RESEARCH

Fine-grained Multilayer Virtualized Systems Analysis

Cédric Biancheri* and Michel R Dagenais

Abstract

With the consolidation of computer services in large cloud-based data centers, almost all applications and even application development execute in virtualized systems (VS's), sometimes nested. Whether it is inside a container, a virtual machine (VM) running on a physical host, or in a nested virtual machine, every process eventually runs on a physical CPU. Consequently, multiple virtualized systems might unknowingly compete with each other for physical resources. In this paper we study the interactions between all the VS's running on a physical machine. We introduce an analysis based on kernel tracing that erases the bounds between VS's and their host, to display a multilayer system as a single layer. As a result, it becomes possible to know exactly which process is currently running on a physical CPU, even if it is launched inside multiple layers of containers, themselves enclosed into two layers of VMs.

To use this analysis, we developed in Trace Compass a view that displays a time line for each host CPU, showing across time which process is running. Moreover, the full hierarchy of the VS's is retrieved from the analysis and is displayed in the view. By using a system of dynamic and permanent filters, we added the possibility to highlight in this view either traced VMs, virtual CPUs, specific processes and containers. This last feature, combined with our view, allows to thoroughly apprehend the execution flow on the physical host, although it may involve multiple nested virtualized systems.

Keywords: Virtualized System; KVM; LXC; Tracing; LTTng

Introduction

Among the advantages of cloud environments we can cite their flexibility, their lower cost of maintenance, and the possibility to easily create virtual test environments. Those are some of the reasons explaining why they are widely used in industry. However, using this technology also brings its share of challenges in terms of debugging and detecting performance failures. Indeed, it can be more straightforward, when using the right tools, to detect performance anomalies while working with a simple layer of virtualization. For instance, if we have information about all the processes running on a machine through time, it is then possible to know for a specific thread which processes interrupted it. Because virtual machines (VM) are running in a layer independent of their host, it becomes more tedious to detect direct and indirect interactions between tasks happening inside a VM, on the host, inside a container, or even on nested or parallel VMs.

*Correspondence: cedric.biancheri@polymtl.ca
Department of Computer and Software Engineering, Polytechnique Montréal, Boulevard Edouard-Montpetit, QC H3T 1J4 Montréal, Canada

In this study, we focus on a way to analyze information, coming from a host, multiple VMs and linux containers (LXC) [1], as if all the execution was only happening on the host. The main objective is to erase as much as possible the boundaries between a host and the different virtual environments, to help a user visualize in a clearer way how the processes are interacting with each other.

To achieve this, we use kernel tracing on both the host and VMs, synchronize those traces, aggregate them into a unique structure and finally display the structure inside a view showing the different layers of the virtual environment during the tracing period. Considering the set of recorded traces as a whole system is the core concept of our fused virtualized systems (FVS) analysis presented here.

This paper is structured as follow: Section 2 exposes some related work about performance anomalies related to virtual environments. Section 3 explains in more details the multiple steps of the FVS analysis, including the single layered VMs (SLVMs), nested VMs (NVMs) and containers detection strategies. The same section introduces the view created to visualize the

whole system. Section 4 presents some use cases for the FVS analysis and view. Section 5 concludes this paper.

1 Related Work

Dean et al. [2] created an online performance bug inference tool for production cloud computing. To accomplish this, they created an offline function signature extraction using closed frequent system call episodes. The advantage of their method is that the signature extraction can be done outside the production environment, without running a workload that usually triggers a performance default. By using their tool, they can identify a deficient function out of thousands of functions. However, their work is not adapted to performance anomalies involving multiple virtual machines.

The research investigated by Sambasivan et.al. [3], proposes an approach to find, categorize and compare similar execution flows of different requests to diagnose performance changes. Their way of extracting similarities between different requests comprises some similarity to our method. However, our solution can be used in different purposes, from comparing the different execution flows to understanding the overall execution of VMs and extracting the relations between the different executions of different processes of the VMs and the host machine.

In their work, Shao et al. [4] proposed a scheduling analyzer for the Xen Virtual Machine Monitor [5]. The analyzer uses a trace provided by Xen to reconstruct the scheduling history of each virtual CPU. By doing so, it is possible to retrieve interesting metrics like the block-to-wakeup time. However, this approach is limited to Xen and not directly applicable to other hypervisors. Furthermore, a trace produced by Xen is not sufficient to identify a process inside a VM that creates a perturbation across the VMs.

To gain in generality and not rely too much on hypervisors and application code, some work was initiated with the intention to detect performance anomalies across virtual machines by using kernel tracing.

With PerfCompass [6], Dean et al. used kernel tracing on virtual machines and created an online system call trace analysis, able to extract fault features from the trace. The advantage of their work is that it only needs to trace the virtual machine's system calls and not the host. Consequently, their solution has a low overhead impact and is able to distinguish between external and internal faults. However, it is not possible to see the direct interactions of the VM with neither the host nor the other VMs and the containers.

Another work proposed by Gebai et al. [7] focused more on the interactions between several machines. The authors proposed at first an analysis and a view showing, for each virtual CPU, when it is preempted. They also created a way to recover the execution flow of a specific process by crossing virtual machine boundaries to see which processes preempted it.

Their work is similar to ours but differs on multiple points. For instance, in their work, the Virtual Machine view displays one row for each virtual CPU. This number can easily grow if numerous VMs are traced. Consequently, the readability of the view can be altered. Additionally, by doing so, information about physical CPUs is lost. It is therefore impossible to track a VM, a virtual CPU or a process on the host. Finally, their work is dedicated to the analysis of single layered VMs, unlike our work that focuses also on nested VMs and containers.

In [8], authors used the recently introduced Intel PT ISA extensions on modern Intel Skylake processors to analyse performance of VMs. They developed interactive Resource and Process Control Flow visualization tools to analyze the hardware trace data for VM. They could trace proprietary close-sourced operating systems to diagnose abnormal executions. Despite its merits, it is limited to new Intel processor and works only for hardware-assisted virtualization, thus it cannot be used with other virtualization methods, which does not meet our flexibility requirement.

Nemati et al.[9] proposed a low-overhead technique that uses the trace from Host hypervisor to detect overcommitment of resources in host machine. Their work can detect some problems related to resource contention but is not able to detect problems occurring within the VMs.

To our knowledge, no previous work tried to retrieve information about containers from a kernel trace. Other projects, like Docker [10], give access to runtime metrics such as CPU and memory usage, memory limit, and network IO metrics, exposed by the control groups [11] used by LXC. No previous work tries to represent the full execution of a multilayered system as if everything was happening on the host. Nonetheless, in reality, every process, even in nested VMs, eventually runs on a physical CPU of the host. Our contribution is to fulfill this gap.

2 Fused Virtualized Systems Analysis

A multilayered architecture is often the chosen strategy regarding the development of a software architecture. Each layer is dedicated to a specific role, independently of other layers, and is hosted by a tier, or a physical layout, that can contain multiple layers at once.

