Journal of Systems Architecture 106 (2020) 101709

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Using Xen and KVM as real-time hypervisors

Luca Abeni[a,*], Dario Faggioli[b]

[a] Scuola Superiore Sant' Anna, Pisa, Italy
[b] SUSE Software Solutions, Italy

虚拟化技术的最新发展使在虚拟机中执行复杂且对性能至关重要的应用程序变得可行。 其中一些应用程序具有实时约束，需要在物理核心上对虚拟机进行可预测的调度，因此，实时文献中的一些工作提出了高级调度和设计技术来遵守应用程序约束。 本文对这些工作进行了补充，调查了两个使用最广泛的开源虚拟机管理程序Xen和KVM引入的延迟。 还提供了一些适当配置VM的准则，以减少引入的延迟（以便可以在实践中使用以前的理论分析和算法），这表明KVM和Xen都可用作实时虚拟机管理程序。

ARTICLE INFO

Keywords:
Real-Time
Virtualisation
Xen
KVM

ABSTRACT

The recent developments in virtualisation technologies have made feasible the execution of complex and performance-critical applications in virtual machines. Some of such applications are characterised by real-time constraints and require a predictable scheduling of virtual machines on physical cores, hence several works in real-time literature have proposed advanced scheduling and design techniques to respect the application constraints. This paper complements those works, investigating the latencies introduced by two of the most widely used open-source hypervisors, Xen and KVM. Some guidelines for properly configuring the VMs in order to reduce the introduced latencies (so that previous theoretical analysis and algorithms can be used in practice) are also provided, showing that both KVM and Xen are usable as real-time hypervisors.

1. Introduction

In the last years, there has been an increasing interest in various forms of virtualisation technologies. As a consequence, the usage of Virtual Machines (VMs) is common across many different systems, ranging from large-scale servers (used in data centers composing large clouds) to small embedded systems, for which the composition of software developed by different vendors is a standard software engineering practice.

If the virtualised software components/applications are characterised by some kinds of temporal constraints, it is important to make sure that it is possible to provide *a-priori guarantees* on respecting such temporal constraints. Hence, it is becoming more and more important to provide real-time performance to applications running in VMs. This can be achieved by adopting well-known techniques from real-time literature, which propose various kinds of scheduling algorithms for VMs, and the corresponding theoretical schedulability analysis. Theoretical background on appropriate VM scheduling already exists and has also been implemented [1,2] in widely used hypervisors such as Xen [3].

Only few works, instead, investigated the real-time performance of existing hypervisors, which experimentally evaluates how the implementations of well-known scheduling techniques perform in practice. Such an analysis can be performed by evaluating (and controlling) the discrepancies between the theoretical VM schedule and the actual one generated by real hypervisors (answering the question: *given a VM scheduling algorithm, how much accurate is its implementation in the hypervisor?*).

Every hypervisor introduces some *latencies* in the VM schedule, and such latencies can affect the correctness and applicability of the existing real-time analysis. Most of the existing techniques from the literature are based on the assumption that the virtualisation mechanism introduces negligible latencies. Hence, being able to provide an upper bound to the latencies introduced by the hypervisor is fundamental to provide a controlled Quality of Service in clouds, micro-services, distributed applications based on VMs, or even embedded applications based on a component-based development approach. A similar analysis has been performed in the past considering μ-kernels [4], but its application to widely used hypervisors has not been investigated yet.

To provide a solid foundation for the development of such services/micro-services/applications, a preliminary version of this paper [5] evaluated and compared the latencies introduced by Xen and KVM, two of the most widely-used hypervisors. However, the previous investigation highlighted some issues with the Xen hypervisor, without providing any solution. This paper presents a more accurate analysis of the sources of those latencies and shows how to reduce them so that both KVM and Xen can be used as real-time hypervisors.

2. Definitions and background

A real-time application can be modelled as a set of real-time tasks, characterised by temporal constraints. A real-time task τ is a process or thread that can be seen as a sequence of jobs J_j, with job J_j arriving (the task becomes executable) at time r_j and finishing (the task blocks) at time f_j after executing for a time c_j. Each job is also associated with an absolute deadline d_j, which is respected if $f_j \leq d_j$. If $\forall j, r_{j+1} = r_j + T$ and $d_j = r_j + D$, then the task is said to be periodic with period T and relative deadline D. The response time of job J_j is defined as $f_j - r_j$.

* Corresponding author.
E-mail addresses: luca.abeni@santannapisa.it (L. Abeni), dfaggioli@suse.com (D. Faggioli).

https://doi.org/10.1016/j.sysarc.2020.101709
Received 29 July 2019; Received in revised form 18 November 2019; Accepted 7 January 2020
Available online 15 January 2020

The finishing time of each job depends on the CPU scheduler's decisions (if job J_j is scheduled as soon as it arrives, then $f_j = r_j + c_j$, otherwise it is larger, and $f_j - r_j - c_j$ depends on the scheduler). Hence, to respect the temporal constraints associated with each task, it is essential to use an appropriate scheduling algorithm and to use an admission test (a test that given a set of real-time tasks and a scheduling algorithm can check in advance if the tasks' deadlines will be respected). Real-time literature provides a lot of different scheduling algorithms (and associated schedulability tests), starting from the seminal paper by Liu and Layland [6].

If the considered real-time applications are executed inside VMs, then the system can be modelled as a hierarchy of 2 schedulers: a *VM scheduler*[1], that selects the virtual machine to be executed on the physical CPUs, and multiple *guest schedulers*[2] (one per VM). Again, real-time theory provides algorithms and mathematical tools that allow performing schedulability analysis for scheduling hierarchies like these, both in case of single-processor [7–11] or multi-processor/multi-core systems [12–16].

However, most of the previous research on real-time VMs focused on (root or guest) scheduling algorithms and theoretical schedulability analysis assuming that the schedule is respected, without any delays. Unfortunately, when considering real schedulers the scheduling algorithm is only a part of the equation, and implementation details are important: the actual schedule is a consequence of the theoretical scheduling algorithm *and* the accuracy of the implementation.

In practice, it can happen that tasks are scheduled with some delay with respect to the theoretical schedule. This delay, which is generally referred to as *latency* is due to non-preemptable sections in the system software (OS kernel, hypervisor, or even in other components, like language runtime, etc.) or to some other implementation details that are often not considered.

Informally speaking, the latency can be seen as a measure of the difference between the theoretical schedule and the actual one. More formally, latency and some other related concepts can be defined as follows:

Definition 1. Let e be an event, supposed to occur at time t according to the theoretical scheduling algorithm, that actually happens at time $t' > t$ on a real system.

The latency of the event e is defined as $l_e = t' - t$.

Definition 2. Given any possible event e, a latency upper bound L is defined as $\forall e,\ l_e \leq L$.

Definition 3. The smallest possible value *WL* that can be used as upper bound for the latency (i.e., that satisfies Definition 2) is called the worst-case latency.

Latencies introduced by the OSs (and, in particular, by the OS kernels) have been extensively studied in literature [17], and two main sources of latencies have been identified:

- The *timer resolution latency* is due to the periodic interrupt mechanism (the so called "jiffy", or "tick") used by many OS kernels
- The *non-preemptable section latency* is due to non-preemptable sections in the OS kernel, or to some mechanisms used by the kernel to handle concurrency (bottom halves, the Linux tasklets, soft-irqs, etc...)

Techniques and mechanisms for controlling and reducing the latencies caused by OS kernels have been envisioned, designed and developed. The latency being introduced by hypervisors, or, in general, by the various components of a virtualisation solution, on the other hand, has often been overlooked.

[1] Also known as host scheduler, or root scheduler.

[2] Also known as second-level schedulers, or local schedulers.

What can happen is that a VM should start its execution, according to the (theoretical) host scheduling algorithm, at time t. However, the hypervisor's scheduler may be, for some reason, invoked with some delay. The VM, therefore, is scheduled and starts running at time $t' = t + l$. In Section 6 it will be shown that, while OS kernels introduce latencies only due to timer resolution or non-preemptable sections, hypervisors' latencies can also be due to some forms of priority inversion.

If a worst-case latency *WL* exists and is well-known, for a given virtualisation solution, then it is possible to account for it in the schedulability analysis, by modelling it as a blocking time as big as of *WL* experience by all the VMs.

Hence, a "real-time hypervisor" should guarantee that the value of the latency is upper-bounded by a well-known value *WL*. Similar considerations apply to OS kernels: it can be stated that the difference between a real-time kernel and a general-purpose kernel is that the former provides a reasonable upper bound for the kernel latency, while the latency experienced in the latter can be much larger, or even unbounded.

If the latency introduced by the hypervisor is too high, real-time tasks running in a VM can miss deadlines even if theoretical analysis seems to indicate that they are schedulable. This is a real issue because some discrepancies between theoretical analysis and experimental results have already been observed in practice (see Section 7 in [16], for example). The delays in scheduling VMs can sometimes be due to hardware bottlenecks or to unpredictabilities of the CPU caches, and some works try to address some of the issues due to the hardware [18–20]. However, software issues due to the hypervisor structure have not been investigated yet.

As previously mentioned, techniques and mechanisms for reducing the latency introduced by an OS kernel (kernel latency, for short) have been developed and tested in literature. For example, the kernel latency can be reduced by using the so-called *dual kernel approach* [21–23]. An alternative approach would be modifying the kernel to reduce the size of non-preemptable sections. This second solution can be implemented by introducing fine-grained locking in the kernel and by using sleeping locking mechanisms (such as mutexes) instead of busy waiting (such as in spinlocks) to protect the kernel's critical sections. Besides reducing the kernel latency, mutexes also have the advantage that they allow to use some real-time resource sharing protocol (one of the most famous of which is priority inheritance [24], but there are many others) so that it is possible to compute an upper bound for the blocking times. However, to use mutexes in interrupt handlers, the handlers must be executed in dedicated kernel threads (so that they can block when trying to lock an already locked mutex). This approach is implemented, for example, by the Preempt-RT kernel patch for the Linux kernel [25,26].

Note that, when an OS is executed inside a VM, employing a real-time kernel is useless, if the hypervisor can introduce large, or unbounded, latencies. Hence, it is essential to measure and reduce the latencies introduced by the virtualisation mechanisms.

