Gebai *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2014) 3:23
DOI 10.1186/s13677-014-0023-3

Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Fine-grained preemption analysis for latency investigation across virtual machines

Mohamad Gebai*, Francis Giraldeau and Michel R Dagenais

Abstract

This paper studies the preemption between programs running in different virtual machines on the same computer. One of the current monitoring methods consist of updating the average steal time through collaboration with the hypervisor. However, the average is insufficient to diagnose abnormal latencies in time-sensitive applications. Moreover, the added latency is not directly visible from the virtual machine point of view. The main challenge is to recover the cause of preemption of a task running in a virtual machine, whether it is a task on the host computer or in another virtual machine.

We propose a new method to study thread preemption crossing virtual machines boundaries using kernel tracing. The host computer and each monitored virtual machine are traced simultaneously. We developed an efficient and portable trace synchronization method, which is required to account for time offset and drift that occur within each virtual machine. We then devised an algorithm to recover the root cause of preemption between threads at every level. The algorithm successfully detected interactions between multiple competing threads in distinct virtual machines on a multi-core machine.

Keywords: Virtual machine; Tracing; KVM; LTTng; Performance; CPU

Introduction

Cloud environments present advantages of increased flexibility and reduced maintenance cost through resource sharing and server consolidation [1]. However, virtual machines (VMs, or guests) on the same host computer may compete for shared resources, introducing undesirable latency. Previous study found that jitter impacts response time of programs on popular commercial cloud environment [2]. In cloud environments, virtual machines have the illusion of absolute and exclusive control over the physical resources. However, the host's resources are more often than not overcommitted, whereas they appear to guest operating systems as being more available than they actually are [3]. As a result, virtual machines on the same host computer may interfere with each other without their knowledge, inducing invisible yet real latency.

The diagnosis is more complex when the guest is isolated from its external environment and an additional virtualization layer is introduced. It is therefore necessary to have powerful and efficient tools to diagnose the root cause of unexpected delays at low granularity when they occur in a virtualized environment. To our knowledge, no such tool was available.

This study focuses on processor multiplexing across virtual machines. In particular, we are interested in automatically identifying the root cause of task preemption crossing virtual machines boundaries. The approach we propose is based on kernel tracing, which is an effective and efficient way to investigate latency problems [4]. The method we propose consists of aggregating kernel traces recorded simultaneously on the host and each virtual machine. However, more often than not, timekeeping is a task left to each of the operating systems. In such cases, timestamps from different traces are not issued using the same clock reference. As a result, trace merging without an appropriate synchronization method to account for clock differences would produce incoherent results.

The challenge is to consider the system as a whole, while preserving virtual machine isolation. Flexibility and portability constraints are also important for practical considerations. The approach should be independent from the underlying architecture and the operating system to

*Correspondence: mohamad.gebai@polymtl.ca
Department of Computer and Software Engineering, Polytechnique Montreal, 2900 Boulevard Edouard-Montpetit, Montreal, QC H3T 1J4, Canada

account for portability, whereas flexibility requires independence from the hypervisor and the tracer.

Three main contributions are presented in this paper. First, we propose an approach for trace synchronization. At the trace merging step, we propose an algorithm that modifies timestamps of the guests' traces to bring them back to the same timespan as the host. Secondly, we implemented an analysis program that transforms aggregated kernel traces to a graphical view that shows the states of the virtual machines and their respective virtual CPUs (vCPUs) through time while taking into consideration virtualization and its impact. Thirdly, we implemented an additional analysis program that presents the interactions of threads across different systems. Such an analysis can be performed by recovering the execution flow centered around a particular thread.

The rest of this paper is structured as follows: Section 'Related work' goes through different approaches currently used for virtual machine monitoring. Section 'Problem statement and definitions' introduces the required concepts in virtualization and tracing, and states the problem addressed by this paper. Section 'Trace synchronization' explains our approach for trace synchronization at the aggregation step. Each of sections 'Multi-level trace analysis' and 'Execution flow recovery' introduces an analysis module and its inner working. Section 'Use cases' shows some representative use cases and their analysis results. Section 'Flexibility and portability' reiterates over flexibility and portability. Section 'Conclusion' concludes.

Related work

On Linux kernels supporting paravirtualization, `top` reports a metric specific to virtual machines, named *steal time*. This metric shows the percentage of time for which a vCPU of the VM is preempted on the host. While this information can give a general idea or a hint of overcommitment of the CPU, it does not report the actual impact on the running threads nor the source cause of preemption. Additionally, this approach is specific for Linux paravirtualized systems and thus limits portability. Moreover, `top` adds significant overhead as it gathers information by reading entries in the `proc` pseudo-filesystem, and offline analysis or replay of the execution flow are not possible.

Perf has been extended to support profiling and tracing specifically for KVM. Using its "`kvm`" subcommand, one can use Perf to get runtime statistics and metrics about each virtual machine. Common metrics include the number of traps caught by the hypervisor, their cause and the time to process each of them. The information reported by Perf also includes CPU time for the guest kernel, host kernel and the hypervisor, which are good indicators about the overhead introduced by virtualization. Perf also reports information about the Performance Monitoring Unit (PMU), which is a set of counters that keep track of particular events such as cache misses, TLB misses, CPU cycles, etc. However, these performance counters aren't available for virtual machines. In [5], an approach for PMU virtualization is proposed, which are then used to monitor the runtime behavior of virtual machines in more detail. In [6] and [7], the authors also use Perf for virtual machine profiling and resource utilization. Such methods may also require exporting the symbol table of the guest kernel to the host to resolve. While it is possible to detect performance degradation due to resource sharing among virtual machines, the analysis doesn't cover detailed fine-grained information about the root causes of preemption. However, the interactions between the virtual machines through the usage of shared resources are essential to understand performance degradations and easily pinpoint their cause in order to remedy them. Finally, the approach using `perf kvm` is dependent on both the operating system and the hypervisor, which doesn't meet the portability requirement.

Shao et al. use an approach based on tracing within Xen to generate useful metrics for virtual machines [8]. Based on scheduling events, latency due to virtual CPU preemption can be easily calculated. Other metrics of interest are also presented such as the wake-to-schedule time. However, these metrics are mostly useful for analyzing Xen's scheduler itself. Such an analysis would be less relevant in the case of KVM (or some other hypervisors) as it is a an "extension" to the Linux kernel via loadable kernel modules and thus uses its scheduler. Moreover, the impact of the applications running inside the virtual machines on the system as a whole can not be retrieved from Xen traces. Differently put, perturbations caused by userspace applications across different virtual machines cannot be analyzed or quantified using solely Xen traces.

As for trace synchronization, previous studies [8,9] have used the TSC (TimeStamp Counter) as a common time reference to approach timekeeping and clock drift issues among VMs. The TSC is a CPU register on x86 architectures which counts CPU cycles since the boot of the system (uptime). When read from a virtualized system, the TSC is usually automatically offset in hardware to reflect the uptime of the guest operating system. The value of the offset is specified by the `TSC_OFFSET` field in the Virtual Machine Control Structure (VMCS). Each VM has its own `TSC_OFFSET` value, and reading the TSC from different systems always returns a coherent value with respect to their respective uptime. Once traces are recorded on different systems, converting guest TSC values to host TSC values comes down to subtracting the value of `TSC_OFFSET` from each timestamp. However, the TSC offset may have to be adjusted during the execution of the VM upon certain events, such as virtual machine migration. As a result, `TSC_OFFSET`

adjustments have to be tracked down by the tracer at runtime. If tracing is not enabled before the creation of the virtual machine, the initial value of the TSC offset cannot be obtained, unless explicitly requested by the tracer. Additionally, this approach does not allow for the possibility of lost events since a TSC adjustment event could be lost. In any manner, even if the TSC isn't virtualized and is unique across all systems, synchronization using the TSC does not meet our requirement of portability, as it is an x86-specific register. Moreover, TSC offsetting is specific to hardware-assisted virtualization, thus it cannot be used with other virtualization methods, which does not meet our flexibility requirement. Finally, the TSC register only counts CPU cycles since boot time, which is not as meaningful as an absolute wall clock time, especially on computers with a non-constant TSC where the conversion from TSC to real time would be an additional challenge (CPU flag `constant_tsc` can be queried to verify this property).

