Container-Based Real-Time Scheduling in the Linux Kernel

Luca Abeni, Alessio Balsini, Tommaso Cucinotta
Scuola Superiore Sant'Anna
Pisa, Italy
first.last@santannapisa.it

ABSTRACT

In recent years, there has been a growing interest in supporting component-based software development of complex real-time embedded systems. Techniques such as machine virtualisation have emerged as interesting mechanisms to enhance the security of these platforms, while real-time scheduling techniques have been proposed to guarantee temporal isolation of different virtualised components sharing the same physical resources. This combination also highlighted criticalities due to overheads introduced by hypervisors, particularly for low-end embedded devices. This led to the need of investigating deeper into solutions based on lightweight virtualisation alternatives, such as containers. In this context, this paper proposes to use a real-time deadline-based scheduling policy built into the Linux kernel to provide temporal scheduling guarantees to different co-located containers. The proposed solution extends the `SCHED_DEADLINE` scheduling policy to schedule Linux control groups, allowing user threads to be scheduled with fixed priorities inside the control group scheduled by `SCHED_DEADLINE`. The proposed mechanism can be configured via control groups, and it is compatible with commonly used tools such as LXC, Docker and similar. This solution is compatible with existing hierarchical real-time scheduling analysis, and some experiments demonstrate consistency between theory and practice.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; **Real-time operating systems**; • **Software and its engineering →** *Virtual machines*;

KEYWORDS

Real-Time scheduling, Virtualization, Containers, Linux

ACM Reference Format:
Luca Abeni, Alessio Balsini, Tommaso Cucinotta. 2018. Container-Based Real-Time Scheduling in the Linux Kernel. In *Proceedings of Embedded Operating System Workshop (EWiLi'18).* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Component-based software design and development is a well-known approach used in software engineering to address the complexity of the applications to be developed and to improve the software re-usability. Using this technique, complex software systems are decomposed into *components* described by well-defined interfaces; each component can be developed independently, and various components can be integrated later on the hardware target platform.

When developing real-time applications, the "traditional" software interface of each component must be complemented with non-functional attributes, e.g., those describing its timing requirements. In real-time literature, component-based design has been supported by modelling a component-based application as a scheduling hierarchy, as done for example in the Compositional Scheduling Framework (CSF) [14, 15], that has been extended to support multiple CPUs and / or CPU cores in various ways [3–5, 8, 18].

In the past, these frameworks have been generally implemented by running each software component in a dedicated Virtual Machine (VM) and using a hypervisor scheduler as the root of the scheduling hierarchy [7, 9, 17]. In some situations (especially in embedded devices), it can be interesting to reduce the overhead introduced by a full VM, and to use a more lightweight virtualisation technology such as container-based virtualisation.

This paper presents a CSF implementation based on containers (specifically, Linux control groups and namespaces). To support CSF analysis, this technology (on which many widely used programs such as Docker, LXC, LXD and similar tools are based) has been extended by implementing a theoretically sound 2-levels scheduling hierarchy. The `SCHED_DEADLINE` scheduling policy (implementing the CBS [1] algorithm) is used as the root of the scheduling hierarchy, and the standard fixed priority scheduler (`SCHED_FIFO` or `SCHED_RR` scheduling policy) is used at the second level of the hierarchy. The main difference with respect to previous implementations [6] is that the standard `SCHED_DEADLINE` code-base in the mainline Linux kernel is extended, rather than reimplementing an independent reservation-based scheduler.

The proposed approach is compatible with existing real-time analysis (some experiments show that the obtained results are compatible with MPR analysis [8], but different kinds of analysis and design techniques can be used as well) and with commonly used software tools, that do not need to be modified (an unmodified LXC installation has been used for the experiments).

2 DEFINITIONS AND BACKGROUND

As previously mentioned, isolation among software components can be achieved by running each component in a separate VM. By using appropriate scheduling techniques, it is possible not to limit isolation to spatial isolation, but to implement temporal isolation too (meaning that the worst case temporal behaviour of a component is not affected by the other components)[1].

2.1 Virtualisation and Containers

The software components of a complex real-time application can be executed in different kinds of dedicated VMs (a VM per component), based on different virtualisation technologies and providing different performance and degrees of isolation among components.

[1]While in clouds or large-scale servers temporal isolation is sometimes implemented by dedicating entire CPU cores to components, in embedded systems the number of available CPU cores is too small, and this solution is often not appropriate.