In this paper, we focus on a tier, or physical machine, hosting multiple layers of virtualized systems (VS) also called virtualized execution environment. A

virtualized system will be considered as a virtual machine or a container. Figure 1 shows how the different layers can be organized in practical cases. Without using multilayers of virtual environments, the system is reduced to a single layer which is the host, also called the physical machine. This layer will be called L_0. Virtual machines adding a layer above the host will be labeled as L_1 VMs, and recursively, any VM above a L_n VM will be a L_{n+1} VM. Containers will not be labeled but will be associated to the machine directly hosting them. Containers can be running directly in L_0 but, for security reasons [12], they are most often used within virtual machines.

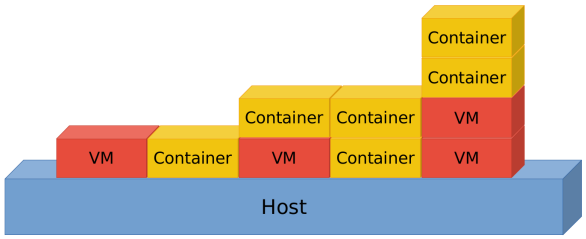

Figure 1: **Examples of different configurations of layers of execution environment.**

The idea we introduce here is to erase the bounds between L_0, its VMs in L_1 and L_2 and every container, to simplify the analysis and the understanding of complex multilayer architectures. Some methods for detecting performance degradations already exist for single-layer architectures. To reuse some of these techniques on multilayer architectures, one might remodel such systems as if all the activity was involving only one layer.

2.1 Architecture

The architecture of this work is described as follows: first we need to trace the host and the virtual machines, then because of clock drift [13] we have to synchronize those traces. After this phase, a data analyzer fuses all the data available from the different traces to put them in a data model. Finally, we need to provide an efficient tool to visualize the model that will allow the user to distinguish easily the different layers and their interactions. Those steps are summarized in Figure 2.

A trace consists of a chronologically ordered list of events characterized by a name, a time stamp and a payload. The name is used to identify the type of the event, the payload provides information relative to the event and the time stamp will specify the time when the event occurred.

In this study, we use the Linux Trace Toolkit Next Generation (LTTng) [14] to trace the machine kernels.

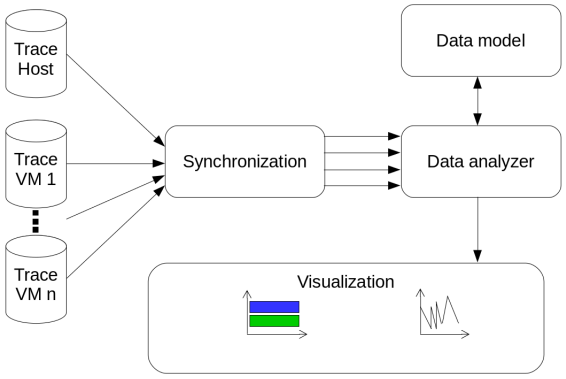

Figure 2: **Architecture of the fused virtual machines analysis.**

This low impact tracing framework suits our needs, although other tracing methods can also be adopted. By tracing the kernel, there is no requirement to instrument applications. Therefore, even a program using proprietary code can be analyzed by tracing the kernel. However, some events from the hypervisors managing the VMs are needed for the efficiency of the fused analysis. The analysis needs to know when the hypervisor is letting a VM run its own code or when it is stopped. Since, in our study, we are using KVM [15], merged in the Linux kernel since version 2.6.20 [16], and because the required trace points already exist, there is no need for us to add further instrumentation to the hypervisor. In our case, with KVM using Intel x86 virtualization extensions, VMX [17], the event indicating a return to a VM running mode will always be recorded on L_0 and will be generically called a VMEntry. The opposite event will be called a VMExit.

Synchronization is an essential part of the analysis. Since traces are generated on multiple machines by different instances of tracers, we have no guaranty that a time stamp for an event in a first trace will have any sense in the context of a second trace. Each machine may have its own timing sources, from the software interrupt timer to the cycle counter. When tracing the operating system kernel, each system instance (i.e., host, VM, container, etc.) uses its own internal clock to specify the events time stamps. But, in order to have a common sense of all systems behaviors, which are recorded as trace events separately in each system, it is essential to properly measure the differences and drifts between these machines.

Figure 3 shows that without synchronization two traces recorded at the same time may seem to be created at two different times. The right scheduling of events, even coming from different traces, is crucial

because, when fusing the traces of a VM with its host, the events of the VM will have to be handled exactly between the VMEntry and the VMExit of L_0, relative to this specific VM. An imperfect synchronization can be the vector of incoherent observations that would impede the fused analysis. Figure 4 shows the difference between an analysis done on two pairs of traces with respectively an accurate and inaccurate synchronizations. The inexact synchronization can lead to false conclusions. In this case, a process from the VM seems to continue using the processor while in reality the VM has been preempted by the host.

There are different possible solutions to synchronize the trace events between host kernel and VMs. One way is using TSC (Time Stamp Counter) that is built in the processors as a register. TSC is a 64-bit register which counts CPU cycles since the boot time of the system, and can be read by single assembly instruction (rdtcs) and therefore could be considered as a time reference, anywhere in the system (i.e., both kernel, hypervisor, and application). However, using TSC for timekeeping in a virtual machine has several drawbacks. The TSC_OFFSET field for VM can be changed especially during VM migration which forces tracer to keep track of this field in VMCS. If this event is lost, or the tracer is not started at that time, the calculated time will not be true anymore. Furthermore, some processors stop the TSC in their lower-power halt states which causes time shifting in VM. Also, timekeeping for full virtualization is not possible since TSC_OFFSET is part of Intel and AMD virtualization extensions.

Because VMs can be seen as nodes spread through a network, a trace synchronization method for distributed systems [18] can be adapted. As [7] we use hypercalls from the VMs to generate events on the host that will be related to the event recorded on the VM before triggering the hypercall. With a set of matching events, it is possible to use the fully incremental convex hull synchronization algorithm [19] to achieve trace synchronization. Because of clocks drift, a simple offset applied on the time stamps of a trace's events is not enough to synchronize the traces. To solve this issue, the fully incremental convex hull algorithm will generate two coefficients, a and b, for each VM trace while the host's trace is taken as time reference. Each event e_i will have its time stamp t_{e_i} transformed to t'_{e_i} with the formula:

$$t'_{e_i} = at_{e_i} + b$$

[7] used the hypercall only between L_0 and L_1. However, the method also applies between L_n and L_{n+1}, since an hypercall generated in L_{n+1} will necessarily be handled by L_n. In our case, synchronization events will be generated between L_0 and all its machines in L_1, and between machines of L_1 and their hosted machines. Consequently, a machine in L_2 will be synchronized with its host that will have previously been synchronized with L_0.