Problem statement and definitions

Addressed problem and motivation

We noticed that one of the main limitations of current approaches for virtual machine monitoring is the lack of a general approach, which takes into account in-depth analysis of all the involved systems. Most of the monitoring tools are designed to be centered either around the hypervisor or the guest OS. In the former case, only an analysis from the host point of view, abstracted by the virtualization layer, is possible. In the latter case, the analysis is too restricted inside the guest OS and doesn't consider the outside environment for detecting the root causes of performance degradations.

As mentioned in section 'Introduction', investigating latency problems in virtualized systems is a non-trivial task. The isolation of virtual machines from their environments imposes limits on the scope of traditional analysis tools. Moreover, the virtualization layer itself adds overhead due to the involvement of the hypervisor for privileged operations [10]. Furthermore, the assumption of exclusive access to the hardware layer by each virtual machine inevitably induces hidden latency due to the overcommitment of resources, particularly the CPU. As a result, the CPU becomes a scarce resource, which has to be shared among running VMs. As we presented in the previous section, there is no obvious way for a VM to detect runtime perturbation caused by the "outside world". While a guest OS may perceive one of its processes taking full use of the CPU for a certain amount of time, this might not be effectively the case on the actual hardware. Indeed, when a process is allocated a limited amount of vCPU time in a guest OS, it might get deprived of this resource by the host's scheduler which might elect a different VM for execution at any moment. Analyzing preemption across virtual machines boundaries (inter-VM) allows the user to detect such perturbations and take actions to remedy them.

In this paper, we explain how we used kernel traces recorded in each VM and on the host simultaneously to investigate such problems. As we present in section 'Use cases', the tools resulting from our study help the users to easily find the latency cause due to CPU sharing among virtual machines, as well as the actual threads that affect the completion time of a certain workload. However, merging distributed traces is a problem in itself as each operating system is solely responsible for its own timekeeping. The next sections present prerquisites in order to understand all of the parts used in our final solution.

Hypervisor

CPU vendors introduced extensions at the hardware level which allow for efficient architecture virtualization and overcome design issues, as presented in [10] for x86. On Intel hardware, this CPU extension is called VMX (Virtual Mahine eXtension), while AMD-V is its counterpart from AMD. On hardware-assisted virtualization, the CPU transits between non-root and root modes. On Intel CPUs, these modes are respectively called VMX non-root and VMX-root. The former is entered using the `vmentry` instruction by the hypervisor, giving control to the VM's native code. The later is reached when the VM executes an instruction that triggers a trap, called `vmexit`. A trap is usually a sensitive instruction such as writing to a privileged register, and allows the hypervisor to take control of the execution and react to the trapped instruction, usually through emulation. `Vmexit` can be thought of as a *reaction*, as opposed to `vmentry` which is an actual instruction. Moreover, a data structure called VMCS (Virtual Machine Control Structure) [11] contains runtime information about a virtual machine. This data structure is used as an interaction mechanism between the VM and the hypervisor [12] (i.e. between non-root and root modes), as well as a way to define behavioral elements, such as enabling or disabling hardware TSC offsetting.

The software that interacts with these hardware extensions is called a hypervisor. KVM [13] is an example of such software and is included in Linux as a loadable kernel module. Its role is to exploit and manage the virtualization capabilities of the hardware, and provide easy access to these capabilities to any userspace component via the `ioctl` interface. As a result, many userspace emulators can be built atop KVM without reimplementing hardware-specific functionalities. We use QEMU as this userspace component that interacts with KVM to take advantage of hardware assistance. Moreover, as KVM is an extension to the Linux kernel, it can take advantage of its basic functionalities, such as the scheduling, NUMA

node management, and even its tracing infrastructure. Thus, KVM is instrumented with tracepoints which can be traced using any kernel tracer. In QEMU/KVM, each virtual machine is a QEMU process, and each of its virtual CPUs (vCPUs) is emulated by a separate thread that belongs to that process.

In this article, the terms hypervisor and VMM (Virtual Machine Monitor) will be used interchangeably. The same applies for the terms VM, guest system and virtualized system.

Trace indexing

We implemented our trace analysis algorithm using the Trace Compass trace viewer (previously TMF - Tracing and Monitoring Framework) [14]. Trace Compass is a Free and Open tool for viewing traces in different graphical views. Views are usually designed for specific kind of analyses. The most common views in Trace Compass are the Control Flow view and the Resource view. The former shows the states of all threads on a system throughout the tracing session (Running, Idle, Preempted, Blocked), whereas the latter shows the states of different resources such as the CPU and IRQ lines. This project resulted in two additional views integrated to Trace Compass, which can be used for Virtual Machine runtime analysis of inter-VM preemption.

Trace Compass indexes the trace using a State History Tree (SHT) [15]. The SHT represents the state of the whole system, and is updated at each event to define time intervals [16]. This index allows efficient stabbing queries, returning the complete state of the system at a given time. A node of the tree is a key-value pair, where the key is a path component, and the value is an attribute associated with a duration, that gets updated as the trace is being processed. The rules, by which attributes are updated, are established by our algorithm presented in section 'Multi-level trace analysis'.

Our algorithm requires kernel traces from all systems in the setup, i.e., the host and guests operating systems. Events from these traces are then merged and sorted by chronological order for processing. Trace Compass reads the trace one event at a time and modifies the SHT attributes. Figure 1 shows a part of our SHT. For instance, the path "`/Virtual Machines/Ubuntu/CPUs/CPU0/Current Thread`" contains the thread ID executing on CPU 0 of the VM named "Ubuntu". When a scheduling event such as *sched_switch* from the VM's trace is processed, the value of the attribute is changed from the TID of the former thread to the latter's. Similarly, the attribute at path "`/Host/Threads/Thread 1234/Status`" holds the status of the thread whose TID is 1234 on the host. This attribute may be modified when a context switch event involving the thread 1234 is being processed.

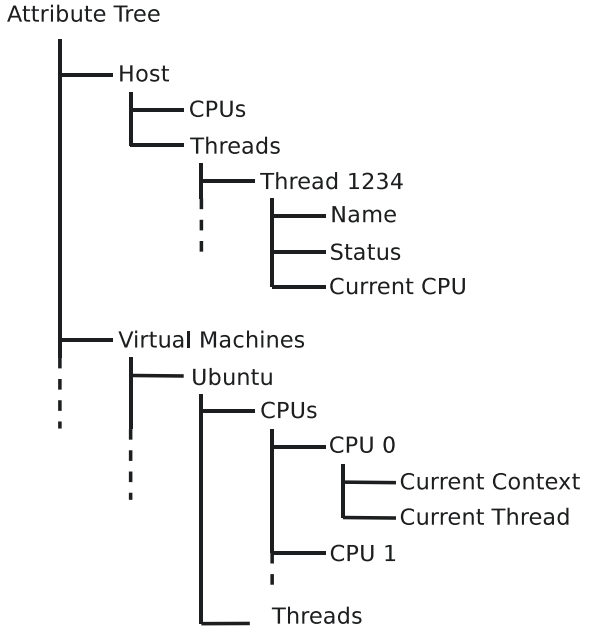

Figure 1 Example of a state history tree.

