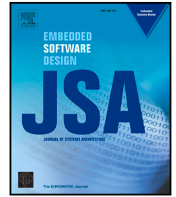
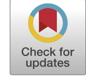
Journal of Systems Architecture 154 (2024) 103238

Contents lists available at ScienceDirect

Journal of Systems Architecture

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





Virtualized real-time workloads in containers and virtual machines

Luca Abeni

Scuola Superiore Sant'Anna, Pisa, Italy

ARTICLE INFO

Keywords:
Real-time computing
Virtual machines
Containers
Operating systems
Unikernels

ABSTRACT

Real-time virtualization is currently a hot topic, and there is much ongoing research on real-time Virtual Machines and hypervisors. However, most of the previous research focused either on reducing the latencies introduced by the virtualization stack (hypervisor, host Operating System, Virtual Machine scheduling, etc...) or analyzing the virtual CPU scheduling algorithms. Only a few works investigated the impact of the guest Operating System architecture on real-time performance or considered multiple performance metrics (latency, schedulability, startup times, resource consumption) at the same time. This paper compares various features of different virtualization technologies and guest Operating Systems, evaluating their suitability for serving real-time applications. The results indicate that solutions based on KVM (and an appropriate microvm) and the OSv unikernel can be considered viable alternatives to more traditional VMs or containers.

1. Introduction

With the ever-growing diffusion of cloud computing, various services or entire OSs can be remotely executed in distributed virtualized environments. This results in an increasing interest in various virtualization technologies, leading people to virtualize even applications that have not traditionally been considered candidates for execution in virtual environments. An interesting example is the virtualization of applications characterized by some kinds of temporal constraints, which is challenging because it requires providing *a-priori guarantees* that such temporal constraints are respected. Such guarantees can be provided by using well-known techniques from real-time literature, real-time OSs/hypervisors, and appropriate management software. In particular, it is important to schedule the virtualized real-time applications according to predictable/analyzable algorithms and to introduce *bounded* latencies in their execution.

Analyzable and deterministic scheduling algorithms have been proposed in the literature and have already been implemented [1–4] both in widely used hypervisors [5] or host OSs and in guest OSs. Moreover, modern hypervisors or host OSs introduce latencies that are low enough for running real-time applications [6–8].

Although the usage of Virtual Machines (VMs) is widespread across many different kinds of systems, from cloud/edge servers to low-power embedded devices, real-time virtualization solutions mainly focus on embedded devices or dedicated machines [9–11]. In most of these situations, the guest OS can be based on small real-time executives or dedicated kernels. When real-time applications are executed on larger-scale cloud systems, additional issues must be considered. For example, to consider the execution of a real-time application in a virtual environment as a viable alternative to the execution on bare hardware, the startup delay introduced by the virtualization mechanisms should be limited (users should not notice any additional delay when starting an application in a virtual environment). Moreover, to avoid scalability issues, it is important to reduce the overhead in resource consumption and the need for resource over-allocation.

While multiple previous works measured different performance metrics for VMs, containers, and/or unikernels, a comprehensive comparison of different technologies focused on real-time performance is still missing. For example, previous research on real-time VMs focused on scheduling and latency aspects [7], ignoring other important metrics such as the application's startup time and the VM's resource consumption, while previous research on containers and unikernels (dedicated and/or specialized library OSs optimized to execute inside a VM) mainly focused on average throughput or average startup time [12–14], generally ignoring scheduling, worst-case latencies, and worst-case startup times.

This paper investigates how different virtualization technologies and guest Operating Systems comply with the most important requirements of real-time applications. It considers host operating systems based on a real-time version of Linux, running containers or different kinds of Virtual Machines, comparing the real-time performance, boot times, and resource usage of guest OSs based on a real-time version of Linux or a unikernel. Unikernels have not been traditionally considered as an option for executing real-time applications (hence, many unikernels miss some of the features needed to execute real-time applications or provide unsuitable performance); however, recent work shows that it is possible to patch the OSv unikernel to provide the needed support [15].

E-mail address: luca.abeni@santannapisa.it.

https://doi.org/10.1016/j.sysarc.2024.103238
Received 29 November 2023; Received in revised form 13 June 2024; Accepted 12 July 2024
Available online 23 July 2024
1383-7621/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

In more detail, the analysis focuses on the following metrics, which are evaluated through some significant micro-benchmarks:

- Latency introduced by the VM or container (evaluated through the standard "`cyclictest`" tool)
- Startup time
- Temporal isolation provided by the scheduler (evaluated by running periodic real-time tasks in a VM or container, with and without some competing real-time and non-real-time workload)
- Resource (memory and CPU time) consumption

One of the interesting results of this evaluation is that hypervisor-based virtualization can be competitive with OS-level virtualization even when the guest runs real-time applications; moreover, the OSv unikernel further decreases the applications' startup times and resource consumption and can be an interesting alternative to run real-time applications in a cloud environment, especially if it is associated with microvm technologies. While similar results were known for general-purpose applications focusing on average throughput, this paper extends the results to real-time applications.

The paper is organized as follows: Section 2 provides some definitions and recalls some concepts and ideas used in the rest of the paper, then Section 3 introduces various virtualization technologies and guest OS structures that will be evaluated in Sections 4, and 5 finally presents the conclusions.

2. Background and definitions

This paper considers the respect of temporal constraints for real-time applications running on a *guest OS* hosted by some kind of virtual environment.

Such a virtual environment can be implemented by using different technologies; for example, it can be based on *hardware virtualization* or *OS-level virtualization*. In the case of full hardware virtualization, a software component called *hypervisor* virtualizes the CPU and the memory; all the devices of a real machine are also virtualized (by the hypervisor or by some kind of higher-level software component often called Virtual Machine Monitor — VMM). In contrast, in the case of OS-level virtualization, only the OS kernel functionalities (and not all the hardware devices) are virtualized.

Using hardware virtualization, the guest OS executes in the VM as if it was on real hardware, and unmodified OSs can easily run in virtualized environments without being aware of the virtualization layer. To improve I/O performance, the VM can implement some "special" para-virtualized devices; in this case, the performance boost is "paid" by the need to modify the guest OS kernel (that must be aware of running inside a VM and must know how to handle the para-virtualized devices). If the CPU's Instruction Set Architecture (ISA) is virtualizable [16], the virtualization performance can be improved by directly executing the guest machine instructions on the host CPU. All the modern CPUs have a virtualizable ISA (or provide some hardware-assisted virtualization features that make the ISA virtualizable). Hence, the most commonly used hypervisors can take advantage of this technique.

When OS-level virtualization is used, instead, the system runs one single OS kernel (the host kernel), and such a kernel is in charge of virtualizing its services for the guest OSs (and providing isolation between the guests). Hence, all the OSs running on the physical node share the same kernel; this implies that the guests must run the same hardware and software architecture (if the host OS is Linux-based and the physical machine is based on Intel x86_64 CPUs, then the guest OSs must also be based on an x86_64 Linux kernel). This form of virtualization is often used to implement the *container* abstraction [17].

Real-time applications are often modeled as sets $\Gamma = \{\tau_i\}$ of (periodic or sporadic) real-time tasks, where each task τ_i is a sequence of activations, or jobs, $J_{i,j}$. Each job $J_{i,j}$ is activated at time $r_{i,j}$ (with $r_{i,j+1} \geq r_{i,j} + T_i$, where T_i is the task's period or minimal inter-arrival time) and finishes at time $f_{i,j}$ after executing for a time $c_{i,j}$ (with $c_{i,j} \leq C_i$, where C_i is the task's Worst Case Execution time - WCET). If $\forall j, f_{i,j} \leq r_{i,j} + D_i$ (where D_i is the task's relative deadline), then the task respects all of its temporal constraints.