3. Latencies in virtual machines

To measure (and try to cope with) the latency experienced inside a VM, it is important to understand how modern virtualisation solutions work.

The software component responsible for controlling the execution of multiple OSs on the same physical node (creating the VM abstraction) is called *hypervisor*. The hypervisor is what lets entire OSs (i.e., their kernels but also all their application programs) safely share the same hardware, allowing the guest kernel and applications to run directly on the host CPU (as much as possible, at least).

3.1. Basic virtualisation concepts

Virtualisation is not a new idea (the first works about virtualisation date back to the '70s, or even before), and started from the "trap-and-emulate" technique, identifying some *sensitive instructions* that must be

intercepted by the hypervisor and emulated. It has been proved [27] that if all the sensitive CPU instructions (instructions that the hypervisor has to emulate) are also privileged instructions (instructions that generate a trap or an exception if executed when the CPU has a low privilege level), then the CPU can be easily and efficiently emulated (the basic idea is to execute the guest code at a low privilege level, and let the hypervisor handle the traps/exceptions generated by the execution of privileged instructions in the guest).

If the CPU does not fulfil these basic virtualisation requirements (that is, if its ISA has sensitive instructions that are not privileged), then implementing a VM is more difficult and requires, for example, a technique called *Binary Translation*, first described in [28]. For a long time, binary translation has been the only viable way for virtualising one of the most widely used CPU architectures (the Intel and AMD x86 "PCs") because the original x86 architecture did not fulfil the requirements mentioned above. This prompted the development of *para-virtualisation* technologies, advocated for by Xen [3], that are both simpler and more efficient than binary translation. The basic para-virtualisation idea is to make the guest OS aware of being executed in a VM, and to modify accordingly (for example, by replacing sensitive instructions with instructions that generate a trap invoking the execution of the hypervisor, also called hypercalls).

On the other hand, many CPUs (for example recent x84 CPUs, but also ARM or PowerPC) introduced some virtualisation extensions, that made them "virtualisable" (compliant with the basic virtualisation requirements [27]), removing the need for binary translation or para-virtualisation. Modern CPUs went way beyond just enabling the basic virtualisation requirements and also introduced features for virtualising in-hardware the guest OS's page tables, interrupt delivery and other mechanisms. This evolution in CPU architectures triggered the development of virtualisation solutions like KVM [29], often referred as *Hardware Virtualisation*, or HVM. Guests running inside a VM based on hardware virtualisation are hence referred as HVM guests (or just HVM). In the Intel world, a physical CPU with VT-x support that is executing hypervisor code is said to be in "VMX-root mode". When it is executing code from the guest OS, it is said to be in "VMX-nonroot mode".

Similarly, a VM based on para-virtualisation is called a PV-VM and the code running inside it is a PV guest. As previously explained, for running as a PV guest the OS kernel needs to be modified, by replacing some of the privileged routines and instructions of the OS kernel (e.g., manipulating the MMU) with hypercalls. While this technique is born as a clever and very effective way for virtualising architectures that were not virtualisation capable, it can still be useful nowadays, on modern hardware. For example, even on fully virtualisable CPUs some sensitive instructions must be intercepted and treated specially by the hypervisor, introducing some overhead. Some of these operations can be faster if they are para-virtualised.

3.2. Xen and KVM

Xen[3] and KVM are the two most widely adopted open source virtualisation solutions, for server consolidation and cloud computing, but they can also be used in embedded systems. This paper, therefore, focuses on them and experimentally measures the latencies they introduce.

Xen is a *bare-metal* (sometimes referred to as "Type-I") hypervisor. This means that it runs directly on the host system hardware, and it only deals with virtual machines, not with applications or any other kind of "tasks". A bare-metal hypervisor has its own VM scheduler, which is used to multiplex the execution of the *virtual CPUs* of the VMs on the physical CPUs. The entities which are scheduled by the scheduler of a bare-metal hypervisor are the virtual CPUs of the various VMs. If there are N VMs, each with M virtual CPUs, and the host has P physical CPUs, the hypervisor scheduler has to figure out how to make these $M\dot{N}$ entities run (typically in a time-sharing manner) on the P physical CPUs.

KVM is a *hosted* (or "Type-II") hypervisor. This means it runs in a host OS kernel, using the kernel's functionalities. The host kernel gains a new component, capable of executing VMs by making their virtual CPUs look exactly like other host tasks. In a hosted hypervisor scenario, therefore, it is the host kernel scheduler that is responsible for scheduling the VMs. In a KVM setup, the virtual CPUs of the VMs are implemented as threads (the so-called "virtual CPU" — or vcpu — threads) scheduled by the Linux kernel scheduler. As seen by Linux, KVM is basically a driver for the processor virtualisation extension. Whenever the Linux scheduler selects a task to be run on a physical CPU, and that task reveals to be a virtual CPU of a VM, KVM is "contacted" to make sure that what actually runs on the hardware is code from the guest OS (or, in general, from whatever is running inside the VM).

3.3. Hypervisors and latencies

From a latency perspective, if a hypervisor is hosted on a real-time OS kernel, we can expect that the same techniques already in place for controlling the kernel latency are also effective in providing good real-time virtualisation performance, reducing the latencies experienced by the guest OSs. A bare-metal hypervisor, on the other hand, must implement its own mechanisms to support low latencies and real-time performance. Bare-metal hypervisors are generally small, and this usually means that critical sections and latency sensitive paths in the hypervisor code are short and fast enough. In this sense, bare-metal hypervisors can be compared to μ-kernels [4] or the dual kernel real-time systems [30].

It is possible to provide real-time performance to KVM by applying the Preempt-RT patch to the host kernel. Although the Xen hypervisor has been equipped with a real-time scheduler [2], to the best of the authors' knowledge, there has been no effort to try to reduce, or even to measure, the latency introduced by the hypervisor.

Note that the idea of using KVM-based VMs to host real-time applications is not new [31], and some previous works reported very low latencies (in the order of tens of μs) in a properly configured VM [32]. However, such works were based on dedicating a physical CPU to each vcpu thread (using the Linux cpuset mechanism), while this work focuses on scheduling the virtual CPUs.

4. Virtualisation in KVM and Xen

To better understand the experiments and the results from the next sections, it is important to quickly recall the Xen and KVM architectures, and the basic virtualisation technologies they use.

4.1. Hardware virtualisation and paravirtualisation

In virtual machines based on HVM, code execution on the virtual CPUs and access to main memory are handled by the hypervisor and by the virtualisation support of the CPU, but I/O operations require some assistance. I/O devices (like network cards and disk drives) need to be emulated in software by a dedicated component, called the *Device Model* (DM). The piece of software which is most widely used as a DM is QEMU [33], but other choices are possible.

Both Xen and KVM use a combination of hardware virtualisation and para-virtualisation to achieve the best performance. For instance, while taking advantage of hardware virtualisation technology for CPU and memory, I/O is often handled using para-virtualisation, within both Xen[4] and KVM.[5]

[3] It should be called, for correctness, the "Xen-Project hypervisor", but, in the remainder of the paper, it will be referred to as Xen, for the sake of brevity.

[4] https://wiki.xenproject.org/wiki/PV_Protocol.

[5] https://www.linux-kvm.org/page/Virtio.

4.2. KVM Architecture

When using KVM, the CPU is always virtualised through HVM, while I/O devices can be emulated (reproducing the hardware interface of real devices, so that existing drivers can be re-used in the guest) or para-virtualised (modifying the code of the guest kernel). In any case, QEMU is used as a Device Model for implementing I/O devices or emulating other operations (while starting a VM, for instance). Since the benchmarks used in this paper are not I/O intensive, the details about devices emulation or para-virtualisation do not have impact on the presented experiments and analysis (basically in the experiments presented in this paper QEMU is only used to set up the VM and drive the KVM module). Hence, in the context of this paper the generic term "guest" or "KVM guest" is enough to describe a guest OS running in a KVM-based VM (and it is not necessary to specify more details about the VM).

Every KVM-based VM consists in a QEMU process containing one thread (referred to as "*vcpu thread*") for each virtual CPU, plus one main QEMU thread acting as Device Model and eventually some I/O threads (in many cases the setup can be much more complex, but they are out of the scope of this paper). All the threads of the QEMU process (including the vcpu threads) are, as already explained, scheduled by Linux scheduler, acting as the host kernel scheduler.

4.3. Xen guest types and architecture

Since Xen was designed and developed when x86 CPUs did not provide any hardware virtualisation support, it originally used para-virtualisation, supporting VMs only able to run OS kernels that have been modified accordingly. Note that a pure PV guest does not need any emulation, not even for I/O, or for bringing up and booting the VM (as that is para-virtualised too). So, when using a PV guest, there is no need for any Device Model.

Today, Xen also supports HVM guests: the hardware extensions are used to virtualise CPU and memory, and a QEMU DM is used for VM boot, management, and for I/O. Depending on the configuration of the guest itself, some I/O can happen via para-virtualisation instead of via DM emulation. However, as already said, this is not interesting for our work. In any case, HVM guests always need a Device Model.

Recently, another virtualisation mode has been introduced to combine the best of PV and HVM which, broadly speaking, means: (1) needing no DM, like PV guests; (2) taking advantage of hardware support, like HVM guests. This virtualisation mechanism has recently become usable and is called PVH (or HVMLite). An OS kernel needs to be modified to run as a PVH guest (the Linux kernel has been already), and PVH guests, usually, need no DM.

4.4. Xen Dom0 and DomU

As Xen is a bare-metal hypervisor, it runs on top of the hardware, and there is no host OS. However, to keep the hypervisor small and simple (for example, to avoid the need for device drivers in the hypervisor) Xen creates, as one of the first things it does during boot, a special PV guest, called *Dom0*. This Dom0 guest is sometimes seen as a sort of host OS, but it is important to recall that is, to all the effects, a guest, although a special (privileged) one (for example, it has access to hardware devices).

Dom0 is used, for instance, to allow user interactions with the Xen hypervisor, to create other guests, etc. Typically, Dom0 is a Linux OS or distribution. So, in every Xen installation, there is at least the hypervisor, and a PV guest acting as Dom0. The virtual CPUs of Dom0 are scheduled by the Xen scheduler, together with all the other virtual CPUs of all the various guests that are created during the system usage.