The purpose of the data analyzer is to extract from the synchronized traces all relevant data and to add them in a data model. Besides analyzing events specific to VMs and containers, our data analyzer should handle events generally related to the kernel activity. For this reason, the fused analysis is based on a pre-existing kernel analysis used in Trace Compass [20], a trace analyzer and visualizer framework. Therefore, the fused analysis will by default handle events from the scheduler, the creation, destruction and waking up of processes, the modification of a thread's priority, and even the beginning and the end of system calls. Unlike in a basic kernel analysis, the fused analysis will not consider each trace independently but as a whole. Consequently, the core of our analysis is to recreate the full hierarchy of containers and VMs, and to consider events coming from VMs as if they were directly happening in L_0. As shown in Figure 5, for the simple case of SLVMs, the main objective is to construct one execution flow by fusing those occurring in L_0 and its VMs. The result is a unique structure encompassing all the execution layers at the same time, replacing what was seen as the hypervisor's execution, from the point of view of L_0, by what was really happening inside L_1 and L_2.

KVM works in a way such that each vCPU of a VM is represented by a single thread on its host. Therefore, to complete the fused analysis, we need to map every VM's vCPU with its respective thread. This mapping is achieved by using the payloads of both synchronization and VMEntry events. On the one hand, a synchronization event recorded on the host contains the identification number of the VM, so we can match the thread generating the event with the machine. On the other hand, a VMEntry gives the ID of the vCPU going to run. This second information allows the association of the host thread with its corresponding vCPU.

2.2 Data model

The data analysis needs an adapted structure as data model. This structure needs to satisfy multiple criteria. A fast access to data is preferred to provide a more pleasant visualizer, so it should be efficiently accessible by a view to dynamically display information to users. The structure will also need to provide a way to store and organize the state of the whole system, while keeping information relative to the different layers. For this reason, we need a design that can store information about diverse aspects of the system.

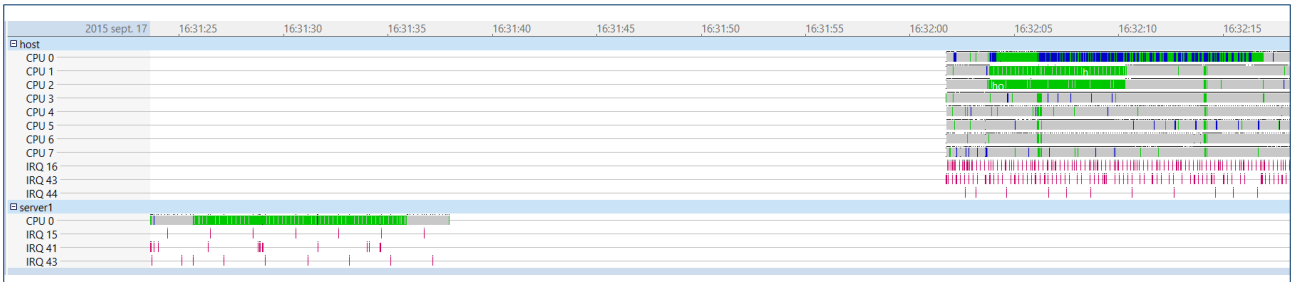

Figure 3: **Traces visualization without synchronization.**

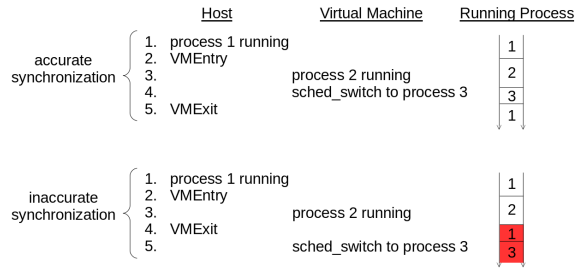

Figure 4: **Wrong analysis due to inaccurate synchronization.**

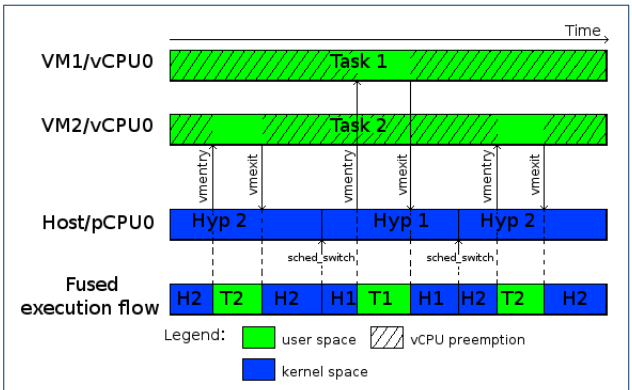

Figure 5: **Construction of the fused execution flow.**

As seen in Figure 6, the structure contains information relating to the state of the different threads but also of the numerous CPUs, VMs and containers. Each CPU of L_0 will contain information concerning the layer that is currently using it, like the name of the VM running and with which thread and which virtual CPU. The Machine node will contain basic information about VMs and L_0, like the list of physical CPUs they have been using, their number of vCPUs or their list of containers. This node is fundamental since it is used to recreate the full hierarchy of the traced systems, in addition to the hierarchy of all the containers inside each machine.

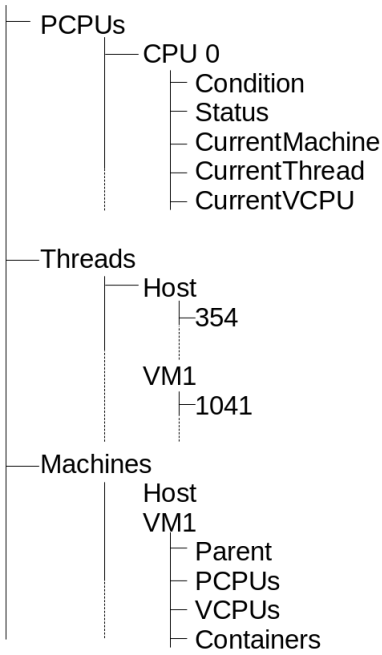

Figure 6: **Structure of the data model.**

Finally, the data model provides a time dimension aspect, since the state of each object attribute in the structure is relevant for a time interval. Those intervals introduce the need for a scalable model, able to record information valid from a few nanoseconds to the full trace duration.

In this study, we chose to work with a State History Tree (SHT) [21]. A SHT is a disk-based data structure designed to manage large streaming interval data. Furthermore, it provides an efficient way to retrieve, in logarithmic access time, intervals stored within this tree organization [22].

Algorithm 1 constructs the SHT by parsing the events in the traces. If the event was generated by the

host, then the CPU that created the event is directly used to handle the event. However, if the event was generated by a virtual machine, we need to recursively find the CPU of the machine's parent harboring the virtual CPU that created the event, until the parent is L_0. Only then, the right pCPU is recovered and we can handle the event. This process is presented in Algorithm 2.

Algorithm 1 Handling multilayer kernel traces

```
Input: StateHistoryTree s, List <Event> list
for each event in list do
    machine = Query the machine that generated event;
    if machine is a VM then
        // translation between virtual and physical CPU
        cpu = Query the pCPU currently running machine's cpu;
    else
        // the event happened in L0
        cpu = Query the CPU that generated event;
    end if
    handleEvent(s, event, cpu);
end for
```

Algorithm 2 Retrieving the physical CPU

```
Input: Machine machine, Cpu cpu
Output: The physical CPU harboring cpu
while machine is a VM do
    // cpu is a vCPU
    parent = Query machine's parent;
    cpu = Query parent's CPU currently running cpu;
    machine = parent;
end while
return cpu
```

The fundamental aspect of the construction of the SHT is the detection of the frontiers between the execution of the different machines and the containers. This detection is achieved by handling specific events and the application of multiple strategies.