Relevant tracepoints

In this study, we use LTTng as a kernel tracer. LTTng was designed for high throughput tracing while reducing as much as possible its impact on the traced system [17]. We now introduce the key tracepoints for our analysis. We present their significance, as well as the content of their respective payload. This section is complementary to section 'Multi-level trace analysis' which explains how these tracepoints are used to update the SHT.

The `sched_switch` tracepoint indicates a context switch on the CPU which recorded the event. Useful payload fields are the names and the TIDs (Thread Identifiers) of the former and new threads involved in the context switch. Since all events are timestamped using the system's time at the nanosecond scale, the amount of time spent on each CPU by a specific thread is easily computed by subtracting the timestamps of the `sched_switch` events involving a particular thread.

Tracepoint `sched_migrate_task` indicates the migration of a thread from one CPU to another. Its payload holds the TID of the migrated task, as well as the origin and the destination CPU identifiers. Tracepoint `sched_process_ fork` indicates the creation of a new process, and exposes the names, PIDs (Process Identifiers) and TIDs of the newly created process as well as its parent's. Its complementary event, `sched_process_exit`, records the end of life of a thread. The payload contains the name and TID of the process.

VMX mode transitions by KVM can be tracked by enabling the `kvm_entry` and `kvm_exit` events. Tracepoint `kvm_entry` indicates a transition from root to non-root modes, and thus the beginning of the execution of the VM's native code. On the other hand, tracepoint `kvm_exit` indicates the opposite transition, which interrupts the execution of the VM and gives control to KVM. Elapsed time between consecutive `kvm_exit` and `kvm_entry` events represents overhead introduced by the hypervisor.

Trace synchronization

System timekeeping

LTTng uses the monotonic clock of the kernel for timestamping events, rather than the raw TSC. It avoids architecture-dependent limitations inherent to the TSC, such as TSC synchronization between cores and non-constant TSC on variable frequency CPUs. Even in the case of an ideal TSC (invariant and synchronized between cores), the value is based on the processor frequency, and thus needs to be scaled, or translated, for the user. Moreover, the TSC is an x86-specific register and using it as a clock source does not meet our requirement of portability. However, it is worth mentionning that the monotonic clock internally scales the TSC to nanoseconds and applies an offset to represent the current time, as shown in Equation 1:

$$t = T + f(TSC) \tag{1}$$

where T is a coarse-grained value updated on system timer interrupts. For a finer timekeeping, T needs to be adjusted using the TSC to account for the elapsed time since the last update (last timer interrupt). This is done using function *f()*, which translates the TSC to an actual time value that can be used for fine-grained timekeeping.

In addition, the monotonic clock guarantees total ordering, even in the case of modification of the system's wall clock time while tracing, and therefore is an ideal source for event timestamps.

Although the TSC is paced at the same rate across the different virtual systems, the offset values T of each system are not, and thus are subject to drifting apart as time goes by. In fact, modern tickless operating systems disable timer interrupts on idle processors to reduce energy consumption. As a result, the update period of T is variable, which may contribute to increase the time difference between systems. Furthermore, virtual machines may be set to different timezones, introducing even more incoherent timestamping when traces are merged together, which would make them appear as being recorded at different moments. As a result, high precision timestamping and clock drifting do not allow for simple clock offsetting to ensure coherency between traces, and require a specific synchronization method. The next section presents our approach to ensure coherent trace merging.

Event matching

We use the fully incremental convex hull synchronization algorithm to achieve offline trace synchronization, introduced by [18] for distributed traces synchronization. Each guest trace is processed individually and synchronized according to the host's trace whose timeline is taken as a reference. This approach is based on event matching between two traces. In order to use the synchronization algorithm, an event a from one trace must be associated to another complement event b from the other. Each couple of events $\{a, b\}$ must respect the following equation:

$$a_{T_1} \xrightarrow{k} b_{T_2} \tag{2}$$

More formaly, the following requirements have to be met:

1. Causality: a must (quickly) trigger b;
2. Bijection: a and b must share a common and unique key k in their payloads;
3. Every event b must be matched to at most one event a (one-to-one). Unmatched events b are ignored.

The key k is used to ensure a one-to-one relation between a and b. A lower delay between events a and b results in a more precise synchronization scheme. A synchronization formula is then derived by the algorithm which is a function of clocks offset and time drift. This formula is then applied to all timestamps of the guest's trace, bringing them to the same timebase as the host. By using the relation "a triggers b", a lower bound is imposed on the timestamps of events b as they cannot appear before their matching event a. Events between two consecutive events b are then adjusted to respect this constraint. With that being said, an upper bound has to be imposed as well to events b. To set an upper bound on events b, we use the same matching approach in the opposite direction between systems. If a is an event in the guest OS that triggers b on the host, then an event c on the host that triggers an event d on the guest is needed. Figure 2 shows events a, d, b, c from the original traces correctly reordered as a, b, c, d after synchronization. The next section explains how we used and adapted this method for virtual machines.

Implementation in virtualized systems

Previous section 'Event matching' explained the theory of the fully incremental convex hull algorithm for trace synchronization. However, the requirements for this algorithm are not directly met in the case of virtual machines. Originally, the algorithm was built based on TCP packet exchange events, where `send` and `receive` events are

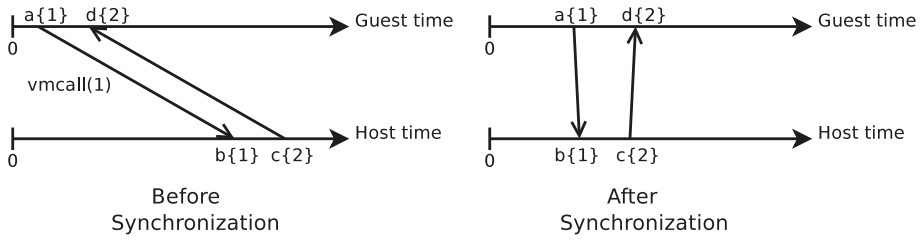

Figure 2 Lower and upper time bounds with matching events are used to synchronize traces.

respectively a and b: `send` triggers `receive` and the TCP packet number is the key k. In Cloud environments, virtual machines do not necessarily exchange TCP packets with each other or with the host, as VMs are usually provided to different clients. As a result, we need to customize the setup of the virtual machines to generate events both in the VMs and on the host that would respect the requirements established earlier. This section introduces our approach to obtain events that can be used to achieve trace synchronization.

We added tracepoints to the kernel through a loadable module for flexibility, so no modification to kernel code would be required to perform trace synchronization. Upon loading, this module registers a probe to the system timer's interrupt. In other words, every time the system timer issues an interrupt to the CPU, our synchronization routine will be invoked. The synchronization routine can be summed up as follows:

- Guest: Trigger hypercall (event a)
- Host: Acknowledge hypercall (event b)
- Host: Give control back to the VM (event c)
- Guest: Acknowledge control (event d)

The first pair of events (a, b) can be simulated by issuing a hypercall. When executing the `vmcall` instruction from the guest OS (event a), a trap is generated by the CPU and control is given to the hypervisor, which in turn acknowledges the trap (event b). A counter X is passed to the host OS as a parameter to the hypercall. This parameter will serve as the shared key required by the synchronization algorithm. As a result, events a and b are both recorded in a short period of time on the guest and host OS respectively, both holding the same value X as their payload.