Respecting such temporal constraints requires that system resources are timely allocated to the virtualized real-time applications. In other words, the applications' tasks must be scheduled by using a theoretically sound scheduling algorithm that can be analyzed and allows providing real-time guarantees.

Moreover, such a scheduling algorithm has to be implemented correctly, limiting the differences between the theoretical scheduling and the actual implementation. The hypervisor (or the host OS kernel) and the guest OS kernel risk introducing some so-called *latencies* which, informally speaking, affect the accuracy of the scheduler's implementation. Hence, such latencies have to be measured and analyzed to guarantee the predictability of the virtualized system. In more detail, latency is experienced every time that application tasks are not scheduled at the time when the theoretical scheduling algorithm would schedule them, but their execution is somehow delayed. This latency can be caused by non-preemptable sections in the kernel or in the hypervisor, by the language runtime (think about garbage collection in the JVM), or by similar implementation details. It was originally studied in the context of real-time OSs [18,19], but the definitions and analysis can be extended to virtualized environments, too [6,7,20–22] (see the cited papers for more formal definitions).

When hardware virtualization is used, two different strategies can be used to map virtual CPUs to physical CPU cores: the so-called *partitioning hypervisors* [9–11,23] or separation kernels [24] directly map each virtual CPU to one entire physical CPU core, while other hypervisors can schedule virtual CPUs to multiplex various virtual CPUs on a single physical CPU core. Partitioning hypervisors introduce very small latencies at the cost of dedicating a physical CPU core to each virtual CPU. In other words, even if a VM hosts a real-time application with a very small utilization, a full physical CPU core is reserved for it. This constraint obviously imposes a limit to the total number of virtual CPUs (and to the number of hosted VMs), creating a scalability issue that can be addressed by scheduling multiple virtual CPUs on a smaller number of physical CPU cores through a virtual CPU scheduler implemented in the host OS kernel or in the hypervisor.

Hypervisors that schedule virtual CPUs often result in a 2-level scheduling hierarchy: the host/hypervisor scheduler schedules VMs (or, better, the VMs' virtual CPUs), and the guest schedulers running inside each VM schedule the applications' tasks when the first scheduler selects the VM. The behavior of such a scheduling hierarchy has been previously analyzed [25]. For example, real-time guarantees can be easily provided if the host kernel/hypervisor provides CPU reservations and the guest scheduler provides fixed priorities [26–31].

Since the latency can be accounted for in schedulability analysis as a blocking time, if an upper bound L for the latency is known, then standard real-time analysis can be used to guarantee the respect of the temporal constraints of virtualized applications even if the host kernel or hypervisor introduces some latency. Of course, if this upper bound is too high, then no guarantee can be provided for real-time tasks running inside a VM.

When OS-level virtualization is used, there is only one single CPU scheduler (the host kernel scheduler) which is in charge of implementing the scheduling hierarchy. This means that such a scheduler has exact knowledge of the tasks scheduled inside each virtual environment (while in the case of hypervisor-based virtualization, the host scheduler is not aware of the guest scheduler's decisions). As a consequence, global scheduling in the guest can be analyzed [32] without the issues experienced in the case of full hardware virtualization [4]. Moreover, the host kernel is the only source of latencies for OS-level virtualization.

Standard tools such as `cyclictest` are commonly used to measure the worst-case latency introduced by an OS or experienced in a virtualized environment (introduced by the hypervisor in case of full hardware virtualization or by the host OS kernel in case of OS-level

virtualization). Previous work already showed that with off-the-shelf hardware and properly tuned host kernels or hypervisors, it is possible to reduce the latency experienced inside a virtual environment to values that are acceptable for many real-time applications [7].

Finally, in some situations such as the offloading of real-time computations to cloud or edge nodes, the execution of a real-time application in a VM (or container) can be considered as an alternative to execution on bare metal only if the time needed to start the VM is comparable with the time needed to start the real-time application. In these situations, even if appropriate scheduling algorithms are used and acceptable upper-bounds for the latencies are provided VMs are usable to serve real-time applications only if they can be started in very short times.

Hence, another important performance metric to be considered in real-time virtualization is the worst-case *startup time*. It is known that OS-level virtualization can help reduce average startup times and resource consumptions, even if containers are generally considered less secure than VMs [33]. Startup times and resource consumption can also be reduced without renouncing full hardware virtualization by modifying the guest operating system (and its kernel); for example, using unikernel architectures or small embedded RTOSs.

3. Real-time virtual environments

With some notable exceptions [15], previous work on real-time virtualization focused on the host OS or hypervisor and only marginally investigated the effects of other parts of the virtualization stack (or of the guest OS) on real-time performance, resource usage, and startup times. However, the guest OS structure can have a huge impact on the application's startup time and resource usage. This paper investigates the latencies, scheduling, resource consumption, and startup time issues in hypervisor-based and OS-level virtualization when different kinds of guest OSs are used. In particular, the presented experiments focus on:

- Real-time (Preempt-RT) Linux kernels for OS-level virtualization (in this case, the host OS kernel is used by the guest, too, and the rest of the guest OS can be reduced to only a few libraries and programs)
- Real-time (Preempt-RT) Linux kernels and unikernels (OSv, in particular) for hypervisor-based virtualization

3.1. OS-level virtualization

In Linux, OS-level virtualization is implemented by using *control groups* to account for the usage of kernel resources and *namespaces* to control their visibility and provide isolation. User space tools such as `Docker` or `podman` are used to set up the containers; in turn, these tools use a lower-level *container runtime* for properly configuring control groups and namespaces to execute the guest OS inside them.

As previously mentioned, when using OS-level virtualization, the latencies experienced by guest real-time applications are introduced by the host OS kernel. Hence, the Preempt-RT patchset [34] has been applied to the host Linux kernel to provide low latencies to containerized real-time applications.

The startup times and resource usage can be reduced by optimizing the container runtime and by running the real-time application in a reduced guest OS (based on `busybox`[1]) that avoids starting useless daemons and services. Even with these optimizations, however, some container engines seem to introduce additional startup delays, as shown in Section 4.

3.2. Hypervisor-based virtualization

Different kinds of hypervisors can be used to implement full hardware virtualization; for example, the hypervisor can directly access the hardware without using any functionality provided by other software components (and in this case, it is called "bare-metal hypervisor"), or can use some functionalities provided by a host OS kernel (and in this case it is called "hosted hypervisor"). In the Linux environment, the most common example of bare-metal hypervisor is Xen [5], which is directly started by the bootloader (or by the BIOS) and takes care of initializing the hardware, starting at least a guest OS (generally Linux based), and managing CPU, memory, and other resources without any help from external components. Moreover, it contains a *virtual CPU (vCPU) scheduler*. The most common example of hosted hypervisor, instead, is KVM [35], which uses the Linux kernel for managing the hardware and scheduling the virtual CPUs. Summing up, bare-metal hypervisors are self-contained and have no external dependencies. On the other hand, they have to duplicate many functionalities generally implemented in the OS kernel.