Note that, when creating a guest that needs a Device Model, i.e., an HVM guest, the Device Model is an instance of QEMU, that runs as a process inside Dom0. This means that, from a scheduling point of view, a Xen HVM guest is composed by: (1) the virtual CPUs scheduled by the Xen scheduler; (2) the QEMU process acting as Device Model, which is scheduled by the Dom0 scheduler and runs on Dom0 virtual CPUs (which in turn are scheduler by the Xen scheduler).

From a latency perspective, an operation that requires interactions with the DM can be problematic: for the action to be performed, the Xen scheduler must schedule (at least one of the) Dom0 virtual CPUs, and the Dom0 scheduler must schedule the DM (QEMU) process. To better deal with these situations, Xen supports different setups.[6]

Finally, the term *DomU* is how, historically, the Xen PV guests where called, in the early days. However, in most context, including this paper, the term is used to refer to a non-privileged Xen guest (a Xen guest other than Dom0). The type of guest can be explicitly specified, such as in "PV DomU" or "PVH DomU".

5. Latency evaluation

The latency experienced by applications running inside KVM and Xen VMs has been measured and analysed, by running an extensive set of experiments, described in this section.

5.1. Experimental setup

To make sure that the conclusions do not depend on the specific hardware used for the tests, the experiments have been repeated on various machines, with different types of CPUs, focusing on the Intel 64bit ISA.[7] The various CPUs used for testing differ in number of cores, speed and provided features, presenting some small ISA variations (some machine instructions are available on some CPUs and not on other, etc...), but the experiments revealed that the latency experienced in a VM (either Xen-based or KVM-based) is mostly impacted by how the VM handles the CPU's timestamp counter. For example, when a guest tries to execute the `rdtsc` (read timestamp counter) instruction, this instruction is emulated on some CPUs (such as the Intel Core Duo) or directly executed on other CPUs (such as the Intel Core i7). If para-virtualisation is used, instead, the `rdtsc` instruction is directly removed from the guest kernel. Since the timestamp counter is often accessed by the guest kernel to handle its timers, using "trap-and-emulate" on `rdtsc` (or similar instructions) can have a huge impact on the latency experienced in the guest (in particular, on the timer resolution latency).

The latency measurements have been performed by running the `cyclictest` tool[8] in the guest OS. The `cyclictest` tool is the de-facto standard for measuring latencies on Linux-based systems and works by measuring the response time of a high-priority periodic task. If this is the highest priority task in the system, its response time should be equal to its execution time, which is almost 0; so, the response time can be considered as an upper bound to the latency experienced by the task. Notice that technically `cyclictest` measures the latency experienced by the hardware timer's interrupt, that is actually the sum of the "timer resolution latency" and other sources of latency [17]. Since when serving all the other interrupts the "timer resolution latency" will not be experienced, the latency measured by `cyclictest` is an upper bound for all the latencies that can be experienced in the (guest) OS. This is why the `cyclictest` results are generally considered a good measure of the system's real-time performance.

In the first experiments, the period of the `cyclictest` periodic task has been set to 50 µs, since it is known that it should be set to no more than twice the expected worst-case latency [34]. All the VMs have been configured with only one virtual CPU, used to run

[6] https://wiki.xen.org/wiki/Dom0_Disaggregation.

[7] The CPUs used for the experiments include an "Intel(R) Xeon(TM) W3530" CPU running at 2.80 GHz, an "Intel(R) Core(TM) i7-6700" CPU running at 3.40 GHz, an "Intel(R) Core(TM)2 Duo E8500" CPU running at 3.16 GHz, an "AMD Ryzen(TM) Threadripper 2950X" CPU running at 2.4 GHz and others.

[8] wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/.

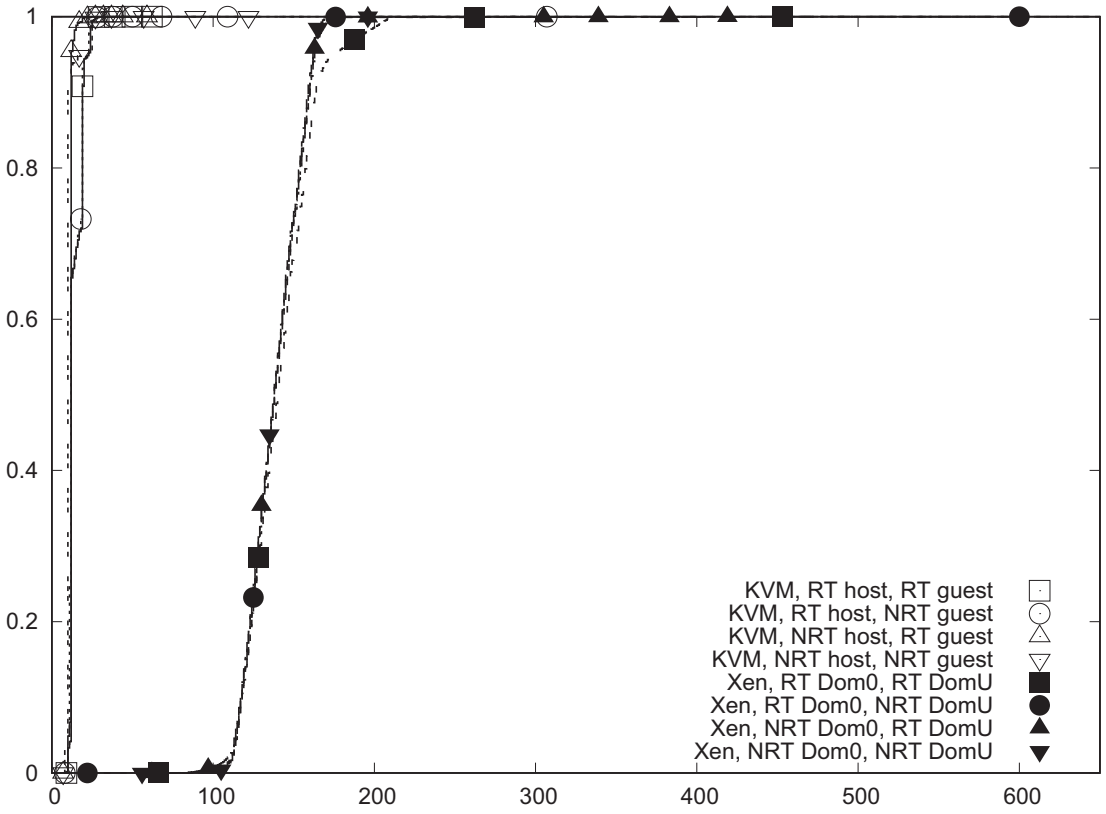

Fig. 1. Experimental CDF of the latencies experienced in a VM with various configurations and without any load in background. Experiments performed on an Intel Core Duo.

the `cyclictest` periodic thread. To make sure that the real hypervisor introduced latencies (and not some delays due to preemptions by other tasks) were measured, the KVM vcpu thread was given the `SCHED_FIFO` policy and the highest possible priority (99). On Xen, we used the default hypervisor scheduler, but we made sure that the DomU had a dedicated physical CPU, all to itself, and not used by Dom0 or any other VM. In this way, the DomU virtual CPU is always the only virtual CPU that can run on its physical CPU, and the KVM vcpu thread is the highest priority thread in the system (hence, the hypervisor should schedule the virtual CPUs of the VMs where `cyclictest` is running as soon as they are ready for execution).

To trigger higher latencies, in some experiments a "stress workload" has been added in background (using the `SCHED_OTHER` policy). Such a stress workload is based on scenarios that are known to cause high latencies[9]; it is executed on the host kernel for the experiments involving KVM and in Dom0 for the Xen experiments.

The measurements have been performed using both a real-time kernel (Preempt-RT, version 5.0.19-rt11) and a non-real-time kernel (5.0.19, with kernel preemption set to "none") as KVM host kernel or Xen Dom0 kernel. The same kernel versions have been used for the guest. In all the following figures, tables and discussions, "RT kernel" refers to a Linux kernel with Preempt-RT enabled, while "NRT kernel" refers to Linux compiled with kernel preemption set to "none". In all the tables, the "Kernels" column indicates the host (or Dom0) / guest (or DomU) kernels used in the experiment. In the first set of experiments, the Xen VM used HVM for virtualisation. Finally, all the experiments with Xen have been performed using the latest stable version (version 4.11.1).

[9] wiki.linuxfoundation.org/realtime/documentation/howto/tools/worstcaselatency.

5.2. Timer resolution latency

A first set of experiments has been performed to evaluate the timer resolution latency introduced by Xen and KVM. For these experiments, `cyclictest` has been executed in a VM *without running any stress workload* on Dom0 or the host. In this setup, since kernels and hypervisor are not subject to additional workloads it is reasonable to expect that the measured latencies are almost completely due timer resolution issues.

Figs. 1 and 2 report the experimental Cumulative Distribution Function (CDF) of the latencies measured on the Intel Core Duo (a CPU on which accesses to the timestamp counter are emulated) and the Intel i7 (a CPU on which the timestamp counter can be safely accessed by the guest without being emulated by the hypervisor) CPUs. The other CPUs used in the experiments provided similar results. Notice that the long tails of some CDFs have been removed from the plots to make them more readable. The first important thing to be noticed is that changing the kernels used in the experiments does not seem to significantly affect the shape of the CDFs (all the "KVM curves" look overlapping, and the same holds for the "Xen curves"). It is also possible to see that KVM achieves significantly smaller latency and hence better real-time performance than Xen.

As a reference, Fig. 3 compares the latencies experienced by a KVM RT guest running on an RT host, with the latencies obtained by directly running `cyclictest` on the RT host. Similarly, Fig. 4 compares the latencies measured in a Xen HVM DomU with the ones measured in Xen Dom0 (with an RT kernel or NRT kernel). On KVM, latencies from inside the guest are near to the ones measured on real hardware. On Xen, the latencies experienced inside the DomU and inside Dom0 are similar, and both are sensibly higher than what is measured on hardware directly.