EWiLi'18, October 4th, 2018, Torino, Italy

For example, some VMs are based on *full hardware virtualisation* whereas other VMs are based on *container-based virtualisation* (or *OS-level virtualisation*).

When full hardware virtualisation is used, the VM software implements and emulates all the hardware details of a real machine (including I/O devices, etc.) and the guest OS running in the VM can execute as if it was running on real hardware (in theory, any unmodified OS can run in the VM without being aware of the virtualisation details - but this is generally bad for performance, and some kind of *para-virtualisation* is generally used).

When OS-level virtualisation is used instead, only the OS kernel services are virtualised. This is done by the host kernel, which uses virtualisation of its services to provide isolation among different guests. This means that the kernel is shared between the host OS and the guest OSs; hence all the guests have to be based on the same hardware architecture and kernel. As an example, this technique allows for using the same hardware platform to run multiple Linux distributions based on the same kernel and it is often used to run multiple Linux applications/containers on the same server (think about Docker). Every distribution will be isolated from the others, having the impression to be the only one running on the kernel.

The Linux kernel supports OS-level virtualisation through *control groups* (also known as `cgroups`, a mechanism originally inspired by *resource containers* [2]), and *namespaces*, that can be used to isolate various kernel resources. User-space tools like Docker or LXC are used to set-up the control groups and namespaces as appropriate to execute guest OSs (or isolated applications) inside them.

Most of the previous implementations of the Compositional Scheduling Framework focused on full hardware virtualisation, directly executing guest machine language instructions on the host CPUs to improve performance (and relying on special CPU features to make this possible - more technically, to make the CPU fully virtualisable [13]). The software responsible for controlling the execution of guest code, the *hypervisor*, contains a scheduler that is responsible for selecting the VM to be executed and implements the root of the scheduling hierarchy. In previous works, this scheduler has been modified to support real-time resource allocation [6, 17].

When using container-based virtualisation, on the other hand, the host kernel is responsible for scheduling all of the tasks contained in the various VMs, and implements the root of the scheduling hierarchy. Since the host scheduler can easily know if a guest is executing a real-time task (using the POSIX `SCHED_FIFO` or `SCHED_RR` policy) or not, it is easy to schedule only real-time tasks through a CPU reservation (implemented by the `SCHED_DEADLINE` policy) serving the VM. This is another advantage of container-based virtualisation that will be discussed later.

2.2 Real-Time Scheduling

A real-time application can be built by integrating a set of *components*, where each component C can be modelled as a set of real-time tasks $C = \{\tau_1, ...\tau_n\}$. A real-time task τ_i is a stream of activations, or jobs, $J_{i,j}$ ($J_{i,j}$ is the j^{th} job of task τ_i) with job $J_{i,j}$ arriving (becoming ready for execution) at time $r_{i,j}$ and executing for a time $c_{i,j}$ before finishing at time $f_{i,j}$ (the finishing time $f_{i,j}$ depends on the arrival time $r_{i,j}$ and the scheduling decisions).

The Worst Case Execution Time (WCET) of task τ_i is defined as $C_i = \max_j\{c_{i,j}\}$, while the period (or minimum inter-arrival time) is defined as $T_i = r_{i,j+1} - r_{i,j}$ (or $T_i = \min_j\{r_{i,j+1} - r_{i,j}\}$).

Each job is also characterised by an *absolute deadline* $d_{i,j}$, representing a temporal constraint that is respected if $f_{i,j} \leq d_{i,j}$. The goal of a real-time scheduler is to select tasks for execution so that all the deadlines of all the jobs are respected.

As previously mentioned, a component C is executed in a dedicated VM having m virtual CPUs[2] π_k (with $0 \leq k < m$). When container-based virtualisation is used, the host scheduler is responsible for:

(1) selecting the virtual CPUs that execute at each time;
(2) for each selected virtual CPU, select the task to be executed.

In this paper, virtual CPUs are scheduled using a reservation-based algorithm: each virtual CPU π_k is assigned a *maximum budget* (or *runtime*) Q_k and a *reservation period* P_k, and is reserved an amount of time Q_k every P_k for execution. In more details, the CBS algorithm [1] as implemented in the `SCHED_DEADLINE` scheduling policy [10] is used (see the original papers for more details about the algorithm and its implementation).

3 REAL-TIME FOR LINUX CONTAINERS

The solution presented in this paper is based on implementing a hierarchical scheduling system in a way that is compatible with the most commonly used container-based virtualisation solutions. Unmodified LXC has been used for the experiments as an example (because of its simplicity), but other tools can be used as well.