When KVM is used, a user-space VMM is responsible for driving the hypervisor, creating VMs, populating them with vCPUs and memory, and implementing virtual devices. QEMU [36] is generally used as a user-space VMM for KVM, but some smaller and more lightweight VMMs, such as the Intel Cloud Hypervisor[2] or Amazon's FireCracker [37] are also available. In general, these lightweight VMMs allow for reducing the boot times and the resource consumption, and even QEMU implemented a lighter VM model called microvm.[3]

All the KVM-based VMMs create a thread per vCPU (this is often referred to as vCPU thread); since such threads are scheduled by the Linux CPU scheduler, the kernel is responsible for scheduling the vCPUs. This means that real-time scheduling policies such as `SCHED_FIFO`, `SCHED_RR`, or `SCHED_DEADLINE` can be used to schedule the VMs' vCPUs.

3.3. Guest OS optimization

Unikernels are a specialized form of library OSs [38,39] in which the OS kernel is designed as a library to be linked to user applications. This allows for reducing the system call overhead and opens the way for kernel customization by removing the subsystems that are not needed/used by the application.

The unikernel approach takes this idea to the limit [40], allowing to build kernels that serve one single application and removing the overhead introduced by memory protection (a unikernel is designed to execute inside a VM, which already implements a protection domain; hence, the kernel does not need to implement any kind of protection). As a result, a unikernel-based system consumes fewer resources and provides better performance.

Although there has not been much previous research on real-time unikernels, it might be conjectured that unikernels also provide better real-time performance (lower latencies). However, this kind of reasoning can sometimes be dangerous and lead to wrong conclusions; for example, since bare-metal hypervisors like Xen [5] are much smaller and simpler than hosted hypervisors (plus the host kernel) such as KVM+Linux [35], many people think that bare-metal hypervisors introduce smaller latencies than hosted hypervisors. This is, however, not the case, as latencies introduced by KVM+Linux actually have a smaller upper bound than the latencies introduced by Xen [7] (probably because Linux and KVM have been fine-tuned for real-time performance for a long time, while support for real-time applications in Xen did not receive as much attention). Hence, it is important to run some

[1] http://www.busybox.net.

[2] https://www.cloudhypervisor.org.

[3] https://www.qemu.org/docs/master/system/i386/microvm.html.

systematic sets of experiments to evaluate the suitability of the guest OS (unikernel-based, in this case) for real-time applications.

While some unikernels provide a specialized Application Programming Interface (API) and require developing the guest applications according to such a non-standard API, others try to support some standard API such as POSIX. Other unikernels even try to provide Application Binary Interface (ABI) compatibility with Linux so that native Linux applications can be executed on the unikernel without requiring a recompilation. However, not all the Linux system calls (or the glibc functions) are currently implemented, and some functionalities might be missing. To support the execution of real-time applications, the guest OS must support fixed-priority scheduling [41] so that the compositional scheduling framework analysis [25] can be used. Moreover, real-time applications need high-resolution timers and are often composed of periodic tasks that can be implemented efficiently, accurately, and without race conditions if the `clock_nanosleep()` call (with the `TIME_ABSTIME` flag) is supported. These are the main features that have be analyzed in previous work [15] for various unikernels. In such a previous work, some candidate unikernels have been analyzed, identifying in OSv [42] the best candidate for serving real-time applications.

OSv is based on a kernel developed from scratch in C++, which is able to parse dynamic ELF executables and directly link to them. While this approach does not allow discarding the kernel subsystems not used by the application (for example, the virtual filesystem and the ZFS code are used even if the application does not access the filesystem), it greatly improves the applications' performance. Currently, OSv provides ABI compatibility with Linux/glibc so that it can execute native Linux applications without requiring to recompile them. Patches for supporting fixed-priority scheduling with the POSIX API and the `clock_nanosleep()/clock_gettime()` system calls are available and provide the features needed to run real-time applications. Hence, OSv will be used in this paper as an example of unikernel to compare the unikernel approach with traditional guest OS kernels such as Linux.

4. Evaluation

This section presents some measurements to evaluate the real-time performance of different virtualization mechanisms and guest solutions. A large number of experiments have been performed on different machines based on the x86_64 architecture, using both AMD and Intel CPUs. All the experiments provided consistent results, and this section reports the ones performed on a system based on an AMD Ryzen 7 5700U CPU running at 1.4 GHz with 16 GB of RAM and an Intel NVMe disk large 1024 GB. Since previous work [7,15] showed that using a real-time Linux version in the host greatly improves the real-time performance, a Preempt-RT Linux kernel (in particular, 6.2.14-rt3 configured to maximize the determinism of the execution times [43]) has been used. The experiments have been performed using Ubuntu 23.10 (installed in a single ext4 partition).

4.1. Latency

The latencies introduced by the various virtualization mechanisms have been evaluated by running `cyclictest` in a guest with OS-level virtualization and full hardware virtualization using Linux or OSv as a guest kernel. To increase the probability of triggering the worst-case latency in a short time, a "stress workload" (based on benchmarks that are known to cause high latencies[4]) has been executed as non-real-time on the host.

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

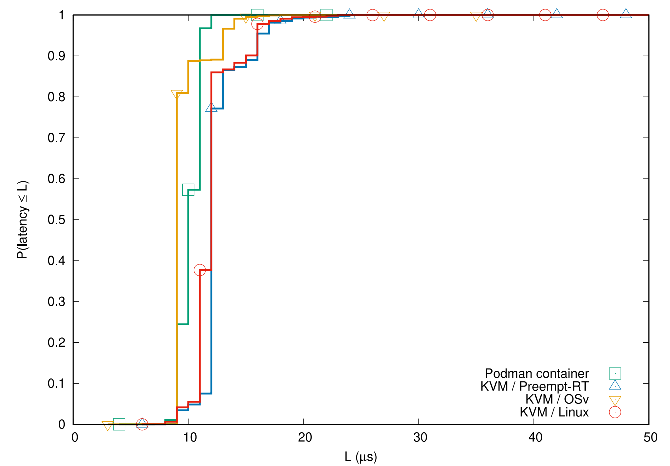


Fig. 1. Experimental CDF of the latencies measured by `cyclictest` with period 100 μs in various kinds of virtual environments on an AMD Ryzen 7 5700U CPU running at 1.4 GHz.

Table 1
Worst-case latencies measured by `cyclictest` with period 100 μs in various kinds of virtual environments on an AMD Ryzen 7 5700U CPU running at 1.4 GHz.

Podman	KVM/Preempt-RT	KVM/OSv	KVM/Linux
51 μs	85 μs	92 μs	484 μs

Fig. 1 shows the experimental Cumulative Distribution Function (CDF) of the latencies measured on the AMD Ryzen CPU mentioned above, when the cyclictest period is set to 100 μs. The x-axis is cut to 50 μs to make the figure more understandable, so the long tails of some of the CDFs are not visible; hence, the worst-case latencies are reported in Table 1. Here, "podman" refers to cyclictest running in a podman container, while KVM / {Preempt-RT, OSv, Linux} refer to cyclictest running in a KVM/QEMU[5] VM with a guest OS based on Preempr-RT, OSv, or vanilla (non-real-time) Linux.

From the figure, it is possible to notice that all the virtual environments introduce small latencies; the OSv curve initially increases faster than the other curves, but then has a longer tail (this seems to indicate that OSv has been optimized for reducing the average latency, but has a larger worst-case). The table shows that OS-level virtualization allows minimizing the worst-case latency, while when using KVM/QEMU, Preempt-RT performs slightly better than OSv; the non-real-time Linux kernel, instead, provides a much larger worst-case latency even if the shape of its CDF plot is similar to the Preempt-RT one.