Looking at the worst-case latency values, reported in Tables 1 and 2, we notice that switching between RT and NRT guest and host (Dom0)

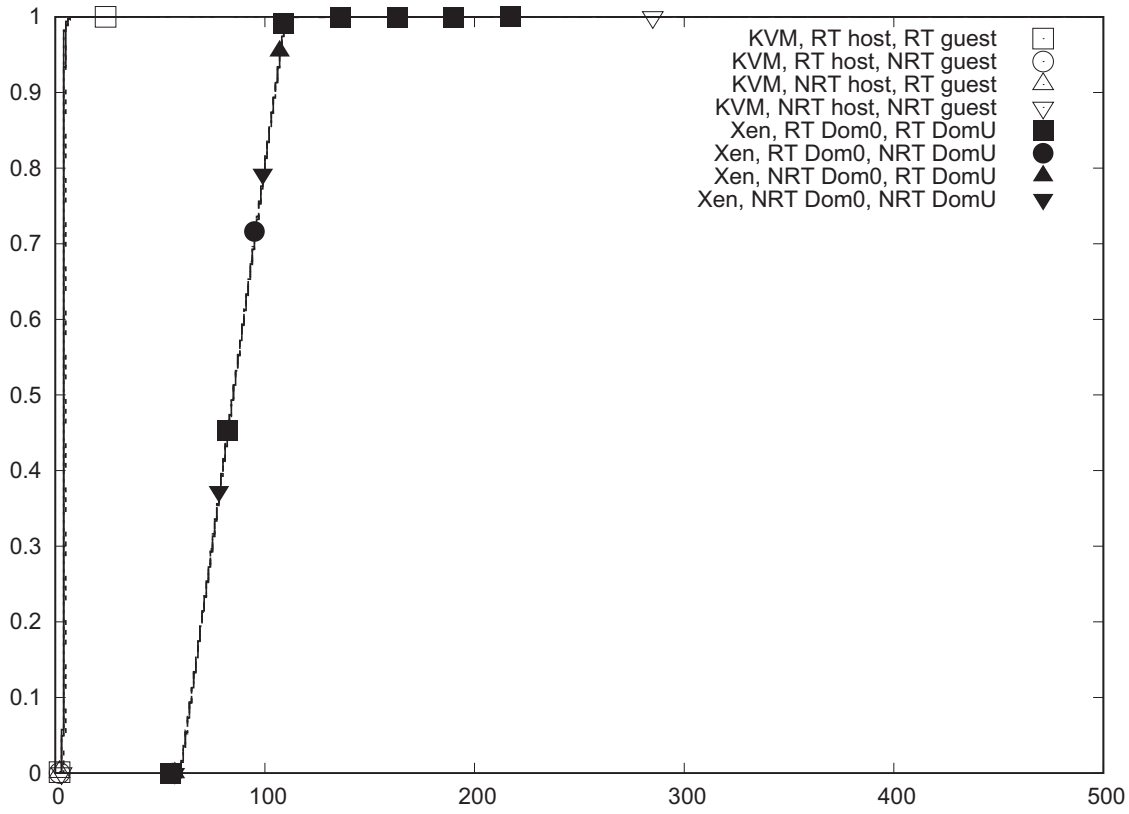

Fig. 2. Experimental CDF of the latencies experienced in a VM with various configurations and without any load in background. Experiments performed on an Intel i7 CPU.

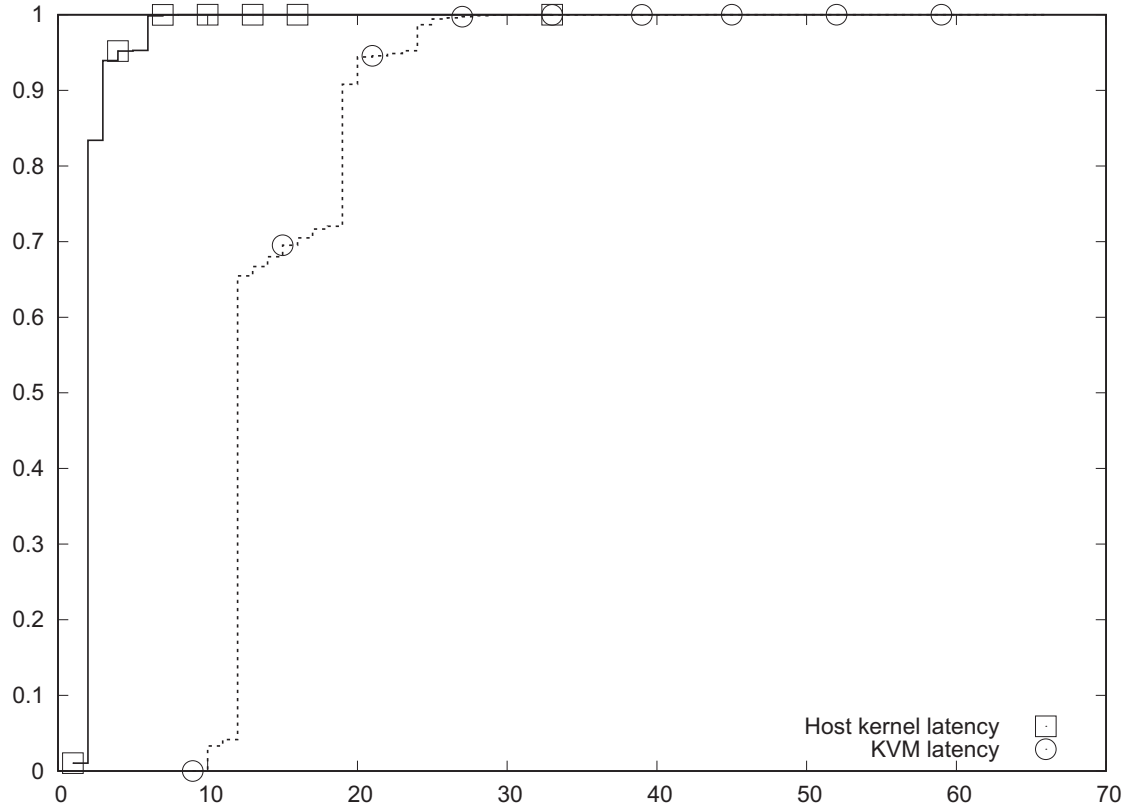

Fig. 3. Experimental CDF of the latencies experienced in a real-time KVM-based VM and on the host (with an RT kernel).

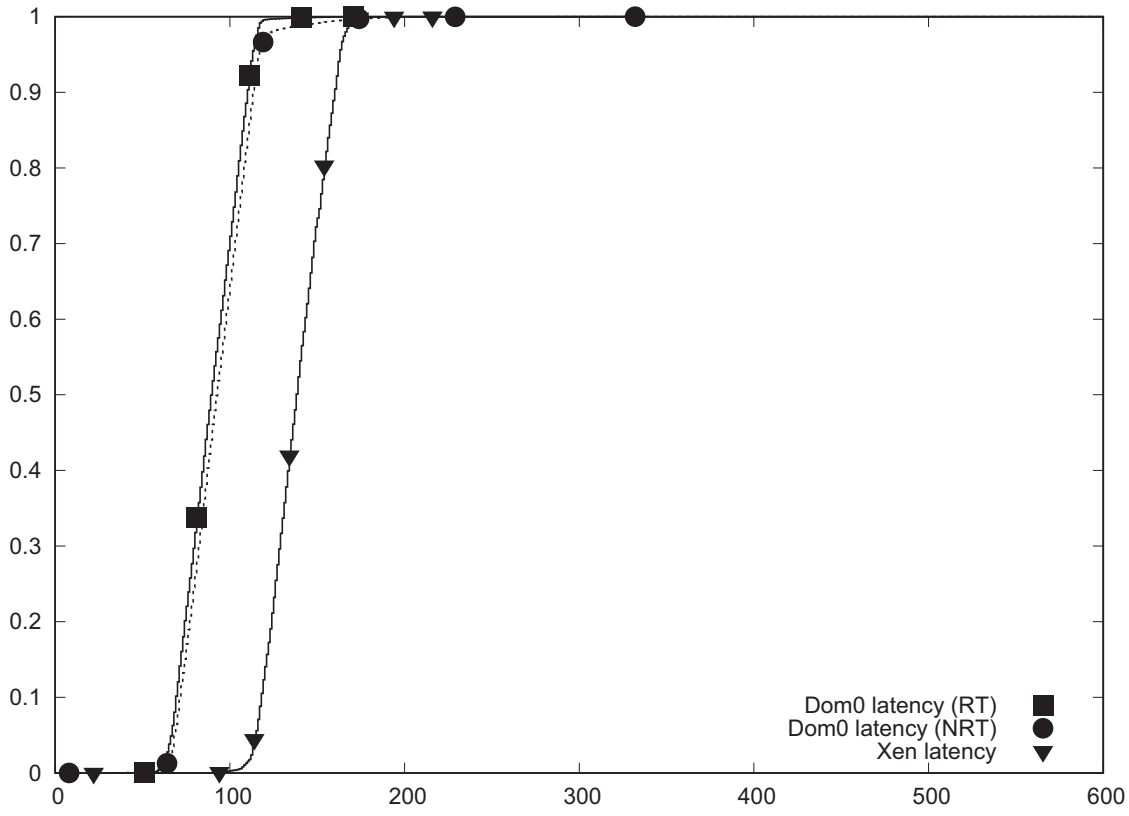

Fig. 4. Experimental CDF of the latencies experienced in a Xen-based VM (DomU) and on the Xen Dom0 (with an RT and an NRT kernel).

Table 1
Worst case latency experienced in a VM, with various configurations and without any load in background. Experiments performed on an Intel Core Duo.

Kernels	Xen	KVM
NRT/NRT	2303 µs	344 µs
NRT/RT	602 µs	116 µs
RT/NRT	2502 µs	307 µs
RT/RT	577 µs	66 µs

Table 2
Worst case latency experienced in a VM, with various configurations and without any load in background. Experiments performed on an Intel i7 CPU.

Kernels	Xen	KVM
NRT/NRT	529 µs	294 µs
NRT/RT	544 µs	50 µs
RT/NRT	660 µs	34 µs
RT/RT	240 µs	28 µs

kernels has an impact on real-time performance. In particular, on KVM, using an RT kernel in host and guest provides good performance, limiting the worst-case latency to 66 µs for the Intel Core Duo and 28 µs for the Intel i7. On Xen, however, using an RT kernel does not reduce too much the worst-case latencies, which are almost 10 times larger than the ones experienced when using KVM. While using an RT kernel in the guest allows reducing the latencies (to about 500 µs on the Intel Core Duo and to about 200 µs on the Intel i7), the usage of an RT kernel in Dom0 does not always reduce the latencies experienced in the VM (this can be expected, because the Xen hypervisor is not hosted, hence it does not use the Dom0 kernel).