3.1 A Hierarchical Scheduler for Linux

Independently from the used user-space tool, container-based virtualisation in Linux is implemented by combining two different mechanisms: control groups (or cgroups) and namespaces; the user-space tools like LXC and Docker are only responsible for setting up the namespaces and cgroups needed to implement a VM.

Namespaces are used to isolate and virtualise system resources: a process executing in a namespace has the illusion to use a dedicated copy of the namespace resources, and cannot use nor see resources outside of the namespace. Hence, namespaces affect the resources' visibility and accessibility.

Control groups are used to organise the system processes in groups, and to limit, control or monitor the amount of resources used by these groups of processes. Hence, cgroups affect the resource scheduling and control. In particular, the real-time control group can be used to limit somehow and control the amount of time used by `SCHED_FIFO` and `SCHED_RR` tasks in the cgroup. Traditionally, it allows for associating a *runtime* and a *period* to the real-time tasks of the control group (hence, it could potentially be used for implementing a scheduling hierarchy), but its behaviour is not well defined. Hence, the resulting scheduling hierarchy is not easy to analyse; for example, it implements a runtime "balancing" mechanism among CPUs[3], it uses a scheduling algorithm similar

[2]If full hardware virtualisation is used, the number of virtual CPUs m can theoretically be larger than the number of physical CPUs M; if container-based virtualisation is used, $m \leq M$.

[3] For more information, refer to the `do_balance_runtime()` function in https://github.com/torvalds/linux/blob/master/kernel/sched/rt.c.

to the deferrable server [16] algorithm (but not identical - hence problematic for the analysis) and it has a strange behaviour for hierarchies composed of more than two levels. Moreover, real-time tasks are migrated among CPUs without looking at the runtimes available on the various cores.

In our approach, we re-use the software interface of the real-time control groups, changing the scheduler implementation to fit in the CSF. A CBS-based reservation-based algorithm, `SCHED_DEADLINE`, was already available in the mainline kernel, so we decided to use it to schedule the control groups. Linux implements global scheduling between available CPUs by using per-core ready task queues, named *runqueues*, and migrating tasks between runqueues when needed (for example, "push" and "pull" operations are used to enforce the global fixed-priority invariant - the M highest priority tasks are scheduled - between fixed priority tasks). For each CPU, there is a runqueue for non-real-time (`SCHED_OTHER`) tasks, one for fixed priority tasks (the real-time runqueue) and one for `SCHED_DEADLINE` tasks (the deadline runqueue). Each task descriptor contains some *scheduling entities* (one for `SCHED_OTHER`, one for `SCHED_FIFO` / `SCHED_RR` and one for `SCHED_DEADLINE` - the deadline entity) that are inserted in the runqueues.

In our scheduler, we added a deadline entity to the real-time runqueue associated with every CPU in a control group. This way, the real-time runqueues of a control group can be scheduled by the `SCHED_DEADLINE` policy. When `SCHED_DEADLINE` schedules a deadline entity, if the entity is associated with a real-time runqueue, then the fixed priority scheduler is used to select its highest priority task. The usual migration mechanism based on push and pull operations is used to enforce the global fixed priority invariant (the m highest priority tasks are scheduled).

The resulting real-time control group now implements a 2-levels scheduling hierarchy with a reservation-based root scheduler and a local scheduler based on fixed priorities: each control group is associated with a deadline entity per runqueue (that is, per CPU), and fixed priority tasks running inside the cgroup are scheduled only when the group's deadline entity has been scheduled by `SCHED_DEADLINE`. Each deadline entity associated with a real-time runqueue is strictly bound to a single physical CPU.

Currently, all the deadline entities of the cgroup (one per runqueue / CPU) have the same runtime and period, so that the original cgroup interface is preserved. However, this can be easily improved if needed (for example, using an "asymmetric distribution" of runtime and period in the cgroup cores can improve the schedulability of the cgroup real-time tasks [11, 18]). On the other hand, if deadline entities with different parameters are associated with the runqueues of a cgroup, then the resulting container is characterised by virtual CPUs having different speeds, so the push and pull operations need to be updated.

This container-based implementation of a hierarchical scheduler has an important advantage compared to full virtualisation: when the runtime of a virtual CPU (vCPU) is finished and the vCPU is throttled, the scheduler can migrate a real-time task from that vCPU to others (using the "push" mechanism). With machine virtualisation, instead, the guest OS kernel cannot know when a vCPU is throttled and its tasks must be migrated to other vCPUs; hence, it may throttle some real-time tasks while some other vCPUs still have an available runtime that is left unused.