The results obtained using the Intel Cloud Hypervisor are not shown but are similar to the KVM/QEMU ones.

Summing up, as noticed in previous papers, OSv results in a smaller average latency, although the OSv CDF has a longer tail that is not present in the Linux CDF when Preempt-RT is used. This probably happens because Preempt-RT kernels have been optimized to reduce the maximum latency at the cost of increasing the average values.

4.2. Boot times and memory consumption

The results presented in the previous section confirm that using both a Linux-based OS or OSv as a guest makes it possible to support real-time applications with quite strict temporal requirements. However, the boot times and the resource consumption of a Linux-based OS can be too large.

[5] Here, KVM/QEMU indicates the usage of the QEMU VMM with the KVM hypervisor.

Table 2
Linux Boot/Shutdown times.

Virtualization mechanism	Worst-case	Average
QEMU/Linux	1489 ms	1408 ms
QEMU/Linux with PVH	1310 ms	1235 ms
QEMU/Linux microvm	569 ms	484 ms
Linux with Cloud Hypervisor	451 ms	386 ms

The fact that the Linux boot time can be too large has been verified by starting a VM containing a simple busybox-based guest that immediately shuts down and measuring the time QEMU needs to boot and shut down the OS. The first experiments were based on a minimal KVM/QEMU VM, with one vCPU and 1 GB of RAM; the guest is stored in an initramfs (large about 5 MB). The worst-case (maximum) boot/shutdown time over 100 runs has been measured as 1489 ms, with an average value of 1408 ms; in some situations, having to wait for almost 1.5 s to start an application is not acceptable. These boot times can be reduced by using a para-virtualized boot protocol (for example, using PVH[6]) and a lighter VM model, such as QEMU's microvm.

Table 2 shows the worst-case and average times needed to boot/shutdown a KVM-based VM when a standard Linux kernel is used for the guest OS. It can be noticed that while PVH provides some improvements, the lighter VMM reduces the times to less than half. Moreover, the kernel build-time configuration can be optimized to reduce the boot times further, as will be shown in Table 3.

Another way to decrease the startup times for virtualized applications is to use OS-level virtualization or unikernel guests. To verify this, the experiment has been repeated using a container started with `podman` (using a pre-downloaded container image based on busybox). In this case, the worst-case boot/shutdown time over 100 runs is 552 ms, and the average is 325 ms, showing a surprisingly large variability in the execution times. In order to investigate the source of these large startup times, the experiment has been repeated using `Docker` instead of `podman`, obtaining again a very high worst-case boot/shutdown time (593 ms). The `podman` experiment has then been repeated using an already-setup rootfs instead of a container image that the container manager has to uncompress and extract somewhere on the filesystem. However, this change did not result in noticeable improvements in the startup times, showing that the delay is not due to the rootfs setup mechanism. Further investigation revealed that most of the large startup delays were caused by the time spent in setting up the network namespace (the mechanism used to isolate the container from the network). Creating containers without network isolation resulted in a worst-case startup/shutdown time of 285 ms (294 ms using `Docker`). Finally, the containers have been created by directly invoking the low-level container runtime (`runc` and `crun` have been tested) instead of using a higher-level container manager such as `podman`. In this case, the startup times decreased to a few tens of milliseconds.

Table 3 reports the boot/shutdown times for KVM-based virtual machines (using QEMU or the Intel Cloud Hypervisor as a VMM) with an optimized Linux kernel or OSv as a guest, and for OS-level virtualization using high-level container engines (`Docker` and `podman`) or low-level container runtimes (`crun` and `runc`).

The table shows that unikernels allow decreasing the boot times without renouncing full hardware virtualization (for example, OSv provides boot/shutdown times smaller than `podman`). When using the Intel Cloud Hypervisor, OSv provides boot/shutdown times smaller than 100 ms. Moreover, the Intel Cloud Hypervisor allows achieving boot times smaller than the `podman`/`Docker` ones even when using a standard Linux kernel (and even when the containers are started without network isolation).

[6] https://stefano-garzarella.github.io/posts/2019-08-23-qemu-linux-kernel-pvh.

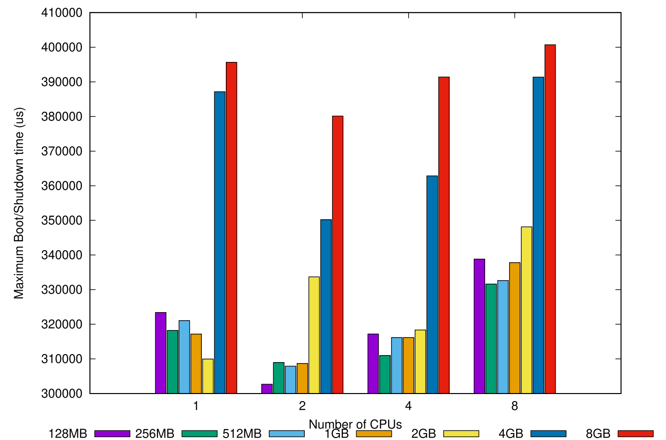


Fig. 2. Worst-case Boot/shutdown times for the KVM/QEMU microvm with different configurations.

Table 3
Boot/Shutdown times with various virtualization technologies.

Virtualization	Worst-Case	Average
Linux	1276 ms	1198 ms
Linux with PVH	1069 ms	985 ms
Linux with microvm	316 ms	274 ms
Linux with Cloud Hypervisor	237 ms	188 ms
OSv	235 ms	168 ms
OSv with microvm	133 ms	88 ms
OSv with Cloud Hypervisor	78 ms	58 ms
PodMan	552 ms	325 ms
PodMan from rootfs	510 ms	305 ms
Docker	593 ms	462 ms
PodMan no network isolation	285 ms	254 ms
Docker no network isolation	294 ms	240 ms
crun	19 ms	15 ms
runc	42 ms	25 ms

Finally, the results obtained with the low-level container runtimes (and especially with `crun`, which has been optimized to reduce the startup times) show that they provide good performance, even if they are not comparable with the native execution of the application on the host OS (on the test machine, a simple "`/bin/true`" application is executed in less than 2 ms). Anyway, the table also shows that full hardware virtualization is competitive with containers handled by a high-level container manager (and even KVM/QEMU – using the microvm architecture – with an optimized Linux guest can provide boot times comparable with the startup times of `podman` containers).

To check how the various solutions scale, the experiment has been repeated with different numbers of vCPUs and different amounts of virtualized RAM. When using OS-level virtualization, the number of vCPUs and the amount of virtualized RAM are imposed by the container runtime by limiting the amount of resources used by the containerized applications. Hence, the only variations in boot times can be caused by the overhead introduced by CPU scheduling or by the memory controller. Some experiments with podman, however, revealed that such an overhead is not significant, and the containers' startup times are not affected by the number of vCPUs or by the amount of RAM. Using hypervisor-based virtualization, instead, the VMM has to create a thread for each vCPU, and to perform a large memory allocation for the guest's memory. Hence, the boot times can be affected by the number of vCPUs and by the VM memory size.