2.2.1 Single Layered VMs Detection

In the case of SLVMs, the strategy is straightforward. The mapping is direct between the vCPUs of a VM in L_1 and its threads in L_0, a VM will be running its vCPU immediately after the recording of a VMEntry on its corresponding thread. Conversely, L_0 stops a vCPU immediately before the recording of a VMExit.

Algorithm 3 describes the handling of a VMEntry event for the construction of the SHT. In this case, we query the virtual CPU that is going to run on the physical CPU. Then, we restore the state of the virtual CPU in the SHT, while we save the state of the physical CPU. The exact opposite treatment is done for handling a VMExit event.

Algorithm 3 Handling vmentry event for Single Layered VMs

```
Input: StateHistoryTree s, Event event, Cpu cpu
if event == vmentry then
    vcpu = Query the virtual CPU going to run on cpu;
    Save the state of cpu contained in s;
    Restore the state of vcpu in s;
end if
```

2.2.2 Nested VMs Detection

For VMs in L_2, the previous strategy needs to be extended. Being a single-level virtualization architecture [23], the Intel x86 architecture has only a single hypervisor mode. Consequently, any VMEntry or VMExit happening at any layer higher or equal than L_1, is trapped to L_0. Figure 7 shows an example of the sequence of events and the hypervisors executions occurring on a pCPU when a VM in L_1 wants to let its guest execute its own code, and when L_2 is stopped by L_1. The dotted line represents the different hypervisors executing while the plain line shows when L_2 uses the physical CPU.

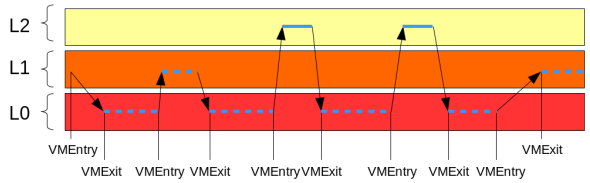

Figure 7: **Entering and exiting L_2**

This architecture supersedes the previous strategy used for SLVMs. A VMEntry recorded in L_1 does not imply that a vCPU of a VM in L_2 is going to run immediately after. Likewise, L_2 does not yield a pCPU shortly before an occurrence of a VMExit in L_1, but when the hypervisor in L_0 is running, preceded by its own VMExit.

The challenge we overcome here is to distinguish which VMEntries in L_0 are meant for a VM in L_1 or L_2. Knowing that a VM of L_2 is stopped is straightforward, if the previous distinction is done. If a thread of L_0 resumes a vCPU of L_1 or L_2 with a VMEntry, then a VMExit from this same thread means that the vCPU was stopped.

We created two lists of threads in L_0. The waiting list and the ready list. If a thread is in the ready list, it means that the next VMEntry generated by this thread is meant to run a vCPU of a VM in L_2. The second part of Algorithm 4 shows that we retrieve the vCPU of L_2 going to run by querying it from the vCPU of L_1 associated to the thread. The pairing between

the vCPUs of L_1 and L_2 is done in the first part of the algorithm, during the previous VMEntry recorded on L_1. It is also at this moment that the thread of L_0 is put in the waiting list.

Algorithm 4 Handling vmentry event for nested VMs

```
Input: StateHistoryTree s, Event event, Cpu cpu
 1: if event == vmentry then
 2:     vcpu = Query the virtual CPU going to run on cpu;
 3:     machine = Query the machine that generated event;
 4:     if machine is a VM then
 5:         // the vmentry is from L1 and cpu is a vCPU
 6:         Mark cpu as wanting to run vcpu;
 7:         Mark L0's thread running cpu as waiting;
 8:         return
 9:     end if
10:     // the vmentry is from L0
11:     thread = Query the thread running on cpu;
12:     if thread is ready for next layer then
13:         // L0's thread is ready to run L2
14:         // retrieve the real vCPU going to run
15:         vcpu = Query the vCPU that vcpu wants to run;
16:     end if
17:     Save the state of cpu contained in s;
18:     Restore the state of vcpu in s;
19: end if
```

Algorithm 5 shows that the same principle is used for handling a VMExit in L_0. If the thread was ready, then we need again to query the vCPU of L_2 before modifying the SHT.

Algorithm 5 Handling vmexit event for nested VMs

```
Input: StateHistoryTree s, Event event, Cpu cpu
 1: if event == vmexit then
 2:     vcpu = Query the virtual CPU stopped;
 3:     machine = Query the machine that generated event;
 4:     if machine is a VM then
 5:         // vmexits are not relevant for L1
 6:         return
 7:     end if
 8:     // the vmexit is from L0
 9:     thread = Query the thread running on cpu;
10:     if thread is ready for next layer then
11:         // L0's thread is stopping a vCPU in L2
12:         // retrieve the real vCPU stopped
13:         vcpu = Query the vCPU that vcpu was running;
14:     end if
15:     Save the state of vcpu contained in s;
16:     Restore the state of cpu in s;
17: end if
```

When a thread of L_0 is put in the waiting list, it means that a vCPU of L_2 is going to be resumed. However, at this point, we don't know for sure which VMEntry will resume the vCPU. The kvm_mmu_get_page event solves this uncertainty by indicating that the next VMEntry of a waiting thread will be for L_2. Algorithm 6 shows the handling of this event and the shifting of the thread from the waiting list to the ready list.

Algorithm 6 Handling kvm_mmu_get_page event

```
Input: Event event, Cpu cpu
 1: if event == kvm_mmu_get_page then
 2:     machine = Query the machine that generated event;
 3:     if machine is a VM then
 4:         // kvm_mmu_get_page is not relevant for VMs
 5:         return
 6:     end if
 7:     thread = Query the thread running on cpu;
 8:     if thread is waiting then
 9:         Remove thread from the waiting list;
10:         Mark thread as ready;
11:     end if
12: end if
```

As seen in Figure 7, it is possible to have multiple entries and exits between L_0 and L_2 without going back to L_1. This means that a VMExit recorded on L_0 does not necessarily implies that the thread stopped being ready. In fact, the thread stops being ready when L_1 needs to handle the VMExit. To do so, L_0 must inject the VMExit into L_1 and this action is recorded by the kvm_nested_vmexit_inject event. Algorithm 7 shows that the handling of this event consists in removing the thread from the ready list.

Algorithm 7 Handling kvm_nested_vmexit_inject event

```
Input: Event event, Cpu cpu
 1: if event == kvm_nested_vmexit_inject then
 2:     machine = Query the machine that generated event;
 3:     if machine is a VM then
 4:         // kvm_nested_vmexit_inject is not relevant for VMs
 5:         return
 6:     end if
 7:     thread = Query the thread running on cpu;
 8:     Remove thread from the ready list;
 9: end if
```

The process will repeat itself with the next occurrence of a VMEntry in L_1.

