with also significant CPU usage. The latter group should be allocated adequate CPU resources to prevent bottlenecks, whereas the former group can be co-located with CPU-intensive VMs on a server that otherwise has no network usage. Hence, although VMs in (C2, C0) are compatible with those of all other first-stage groups, (C2, C1) may only be co-located on physical servers hosting (C1, C2) or (C0, C4).

As for the Disk-intensive group, we have discovered three clusters with contrasting behaviors in the composition of disk and task usage as well as timer interrupt rates. More specifically, we observe two contrasting sets of workloads (C1, C1) with large disk usage but small inter-task dependencies and (C1, C0) with moderate disk usage and significant task dependencies. Alternatively, (C0, C0) and (C0, C1) make heavy use of CPU and timer resources. This indicates that these VMs contain contending processes that are CPU intensive.

In the next experiment, we applied the de-noising PRA on our data to find significant processes in each VM. Then, we built a new feature vector based on significant processes. Similar to previous experiments, we followed a two-stage clustering strategy.

Figure 14 illustrates the similarity plot for the C0 cluster that is CPU intensive. The first stage cluster C0 turns out to be best characterized by five distinct groups, with a total silhouette score of

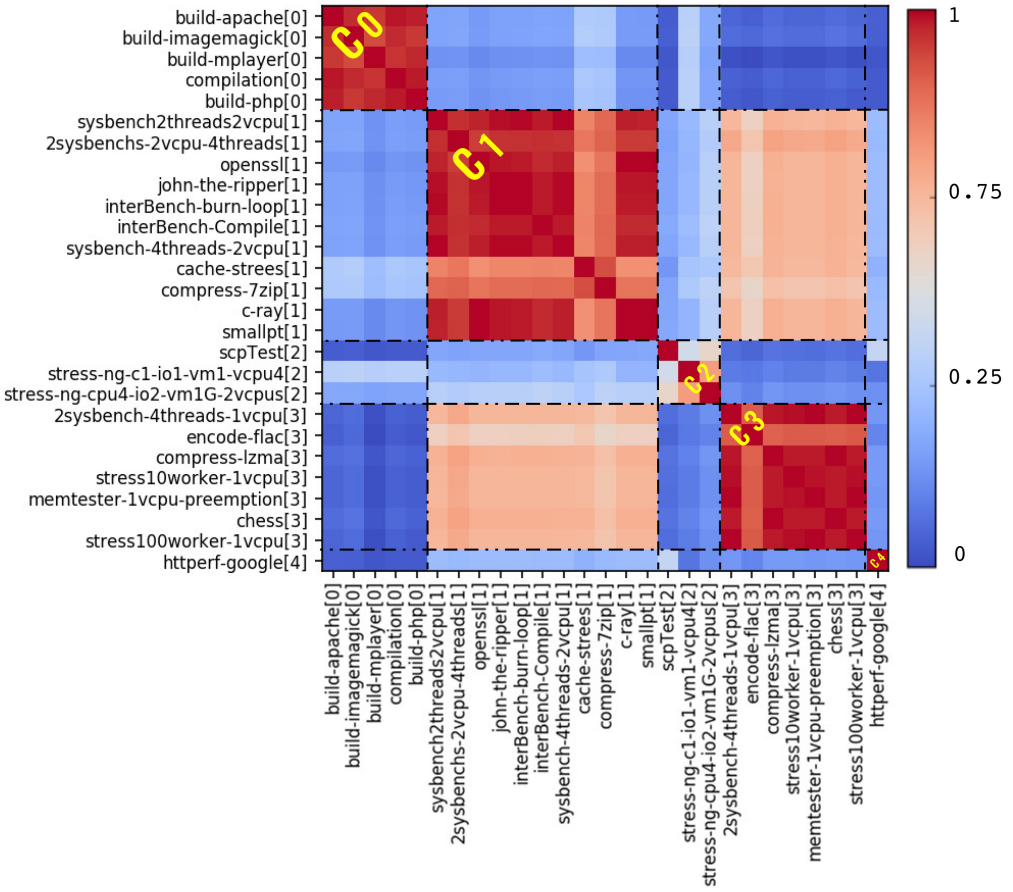


Fig. 14. Similarity plot for the five workload clusters discovered in the first group using PRA (i.e., (C0, Cj), $j \in [0, 4]$ (Silhouette = 0.64)).