The worst-case latency measured when running `cyclictest` on real hardware (with an RT kernel) is 22 µs on the Intel i7 and 33 µs for the Intel Core Duo. Since the worst-case latency experienced by a KVM-based VM (with RT host kernel and RT guest kernel) is 28 µs on the Intel i7 and 66 µs on the Intel Core Duo, it is possible to say that KVM virtualisation does not introduce a relevant timer-resolution latency on the Intel i7 and introduces an additional latency that is comparable with the worst-case latency of a Preempt-RT kernel on the Intel Core Duo. This is due to the overhead caused by the emulation of the `rdtsc` instruction needed on some CPUs such as the Intel Core Duo.

When using Xen, the worst-case latency measured when running `cyclictest` on Dom0 is 329 µs on the Intel Core Duo and 201 µs on the Intel i7 when using an RT kernel, and 1775 µs on the Intel Core Duo (630 µs on the Intel i7) when using a NRT kernel (when measuring the latency in Dom0, the type of the Dom0 kernel obviously affects the results). The worst-case latency measured when running `cyclictest` in DomU, instead, is 577 µs for the Intel Core Duo and 240 µs on the Intel i7 (when using an RT kernel). The increase in latency experienced on the Intel Core Duo is again due to the emulation of the `rdtsc` instruction (which is emulated for the DomU kernel on the Intel Core Duo, but is not emulated on the Intel i7 and is not used by the Dom0 kernel that is para-virtualised).

Since it became clear that Xen is not able to run tasks with period $T = 50$ µs inside guests due to its timer resolution latency, some more additional experiments have been performed with larger `cyclictest` periods. When using either 500 µs or 1000 µs, the observed latency decreased sensibly, as shown in Fig. 5.

These results raised the suspect that the high timer resolution latency inside Xen guests is related to the mechanism used to route the timer interrupts from the hypervisor to the guest itself. It seems that

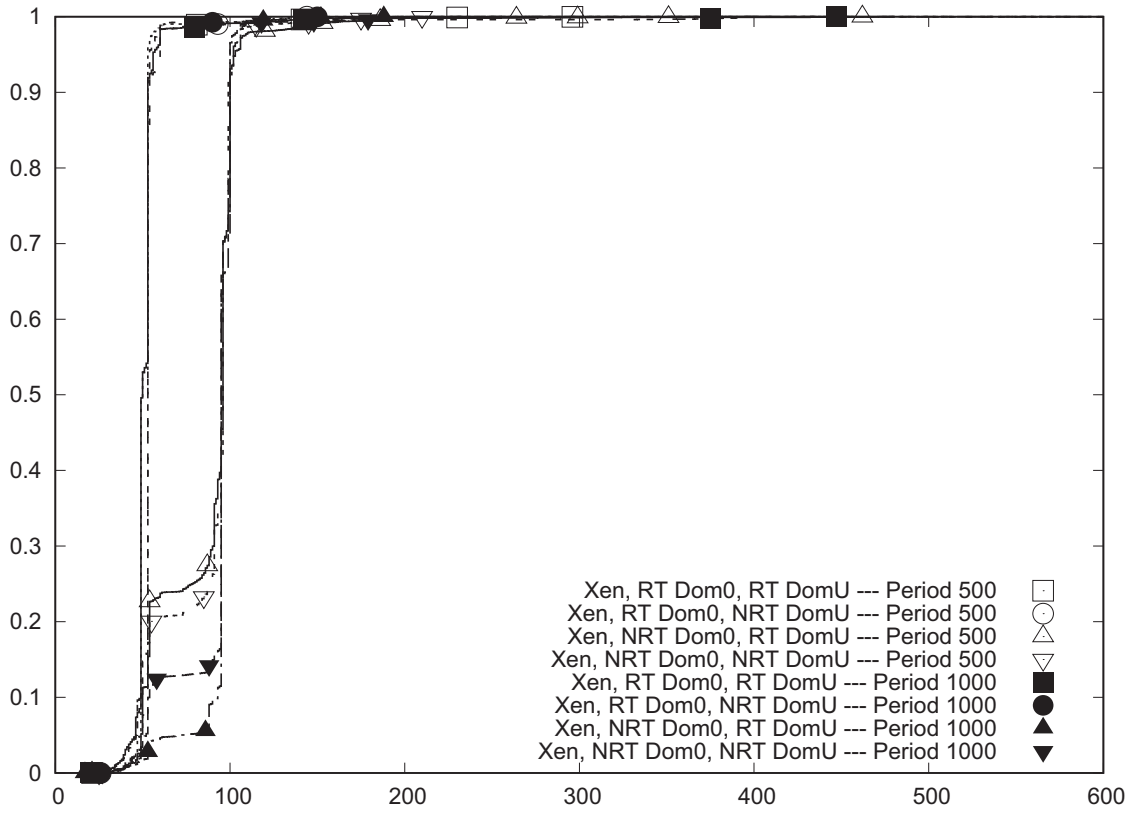

Fig. 5. Experimental CDF of the latencies experienced in a Xen VM when using 500 µs and 1000 µs for the cyclictest period. Experiments performed on an Intel Core Duo.

Xen somehow enforces a minimum period for the timer interrupts and when the cyclictest period is larger than such a minimum period (as in the cases of Fig. 5), the latency remains under control, mainly between 10 µs and 100 µs (but it is still higher than the latency experienced by KVM). When, on the other hand, the cyclictest period becomes near to the minimum value allowed by Xen the guest domain tends to lag much more, and the observed latency grows substantially.

Based on these considerations, the Linux kernel code responsible for handling the Xen's para-virtualised timers has been analysed, finding that it enforces a minimum distance of 100 µs between two consecutive timers (this is defined by the "`TIMER_SLOP`" constant in `arch/x86/xen/time.c`). Further investigation and discussion with the Xen community revealed that this large value is mainly due to historical reasons; a patch for allowing users to set smaller values has been proposed[10] and has been integrated in version 5.2 of the Linux kernel.

Some tests performed with different values of this constant confirmed that this is the cause of the issue and decreasing `TIMER_SLOP` greatly reduces the Xen's timer resolution latency. As an example, Figs. 6 and 7 show the results obtained with `TIMER_SLOP` reduced to 1 µs (the Intel i7 results are particularly good, again due to the fact that accesses to the timestamp counter are not emulated); reducing `TIMER_SLOP` to less than 1 µs did not have noticeable effects on the experienced latencies. For reference, the figures also show the latencies obtained with KVM (using Preempt-RT as a guest kernel and as a host kernel). This allows noticing that although the latencies measured under Xen are still larger than the ones measured under KVM, the difference between KVM and Xen is reduced (compare Fig. 6 with Figs. 1 and 7 with Fig. 2). The experiments also showed that the worst-case latencies are not affected by `TIMER_SLOP`: the worst-cases measured with `TIMER_SLOP` reduced to 1 µs are similar to the ones experienced with the original `TIMER_SLOP` value. However, as it is visible from the figures, the latency is smaller than 50 µs (the `cyclictest` period) for more than 90% of the samples.

Some additional experiments have been performed using PV and PVH as virtualization techniques, to check if they allow to further reduce the latencies. The experiments revealed that when the CPU architecture does not require to trap and emulate the `rdtsc` instruction (for example, on the Intel i7) using PV or PVH does not significantly change the shape of the latency CDF (but allows reducing the worst-case latencies, as shown in Section 6). In architectures where `rdtsc` needs to be emulated (for example, on the Intel Core Duo), instead, using PV shifts left the latency CDF curve by about 15 µs. This is basically the cost of the "trap-and-emulate" mechanism for the timestamp counter.

5.3. Other sources of latency

Once the Xen's timer resolution latency has been reasonably reduced, the background stress workload has been added to measure latencies due to other sources. The results obtained on the Intel Core Duo are presented in Fig. 8; since in this case all the CPUs showed consistent results (indicating that the `rdtsc` emulation affect the timer resolution latencies, but does not have a big impact on the other latencies), the results obtained on the Intel i7 and other CPUs are not reported. By looking at the figure, it is possible to notice two important things: first of all, the Xen latencies are still higher than the KVM latencies; then, the shapes of the CDFs are similar to the ones measured without the background stress workload (Fig. 6). However, some important differences are visible when looking at the worst case values, presented in Table 3.

The KVM results confirm that using RT kernels in the guest and in the host allows to control the worst-case latencies, which increase from 66 µs without stress to 71 µs with stress for the Intel Core Duo and from

[10] https://lists.xenproject.org/archives/html/xen-devel/2019-03/msg01785.html.