As an example, consider a task set $\Gamma = \{\tau_1, \tau_2\}$, with $C_1 = 40ms$, $T_1 = 100ms$, $C_2 = 60ms$, $P_2 = 100ms$ running in a VM with 2 vCPUs having runtimes $Q_1 = Q_2 = 50ms$ and periods $P_1 = P_2 = 100ms$. Assume that τ_1 is scheduled on the first vCPU and τ_2 is scheduled on the second one. At time $40ms$, the job of task τ_1 finishes, leaving $10ms$ of unused runtime on the first vCPU, and at time $50ms$ the job of task τ_2 has consumed all the runtime of the second vCPU, hence the vCPU is throttled. If the VM is implemented using traditional hardware virtualisation, the guest scheduler has no way to know that migrating τ_2 on the first vCPU the job would still have some runtime to use, while using the container-based approach the host scheduler can push τ_2 from the second vCPU of the control group to the first one, allowing it to use the remaining $10ms$ of runtime and hence to finish in time.

All the control groups are associated with well-specified runtimes Q_i and periods P_i, and the `SCHED_DEADLINE` scheduling policy enforces that the real-time tasks of a cgroup cannot use a fraction of CPU time larger than Q_i/P_i. Hence, we decided to implement only a simple 2-levels scheduling hierarchy. Since the real-time control group interface allows to build deeper hierarchies nesting cgroups inside other cgroups, we decided to support this feature by "flattening" deeper cgroup hierarchies: a cgroup with runtime Q_i and period P_i can contain children cgroups with runtimes Q_k^i and periods P_k^i only if $\sum_k Q_k^i/P_k^i \leq Q_i/P_i$. Every time a child cgroup is created, its utilisation Q_k^i/P_k^i is subtracted from the parent's utilisation Q_i/P_i (technically, the parent's runtime is decreased accordingly) and the child cgroup is associated with a deadline entity that is inserted in the "regular" `SCHED_DEADLINE` runqueue (the same runqueue where the parent's deadline entity is inserted). In this way, the temporal isolation properties and the reservation guarantees of each cgroup are preserved, while user-space tools can still create nested cgroups (for example, LXC does not create its cgroups in the root cgroup, but in a dedicated "LXC" cgroup).

3.2 Schedulability Analysis

The presented kernel modifications result in:

- 2 levels of scheduling (a root scheduler and a local scheduler);
- m vCPUs for each VM;
- reservation-based scheduling of the vCPUs (reservation-based root scheduler): each vCPU π_k is assigned a CPU reservation (Q_k, P_k);
- local scheduler based on fixed priorities.

This kind of scheduling hierarchies has already been widely studied in real-time literature; hence, there is no need to develop new analysis techniques, but previous work can be re-used. In particular, Section 4 will show that the presented implementation provides experimental results that are consistent with the MPR analysis [8].

According to the MPR model, each VM is assigned a total runtime Θ every period Φ, to be allocated over at most m virtual CPUs; hence, the VM is associated with a multi-processor reservation (Θ, Φ, m) to be implemented as m CPU reservations. Using our implementation, this corresponds to using a reservation period $P_k = \Phi$ and a runtime $Q_k = \lceil \Theta/m \rceil$ for each virtual CPU π_k. Note that the original MPR paper only analysed EDF-based local schedulers; however, the paper mentioned that it is possible to extend the analysis to fixed priorities.

Such an extension is already implemented in the CARTS tool [12], that has been used to validate the experimental results.

3.3 Application-Dependent Analysis

As discussed, existing CSF analysis can be applied to container-based hierarchical scheduler presented in this paper (and Section 4 will show that the scheduler provides results consistent with the theory). However, this kind of analysis is often pessimistic, because it has to consider the worst-case arrival pattern for the tasks executing in the container. If more information about the structure of the containerised applications is known, then a less pessimistic application-dependent schedulability analysis can be used.

For example, consider a simple scenario where a pipeline of n audio processing tasks $\tau_1, \dots, \tau_n$ is activated periodically to compute the audio buffer to be played back on the next period. The activation period of the pipeline also constitutes the end-to-end deadline D for the whole pipeline computations. This use-case is recurrent for example when using common component-based frameworks for audio processing, such as the JACK[4] low-latency audio infrastructure for Linux, where multiple audio filter or synthesis applications can be launched as separate processes, but their real-time audio processing threads are combined into a single arbitrarily complex direct acyclic graph of computations. For the sake of simplicity, in the following we consider a simple sequential topology where the tasks $\tau_1, \dots, \tau_n$ have to be activated one by one in the processing workflow (typical of having multiple cascaded filters), and we focus on the simple single-processor case.