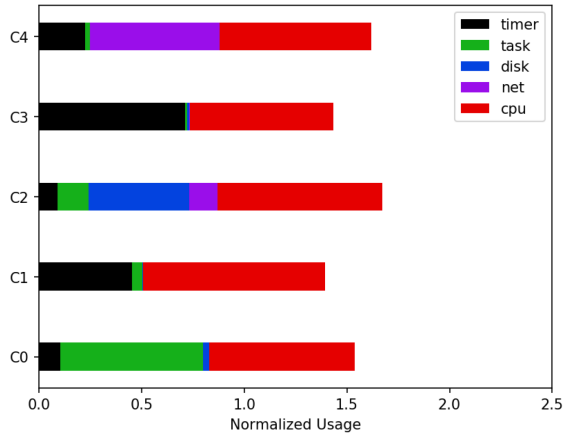


Fig. 15. Workload characteristic of the C0 cluster discovered in the second stage of clustering based on the PRA.

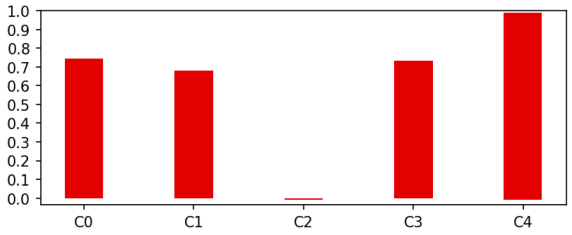


Fig. 16. Silhouette score of the C0 cluster discovered in the second stage of clustering using the PRA.

0.64. According to Figure 14, there is low inter-cluster similarity, which is clear from the moderately cold areas at the clusters' intersections.

There is almost no similarity among VMs in clusters (C0, C1) and (C0, C0). Further investigation of the VMs reveals in clusters C1 and C0 that these two groups are both CPU intensive. However, C0 VMs are executing more task-dependent processes, whereas VMs in C1 have a high timer interrupt rate.

Moreover, there is a moderate similarity among VMs in clusters (C0, C1) and (C0, C3). This is visible from the moderately warm areas at the intersection of these two clusters in the similarity plot of Figure 14. Further inspection of the VMs reveals that these two groups are both CPU intensive, with high timer dependency. This means that both groups have almost single-threaded processes. This is clearly observed from the centroids of C1 and C3 in Figure 15, further showing a sharp contrast between CPU and timer workloads among the two clusters.

Another special cluster is (C0, C2), with a cluster silhouette score of almost zero, as indicated in Figure 16. The similarity plot of Figure 14 for C2 reveals a rather loose cluster in this case (i.e., C0). Although Figure 15 shows that the centroid for the cluster C2 is highly CPU dependent, further inspection of the feature vectors extracted for the VMs in (C0, C2) reveals that they are rather diversely affected by Disk interrupt, Network interrupt, and CPU usage. As a result, we could call this cluster a compound cluster (CPU, Disk, Network). Such cases with low cluster quality, but diversity in interrupt rates, are generally interesting from a resource management point of view, as they represent how resources are utilized.