Figs. 2 and 3 show the results (maximum and average times) obtained with a KVM/QEMU microvm booting a guest Linux kernel. As it is possible to notice, increasing the guest RAM generally increases the boot/shutdown times; the same effect can be seen when increasing the number of vCPUs from 2 to 8. However, increasing the number of

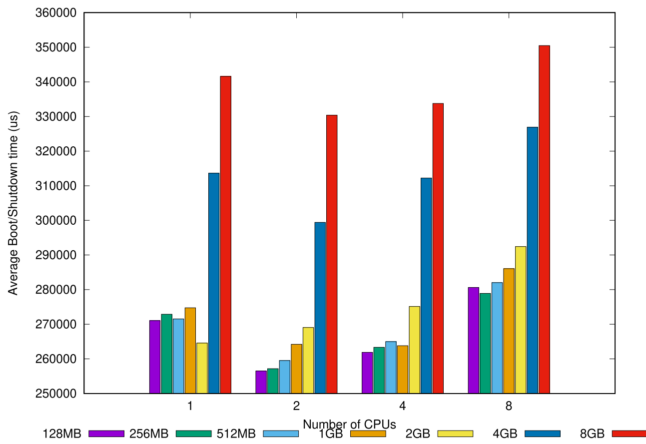


Fig. 3. Average Boot/shutdown times for the KVM/QEMU microvm with different configurations.

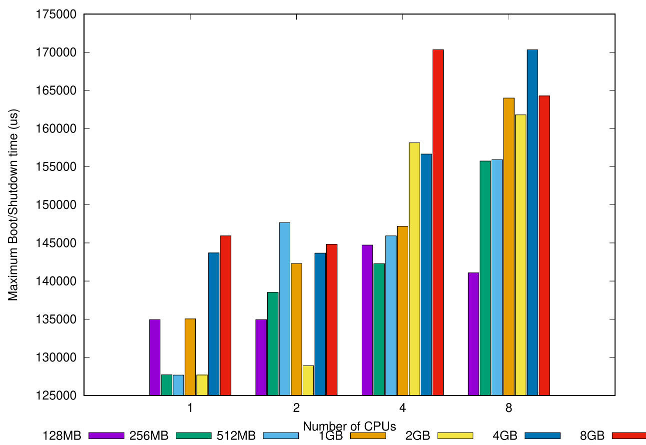


Fig. 4. Worst-case Boot/shutdown times for the OSv unikernel.

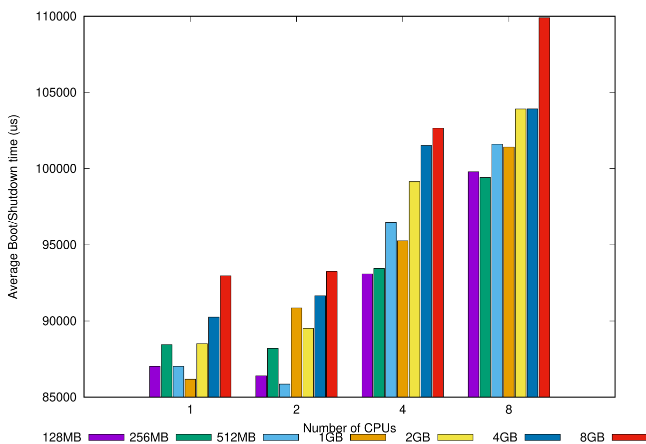


Fig. 5. Average Boot/shutdown times for the OSv unikernel.

vCPUs from 1 to 2 decreases the boot times; this is probably due to the fact that the guest Linux kernel is able to take advantage of the additional vCPU, parallelizing the boot process.

The decrement in boot times when increasing the number of virtual CPUs from 1 to 2 is not visible in the OSv boot/shutdown times (Figs. 4 and 5), indicating that OSv is not able to parallelize the boot process. Finally, Figs. 6 and 7 show the results obtained using the Intel Cloud Hypervisor to boot a Linux kernel: in this case, the number of vCPUs

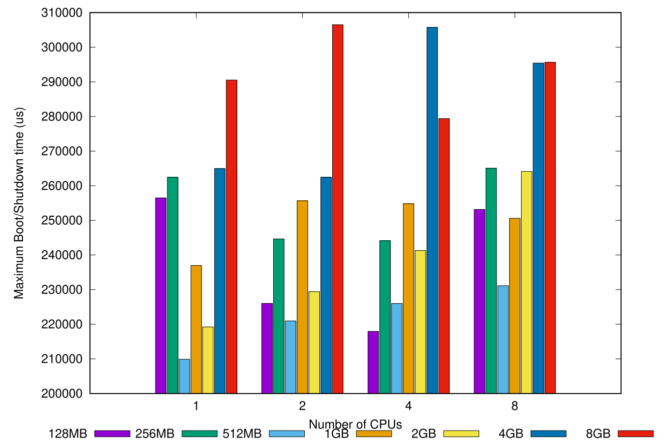


Fig. 6. Worst-case Boot/shutdown times for the Intel Cloud Hypervisor with different configurations.

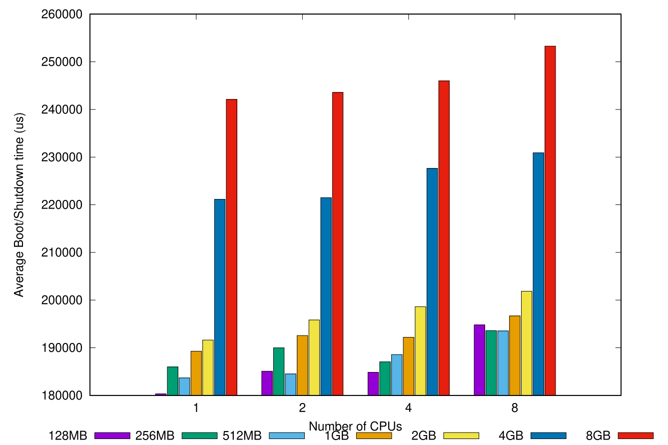


Fig. 7. Average Boot/shutdown times for the Intel Cloud Hypervisor with different configurations.

does not impact too much on the boot times, while increasing the amount of memory allocated to the guest increases the boot times.

The resource consumption of the various virtualization mechanisms evaluated in this section has been estimated by using the `pmap` tool to measure the amount of host memory reserved for the VMs. In the case of full hardware virtualization, the VMM is generally implemented as a single process, and it is possible to apply `pmap` to such a process. Since the guest address space will be part of the VMM process address space, it represents the main contribution to the VM's memory usage. Some simple experiments revealed that a KVM/QEMU VM can boot the Linux kernel and successfully run `cyclictest` if the guest address space is at least 128 MB of memory (this can be set by using the `qemu-system-x86_64` "`-m`" option). According to `pmap`, this results in reserving about 399 MB of the host memory, and using the microvm architecture does not help in reducing the memory usage. A similar experiment revealed that OSv (running the `cyclictest` application) can be booted by a VM equipped with only 16 MB of memory (still reserving about 272 MB of host memory).

Using the Intel Cloud Hypervisor did not help much in reducing the needed amount of memory, as it requires 128 MB of guest memory to boot Linux and successfully run `cyclictest`, reserving about 340 MB of host memory. Similarly, it requires 16 MB of guest memory to boot OSv, reserving about 225 MB of host memory.

On the other hand, running `cyclictest` inside a container started by `crun` only consumed about 18 MB of host memory (11 MB for the `cyclictest` process itself, plus 7 MB used by `crun`), while a container started by `runc` reserved more than 100 MB of host memory.

Table 4
Example real-time taskset.