The traditional way to handle the timing constraints of the audio tasks in this scenario is the one to deploy the whole JACK processing workflow at real-time priority. However, in order to let the audio processing pipeline co-exist with other real-time components, one possibility is to use the `SCHED_DEADLINE` policy, that allows to associate a runtime Q_i and a period P_i with each thread of the pipeline (each task τ_i is scheduled with a CPU reservation (Q_i, P_i)).

Therefore, one would apply a standard end-to-end deadline splitting technique, for example setting each intermediate deadline (and period) P_i proportional to the worst-case execution time C_i of task τ_i in the pipeline, while keeping the intermediate reservation budget Q_i equal to (or slightly larger than) C_i:

$$\begin{cases} Q_i & = C_i \\ P_i & = \frac{C_i}{\sum_{j=1}^{n} C_j} D. \end{cases} \tag{1}$$

This would ensure that each task τ_i in the pipeline gets C_i time units on the CPU within P_i, where all of the intermediate deadlines sum up to the end-to-end value D. The computational bandwidth U_{DL} needed for the whole pipeline is:

$$U_{DL} \quad = \quad \sum_{i=1}^{n} \frac{Q_i}{P_i} = \sum_{i=1}^{n} \frac{C_i}{\frac{C_i}{\sum_{j=1}^{n} C_j} D} = n \frac{\sum_{j=1}^{n} C_j}{D}. \tag{2}$$

Using the container-based hierarchical scheduler, instead, it is possible to configure the system with the same ease in terms of schedulability guarantees, but in a much less conservative way. Indeed, the pipeline tasks can simply be all attached to the same group reservation (scheduling the threads with real-time priorities and inserting them in a dedicated cgroup) with runtime equal to the overall pipeline processing time $Q = \sum_{j=1}^{n} C_j$, and a period equal to the end-to-end deadline constraint $P = D$, achieving an overall computational bandwidth U_{HCBS} of simply:

$$U_{HCBS} = \frac{Q}{P} = \frac{\sum_{j=1}^{n} C_j}{D} = \frac{U_{DL}}{n}, \tag{3}$$

namely n times smaller than needed when associating a CPU reservation for each task of the pipeline.

However, in the most straightforward set-up where the tasks in the pipeline execute strictly one after the other[5], it is clear that, even under `SCHED_DEADLINE`, the whole pipeline can execute with a much lower real-time bandwidth occupation. Indeed, once a task is finished, it blocks after unblocking the next task in the pipeline. As a consequence, only one of the real-time reservations is needed at each time, throughout each activation of the whole pipeline. Therefore, it is sufficient to have a single reservation occupying a bandwidth sufficient for hosting the computationally heaviest task (Q_i/P_i turns out to be a constant anyway, following the above deadline splitting technique in Eq. (1)), and resetting its runtime and deadline parameters forward, following the $(Q_1, P_1), \dots (Q_n, P_n)$ sequence, as the processing across tasks moves on. However, albeit convenient, such a "reservation hand-over" feature is not available in `SCHED_DEADLINE`, and, as shown below, its effect is actually achieved equivalently by the hierarchical scheduler here proposed.

4 EXPERIMENTAL RESULTS

The hierarchical scheduler for Linux presented in this paper has been evaluated through a set of experiments. The patchset used for these experiments is freely available online[6]. The experiments have been performed on different machines, with different kinds of CPUs including an Intel(R) Core(TM) i5-5200U and an Intel(R) Xeon(R) CPU E5-2640, achieving consistent results, with CPU frequency switching inhibited and Intel Turbo Boost disabled.

4.1 Real-Time Schedulability Analysis

A first experiment has been designed to show one of the advantages of the proposed approach, with respect to traditional full hardware virtualisation. To achieve this result, a container-based VM has been started using LXC. The VM has been configured with 4 virtual CPUs, with runtime $10ms$ and period $100ms$. When starting a CPU-consuming task (basically, an empty `while()` loop) in the VM, it has been possible to see that the task was able to consume 10% of the CPU time *on each virtual CPU*. This happens because when the task consumes the whole runtime on a virtual CPU, the corresponding deadline entity is throttled, and the task is migrated to a different virtual CPU, where it is able to consume other $10ms$ of runtime.