6 OVERHEAD ANALYSIS

In this section, the overhead of the WRA algorithm is compared with the agent-based feature extraction method (FAE). To compare the two approaches, we enabled the tracepoints that were needed for extracting the same features from inside of the VM. In addition, it is worth mentioning that our WRA algorithm needs only to trace the host. For the File I/O category, we configured Sysbench to read and write randomly 150 GB of data for a maximum of 300 seconds. Then, we published the average time to serve each IO request. In our experiment, the 95th percentile for our baseline was on average 589.28 ms. For the Memory category, we configured Sysbench to write 10 GB randomly to memory. In our experiment, the 95th percentile for our baseline was on average 615.46 ms. For the CPU category, we configured Sysbench to calculate 2,000,000 prime numbers for a maximum of 300 seconds. In our experiment, the 95th percentile for our baseline was on average 493.72 ms. As shown in Table 4, the FAE approach adds more overhead in all tests, since it needs to trace the VMs and the overhead of virtualization will be added. Note that this table shows the overhead of adding a suitable tracepoint for each algorithm. We used the Sysbench benchmarks to reveal the overhead of both approaches, since Sysbench is 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. Our analysis is a postmortem analysis, meaning that we first trace and then

Table 4. Overhead Analysis of the WRA Algorithm Compared with the Agent-Based Feature Extraction Method

Benchmark	Baseline	AFE	WRA	Overhead	
				AFE	WRA
File I/O (ms)	450.92	480.38	451.08	6.13%	0.03%
Memory (ms)	612.27	615.23	614.66	0.48%	0.39%
CPU (ms)	324.92	337.26	325.91	3.65%	0.30%

run our algorithm afterward. All other algorithms would be run either on a separate machine or on separated cores. This approach helps us to not add extra overhead to the VM itself.

7 EASE OF DEPLOYMENT

Our proposed workload analysis technique needs to enable a few host hypervisor tracepoints (five events). However, other available trace-based approaches [12, 20] require to enable most Linux tracepoints in the host and the VMs. The main challenge in these methods is the absence of a global clock, which synchronizes the events collected from the host and the VMs. Therefore, with the cost of an extra overhead, additional events are defined as a synchronization strategy. This leads to an increase in the total analysis completion time. However, our proposed approach receives all events from the same clock source in the host and thus does not require a synchronization technique. In addition, our method can analyze the VMs with different operating systems (e.g., Linux, Windows, and Mac OS), since it only needs to enable a few events in the host.

8 LIMITATIONS

Our proposed approach gathers information from the host. Thus, detailed information regarding the guest operating system is not considered in our method. For instance, our method cannot detect information about containers in a VM, but it can show a container as a separate process. However, the other trace-based methods [12, 20] provide more information about the processes and their interactions with the guest kernel. In addition, the workload that we used in our benchmarks are a mix of workloads that could be used in the real world. As mentioned, the workloads have phases and would change over time. That is why our algorithm gives a hint to the infrastructure provider to place the VMs in the right place if the issue is constant.

9 CONCLUSION

Efficient distributed resource management, in cloud environments with thousands of VMs, is a demanding task that requires detailed information regarding the behavior of each VM. In this article, a host-level hypervisor tracing approach was proposed, based on an agent-less feature extraction technique that provided fine-grain characterization of VM behavior. A PR algorithm was developed to select features from significant processes running on the VMs. A two-phase K-Means clustering approach was applied to group VMs that had similarity in terms of workload behavior. The first-phase clustering provided a general vision on the behavior of each VM, whereas the second phase enabled us to discover detailed information about the root causes of different issues. Experiments on a real dataset, including various VMs running different software applications, showed that our proposed method could model VM behavior as expected. According to our benchmarks, our proposed method reached an accumulated overhead of around 0.3%, whereas other approaches showed an accumulated overhead in a range of 3.65% to 6.13%. Future work includes incorporating other existing tracing-based metrics, as additional features in the VM analysis, to reveal more detailed information on the root causes of VM behavior.

ACKNOWLEDGMENTS

We thank Genevieve Bastien for her comments.