2.2.3 Containers Detection

The main difference between a container and a VM is that the container shares it's kernel with its host while a VM has its own. As a consequence, there is no need to trace a container since the kernel trace of the host will suffice. Furthermore, all the processes in containers are also processes of the host. Knowing if a container is currently running comes down to whether the current running process is from the said container or not.

The strategy we propose here is to handle specific events from the kernel traces to detect all the PID namespaces inside a machine. Then, we find out the virtual IDs of each thread (vTID) contained in a PID namespace.

A kernel trace generated with LTTng contains at least one state dump for the processes. A lttng_statedump_process_state event is created for each thread and any of its instances in PID namespaces. Furthermore, as seen in Figure 8, the payload of the event contains the vTID and the namespace ID (NSID) of the namespace containing the thread.

```
{ tid = 3887, vtid = 291, pid = 3881, vpid = 285, ppid = 3563, vppid = 1, ns_level = 1, ns_inum = 4026532199 }
{ tid = 3887, vtid = 3887, pid = 3881, vpid = 3881, ppid = 3563, vppid = 3563, ns_level = 0, ns_inum = 4026531836 }
{ tid = 3888, vtid = 292, pid = 3881, vpid = 285, ppid = 3563, vppid = 1, ns_level = 1, ns_inum = 4026532199 }
{ tid = 3888, vtid = 3888, pid = 3881, vpid = 3881, ppid = 3563, vppid = 3563, ns_level = 0, ns_inum = 4026531836 }
{ tid = 3893, vtid = 297, pid = 3893, vpid = 297, ppid = 3563, vppid = 1, ns_level = 1, ns_inum = 4026532199 }
{ tid = 3893, vtid = 3893, pid = 3893, vpid = 3893, ppid = 3563, vppid = 3563, ns_level = 0, ns_inum = 4026531836 }
```

Figure 8: **Payload of lttng_statedump_process_state events.**

Figure 9 shows how this information is added to the SHT. The full hierarchy of NSIDs and vTIDs is stored inside the thread's node to be retrieved later for the view. Moreover, each NSID and their contained threads are stored under it's host node. This allows to quickly know in which namespaces a thread is contained and, reciprocally, to known which threads belong to a namespace.

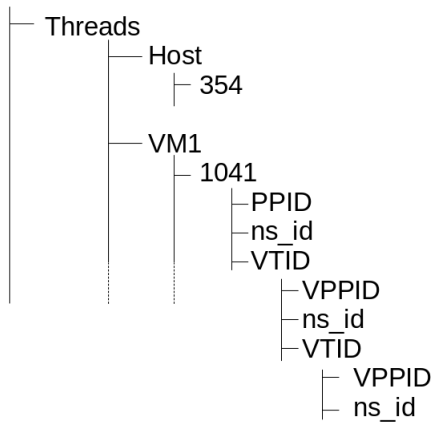

Figure 9: **Virtual TIDs hierarchy in the SHT.**

The analysis also needs to handle the process fork events to detect the creation of a new namespace or a new thread inside a namespace. In LTTng, the payload of this event provides the list of vTIDs of the new thread, besides of the NSID of the namespace containing it. Because the new thread's parent process was already handled by a previous process fork or a state dump, the payload combined with the SHT contains enough information to identify all the name spaces and vTIDs of a new thread.

2.3 Visualization

After the fused analysis phase, we obtain a structure containing state information about threads, physical CPUs, virtual CPUs, VMs and containers through the traces duration. Our intention at this step is to create a view made especially for kernel analysis and able to manipulate all the information about the multiple layers contained inside our SHT. The objective is also to allow the user to see the complete hierarchy of virtualized systems. This view is called the Fused Virtualized Systems (FVS) view.

This view shows at first a machine's entry representing L_0. Each machine's entry of the FVS view can have at most three nodes. A PCPUs node, displaying the physical CPUs used by the machine, a Virtual Machine node, containing an entry for each of the machine's VM, and a Containers node, displaying one entry for each container. Because VMs are considered as machines, their nodes can contain the three previously mentioned nodes. However, a container will at most contain the PCPUs and Containers nodes. Even if it is possible to launch a VM from a container, we decided to regroup the VMs only under their host's node.

Figure 10 is a high level representation of a multilayered virtualized system. When traced and visualized in the FVS view, the hierarchy can directly be observed, as seen in Figure 11.

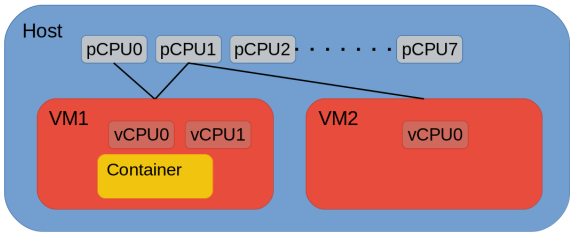

Figure 10: **High level representation of a multilayered virtualized system.**

The PCPUs entries will display the state of each physical CPU during a tracing session. This state can either be idle, running in user space, or running in kernel space. Those states are respectively represented in gray, green and blue. However, there is technically no restriction on the number of CPU states, if an extension of the view is needed.

The Resources view is a time graph view in Trace Compass that is also used to analyze a kernel trace. It normally manages different traces separately and doesn't take into account the multiple layers of virtual execution. Figure 12 shows the difference between the FVS view and the Resources view displaying respectively a fused analysis and a kernel analysis coming from the same set of traces.

In this set, servers 1, 2 and 3 are VMs running on the host. All VMs are trying to take some CPU resources.

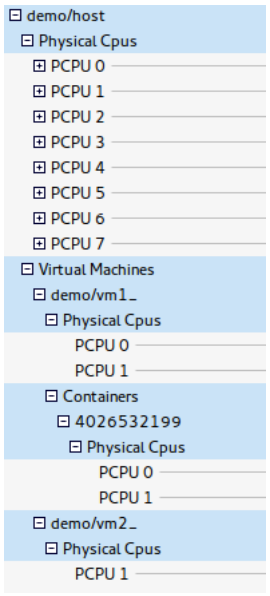

Figure 11: **Reconstruction of the full hierarchy in the FVS view.**

As should be, the FVS view shows all the traces as a whole, instead of creating separate displays as seen in the Resources view. The first advantage of this configuration is that we only need to display the physical CPUs rows instead of one row for each CPU, physical or virtual. With this structure, we gain in visibility. The information from multiple layers is condensed within the rows of the physical CPUs.

To display information about virtual CPUs, VMs and containers, the FVS view asks the data analyzer to extract some information from the SHT. Consequently, for a given time stamp, it is possible to know which process was running on a physical CPU, and on which virtual CPU and VM or container it was running, if the process was not directly executed on the host. Figure 13 shows the displayed tooltip when the cursor is placed on a PCPU entry. These are part of the information used to populate the entry.

We noticed that, in the Resources view, the information is often too condensed. For instance, if several processes are using the CPUs, it can become tedious to distinguish them. Therefore, this situation is worse in the FVS view, because more layers come into play. For this reason, we developed a new filter system in Trace Compass that allows developers of time graph views to highlight any part of their view, depending on information contained in their data model.