Task	C	T
τ_0	10.321 ms	34 ms
τ_1	1.57 ms	40 ms
τ_2	6.3 ms	65 ms
τ_3	1.72 ms	90 ms
τ_4	14.712 ms	98 ms
τ_5	19.691 ms	123 ms
τ_6	3.44 ms	128 ms
τ_7	23.266 ms	164 ms
τ_8	89.843 ms	236 ms
τ_9	20.937 ms	257 ms

4.3. Scheduling

While the vCPUs have been previously scheduled with fixed priorities to reduce the latencies measured by `cyclictest`, reservation-based scheduling is generally used to improve the scalability by sharing physical CPU cores between multiple vCPUs. This means that the ith vCPU of the VM can execute for a reserved runtime Q_i every period P_i.

For KVM-based VMs, the user-space VMM creates one thread per vCPU, and the `SCHED_DEADLINE` scheduling policy can be used to schedule the VMM's vCPU threads. Previous works verified that this kind of setup allows to guarantee the respect the guest's temporal constraints [4] and proposed various solutions for dimensioning the vCPUs runtimes and periods [31].

When using OS-level virtualization, instead, the mainline Linux kernel does not provide appropriate scheduling support for real-time containers. However, out-of-tree patches exist for implementing reservation-based scheduling for control groups (and hence containers) [32]. Moreover, a modified version of Kubernetes named RT-K8 [44] allows the creation of real-time containers that use this real-time control group scheduler. Since the same runtime Q and period P are used on all the vCPUs, this solution requires using a global scheduler in the guest. Notice that this is not an inherent limitation of the scheduler but only a simplification in the interface of the real-time control groups.

Since using a hypervisor and VMM the host scheduler (selecting the vCPU to be executed) and the guest scheduler (selecting the real-time tasks to be executed on the vCPU selected by the host scheduler) do not communicate, this scheduling solution allows to guarantee the respect of temporal constraints only if the guest scheduler is based on a partitioned approach. For example, suppose the guest scheduler uses a global fixed priority algorithm and multiple real-time tasks are ready for execution. In this case, the guest kernel can schedule the highest-priority real-time task on a vCPU which has not been selected for execution by the host scheduler. Hence, the highest-priority real-time task ready for execution is not scheduled on a physical CPU (while lower-priority real-time tasks are), breaking some of the assumptions made by the schedulability analysis (see [4] for more details and examples). Moreover, the amount of CPU time to be reserved when using a partitioned scheduling solution is smaller than the amount of CPU time to be reserved when using a global scheduler in the guest [31]. To see a practical example of these effects, consider one of the tasksets used in previous works to compare various scheduling solutions.[7] This taskset, displayed in Table 4, is composed of 10 tasks τ_i and has a utilization $U = \sum_{i=0}^{9} \frac{C_i}{T_i} = 1.4$. Using the "Ovh" heuristic algorithm [31], the tasks can be partitioned over 2 vCPUs allocating $\{\tau_0, \tau_3, \tau_4, \tau_7\}$ on the first vCPU and $\{\tau_1, \tau_2, \tau_5, \tau_6, \tau_8, \tau_9\}$ on the second one. Then, CSF analysis guarantees that every task respects its deadlines if the first vCPU is scheduled with a $(Q_0 = 7.5\ \text{ms}, P_0 = 11\ \text{ms})$ reservation and the second

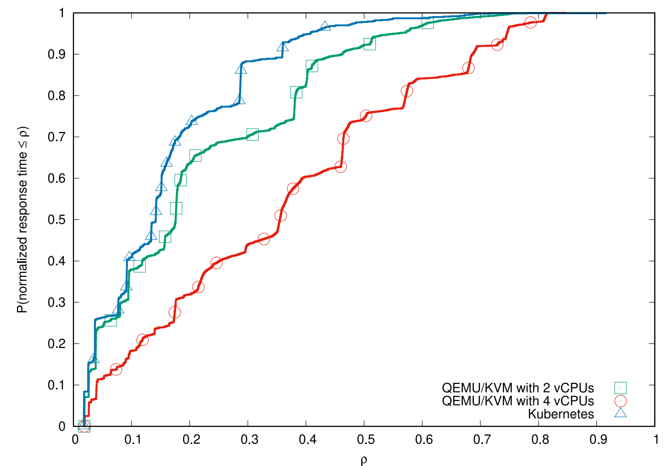


Fig. 8. Experimental CDF of the normalized response times for a set of periodic real-time tasks executed in a QEMU VM with 2 or 4 vCPUs, or in an RT-K8 container.

vCPU is scheduled with a $(Q_1 = 27.5\ \text{ms}, P_1 = 33\ \text{ms})$ reservation. This means that the first vCPU consumes $7.5/11 = 68.18\%$ of a physical CPU core, while the second vCPU consumes $27.5/33 = 83.33\%$ of a physical CPU core. If no physical CPU core has 83.33% idle time, then the VM is not schedulable. Using again the "Ovh" heuristic to partition the taskset on 4 vCPUs, we end up with $\{\tau_0, \tau_8\}$ on the first vCPU, $\{\tau_1, \tau_4\}$ on the second vCPU, $\{\tau_2, \tau_5, \tau_6, \tau_9\}$ on the third vCPU, and $\{\tau_3, \tau_7\}$ on the fourth vCPU. Then, CSF analysis computes $(Q_0 = 8.5\ \text{ms}, P_0 = 12\ \text{ms}), (Q_1 = 2.5\ \text{ms}, P_1 = 11\ \text{ms}), (Q_2 = 5.5\ \text{ms}, P_2 = 14\ \text{ms}), (Q_3 = 2\ \text{ms}, P_0 = 11\ \text{ms})$ consuming 70.83%, 22.73%, 39.29%, and 18.18% of physical CPU cores. Hence, this VM has a higher probability of being schedulable.

If, instead, global scheduling is used, then MPR analysis [46,47], as implemented by CARTS [48] or similar tools,[8] can compute appropriate runtimes and period. For example, the previous taskset results to be schedulable on 2 vCPUs when they are allocated a total runtime of 17.5 ms every 10 ms; since the real-time control group scheduler mentioned above allocates the same runtime on all the control group cores, this means assigning $(Q_0 = 8.75\ \text{ms}, P_0 = 10\ \text{ms}), (Q_1 = 8.75\ \text{ms}, P_1 = 10\ \text{ms})$. The container is hence allocated 175% of a CPU core, while partitioned scheduling requires about 151% of a CPU core.

To verify the practical usability of the previous design, the `rt-app` application[9] has been used to execute the taskset of Table 4 in a KVM/QEMU VM with 2 or 4 vCPUs (scheduled as mentioned above) or in an RT-K8 container with 2 CPU cores reserved as mentioned in the previous analysis. Fig. 8 shows the experimental CDF of the *normalized response times* (defined as response time of a task activation divided by the task period) of the various tasks. Notice that a normalized response time smaller than 1 indicates that the task's deadline is respected. Hence, since all the CDFs shows a 100% probability to have a normalized response time smaller than 1, all the temporal constraints are respected and the practical implementation matches with the theoretical analysis. Being based on CPU reservations, this guarantee is not affected by other real-time or non-real-time containers running on the same host (as soon as the (Q_i, P_i) CPU reservations for all the vCPUs pass an admission test based on schedulability analysis). To verify this property, the experiment has been repeated adding other `SCHED_DEADLINE` tasks and VMs with vCPU threads scheduled by `SCHED_DEADLINE`, with a CPU utilization $U = 5$ (the CPU has 8 cores), verifying that all the tasks are schedulable. The results did not show any visible change in the CDF of the normalized response times, which was almost indistinguishable from Fig. 8.