REFERENCES

[1] GitHub. n.d. Agent-less Virtual Machine Monitoring System. Retrieved July 3, 2020 from https://github.com/Nemati/org.eclipse.tracecompass.incubator/tree/vm2.
[2] Die.net. n.d. Iostat. Retrieved July 14, 2020 from http://linux.die.net/man/1/iostat.
[3] KVM. n.d. Kernel Virtual Machine. Retrieved January 30, 2019 from http://www.linux-kvm.org/page/Main_Page.
[4] LibVMI. n.d. LibVMI: Virtual Machine Introspection Library. Retrieved February 25, 2018 from http://libvmi.com/.
[5] GitHub. n.d. VM Workload DataBase. Retrieved November 3, 2018 from https://github.com/Nemati/VMClassificationDataBase.
[6] Die.net. 2005. Vmstat. Retrieved July 14, 2020 from https://linux.die.net/man/8/vmstat.
[7] Eclipse Foundation. 2018. Eclipse Trace Compass. Retrieved February 12, 2018 from https://projects.eclipse.org/projects/tools.tracecompass.
[8] M. Abdelsalam, R. Krishnan, Y. Huang, and R. Sandhu. 2018. Malware detection in cloud infrastructures using convolutional neural networks. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. 162–169. https://doi.org/10.1109/CLOUD.2018.00028
[9] M. Abdelsalam, R. Krishnan, and R. Sandhu. 2017. Clustering-based IaaS cloud monitoring. In *Proceedings of the 2017 IEEE 10th International Conference on Cloud Computing (CLOUD'17)*. 672–679. https://doi.org/10.1109/CLOUD.2017.90
[10] Adel Abusitta, Martine Bellaiche, and Michel Dagenais. 2018. An SVM-based framework for detecting DoS attacks in virtualized clouds under changing environment. *Journal of Cloud Computing* 7, 1 (April 2018), 9. https://doi.org/10.1186/s13677-018-0109-4
[11] B. Awerbuch and R. Gallager. 1987. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory* 33, 3 (May 1987), 315–322. https://doi.org/10.1109/TIT.1987.1057314
[12] C. Biancheri, N. E. Jivan, and M. R. Dagenais. 2016. Multilayer virtualized systems analysis with kernel tracing. In *Proceedings of the 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW'16)*. 1–6. https://doi.org/10.1109/W-FiCloud.2016.18
[13] C. Canali and R. Lancellotti. 2012. Exploiting classes of virtual machines for scalable IaaS cloud management. *Journal of Communications Software and Systems* 8 (2012), 102–109. https://doi.org/10.24138/jcomss.v8i4.164
[14] Claudia Canali and Riccardo Lancellotti. 2014. Detecting similarities in virtual machine behavior for cloud monitoring using smoothed histograms. *Journal of Parallel and Distributed Computing* 74, 8 (2014), 2757–2769. https://doi.org/10.1016/j.jpdc.2014.02.006
[15] Claudia Canali and Riccardo Lancellotti. 2014. Exploiting ensemble techniques for automatic virtual machine clustering in cloud systems. *Automated Software Engineering* 21, 3 (Sept. 2014), 319–344. https://doi.org/10.1007/s10515-013-0134-y
[16] C. Canali and R. Lancellotti. 2015. Exploiting classes of virtual machines for scalable IaaS cloud management. In *Proceedings of the 2015 IEEE 4th Symposium on Network Cloud Computing and Applications (NCCA'15)*. 15–22. https://doi.org/10.1109/NCCA.2015.13
[17] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut. 2016. PerfCompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds. *IEEE Transactions on Parallel and Distributed Systems* 27, 6 (June 2016), 1742–1755. https://doi.org/10.1109/TPDS.2015.2444392
[18] Mathieu Desnoyers and Michel R. Dagenais. 2006. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the 2006 Ottawa Linux Symposium (OLS'06)*. 209–224.
[19] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. 2004. Kernel K-means: Spectral clustering and normalized cuts. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'04)*. ACM, New York, NY, 551–556. https://doi.org/10.1145/1014052.1014118
[20] Mohamad Gebai, Francis Giraldeau, and Michel R. Dagenais. 2014. Fine-grained preemption analysis for latency investigation across virtual machines. *Journal of Cloud Computing* 3, 1 (Dec. 2014), 23. https://doi.org/10.1186/s13677-014-0023-3
[21] Hani Nemati, Seyed Vahid Azhari, and Michel R. Dagenais. 2019. Host hypervisor trace mining for virtual machine workload characterization. In *Proceedings of the 2019 IEEE International Conference on Cloud Engineering (IC2E'19)*.
[22] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'08)*. ACM, New York, NY, 91–100. https://doi.org/10.1145/1346256.1346269