Simulating the pair of events (c, d) is not as trivial since different constraints are imposed on the host-to-guest communication, as no mechanism of parameter transmission is easily accessible. Implementing shared memory between the guest and host is too intrusive as it would add too much complexity to both systems, and would probably require modification to both kernels. However, the trap generated by the hypercall is virtually invisible to the guest OS, which continues execution "normally" after involvement from the hypervisor. We can take advantage of this property to simulate a parameter transmission when the hypercall handling returns. Event c is recorded on the host right before it finishes the synchronization routine and gives control back to the VM. Event d is recorded on the guest right after the hypercall, which effectively is as soon as the guest OS resumes execution. This model simulates property (1) as c indicates that the host is giving control to the VM and d represents its acknowledgement. Both these events hold $X + 1$ in their payloads to respect the one-to-one relationship.

The downside of this approach is the overhead introduced by the hypercall. Table 1 shows overhead measurements added by the hypercall, with and without tracing. However, registering to the system timer interrupt takes advantage of tickless kernels as they aim to reduce energy consumption by disabling interrupts on idle CPUs. In other words, the synchronization routine is not invoked on idle virtual machines, which otherwise would trigger a costly context switch on the host for no actual work.

Once traces are generated on both systems, the fully incremental convex hull algorithm is applied, which derives a synchronization function applied on all of the guest's timestamps. This approach is resistant to clock drifts as the convex hull algorithm considers this issue and compensates for it in the generated formula. Additionally, it does not require `TSC_OFFSET` tracking or any other architecture-specific configuration.

Synchronization results

To show the results of our trace synchronization algorithm, we traced simultaneously a running virtual

Table 1 Overhead induced by the hypercall

	Time (ns)		
	Without tracing	**With tracing**	**Relative**
One synchronization tracepoint	102	153	50.0%
Hypercall round-trip	5168	5565	7.7%

machine and its host. We then merged the traces recorded from both systems and used Trace Compass to view the result.

We show in Figure 3 the state of threads on different systems (the color legend is shown in Table 2). Thread `qemu:Debian` with TID 7030 serves as a virtual CPU of the VM as seen from the host. Events from the host's trace are used to recreate its state. Thread `wk-pulse` is a periodic CPU workload (in a pulse-like manner) running inside the VM. Therefore, events from the VM's trace are used to show its state. We can already expect that the vCPU of the virtual machine will follow a pulse-like pattern, as the guest system is mostly idle. On Figure 3, the staggered start of `wk-pulse` indicates a time gap of about 6 seconds between the host's and guest's clocks. We then used our synchronization algorithm to correct the guest trace's timestamps and reused the same view in Trace Compass to view the result, as shown in Figure 4. We clearly see that `wk-pulse` is running on vCPU `qemu:Debian` because of their simultaneous state transitions. It is worth mentioning that the states of threads `qemu:Debian` and `wk-pulse` are computed independently from each other, yet they appear almost in perfect sync after applying the synchronization formula.

Multi-level trace analysis

This section presents how the state of each virtual CPU of a VM is recovered and rebuilt by analyzing the merged traces. The purpose of the analysis is to show the state of the vCPU throughout the trace as seen from the host. Our module parses the resulting trace and updates the attributes of the state system after each processed event. Moreover, we want to show the impact of preemption on the threads running within a VM. This analysis is useful as it shows the effective running time and execution of a thread compared to what is visible to the guest operating system. A vCPU at any time can be in one of the following states: `VMM`, `RUNNING`, `IDLE` or `PREEMPTED`. The state attribute of each vCPU can be found in the state system at path "`/Virtual Machines/VM Name/CPUs/vCPU ID/Status`. Figure 5 is a FSM (finite state machine) that shows transitions between these states. All of the events that trigger transitions originate from the host. Although not included in Figure 5, events from the virtual machines' traces are used to rebuild the states of the threads running on each vCPU within a VM. These threads can be found in the state system at paths "`/Virtual Machines/VM Name/Threads/TID/Status`". Following section 'Virtual CPU states' explains these states as well as the transitions by which they can be reached.

For clarity, we introduce the term pCPU which designates a physical CPU, as opposing to a vCPU which is in reality a QEMU thread emulating the CPU of a VM.

Virtual CPU states

VMM

State `VMM` represents the state when a QEMU thread is running hypervisor code instead of virtual machine code. In other words, it represents participation or involvement from the VMM, as to provide emulation, inject an interrupt into the guest's OS, or any other instruction requiring the external help of KVM. As explained in section 'Hypervisor', CPU transitions between non-root and root are instrumented with tracepoints `kvm_entry` and `kvm_exit` respectively. When a `kvm_exit` event is reached, the vCPU's state is set to `VMM` (transition 2). On the other hand, it leaves this state on a `kvm_entry` event, returning to the state it was in prior to the 's involvement (transition 1). This state serves as an intermediate between any two states, as hypervisor cooperation is required for QEMU threads scheduling.

We also noticed that this state is reached everytime a QEMU thread is involved in a context switch, i.e., when a vCPU is scheduled out of a pCPU. Interestingly, when a QEMU thread is selected by the scheduler to run again, it first executes in the `VMM` state before explicitly invoking the `vmentry` instruction to give control to the guest's OS. This procedure is required because KVM needs to execute specific operations related to the Virtual Machine Control Structure of the VM. KVM uses Linux's notifier chains to "register" on context switches involving a vCPU. When a vCPU reaches this state, its current thread's status is set to `PROCESS_VIRT_PREEMPTED`, which designates wasted time due to the virtualization layer (we see it as preemption to execute hypervisor code, the thread is marked as "virtually preempted").

RUNNING

`RUNNING` shows execution of the VM's code. When in this state, a virtual CPU is considered as running without any involvement from the hypervisor, and instructions dedicated to a specific vCPU are running directly on one of

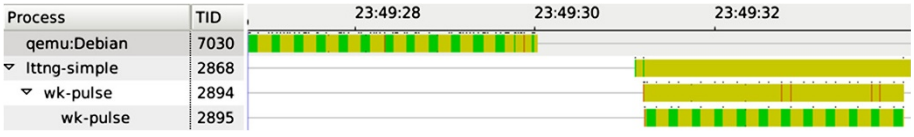

Figure 3 Merged traces without synchronization.

Table 2 Virtual Machine Analysis Color legend

Color	State
Green	Running
Yellow	Blocked
Orange	Preempted
Grayed out green	PROCESS_VIRT_PREEMPT

the host's pCPUs. For this state to be reached, two conditions must be satisfied. First, the QEMU thread emulating a vCPU must be in a running state on the host operating system. Secondly, in the guest operating system, the CPU associated with the specified QEMU thread must be in the running state as well, meaning that any process other than the idle task (`swapper`) is executing on the CPU.

IDLE

`IDLE` represents a state when a vCPU is not executing any code, and thus voluntarily yields the physical CPU. This state is reached when the QEMU thread emulating a vCPU is scheduled out of a pCPU, and if no thread other than the `idle` task is scheduled to run on this vCPU in the guest OS (transition 4). On Linux, the purpose of `swapper` (the `idle` task) is to invoke the scheduler to choose potential threads ready for execution, or to halt the CPU in case no thread is ready to run. The vCPU goes out of this state as soon as the thread emulating it gets scheduled back on the host (transition 3).