Using this filter, it is possible to highlight one or more physical or virtual machines, containers, some physical or virtual CPUs, and some specifically selected processes. In particular, this filter will display what the user doesn't want to see, as if it was covered with a semi opaque white band. Selected areas will appear highlighted by comparison. Consequently, it is possible to see the execution of a specific machine, container, CPU or process directly in that view.

Figure 14 shows the real execution location of a virtual machine on its host. With this filter, we can distinctively see when the CPU was used by another machine, instead of the highlighted one.

In the FVS view, the states in the PCPUs entries of a virtualized system are a subset of the states visible in the PCPUs entries of the VS's parent. Only the physical host PCPUs display the full state history. The other entries can be considered as permanent filters dedicated to display only a VS and its virtualized subsystems. Figure 15 shows a magnified part of Figure 11 with all PCPUs nodes expanded. We can see that their sum equals the physical PCPUs entries.

3 Use Cases and Evaluation

3.1 Use Cases

The concept of fusing kernel traces can have very interesting applications. In this section, we expose multiple use cases.

Our first use case is selecting a specific process, running in a container inside a virtual machine, in order to observe with the FVS view when and where the process was running.

Figure 16 shows that, from the point of view of the VM, the process vm_forks was running without interruption according to the Control Flow view. The Control Flow view is a view listing all the threads that were running during the tracing session, giving the state of those threads (running, waiting for CPU, blocked...). However, when we highlight the process in the FVS view, we clearly see that the selected process was preempted. If we magnify the view, we can even directly see which process from which machine is preempting our highlighted process, and when the process migrated to an other CPU.

Our next use case benefits from the fact that, by erasing the bounds between virtualized systems and the physical host, this analysis and view provide a tool to better understand the execution of an hypervisor. With the FVS view, it is possible to precisely see the interactions between the hypervisor and the host, depending on the instrumentation used.

In our second use case, we propose to compare the time needed to wake up a sleeping process in L_1 and in L_2. In both L_1 and L_2, we created a process that sleeps for a short amount of time and then yields a pCPU. For both of them we examine the elapsed time between the wake up of the hypervisor in L_0 and the return to the VM's process. Figure 17 shows that resuming a VM

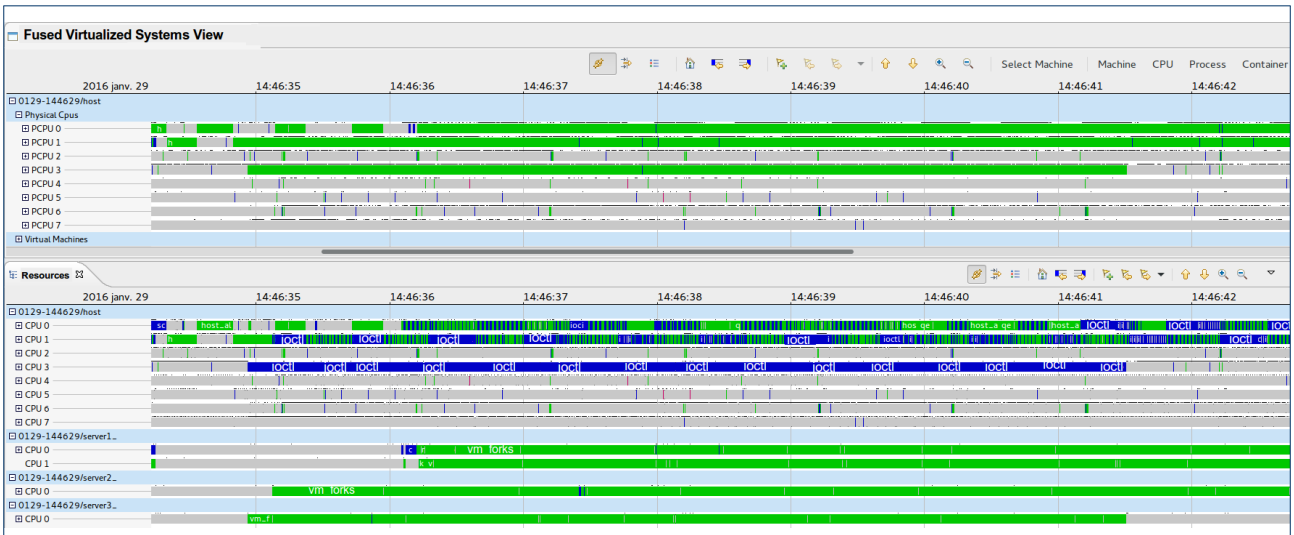

Figure 12: **Comparison between FVS view and Resources view.**

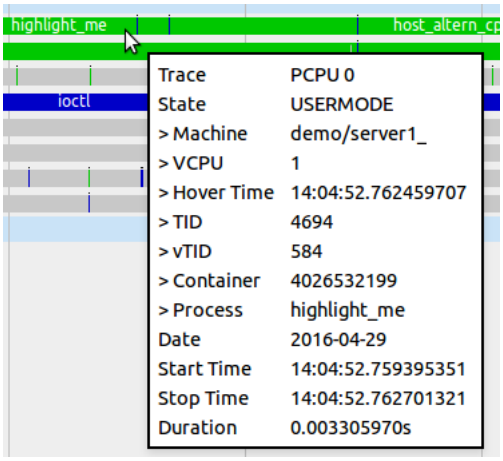

Figure 13: **Tooltip displayed to give more information regarding a PCPU.**

in L_2 necessitates a lot of entries and exits between L_0 and L_1 due to trapped instructions. In our case, it took approximately 300 μs to wake up the process in L_2 while it took only 73 μs to wake up the one in L_1. This observed latency is a reason why deeper nested VMs suffer a higher perceived virtualization overhead.

Our third use case is observing how an interruption is handled inside a VM. Figure 18 shows what occurred during an I/O interruption happening in a VM running on physical CPU 1. We highlighted the execution of the VM to see when the hypervisor is involved. The hypervisor stopped the VM, meaning that the thread went out of guest mode, returned to kernel mode, then to user mode to handle the I/O interruption, then back to kernel mode and finally let the VM run by switching back to guest mode. This behavior is completely consistent with what is expected in [15].

The study of those situations was highly simplified by the use of our tool. To determine if a thread of L_2 is currently running on a pCPU, someone not using our tool should know the functioning of the hypervisor. He will need to determine if one of the current threads running on L_0 is associated to a vCPU of L_1, itself running a thread associated to a vCPU of L_2, executing the thread of interest. This long process is tedious for a human being. Our tool spares the user this waste of time by showing clearly and directly what he wants without having any knowledge of the internal functioning of the hypervisor.

3.2 Evaluation

3.2.1 SHT's Generation Time

If we compare the time needed to complete a fused analysis for a set of traces and the one needed to complete a simple kernel analysis for the same set, we come to the conclusion that the simple kernel analysis is faster. Let T_i be the time needed to analyze trace i. Since the simple kernel analysis doesn't consider the set of traces as a whole but each trace independently, the analysis of the set can be done in parallel, each core dedicated to one trace. If we suppose that we have more cores than traces, then the elapsed time during the analysis will be $max_{1 \leq i \leq n} T_i$ where n is the number of traces.