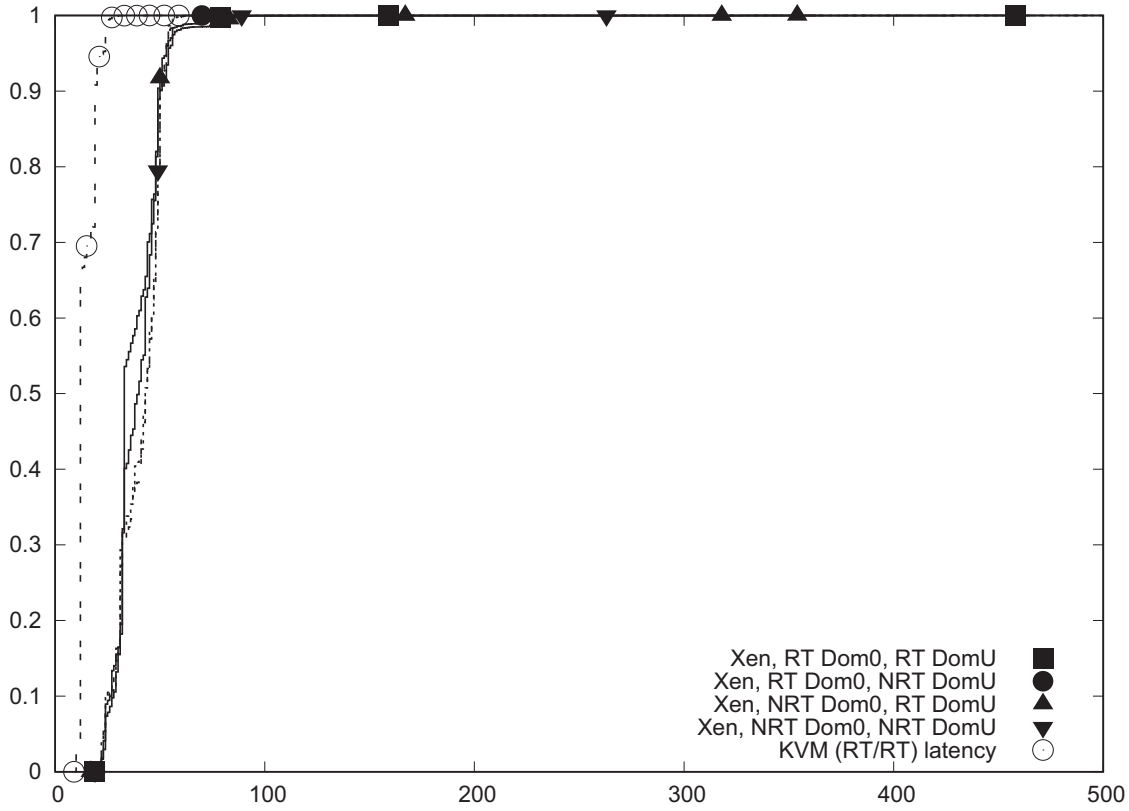

Fig. 6. Experimental CDF of the latencies experienced in a Xen VM when the `TIMER_SLOP` constant of the guest kernel has been set to 1 µs. Experiments performed on an Intel Core Duo.

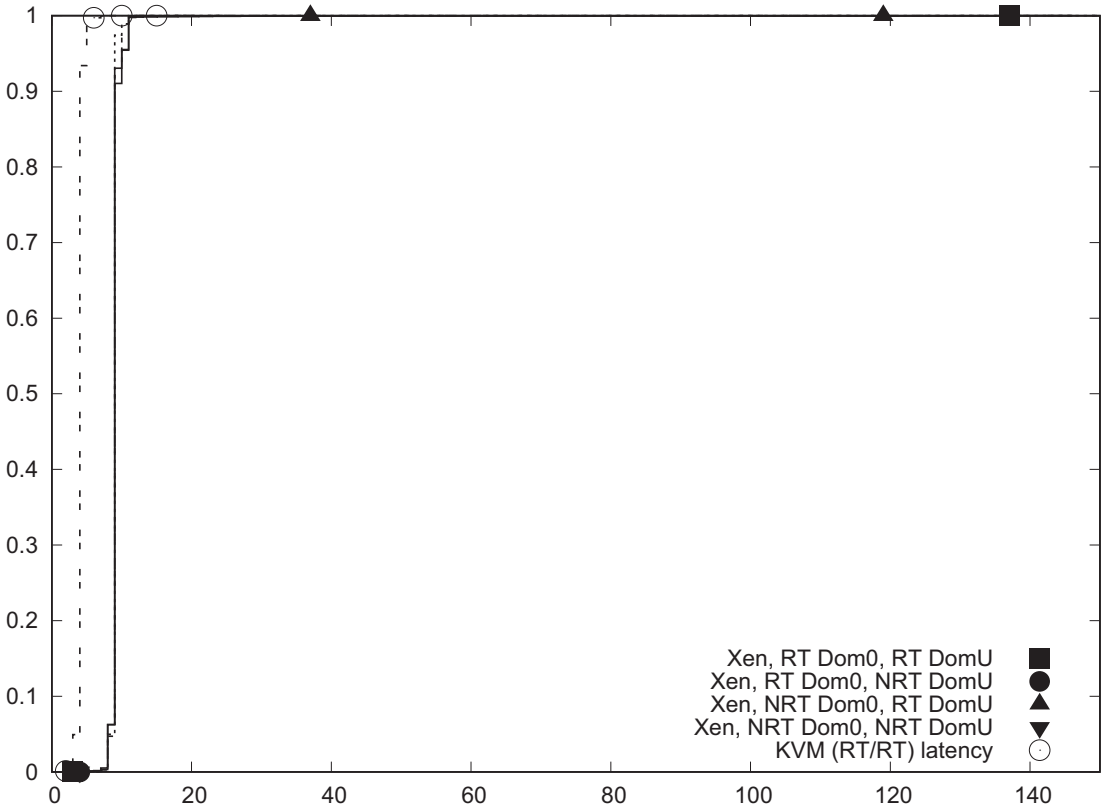

Fig. 7. Experimental CDF of the latencies experienced in a Xen VM when the `TIMER_SLOP` constant of the guest kernel has been set to 1 µs. Experiments performed on an Intel i7.

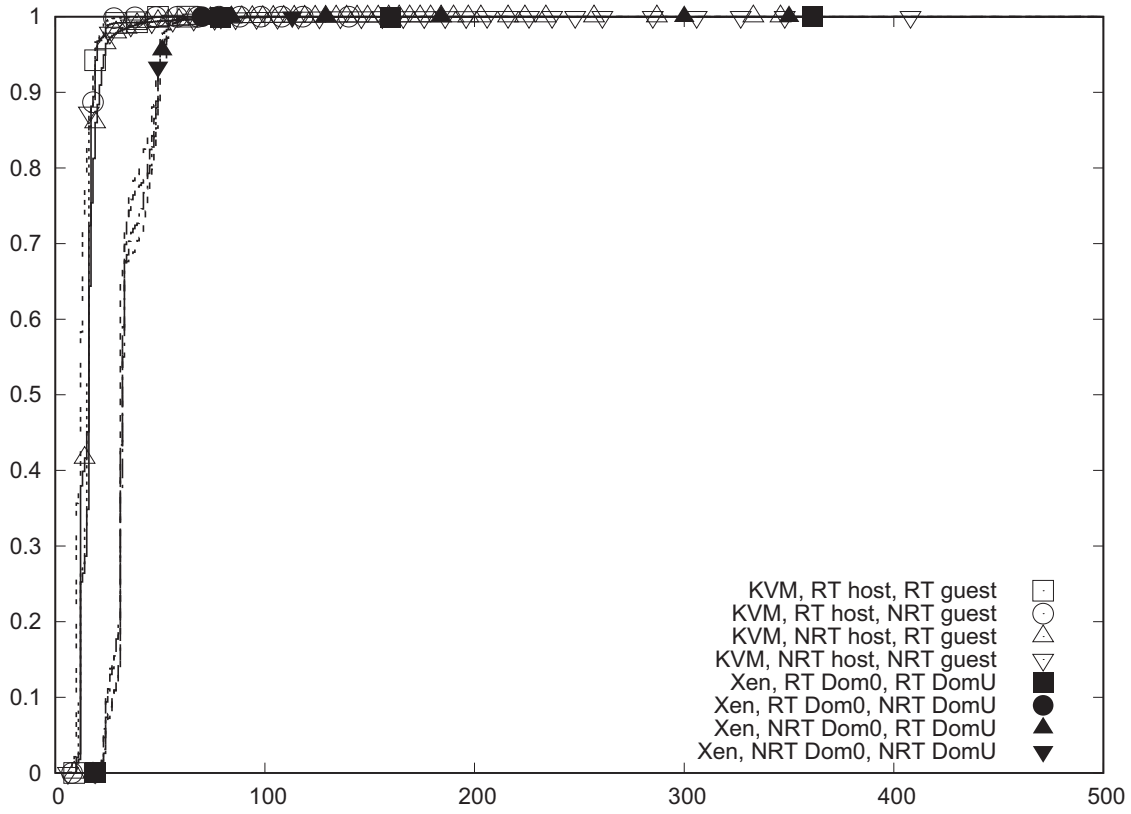

Fig. 8. Experimental CDF of the latencies experienced in a VM with various configurations, with a stress workload in background. Experiments performed on an Intel Core Duo.

Table 3
Worst case latency experienced in a VM, with various configurations and with a stress workload in background.

Kernels	Xen	KVM
NRT/NRT	24230 µs	2683 µs
NRT/RT	1440060 µs	2147 µs
RT/NRT	24889 µs	233 µs
RT/RT	1260548 µs	71 µs

Table 4
Worst case latency experienced in a Xen DomU, with and without load on Dom0, with various configurations.

Kernels	No Stress	Stress
NRT/NRT	2503 µs	24230 µs
NRT/RT	363 µs	1440060 µs
RT/NRT	3419 µs	24889 µs
RT/RT	465 µs	1260548 µs

24 µs to 68 µs for the Intel i7. Notice that in this case the worst case latency is almost the same for the two CPUs (probably because they run at similar frequencies), confirming that the `rdtsc` emulation only affects the timer resolution latency.

The situation with Xen, instead, is more complicated and all the measured worst-case latencies are very high. Moreover, on all the CPUs the results seem to indicate that using an RT kernel in DomU increases the worst-case latency (to very high values!) instead of decreasing it.

6. Investigating the Xen latencies

As shown, after reducing the Xen's timer resolution latency it has been possible to see that Xen consistently introduces large latencies due to other sources, on all the test machines (and, as a consequence, Xen does not seem to be usable for real-time VMs). Hence, additional experiments have been performed to identify the sources of these latencies and reduce them.

6.1. Impact of the Xen scheduler

First of all, we checked whether the high latencies were due to the hypervisor scheduler. Xen supports multiple different schedulers and the experiments described so far were conducted using the Credit scheduler (which is the default for the 4.11 release). To check the impact of the hypervisor scheduler on the experienced latencies, the experiments were repeated using the "null" scheduler which statically assigns each virtual CPU to one and only one physical CPU. The "null" scheduler also avoids any scheduling decision (it is, basically, an "offline" scheduler) [35], removing any kind of scheduling overhead. Since the latencies experienced in the VM were not reduced, it is possible to conclude that their source is not in the scheduler.