PREEMPTED

`PREEMPTED` is the state that indicates direct latency to the execution of a virtual machine. This state is reached when a vCPU is scheduled out of the pCPU by the host's scheduler (transition 5), while the vCPU was effectively serving a thread. Note that the running process on the vCPU stays in the `PROCESS_VIRT_PREEMPTED` state, which indicates that the vCPU on which the thread is running was preempted on the host operating system. Usually, this kind of information is not visible to a virtual machine, though it directly impacts the completion time of a task by introducing delays throughout the execution. As a result, a task may seem to complete in much longer than the effective time during which it was running. When scheduled back in (transition 6), the vCPU passes by the `VMM` state again to finally reach the `RUNNING` state and resume VM code execution.

Illustrative example

We launched a thread that computes a Fibonacci sum on what appeared to be an idle virtual machine. The computer used was an Intel i7 (Nehalem) with 4 hyper-threaded cores (8 logical CPUs), 8 GB of RAM, 1 TB HDD, and running Debian GNU/Linux. Using `top`, no CPU-intensive thread was reported in the VM, and the *steal time* column showed a 0% vCPU preemption. Figure 6 shows the state of the Fibonacci task (thread `fibo`) as seen from the guest operating system. This view shows a monopoly of the CPU and a 100% utilization by the `fibo` task for the whole duration of the trace. This state has been reconstructed by processing only the trace recorded on the guest.

Figure 7 shows the result of our analysis module for the same experiment, after the host's and guest's traces merging and synchronization. We notice that vCPU 0 of the "Debian" VM is constantly transitioning between states `RUNNING` (green) and `PREEMPTED` (purple). With proper zooming, we can see `VMM` state as an intermediate for every transition. These transitions have direct repercussions on the execution of the `fibo` task, which is in turn moving between states `RUNNING` (green) and `PROCESS_VIRT_PREEMPTED` (grayed out green). With a quick look at the graphical view, we can see that the Fibonacci sum could potentially execute approximately twice as fast on a fully available pCPU, or less loaded host system.

The reason why `top` reported a 0% vCPU preemption (steal time), before starting the Fibonacci task, is because the vCPU was mostly idle. As a result, when it asks for CPU time, its request is immediately answered by the host's scheduler as it has the "highest priority" due to its idle nature. We can see that using such a tool to measure resource availability can actually be misleading. The only way to detect vCPU preemption using `top` would be to actively monitor the steal time while running the Fibonacci task.

Execution flow recovery

We now reach the second part of the analysis, which is to reconstruct the execution flow for a specific task of one of the virtual machines. The execution flow with regard to

Figure 4 Merged traces with synchronization.

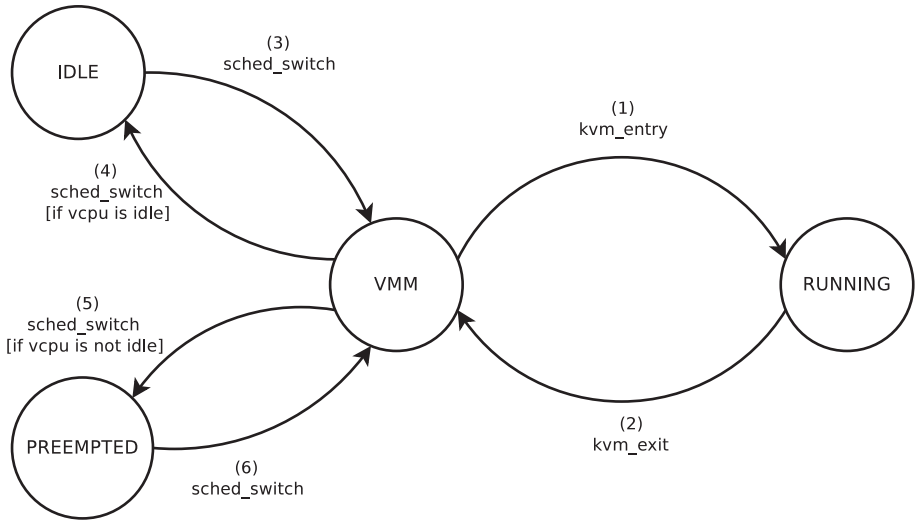

Figure 5 vCPU state transitions.

a certain task `A` is defined as the ordered set of execution intervals of all the tasks affecting the completion time of `A`. The purpose of the execution flow is to show detailed information about the execution of a certain thread as well as its interactions with other threads.

In the scenario shown in Figure 8, the execution flow is computed with regard to task `A`. The timeline shows the start and the end of the lifespan of this task, thus the analysis is time-bounded. In this example, it is clear that task `A` yields the CPU to allow execution of other tasks `B` and `C`. The scheduler then selects `A` after a certain amount of time, letting it complete its execution. Therefore, the completion of `A` was affected by the execution of `B` and `C`. When flattened, the execution intervals of all the threads form one continuous execution interval which represents a busy CPU for the duration of the trace. Although this kind of information could be vaguely suspected from top-like tools, this level of detailed information is necessary for an advanced analysis of latency sources.

For a task executing inside a virtual machine, the computation of the execution flow should be adjusted to take into consideration interactions between different operating systems through the usage of shared resources. The objective of such an analysis is to provide detailed information not only about the execution of a certain task, but also about its interactions with other threads, whether they belong to the same VM, the host, or even a different VM. With such information, major causes of overhead can be easily tracked down by the host's administrator, and adjustments can be made to resolve the issues. Recovering the execution flow comes down to tracking all preemption events involving `A`. Causes of preemption can be within the same operating system and thus easier to investigate, or from a different system making them almost completely hidden. In this section, we show that the execution flow recovery can be computed simply by querying the state system for key attributes modifications, without having to read the trace again.

Implementation

In this section, we call *A* the thread around which we want to recover the execution flow. The first step of the algorithm is to find all the entries involved in the execution flow according to task *A*. In Figure 8, each of tasks A, B and C represents an entry. First, all the threads of all the systems are inserted as entries in the execution flow. This list of all the threads across systems can be recovered by parsing through attributes "`/Virtual Machines/*/Threads/*`" and "`/Host/Threads/*`" in the SHT. The second step of the algorithm is to compute the execution intervals of each entry with regard to task `A`. As a final step,

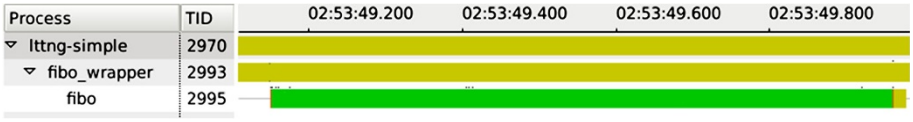

Figure 6 View of Fibonacci experiment with traditional analysis.

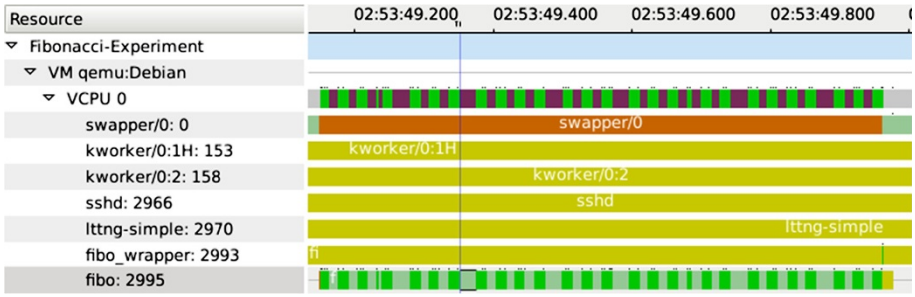

Figure 7 View of Fibonacci experiment with virtual machine analysis.

we remove the entries that have minimal or no impact on the analyzed thread according to a minimal impact threshold. The selection of the threshold doesn't affect the computing time of the algorihtm, as it is performed only after the whole algorithm has executed and all the durations have been computed. The impact of each thread can be measured using Equation 3, as explained in section 'Investigation of execution anomalies'.