[23] Lionel Litty and David Lie. 2006. Manitou: A layer-below approach to fighting malware. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID'06)*. ACM, New York, NY, 6–11. https://doi.org/10.1145/1181309.1181311

[24] J. Macqueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. 281–297.

[25] A. Montplaisir-Goncalves, N. Ezzati-Jivan, F. Wininger, and M. R. Dagenais. 2013. State History Tree: An incremental disk-based data structure for very large interval data. In *Proceedings of the 2013 International Conference on Social Computing*. 716–724. https://doi.org/10.1109/SocialCom.2013.107

[26] H. Nemati and M. R. Dagenais. 2016. Virtual CPU state detection and execution flow analysis by host tracing. In *Proceedings of the 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom'16)*. 7–14. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.13

[27] H. Nemati and M. R. Dagenais. 2020. VirtFlow: Guest independent execution flow analysis across virtualized environments. *IEEE Transactions on Cloud Computing* 8, 3 (2018), 943–956. https://doi.org/10.1109/TCC.2018.2828846

[28] H. Nemati and M. R. Dagenais. 2018. VM processes state detection by hypervisor tracing. In *Proceedings of the 2018 Annual IEEE International Systems Conference (SysCon'18)*.

[29] H. Nemati, F. Tetreault, J. Puncher, and M. R. Dagenais. 2019. Critical path analysis through hierarchical distributed virtualized environments using host kernel tracing. *IEEE Transactions on Cloud Computing*. Early access. November 13, 2019.

[30] J. A. M. Rodrigues, F. M. C. de Oliveira, R. S. Lobato, R. Spolon, A. Manacero, and E. Borin. 2019. Improving virtual machine consolidation for heterogeneous cloud computing datacenters. In *Proceedings of the 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'19)*. 176–179.

[31] Shengming Li, Ying Wang, Xuesong Qiu, Deyuan Wang, and Lijun Wang. 2013. A workload prediction-based multi-VM provisioning mechanism in cloud computing. In *Proceedings of the 2013 15th Asia-Pacific Network Operations and Management Symposium (APNOMS'13)*. 1–6.

[32] W. Xing and A. Ghorbani. 2004. Weighted PageRank algorithm. In *Proceedings of the 2nd Annual Conference on Communication Networks and Services Research*. 305–314. https://doi.org/10.1109/DNSR.2004.1344743

[33] W. Yao, Y. Shen, and D. Wang. 2019. A weighted PageRank-based algorithm for virtual machine placement in cloud computing. *IEEE Access* 7 (2019), 176369–176381.

[34] H. Zhang, J. Rhee, N. Arora, S. Gamage, G. Jiang, K. Yoshihira, and D. Xu. 2014. CLUE: System trace analytics for cloud service performance diagnosis. In *Proceedings of the 2014 IEEE Network Operations and Management Symposium (NOMS'14)*. 1–9. https://doi.org/10.1109/NOMS.2014.6838348

[35] X. Zhang, F. Meng, and J. Xu. 2018. PerfInsight: A robust clustering-based abnormal behavior detection system for large-scale cloud. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. 896–899. https://doi.org/10.1109/CLOUD.2018.00130

Received March 2019; revised July 2020; accepted April 2021