Once it was clear that the Xen scheduler is not responsible for the large latencies, further experiments have been performed to investigate the strange behaviour related to Preempt-RT (in some situations using an RT kernel in DomU increases the DomU latency instead of decreasing it), which turned out to be triggered by the stress workload running in Dom0. The results in Table 4 show that the stress workload affects the DomU latencies both when running an NRT kernel (in this case the worst-case latency increases from 2503 µs to 24230 µs, or from 3419 µs to 24889 µs) and when running an RT kernel (in this case, the worst-case latency increases from less than 500 µs to more than 1 s). This particularly strange behaviour (it could be more reasonable to expect almost complete independence between the latency observed inside the DomU

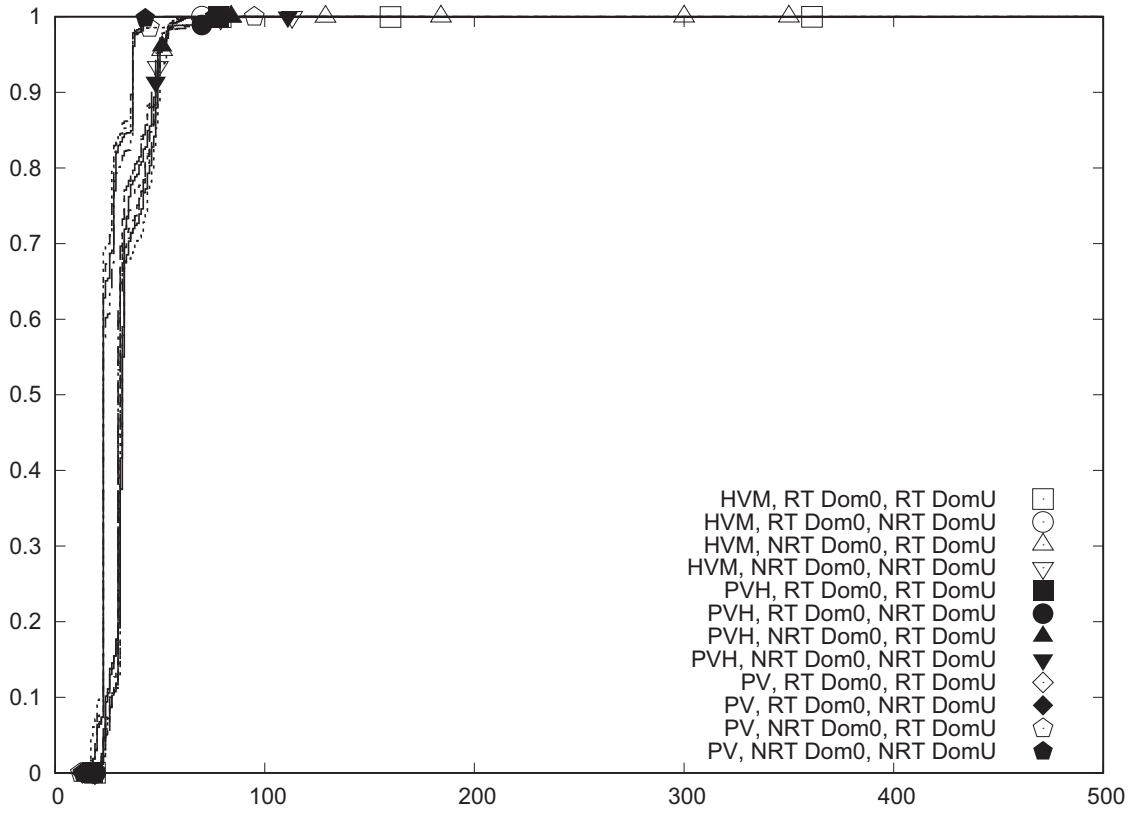

Fig. 9. Experimental CDF of the latencies experienced in a Xen DomU of various type, "HVM", "PV" and "PVH" (with RT and NRT kernels).

Table 5
Worst case latency experienced in a Xen DomU, with various DomU kernels and virtualisation mechanisms.

DomU Kernel	PV	PVH	HVM
NRT/NRT	161 µs	2386 µs	24230 µs
NRT/RT	95 µs	177 µs	1440060 µs
RT/NRT	166 µs	2073 µs	24889 µs
RT/RT	88 µs	182 µs	1260548 µs

and whatever workload runs in Dom0) could be explained if there was some software module running in Dom0 that interacts with the hypervisor or with the DomU. An analysis of the Xen architecture suggested that such a software component can be the QEMU process used as a DM (remember the discussion in Section 4 about the different virtualisation mechanisms supported by Xen).

6.2. Impact of the virtualisation technology

As noticed, one possible source for the high latencies measured in Xen-based VMs could be related to the HVM virtualisation mechanism (used in the experiments described up to now), that requires a QEMU instance running in Dom0 as a DM. Hence, new experiments have been performed using PV and PVH DomUs (in this setup, PV and PVH do not need a DM).

Looking at the results of these new experiments, shown in Fig. 9, it is possible to notice that the shapes of the CDFs are similar, for all guest types. However, the worst-case latencies, reported in Table 5, are not: the numbers in the table show that the strange Preempt-RT behaviour only happens when using HVM guests. It can also be seen that when using PV the worst-case latencies are only slightly larger than the ones experienced with KVM. When using PVH, on the other hand, the worst-case latencies are more than 2 times larger than the ones measured KVM-based VMs. Repeating the experiments on different CPUs showed that the PVH behaviour depends on the CPU architecture: for example, on a Xeon W3530 CPUs the worst-case latency experienced when using PVH is much smaller, and comparable with the one experienced using KVM (PV, instead, has a behaviour similar to the one experienced on other CPUs, and the shapes of the CDF curves are similar to the ones seen on the other CPUs). Fig. 10 shows some results obtained on the Xeon W3530 (the worst-case latencies are 44 µs for KVM, 72 µs for Xen PV, 28 µs for Xen PVH and 695917 µs for Xen HVM).

Summing up, the stress workload in Dom0 has barely any effect when using PV or PVH (in this case, the DomU with RT kernels achieve the best real-time performance, in line with general expectations), but can cause large latencies when using HVM. This same behaviour has been noticed when repeating the experiments on all the CPUs used for the tests and explains the relationship between load in Dom0 and latencies in DomU: the stress workload might delay the execution of the DM, when it is needed by DomU, causing the large latencies. This is a classical example of *priority inversion* [24] (the VM is indirectly delayed by the Dom0 workload, because it waits for the QEMU process that does not execute with a high priority) and shows a source of latency that is not possible for OS kernels (the kernel cannot be delayed waiting for user-space programs), but only for some kinds of hypervisors (that rely on user-space processes for some functionalities). The validity of this analysis has been checked by repeating the experiment with the HVM DomU after assigning the highest possible real-time priority to the QEMU threads. In practice, the QEMU process has been scheduled, inside Dom0, with the `SCHED_FIFO` policy and priority 99, so that it cannot be disturbed by the stress workload. The resulting latencies are similar to ones obtained when using a PVH guest, showing that in this case the presence of the background noise in Dom0 does not impact the DomU latencies.

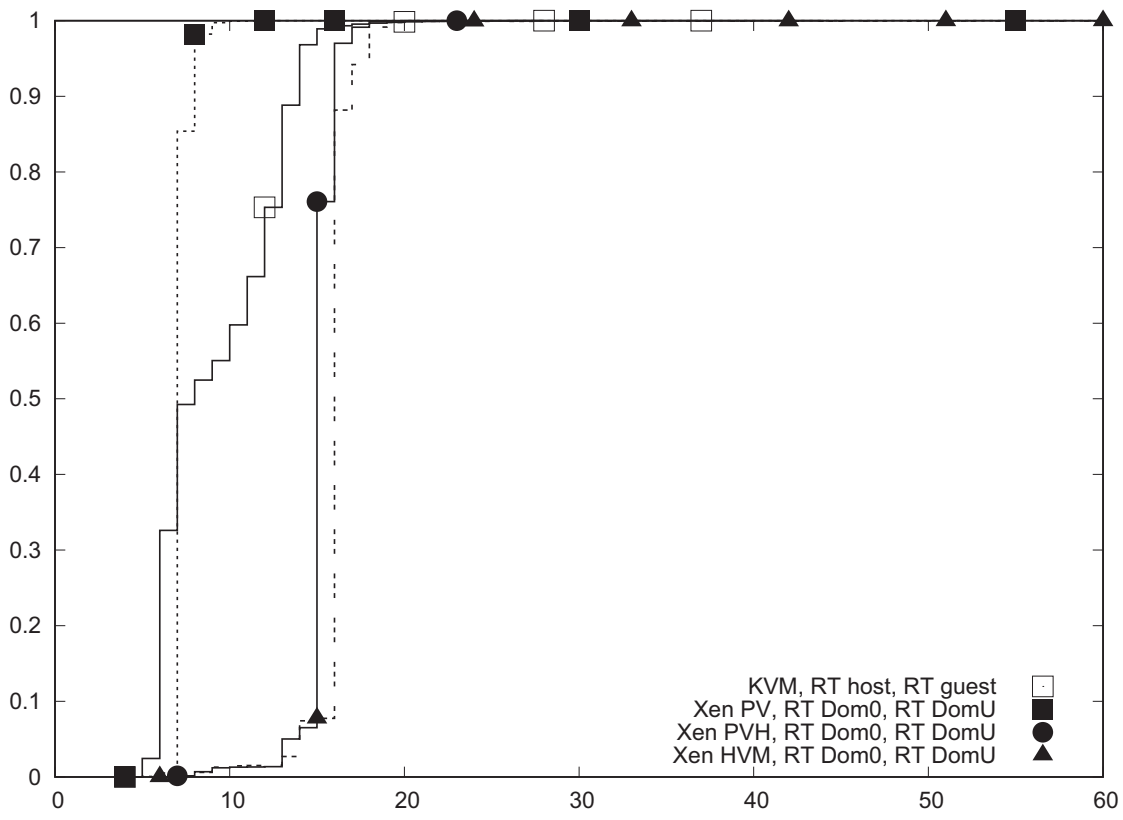

Fig. 10. Experimental CDF of the latencies experienced in a VM running on a Xeon W3530 when using KVM and Xen (PV, PVH and HVM) with RT kernels.