To respect the relationship of affiliation between a thread and its system (host or VM), entries are stored in a tree-like structure with a depth of 2, where each node on the first level represents a system, and its children on the second level represent its threads.

As mentioned earlier, the execution flow can be represented with an ordered list of intervals, where each interval contains a start time, end time, a state, and the TID of the thread executing for the said interval. Algorithms 1 and 2 explain how this list can be built, recovering the execution flow with regard to task A.

Algorithm 1 is used to insert all intervals of A holding the "`RUNNING`" state. As a first step, we query the SHT to retrieve all modifications to the "`Status`" attribute of thread A. The SHT returns a list of intervals for different values of this attribute. Algorithm 1 parses this list and each interval holding the "`RUNNING`" state is directly inserted in the `result` list. However, for each interval holding the "`PREEMPTED`" value, a separate function is invoked to find which thread is preempting A. This function is shown in Algorithm 2.

Algorithm 1: Recovering the execution flow: inserting intervals in the `RUNNING` state

```
Input: StateHistoryTree s
List result; // the list of execution intervals
StatusIntervals intervals = Query status intervals of A
from s;
for each interval in intervals do
    currentPCpu = Query current pCPU of A;
    if interval.state == RUNNING then
        result.insert(interval);
    else
        result.insertAll(resolve(s, interval,
        currentPCpu));
    end
end
return result;
```

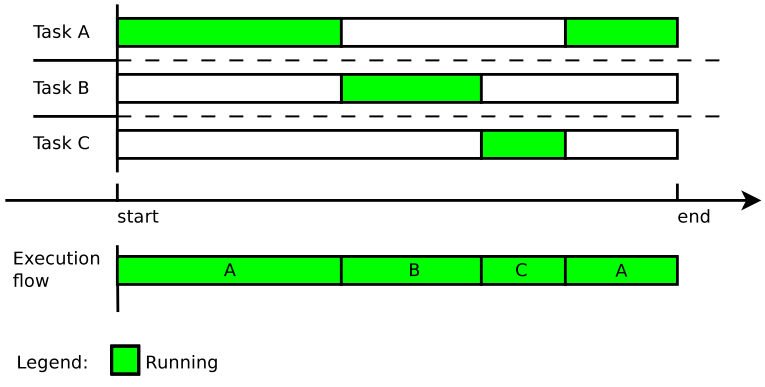

Figure 8 Simple example of an execution flow.

The `resolve` function requires an interval as well as a pCPU Id as input values. The work done by this routine is to find which threads are running on the pCPU for the duration of the interval. In the case where the running thread is the vCPU of another VM, this function will then query the state system to get the running thread inside this VM. Once the running thread preempting *A* is deduced, it is returned to Algorithm 1 which will insert it in the `result` list.

Algorithm 2: Function `resolve()`: Querying the state history tree to get the running threads while `A` is preempted

```
Input: interval
Input: pCpu
Output: outList
start = interval.start;
end = interval.end;
ThreadIntervals = Query "Current Thread" intervals
of pCpu between start and end;
for each interval t in ThreadIntervals do
    if t.tid is a vCPU then
        intervals = Query "Current Thread" intervals
        of t between t.start and t.end
        outList.addAll(intervals);
    else
        outList.add(t);
    end
end
/* All intervals inserted in outList are in RUNNING
state */
return outList;
```

Finally, in the `result` list returned by Algorithm 1, each interval "`interval`" of the list respects the following rules:

$$prevInterval.end = interval.start - 1$$
$$interval.end + 1 = nextInterval.start$$

Where `prevInterval` and `nextInterval` are respectively the previous and next intervals of `interval` in the ordered list `result` from Algorithm 1.

Use cases

This section shows how our work can be used in real-life cases to investigate latency in virtual machines. We start with a follow-up on the Fibonacci example introduced earlier (section 'Follow-up on the Fibonacci Case'). We then present different use cases that show either how to investigate a known issue (section 'Investigation of execution anomalies'), or a general analysis to verify normal execution of the system as a whole (Investigating a residual timer).

Follow-up on the Fibonacci Case

Figure 7 showed that the Fibonacci took longer to execute due to preemption of the vCPU on which it was running. We follow-up on the matter by recovering the execution flow of the experiment. Figure 9 shows the result; we can see that the preemption is due to a single CPU-intensive process on the host called `burnP6`.

Investigation of execution anomalies

We now show a use case of a performance issue which can be easily tracked down using our graphical views. We developed a CPU-intensive task, named `critical_task` which computes for approximately 280 ms. A script spawns a `critical_task` thread in a periodic fashion, asynchronously every second (without waiting for completion). We traced the host and the guest operating systems simultaneously. Figure 10 shows only the guest's trace over 7 seconds, for 7 `critical_task` threads. We can see the rate of thread forking at one thread per second. However, executions 3, 4, 5 and slightly 6 (threads 3523, 3525, 3527 and 3529) show abnormal computing duration although they appear as running (green) without interruption.

We then merged and synchronized kernel traces, and used our first graphical view to analyze the execution, as shown in Figures 11, 12 and 13. In Figure 11, the vCPU 0 line shows transitions between states `IDLE` (gray) and `RUNNING` (green) which indicates that the VM is mostly idle, except when `critical_task` threads are spawned. Additionally, still by looking at the vCPU 0 line, we can clearly see vCPU preemption (purple) for executions 2, 3, 4 and 5. These vCPU preemptions translate into unexpected latency on threads 3523, 3525, 3527 and 3529, as shown by their respective lines in the view

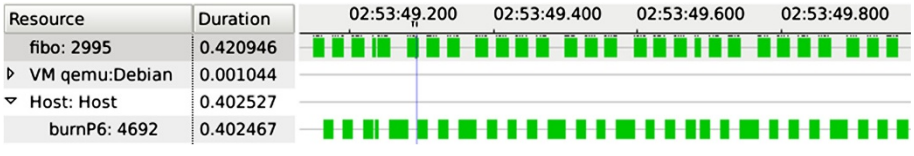

Figure 9 Execution flow recovery of previous Fibonacci experience.

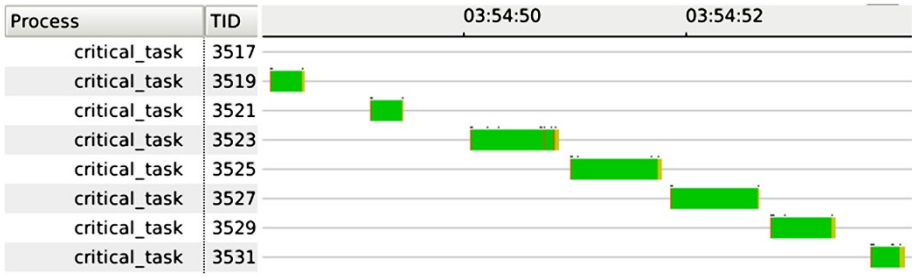

Figure 10 Execution of a periodic task as perceived by the VM. In some cases, the execution inexplicably takes longer to compute although the task appears as running.