[7] The tasksets have been randomly generated using the RandFixedSum algorithm [45].

[8] http://retis.santannapisa.it/luca/RTVM.

[9] https://github.com/scheduler-tools/rt-app.

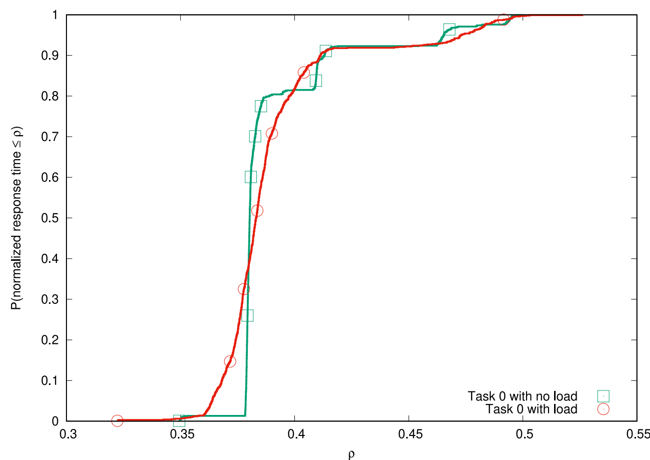


Fig. 9. Experimental CDF of the normalized response times for task $\tau_0 = (10.321, 34)$ executed on the first vCPU of a QEMU VM, scheduled with runtime 7.5 ms and period 11 ms.

This strange result can be explained by remembering that thanks to CPU reservations every vCPU thread is guaranteed to execute for a runtime Q_i every P_i (if the reservation is schedulable), and interferring tasks (or VMs) can only delay the allocation of the runtime inside a reservation period of size P_i. As an example, considering the VM with 2 vCPUs and partitioned scheduling, the first VM is reserved 7,5 ms every 11 ms, hence interferring VMs or real-time tasks can only delay this CPU allocation by 11 ms − 7.5 ms = 3.5 ms. Compared to the tasks' periods, this is a small amount of time, hence the differences in the normalized response times can be hardly noticed. Focusing on one single task with a small period (for example, $\tau_0 = (10.321\ \text{m}, 34\ \text{ms})$) the effect of interferring tasks/VMs becomes more visible, as shown in Fig. 9. As it is possible to notice, the interferring real-time workload makes the response times less predictable, but they are still smaller than the task's period. This effect only depends on the host/hypervisor scheduler and does not depend on the guest scheduling strategy; hence, the interferring workload has the same effect on KVM-based VMs and on containers based on OS-level virtualization.

While the previous examples show that reservation-based scheduling can protect VMs and containers from interference of CPU-hungry co-located containers and VMs, interferring virtualized workload that have a memory-hungry or I/O-hungry behavior can create some hard-to-control interference, that should be accounted for in the schedulability analysis [49–51]. Fortunately, some techniques have been proposed in literature to control such an interference [52–55].

Some modern CPUs even provide hardware mechanisms to control such an interference; for example, Intel Resource Director Technology (RDT) [56] can be integrated both with OS-level virtualization[10] and with hypervisors such as KVM [57].

5. Conclusions

This paper evaluated the possibility of running real-time applications, characterized by some temporal constraints, in virtualized environments. To be considered as a feasible alternative to the direct execution on bare hardware or on an RTOS, such virtual environments must provide low startup times, consume a limited amount of resources, introduce low latencies, and provide appropriate real-time scheduling policies. Hence, the startup times, resource consumption, latencies, and schedulers of different virtualization technologies and guest operating systems (assuming Linux with Preempt-RT as a host kernel) have been evaluated.

The results indicate that containerization mechanisms based on OS-level virtualization introduce very low latencies. Moreover, although some container engines might end up with non-negligible startup times, low-level container runtimes such as `crun` and `runc` provide very low startup times. Finally, hypervisor-based VMs can be competitive with these technologies. In particular, using a lightweight VMM such as the QEMU microvm or the Intel Cloud Hypervisor allows for reducing guest boot times (especially for unikernel guests). Properly configured VMs are also able to limit the introduced latency, if an appropriate guest such as Preempt-RT is used. Guests based on a lightly-patched version of the OSv unikernel also introduce low latencies, slightly larger than the ones introduced by KVM/QEMU but comparable with them.

As future work, the evaluation will be extended using some more realistic and complete real-time applications instead of micro-benchmarks. Moreover, interferences by virtualized real-time applications with memory-intensive or I/O-intensive behaviors will be analyzed to check if OS-level virtualization and hypervisor-based virtualization can help in controlling such interferences; technologies like Intel RDT or similar will be integrated with the hypervisor and the containerization mechanism to verify their effectiveness in virtualized environments.

Other guest OSs (based on different kinds of unikernels or real-time kernels) will also be considered and evaluated. Moreover, inter-task and inter-container communications will be analyzed.

CRediT authorship contribution statement

Luca Abeni: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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] J. Lelli, C. Scordino, L. Abeni, D. Faggioli, Deadline scheduling in the linux kernel, Softw. - Pract. Exp. 46 (6) (2016) 821–839.
[4] L. Abeni, A. Biondi, E. Bini, Hierarchical scheduling of real-time tasks over linux-based virtual machines, J. Syst. Softw. 149 (2019) 234–249.
[5] 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.
[6] 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.
[7] L. Abeni, D. Faggioli, Using Xen and KVM as real-time hypervisors, J. Syst. Archit. 106 (2020) 101709.
[8] H. Li, X. Xu, J. Ren, Y. Dong, ACRN: A big little hypervisor for IoT development, in: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, in: VEE 2019, 2019, pp. 31–44.
[9] A. Biondi, D. Casini, G. Cicero, N. Borgioli, G. Buttazzo, G. Patti, L. Leonardi, L.L. Bello, M. Solieri, P. Burgio, I.S. Olmedo, A. Ruocco, L. Palazzi, M. Bertogna, A. Cilardo, N. Mazzocca, A. Mazzeo, SPHERE: A multi-soc architecture for next-generation cyber-physical systems based on heterogeneous platforms, IEEE Access 9 (2021) 75446–75459.
[10] J. Martins, A. Tavares, M. Solieri, M. Bertogna, S. Pinto, Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems, in: M. Bertogna, F. Terraneo (Eds.), Workshop on Next Generation Real-Time Embedded Systems, NG-RES 2020, in: OpenAccess Series in Informatics (OASIcs), vol. 77, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2020, pp. 3:1–3:14.

[10] https://intel.github.io/cri-resource-manager/stable/docs/policy/rdt.html.

[11] R. Ramsauer, J. Kiszka, D. Lohmann, W. Mauerer, Look mum, no VM exits! (almost), in: Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, OSPERT'17, 2017, http://dx.doi.org/10.48550/arXiv.1705.06932, URL https://ospert2017.uni.lu/.

[12] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and linux containers, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2015, pp. 171–172, http://dx.doi.org/10.1109/ISPASS.2015.7095802.

[13] R.K. Barik, R.K. Lenka, K.R. Rao, D. Ghose, Performance analysis of virtual machines and containers in cloud computing, in: 2016 International Conference on Computing, Communication and Automation, ICCCA, 2016, pp. 1204–1210, http://dx.doi.org/10.1109/CCAA.2016.7813925.

[14] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, F. Huici, Unikraft: Fast, specialized unikernels the easy way, in: Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 376–394, URL https://doi.org/10.1145/3447786.3456248.