6.3. Final considerations

The previous experiments and investigation produced various important results:

- First of all, it has been verified that KVM-based VMs are suitable for serving real-time workloads, causing worst-case latencies smaller than 100 µs. When using KVM, it is important to use RT kernels both in the guest and in the host
- Then, it has been verified that when used as a guest in Xen-based VMs the Linux kernel forces a limit of 100 µs on the resolution of the para-virtualised Xen timers it uses. This is an issue when running real-time tasks in the guest, and can be fixed by modifying the "`TIMER_SLOP`" constant in `arch/x86/xen/timer.c`. Thanks to some interactions with the Xen community, Linux kernels starting to version 5.2 allow to modify this value at run time (without having to recompile the kernel)
- Finally, it turned out that using the Xen's HVM virtualisation mechanism can result in very high latencies in presence of some load in Dom0. This issue can be avoided by using PV or PVH.

Summing up, when running real-time applications in KVM-based VMs it is important to use Preempt-RT as host kernel and as guest kernel, while when running real-time applications in Xen-based VMs it is necessary to modify the guest kernel (this is not necessary starting from version 5.2 of the Linux kernel) and to use the PV or PVH mechanisms. If PV is used, the real-time performance are only slightly worse than the ones experienced with KVM; on some CPUs, using PVH results in even better performance.

7. Conclusions and future work

This paper presented an experimental analysis of the hypervisor latencies generated by Xen and KVM. The latency can be seen as a measure of the difference between a theoretical schedule and the real one, and the worst case latencies introduced by a hypervisor indicate if the hypervisor can be used to serve real-time VMs.

The results showed that a properly configured KVM guest is characterised by a worst-case latency smaller than 100 µs, and can be used for serving a large amount of real-time application. A default installation of Xen, instead, seemed to introduce much larger latencies, but a deep analysis and experimental investigation revealed that Xen can be configured to reduce such latencies and properly serve a good range of real-time applications.

In more details, two sources have been identified for the latencies introduced by Xen. The first one is related to the para-virtualised Xen timers and can be removed by modifying the guest kernel. The second one is related to the QEMU process used as DM when the HVM virtualisation is used and can be removed by using the PV or PVH mechanisms. Based on these considerations, it is possible to configure a Xen-based VM so that the experienced latencies are comparable with the ones experienced in a KVM-based VM.

As a future work, experiments with some realistic real-time applications will be performed, considering the impact of virtualised I/O devices and scheduling the virtual CPUs (as proposed, for example, in [16]).

Declaration of Competing Interest

None.

References

[1] J. Yang, H. Kim, S. Park, C. Hong, I. Shin, Implementation of compositional scheduling framework on virtualization, SIGBED Rev. 8 (1) (2011) 30–37.
[2] S. Xi, M. Xu, C. Lu, L.T.X. Phan, C. Gill, O. Sokolsky, I. Lee, Real-time multi-core virtual machine scheduling in Xen, in: Proc. of 2014 International Conference on Embedded Software (EMSOFT), 2014, pp. 1–10.
[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, SIGOPS Oper. Syst. Rev. 37 (5) (2003) 164–177.

[4] F. Mehnert, M. Hohmuth, H. Härtig, Cost and benefit of separate address spaces in real-time operating systems, in: Proceedings of the 23rd IEEE Real-Time Systems Symposium, RTSS '02, IEEE Computer Society, Washington, DC, USA, 2002, p. 124.
[5] L. Abeni, D. Faggioli, An experimental analysis of the Xen and KVM latencies, in: Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC), 2019, pp. 18–26, doi:10.1109/ISORC.2019.00014.
[6] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard real–time environment, J. Assoc. Comput. Mach. 20 (1) (1973) 46–61.
[7] A.K. Mok, X. Feng, D. Chen, Resource partition for real-time systems, in: Proc. of 7th IEEE Real-Time Technology and Applications Symposium, 2001, pp. 75–84.
[8] X. Feng, A.K. Mok, A model of hierarchical real-time virtual resources, in: Proc. of 23rd IEEE Real-Time Systems Symposium, 2002, pp. 26–35.
[9] G. Lipari, E. Bini, Resource partitioning among real-time applications, in: Proc. of 15th Euromicro Conference on Real-Time Systems, 2003, pp. 151–158.
[10] I. Shin, I. Lee, Periodic resource model for compositional real-time guarantees, in: Proceedings of 24th IEEE Real-Time Systems Symposium, 2003, pp. 2–13.
[11] L. Almeida, P. Pedreiras, Scheduling within temporal partitions: response-time analysis and server design, in: Proc. of 4th ACM International Conference on Embedded Software, 2004, pp. 95–103.
[12] E. Bini, M. Bertogna, S. Baruah, Virtual multiprocessor platforms: specification and use, in: Proc. of 30th IEEE Real-Time Systems Symposium, 2009, pp. 437–446.
[13] A. Easwaran, I. Shin, I. Lee, Optimal virtual cluster-based multiprocessor scheduling, Real-Time Syst. 43 (1) (2009) 25–59.
[14] G. Lipari, E. Bini, A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation, in: Proc. of 31st IEEE Real-Time Systems Symposium, 2010, pp. 249–258.
[15] A. Burmyakov, E. Bini, E. Tovar, Compositional multiprocessor scheduling: the GMPR interface, Real-Time Syst. 50 (3) (2014) 342–376.
[16] L. Abeni, A. Biondi, E. Bini, Hierarchical scheduling of real-time tasks over Linux-based virtual machines, J. Syst. Softw. 149 (2019) 234–249, doi:10.1016/j.jss.2018.12.008.
[17] L. Abeni, A. Goel, C. Krasic, J. Snow, J. Walpole, A measurement-based analysis of the real-time performance of Linux, in: Proc. of 8th IEEE Real-Time and Embedded Technology and Applications Symposium, 2002, pp. 133–142.
[18] M. Xu, L.T.X. Phan, O. Sokolsky, S. Xi, C. Lu, C. Gill, I. Lee, Cache-aware compositional analysis of real-time multicore virtualization platforms, Real-Time Syst. 51 (6) (2015) 675–723.
[19] M. Xu, L. Thi, X. Phan, H.-Y. Choi, I. Lee, vCAT: dynamic cache management using cat virtualization, in: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE, IEEE, 2017, pp. 211–222.
[20] H. Kim, R. Rajkumar, Predictable shared cache management for multi-core real-time virtualization, ACM Trans. Embedded Comput. Syst. 17 (1) (2017) 22:1–22:27.
[21] M. Barabanov, V. Yodaiken, Real-time Linux, Linux J. 23 (4.2) (1996) 1.
[22] P. Mantegazza, E. Dozio, S. Papacharalambous, RTAI: real time application interface, Linux J. 2000 (72es) (2000) 10.
[23] P. Gerum, Xenomai — implementing a RTOS emulation framework on GNU/Linux, in: White Paper, Xenomai, 2004, pp. 1–12.
[24] L. Sha, R. Rajkumar, J.P. Lehoczky, Priority inheritance protocols: an approach to real-time synchronization, in: IEEE Transactions on Computers, vol. 39, 1990, pp. 1175–1185.
[25] S. Rostedt, Internals of the RT patch, in: Proceedings of the Linux Symposium, Ottawa, Canada, 2007, pp. 161–172.
[26] P. McKenney, A realtime preemption overview, aug 2005, https://lwn.net/Articles/146861/.
[27] G.J. Popek, R.P. Goldberg, Formal requirements for virtualizable third generation architectures, Commun. ACM 17 (7) (1974) 412–421.
[28] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, S.G. Robinson, Binary translation, Commun. ACM 36 (2) (1993) 69–81, doi:10.1145/151220.151227.
[29] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: the Linux virtual machine monitor, in: Proceedings of the Linux symposium, vol. 1, 2007, pp. 225–230.
[30] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, D.J. Magenheimer, Are virtual machine monitors microkernels done right? in: Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS), USENIX, 2005.
[31] J. Kiszka, Towards Linux as a real-time hypervisor, in: Eleventh Real-Time Linux Workshop, Dresden, Germany, 2009, p. 205.
[32] R.V. Riel, Real-time KVM from the ground up, KVM Forum 2015, 2015.
[33] F. Bellard, QEMU, a fast and portable dynamic translator, in: USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41–46.
[34] C. Emde, Path analysis vs. empirical determination of a system's real-time capabilities: the crucial role of latency tests, in: Eleventh Real-Time Linux Workshop, Dresden, Germany, 2009, p. 205.
[35] S. Stabellini, D. Faggioli, Xen schedulers and their impact on interrupt latency, Xen-Project Developers and Design Summit, 2017.

Luca Abeni is an Associate Professor at the ReTiS (Real Time Systems) Lab of Scuola Superiore Sant'Anna, Pisa. He graduated in computer engineering from the University of Pisa in 1998 and has been a PhD student at Scuola Superiore Sant'Anna from 1999 to 2002 carrying out research on real-time operating systems, scheduling algorithms, quality of service management, and multimedia applications. In 2000, he was a visiting student at Carnegie Mellon University, working with professor Ragunathan Rajkumar on Resource Kernels and resource reservation algorithms for real-time kernels. In 2001, he was a visiting student at Oregon Graduate Institute (Portland), working with professor Jonathan Walpole on feedback scheduling algorithm s and resource allocation, and on the evaluation and improvement of the real-time performance of the Linux kernel. From 2003 to 2006, Luca worked in Broadsat S.R.L., developing IPTV applications and audio/video streaming solutions over wired and satellite (DVB - MPE) networks. Then, he moved to the Dipartimento di Ingegneria e Scienza dell'nformazione (DISI) of University of Trento as a full-time researcher, where he became later Associate Professor. Since 2017, he is back at Scuola Superiore S. Anna as Associate Professor.

Dario Faggioli graduated in computer engineering at the University of Pisa in 2008, and received the PhD degree in computer engineering from the Scuola Superiore Sant'Anna in 2012. He conducted his research activity within the Real-Time Systems Laboratory (ReTiS). His focus was on both design and efficient implementation of scheduling algorithms and synchronization protocols for real-time operating systems. Since 2011 he works as a Software Engineer in the field of virtualization, and is actively contributing to various Open Source software projects, across the virtualization ecosystem. He currently works in the SUSE R&D department (SUSE Labs) and he is one of the maintainer of the Xen-Project Hypervisor scheduler.