(greyed out green). This is an indicator that the VM is not allocated enough CPU time by the host's scheduler, which is time slicing the pCPU more or less equally among the host processes. We also notice that vCPU0 of the VM Ubuntu woke up from the `IDE` state into the `RUNNING` state for this period of time, which might be the reason for the latencies on the critical tasks running in VM Debian. Figure 12 shows the state of two vCPUs of different virtual machines, respectively Debian and Ubuntu. For simplicity reasons, the Figure shows only the timeframe for the lifespan of `critical_task` with PID 3525. The Figure suggests that these two vCPUs are complimentary in their execution, as when one of them is running, the other one is preempted. This is a strong indicator of a shared resource between these two vCPUs. Moreover, the Figure also shows that both vCPUs are simultaneously being preempted for small amounts of time, which suggests that another thread, outside of the scope of these VMs, is also competing over the same resource. Figure 13 is a magnified view of Figure 11 over the lifespan of `critical_task` 3525, showing how the preemption of vCPU0 is perturbating its execution.

Finally, as a last step, we recovered the execution flow to investigate the source of this latency. The result is shown in Figure 14. The execution flow is centered around thread 3525, which seems to be the execution of the `critical_task` with the most latency. For clarity reasons, we only show a part of the lifespan of the `critical_task`. The view shows that the pCPU is shared amongst three operating systems: Debian (the VM in which the critical tasks are running), Ubuntu (which is another virtual machine) and the host. We notice that threads `burnP6` from the host and `cc` from the Debian VM both strongly preempt the critical task, which explains its excessive duration. The "Duration" column shows the duration of the preemption of each process for the lifespan of thread 3525. For system entries (non-leaf nodes), the "Duration" number indicates the sum of all their threads, ie. the time for which the whole system preempted the analyzed task.

We can see that the critical task ran for 274 ms (as expected, however it was over a longer period), the Ubuntu virtual machine ran for 270 ms and the host ran for 260 ms. These numbers indicate approximately a 33% usage of the CPU for each system, which indicates that the pCPU is strongly shared amongst them. Moreover, we see that process `irq/46-iwlwifi` executes for 296 us, indicating heavy network usage and packet processing.

For each thread T, the proportion of time for which it preempted `A` is computed using Equation 3, where T_{out} is

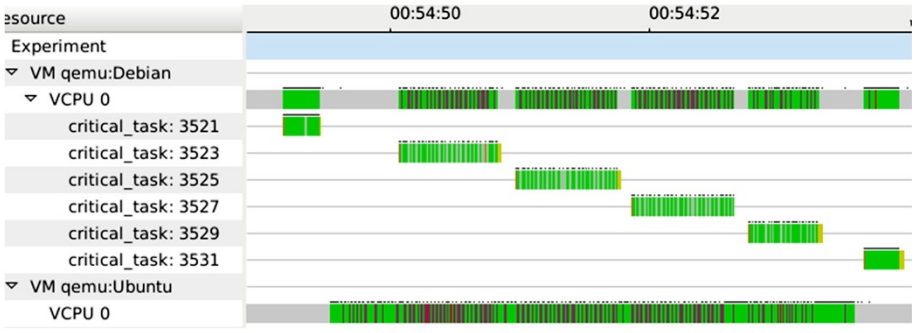

Figure 11 We can see that the vCPU on which the critical task is running is actually being preempted on the host, which impacts the execution of the running thread.

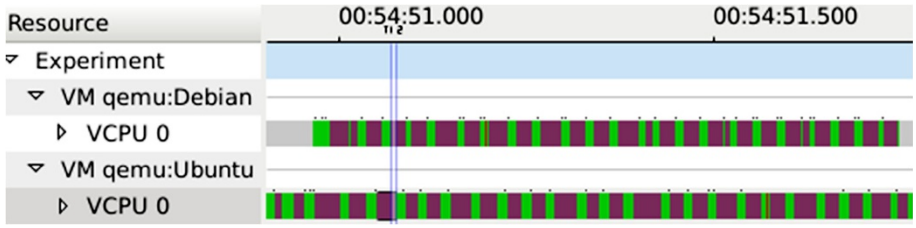

Figure 12 vCPU0 of Debian and vCPU1 of Ubuntu are taking turns in exection, which indicates competition over a shared resource. The figure also shows some timeframes in which both vCPUs are preempted, most probably by a different thread on the host system.

the timestamp indicating a scheduling out and T_{in} is the timsetamp indicating a scheduling in.

$$D(T) = \frac{\sum_{A.start}^{A.end} T_{out} - \sum_{A.start}^{A.end} T_{in}}{A.end - A.start} \quad (3)$$

Investigating a residual timer

We now present a use case that helped us investigate an unexpected operating systems problem. Figure 15 shows the result of our analysis for a workload similar to the one presented in the previous use case (section 'Investigation of execution anomalies'). We first see that the analyzed task is sharing the CPU with threads `cc` from the VM "Ubuntu" and `burnP6` from the host. However, for the second half of the analysis, the ciritcal task is being preempted by `swapper`, the idle task, from "Ubuntu". Such a behavior seems problematic as control is taken away from the analyzed thread to serve an idle thread. With a quick look at the trace when `swapper` is scheduled, we noticed events indicating the expiration of a timer. It turns out that a periodic timer was scheduled in the virtual machine, which would require CPU time for a very short period to acknowledge each timer expiration. This behavior introduces significant overhead as context switches are somewhat costly on the host system. We clearly see the use of such an analysis specifically for virtualized systems. While acknowledging an expired timer on an idle physical machine only consumes a few CPU cycles and little energy, it is much more costly in a virtualized system since it generates a context switch on the host. To sum up this example, we saw how a "forgotten" timer in one virtual machine can affect the execution of others. Such a problem can be easily fixed by the system administrator once it is located.

Flexibility and portability

Throughout this project, we set different constraints to ensure for a portable and flexible solution to our initial problem. First, we used the State History Tree as an abstraction for the traces. The SHT not only delivers performance enhancement for event querying in the trace [15], but allows to dissociate the analysis step from the trace itself. In other words, multiple trace parsers can be used to handle the kernel traces, regardless of the operating system on which they were recorded, or the tracer used, as long as the trace format is open. As long as the backend used for trace representation is the SHT, and given that the required events are reported in the trace, our proposed algorithms will produce the expected results, which accounts for both portability (independant from the OS) and flexibility (independant from the tracers and trace formats). It is worth mentioning that although we used LTTng as a kernel tracer for this project, any other kernel tracer could have been potentially used as long as a trace parser is available.

Moreover, as we explained in previous sections, although a TSC-based approach for trace synchronisation is potentially simpler to use under certain conditions, the TSC is an x86-specific CPU register. Using a higher-level algorithm such as the fully incremental convex hull algorithm provides portability to the synchronisation solution. And, although using hypercalls as a communication

Figure 13 Magnified view of the execution of `critical_task` 3525.

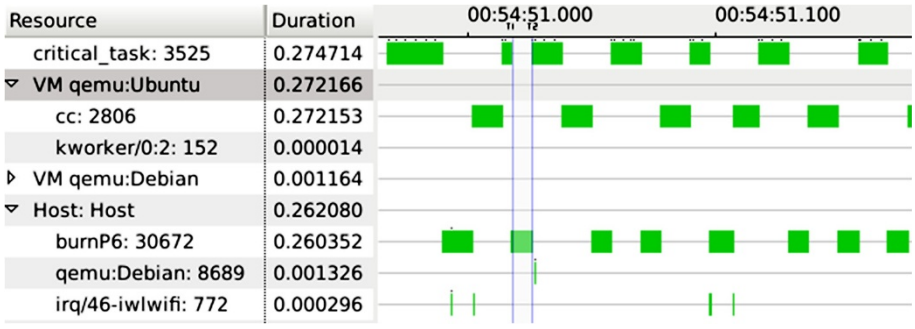

Figure 14 Execution flow recovery of problematic `critical_task`.

mechanism between the host and its guests is specific to hardware virtualization, any pair of events across systems with a causality relation can be a potential replacement, such as network packets exchange.