If the set is considered as a whole, then it is difficult to process the traces in parallel. The elapsed time during the fused analysis will consequently be $\sum_{1 \leq i}^{n} T_i$.

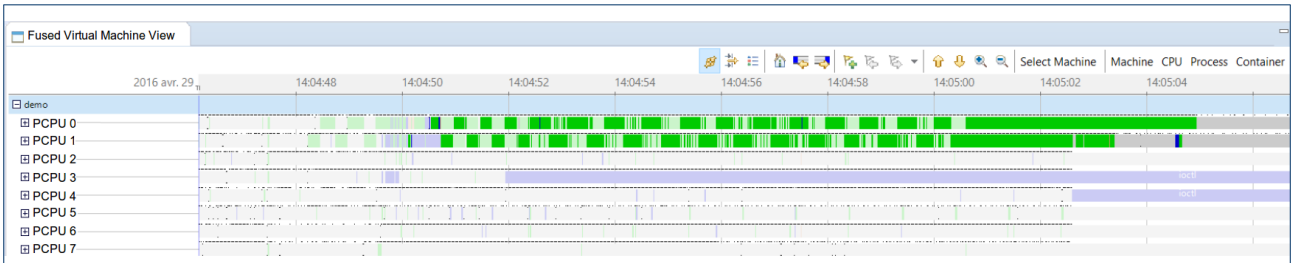

Figure 14: **VM server1 real execution on the host.**

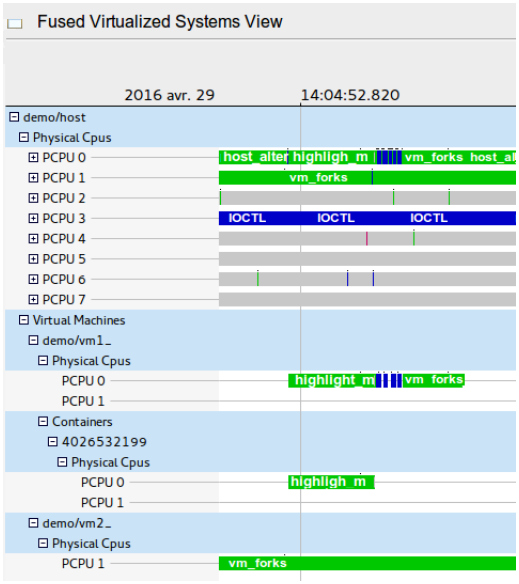

Figure 15: **PCPUs entries of each virtualized system.**

Figure 19 shows experimentally the time needed for the fused analysis and a simple kernel analysis to build SHTs for different sizes of trace sets. We see that the build time for the fused analysis is directly related to the size of the trace set.

3.2.2 SHT's Size on Disk

To evaluate the space on disk necessary to realize the fused analysis, we compared the size of the SHT we created with the sum of the sizes of the SHTs created for each trace by the kernel analysis. Figure 20 shows that our SHT needs less space than the combined kernel analysis SHTs. However, we expected the sizes to be nearly equal since the fused analysis SHT can be seen as a combination of the kernel analysis SHT's. This gap is mainly explained by the fact that the fused analysis starts to build the CPUs attributes of the SHT only when all the machine's roles have been determined.

Those results were obtained with an Intel core i7-3770 and with 16GB of memory.

4 Conclusion and Future Work

In this paper, we presented a new concept of kernel trace analysis adapted to cloud computing and virtualized systems that can help for the monitoring and tuning of such systems and the development of those technologies. This concept is independent of the kernel tracer and hypervisor used. By creating a new view in Trace Compass, we showed that it was possible to display an overview of the full hierarchy of the virtualized systems running on a physical host, including VMs and containers. Finally, by adding a new dynamic filter feature to the FVS view, in addition to a permanent filter for any VS, we showed how it is possible to observe the real execution on the host of a virtual machine, one of its virtual CPUs, its processes and its containers.

In the future, we can expect the concept of the fused analysis to be reused and adapted for more specific utilization like the analysis of I/O or memory usage. We could also use the same principles to analyze more thoroughly systems using applications and programs in virtual execution environments, such as Java or Python. Finally, we can also extend our work to be able to visualize VMs' interactions between nodes to better understand the internal activity of cloud systems.

Competing interests
The authors declare that they have no competing interests.

Author's contributions
CB built the state of the art of the field, defined the objectives of this research, did the analysis of the current virtual machine monitoring tools and their limitations. He implemented the analysis tool presented in this paper, as well as the experiments. MRD initiated and supervised this research, lead and approved its scientific contribution, provided general input, reviewed the article and issued his approval for the final version.

Acknowledgements
The authors would like to thank Francis Giraldeau for resolving some intricated bugs and Naser Ezzati Jivan for reviewing this paper.

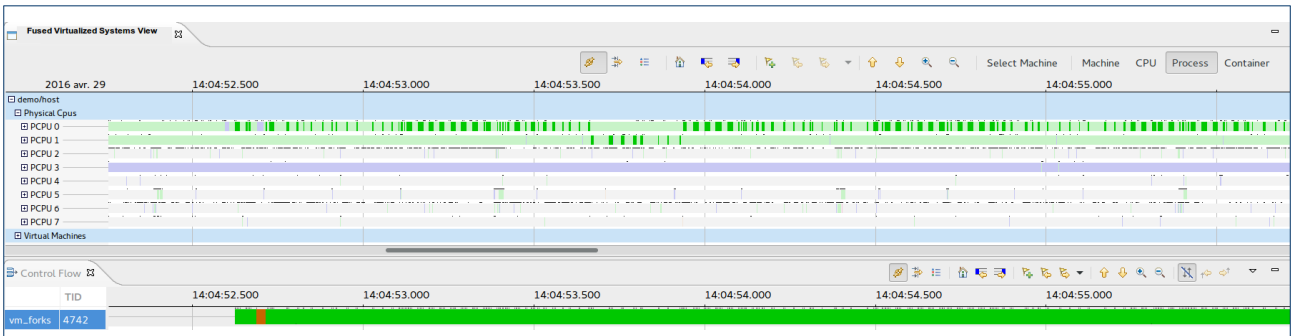

Figure 16: Highlighted process in the FVS view.

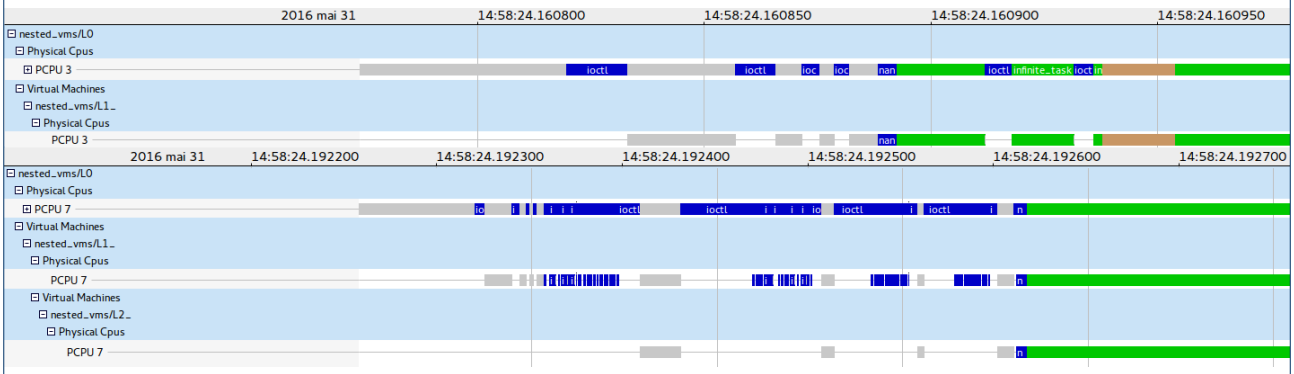

Figure 17: Process wake up time for L_1 and L_2.