[15] L. Abeni, Real-time unikernels: A first look, in: A. Bienz, M. Weiland, M. Baboulin, C. Kruse (Eds.), High Performance Computing, Springer Nature Switzerland, Cham, 2023, pp. 121–133.

[16] G.J. Popek, R.P. Goldberg, Formal requirements for virtualizable third generation architectures, Commun. ACM 17 (7) (1974) 412–421.

[17] G. Banga, P. Druschel, J.C. Mogul, Resource containers: A new facility for resource management in server systems, in: OSDI, Vol. 99, 1999, pp. 45–58.

[18] 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, IEEE, San Jose, California, 2002, pp. 133–142.

[19] D.B. de Oliveira, D. Casini, R.S. de Oliveira, T. Cucinotta, Demystifying the real-time linux scheduling latency, in: M. Völp (Ed.), 32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 165, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020, pp. 9:1–9:23.

[20] A. Moga, T. Sivanthi, C. Franke, OS-level virtualization for industrial automation systems: Are we there yet? in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 1838–1843.

[21] P. Masek, M. Thulin, H. Andrade, C. Berger, O. Benderius, Systematic evaluation of sandboxed software deployment for real-time software on the example of a self-driving heavy vehicle, in: 2016 IEEE 19th International Conference on Intelligent Transportation Systems, ITSC, 2016, pp. 2398–2403.

[22] M. Cinque, R.D. Corte, A. Eliso, A. Pecchia, RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets, in: S. Quinton (Ed.), 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), in: Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2019, pp. 5:1–5:22.

[23] R.V. Riel, Real-time KVM from the ground up, in: KVM Forum 2015, 2015.

[24] R. West, Y. Li, E. Missimer, M. Danish, A virtualized separation kernel for mixed-criticality systems, ACM Trans. Comput. Syst. 34 (3) (2016) URL https://doi.org/10.1145/2935748.

[25] I. Shin, I. Lee, Compositional real-time scheduling framework, in: Proceeding of the 25th IEEE International Real-Time Systems Symposium, 2004, pp. 57–67.

[26] 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.

[27] 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.

[28] G. Lipari, E. Bini, Resource partitioning among real-time applications, in: Proc. of 15th Euromicro Conference on Real-Time Systems, 2003, pp. 151–158.

[29] 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.

[30] 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.

[31] L. Abeni, A. Biondi, E. Bini, Partitioning real-time workloads on multi-core virtual machines, J. Syst. Archit. 131 (2022) 102733.

[32] L. Abeni, A. Balsini, T. Cucinotta, Container-based real-time scheduling in the linux kernel, SIGBED Rev. 16 (3) (2019) 33–38.

[33] S. Sultan, I. Ahmad, T. Dimitriou, Container security: Issues, challenges, and the road ahead, IEEE Access 7 (2019) 52976–52996, http://dx.doi.org/10.1109/ACCESS.2019.2911732.

[34] S. Rostedt, Internals of the RT patch, in: Proceedings of the Linux Symposium, Ottawa, Canada, 2007, pp. 161–172.

[35] 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.

[36] F. Bellard, QEMU, a fast and portable dynamic translator, in: USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41–46.

[37] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D.-M. Popa, Firecracker: Lightweight virtualization for serverless applications, in: 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 20, USENIX Association, Santa Clara, CA, 2020, pp. 419–434.

[38] T. Anderson, The case for application-specific operating systems, in: Proceedings Third Workshop on Workstation Operating Systems, IEEE Computer Society, Los Alamitos, CA, USA, 1992, p. 92,93,94.

[39] D.E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, G.C. Hunt, Rethinking the library OS from the top down, in: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS XVI, Association for Computing Machinery, New York, NY, USA, 2011, pp. 291–304.

[40] A. Madhavapeddy, D.J. Scott, Unikernels: The rise of the virtual library operating system, Commun. ACM 57 (1) (2014) 61–69.

[41] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, J. ACM 20 (1) (1973) 46–61.

[42] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, V. Zolotarov, OSv: Optimizing the operating system for virtual machines, in: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, in: USENIX ATC'14, USENIX Association, USA, 2014, pp. 61–72.

[43] L. Abeni, T. Cucinotta, B. Pinczel, P. Mátray, M.K. Srinivasan, T. Lindquist, On the use of linux real-time features for RAN packet processing in cloud environments, in: H. Anzt, A. Bienz, P. Luszczek, M. Baboulin (Eds.), High Performance Computing. ISC High Performance 2022 International Workshops, Springer International Publishing, Cham, 2022, pp. 371–382.

[44] S. Fiori, L. Abeni, T. Cucinotta, RT-kubernetes: Containerized real-time cloud computing, in: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, SAC '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 36–39.

[45] P. Emberson, R. Stafford, R.I. Davis, Techniques for the synthesis of multiprocessor tasksets, in: Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems, WATERS 2010, 2010, pp. 6–11.

[46] I. Shin, A. Easwaran, I. Lee, Hierarchical scheduling framework for virtual clustering of multiprocessors, in: 2008 Euromicro Conference on Real-Time Systems, 2008, pp. 181–190.

[47] A. Easwaran, I. Shin, I. Lee, Optimal virtual cluster-based multiprocessor scheduling, Real-Time Syst. 43 (1) (2009) 25–59.

[48] L.T.X. Phan, J. Lee, A. Easwaran, V. Ramaswamy, S. Chen, I. Lee, O. Sokolsky, CARTS: A tool for compositional analysis of real-time systems, SIGBED Rev. 8 (1) (2011) 62–63.

[49] H. Yun, R. Pellizzon, P.K. Valsan, Parallelism-aware memory interference delay analysis for COTS multicore systems, in: 2015 27th Euromicro Conference on Real-Time Systems, 2015, pp. 184–195, http://dx.doi.org/10.1109/ECRTS.2015.24.

[50] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding and reducing memory interference in COTS-based multi-core systems, Real-Time Syst. 52 (2016) 356–395.

[51] D. Casini, A. Biondi, G. Nelissen, G. Buttazzo, A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2020, pp. 239–252, http://dx.doi.org/10.1109/RTAS48715.2020.000-3.

[52] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms, in: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2013, pp. 55–64, http://dx.doi.org/10.1109/RTAS.2013.6531079.

[53] H. Yun, R. Mancuso, Z.-P. Wu, R. Pellizzoni, PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms, in: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2014, pp. 155–166, http://dx.doi.org/10.1109/RTAS.2014.6925999.

[54] F. Farshchi, Q. Huang, H. Yun, BRU: Bandwidth regulation unit for real-time multicore processors, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2020, pp. 364–375, http://dx.doi.org/10.1109/RTAS48715.2020.00011.

[55] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, R. Mancuso, MemPol: Policing core memory bandwidth from outside of the cores, in: 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2023, pp. 235–248, http://dx.doi.org/10.1109/RTAS58335.2023.00026.

[56] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, O. Krieger, A closer look at intel resource director technology (RDT), in: Proceedings of the 30th International Conference on Real-Time Networks and Systems, RTNS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 127–139, URL https://doi.org/10.1145/3534879.3534882.

[57] H. Kim, R.R. Rajkumar, Real-time cache management for multi-core virtualization, in: Proceedings of the 13th International Conference on Embedded Software, EMSOFT '16, Association for Computing Machinery, New York, NY, USA, 2016, URL https://doi.org/10.1145/2968478.2968480.