Finally, it is worth mentioning that the only KVM-specific tracepoints are `kvm_entry` and `kvm_exit`, which represent VMX mode transitions. Since these transitions are common to all hypervisors supporting hardware-assisted virtualization, our approach is therefore not specific to KVM. And, although these tracepoints are already included in Linux's source tree, they can be added in any hypervisor by simply instrumenting all calls to `vmentry` and `vmexit` instructions which requires very little effort, thus allowing this model to be used with any hypervisor. Moreover, if the administrator chooses not to instrument these transitions, little information would be lost, as the only state lost in Figure 5 would be `VMM`. Preemption and execution recovery would still be possible with little analysis precision lost (hypervisor involvement would account as effective CPU time instead of overhead due to virtualization). Furthermore, kernel traces generated from other operating systems can be used as well with minimal effort. As long as the events required to cover the FSM presented in section 'Multi-level trace analysis' are available, the model can be ported by simply specifying the names of these events. Moreover, in the case of microcomputers without hardware virtualization, the synchronization approach could potentially be extended to any other type of communication between the guest and the host, such as a TCP packet exchange. The rest of the analysis is based on the state system built, and thus does not depend on the details of the underlying traces.

Conclusion

Cloud computing and virtualization are evolving at a rapid pace. These emerging technologies created a need for analysis tools that can live up to the technological advance. In this paper, we showed that kernel tracing can be used to analyze the execution of virtual machines under such conditions. We first proposed an approach to resolve the problem of clock drift and offset between operating systems. We then showed how the merged traces can be processed to rebuild the state of the virtual machines, as well as their vCPUs, throughout the trace. Finally, we explained how the execution flow with regard to a certain thread can be rebuilt for an in-depth analysis of its execution and interactions with other systems. All

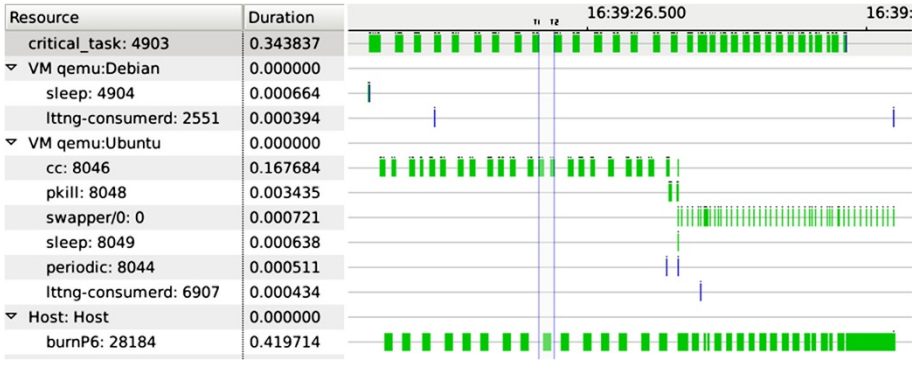

Figure 15 Execution flow recovery of problematic `critical_task` with a periodic timer in another VM.

the solutions proposed in this paper were designed with requirements of portability and flexibility in mind. As a result, all the approaches explained are portable across operating systems, computer architectures, and complementary software (tracer and hypervisor).

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

Authors' contributions
MG 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, did research on the synchronization algorithms and developed an algorithm specific for virtual machines. He also implemented the analysis tool presented in this paper, as well as most of the experiments. FG contributed to the implementations of the analysis tool and the synchronization algorithms, provided general input to the study and oriented the experiments. He also participated in the writing of this article. 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 Ericsson for their input to this study as well as for funding this research project. We also thank Geneviève Bastien for her help in the Open Source project Trace Compass and Naser Ezzati Jivan for reviewing this paper.

Received: 7 August 2014 Accepted: 3 December 2014
Published online: 31 December 2014

References

1. Padala P, Zhu X, Wang Z, Singhal S, Shin K (2007) Performance evaluation of virtualization technologies for server consolidation. HP Labs Technical Report HPL-2007-59
2. Barker SK, Shenoy P (2010) Empirical evaluation of latency-sensitive application performance in the cloud. In: Proceedings of the 1st annual ACM SIGMM conference on Multimedia systems (MMSys'10). ACM, Scottsdale, AZ, USA. pp 35–46
3. Bueso D, Heymann E, Senar MA (2012) Towards efficient working set estimations in virtual machines. Jornadas Sarteco:1. Elx, Spain
4. Fournier P-M, Dagenais MR (2010) Analyzing blocking to debug performance problems on multi-core systems. ACM SIGOPS Oper Syst Rev 44(2):77–87
5. Du J, Sehrawat N, Zwaenepoel W (2011) Performance profiling of virtual machines. In: In Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '11). ACM, New York, NY, USA Vol. 46. pp 3–14
6. Anand A, Dhingra M, Lakshmi J, Nandy SK (2012) Resource usage monitoring for kvm based virtual machines. In: Proceedings of 18th annual International Conference on Advanced Computing and Communications (ADCOM 2012). IEEE, Bangalore, India. pp 66–70
7. Khandual A (2012) Performance monitoring in linux kvm cloud environment. In: Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on. IEEE, Bangalore, India. pp 1–6
8. Shao Z, He L, Lu Z, Jin H (2013) Vsa: an offline scheduling analyzer for xen virtual machine monitor. Future Generation Comput Syst 29(8):2067–2076
9. Yunomae Y (2013) Integrated trace using virtio-trace for a virtualization environment. In: LinuxCon North America/CloudOpen North America, New Orleans, LA. Keynote presentation
10. Agesen O, Garthwaite A, Sheldon J, Subrahmanyam P (2010) The evolution of an x86 virtual machine monitor. ACM SIGOPS Oper Syst Rev 44(4):3–18
11. Intel Corporation (2013) Intel® 64 and IA-32 Architectures Software Developer's Manual. Number 325462-045US
12. Fisher-Ogden J (2006) Hardware support for efficient virtualization. University of California, San Diego, Tech. Rep
13. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A (2007) kvm: the linux virtual machine monitor. In: Proceedings of the Linux Symposium, Ottawa Vol. 1. pp 225–230
14. Toupin D (2011) Using tracing to diagnose or monitor systems. IEEE Softw 28(1):87–91
15. Montplaisir-Gonçalves A, Ezzati-Jivan N, Wininger F, Dagenais M (2013) State history tree: an incremental disk-based data structure for very large interval data. In: Social Computing (SocialCom), 2013 International Conference on. IEEE. pp 716–724
16. Montplaisir A, Ezzati-Jivan N, Wininger F, Dagenais M (2013) Efficient model to query and visualize the system states extracted from trace data. In: Runtime Verification. Springer, Berlin Heidelberg. pp 219–234
17. Desnoyers M, Dagenais MR (2012) Lockless multi-core high-throughput buffering scheme for kernel tracing. ACM SIGOPS Oper Syst Rev 46(3):65–81
18. Jabbarifar M, Dagenais M, Shameli-Sendi A (2014) Online incremental clock synchronization. J Netw Syst Manage:1–15