Figure 18: Handling of an ata_piix I/O interruption by the hypervisor on the physical CPU 1.

References

1. Vaughan-nichols, S.J.: New approach to virtualization is a lightweight. Computer **39**(11), 12–14 (2006)
2. Dean, D.J., Nguyen, H., Gu, X., Zhang, H., Rhee, J., Arora, N., Jiang, G.: Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 1–13 (2014). ACM
3. Sambasivan, R.R., Zheng, A.X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., Ganger, G.R.: Diagnosing performance changes by comparing request flows. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. NSDI'11, pp. 43–56. USENIX Association, Berkeley, CA, USA (2011). http://dl.acm.org/citation.cfm?id=1972457.1972463
4. Shao, Z., He, L., Lu, Z., Jin, H.: Vsa: an offline scheduling analyzer for xen virtual machine monitor. Future Generation Computer Systems **29**(8), 2067–2076 (2013)
5. Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., Neugebauer, R.: Xen and the art of virtualization (2003)
6. Dean, D.J., Nguyen, H., Wang, P., Gu, X.: Perfcompass: toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In: 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14) (2014)
7. Gebai, M., Giraldeau, F., Dagenais, M.R.: Fine-grained preemption analysis for latency investigation across virtual machines. Journal of Cloud Computing **3**(1), 1 (2014)
8. S Sharma, M.D. H Nemati: Low overhead hardware assisted virtual machine analysis and profiling. In: IEEE Globecom Workshops (GC Workshops) (2016)
9. Nemati, H., Dagenais, M.R.: 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), pp. 7–14 (2016)
10. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux Journal **2014**(239), 2 (2014)
11. Process Containers. Accessed: 2016-07-04. http://lwn.net/Articles/236038/
12. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: ACM SIGOPS Operating Systems Review, vol. 41, pp. 275–287 (2007). ACM
13. Marouani, H., Dagenais, M.R.: Internal clock drift estimation in computer clusters. Journal of Computer Systems, Networks, and Communications **2008**, 9 (2008)
14. Desnoyers, M., Dagenais, M.R.: The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In: OLS (Ottawa Linux Symposium), vol. 2006, pp. 209–224 (2006). Citeseer
15. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the linux virtual machine monitor. In: Proceedings of the Linux Symposium, vol. 1, pp. 225–230 (2007)
16. Linux 2 6 20. Accessed: 2016-07-04. http://kernelnewbies.org/Linux_2_6_20

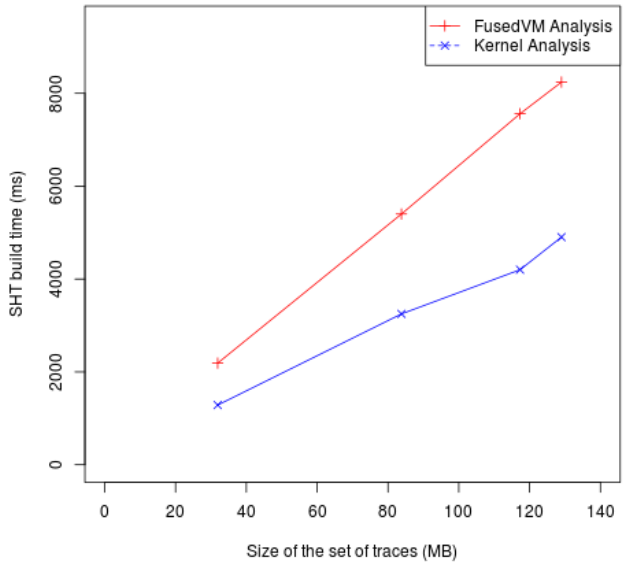

Figure 19: **Comparison of construction time between FusedVS Analysis and Kernel Analysis.**

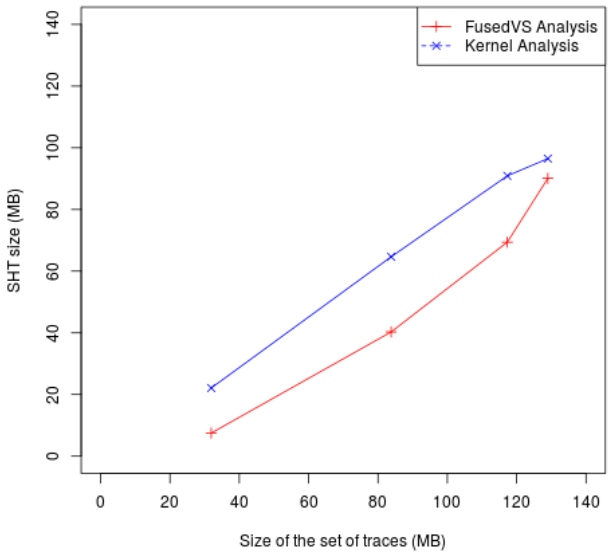

Figure 20: **Comparison of the SHT's size between FusedVS Analysis and Kernel Analysis.**

17. Uhlig, R., Neiger, G., Rodgers, D., Santoni, A.L., Martins, F.C., Anderson, A.V., Bennett, S.M., Kagi, A., Leung, F.H., Smith, L.: Intel virtualization technology. Computer **38**(5), 48–56 (2005)
18. Jabbarifar, M.: On line trace synchronization for large scale distributed systems. PhD thesis, École Polytechnique de Montréal (2013)
19. Poirier, B., Roy, R., Dagenais, M.: Accurate offline synchronization of distributed traces using kernel-level events. ACM SIGOPS Operating Systems Review **44**(3), 75–87 (2010)
20. Trace Compass. Accessed: 2016-07-04. http://tracecompass.org/
21. Montplaisir-Gonçalves, A., Ezzati-Jivan, N., Wininger, F., Dagenais, M.R.: State history tree: an incremental disk-based data structure for very large interval data. In: Social Computing (SocialCom), 2013 International Conference On, pp. 716–724 (2013). IEEE
22. Montplaisir, A., Ezzati-Jivan, N., Wininger, F., Dagenais, M.: Efficient model to query and visualize the system states extracted from trace data. In: International Conference on Runtime Verification, pp. 219–234 (2013). Springer
23. Ben-Yehuda, M., Day, M.D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., Yassour, B.-A.: The turtles project: Design and implementation of nested virtualization. In: OSDI, vol. 10, pp. 423–436 (2010)