This experiment shows that using our hierarchical scheduler the guest's real-time tasks can effectively consume all the runtime allocated to all the virtual CPUs of the VM. When using full hardware virtualisation, instead, this is not possible. For verification, the same

[4]More information at: http://www.jackaudio.org.

[5]JACK can be configured to execute tasks in a real "pipelined" fashion, where multiple tasks are activated at the same time on different pieces of the overall audio buffer being processed.

[6] https://github.com/lucabe72/LinuxPatches/tree/Hierarchical_CBS-patches, applies to the `master` branch of the Linux `tip` repository (git://git.kernel.org/pub/scm/linux/kernel/git/tip/tip.git), as checked out at end of June 2018 (commit `f3a7e2346711`).

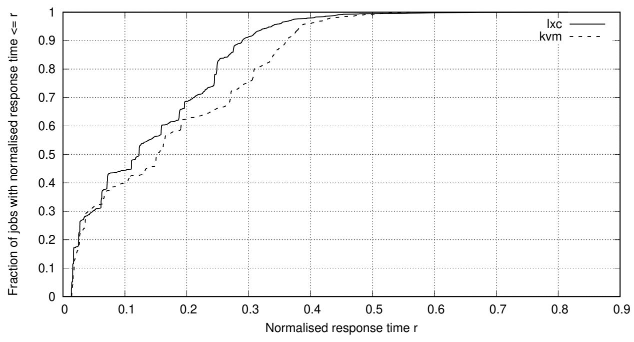

Figure 1: CDF of the normalised response times obtained using LXC and kvm. While the worst case response time is the same, LXC provides better average response times (the LXC CDF is above the kvm CDF).

experiment has been repeated in a kvm-based VM[7] scheduling the 4 virtual CPU threads with SCHED_DEADLINE [10] (runtime $10ms$ and period $100ms$ for each thread), verifying that the CPU-consuming task is able to consume only 10% of the CPU time *of one single virtual CPU*. This happens because when a virtual CPU thread consumes all the runtime, it is throttled, but the guest scheduler does not migrate the thread because the guest scheduler has no way to be notified when the virtual CPUs are throttled.

In the second experiment, the presented hierarchical scheduler has been verified to correctly implement CSF. Different components C with different utilisations $\sum_i \frac{C_i}{T_i}$ have been generated, using the CARTS tool (downloadable from https://rtg.cis.upenn.edu/carts/) to derive some (Θ, Φ, m) parameters that allow for scheduling the component tasks without missing deadlines. Then, the component has been executed in an LXC container (using a minimal Linux-based OS designed to allow reproducible experiments in virtualised environments), verifying that no deadline has been missed.

For example, the task set $\Gamma = \{(2, 32), (2.88, 40), (11.6, 46), (9.125, 48), (0.7555), (17.362), (14.5, 90), (2.6, 103), (4.5, 261), (67, 270), (34.3, 275), (4.45, 283), (8.55, 311), (47.4423), (97.4, 490)\}$ (where (C_i, T_i) represents a periodic task with WCET C_i and period T_i, in ms) is schedulable on 4 virtual CPUs with MPR parameters ($\Theta = 30ms, \Phi = 10ms, m = 4$). When executing such a task set in an LXC container with 4 CPUs, runtime $7.5ms$ and period $10ms$, no deadline has been missed indeed.

The experimental Cumulative Distribution Function (CDF) of the normalised response times (where the normalised response time $r_{i,j}/T_i$ of a job is defined as the job's response time divided by the task period) is represented in Figure 1. For comparison, the figure also contains the experimental CDF obtained when running the same task set Γ in a kvm-based VM (with each one of the 4 virtual CPU threads scheduled by SCHED_DEADLINE, with runtime $7.5ms$ and period $10ms$). From the figure, it is possible to appreciate 3 different things. First of all, both the VMs are able to respect all the deadlines: this can be seen by verifying that both the curves arrive at probability 1 for values of the normalised response time ≤ 1 (if the normalised response time is smaller than 1, the deadline is not missed). Second, the worst-case experienced normalised response time is similar for the two VMs: the two curves arrive at probability 1 for similar values of the worst-case response time. This means that the worst-case behaviour of the two VMs is similar, as expected. The last thing to be noticed is that the LXC curve is generally above the kvm one. This means that in the average case the response times of the container-based hierarchical scheduler are smaller than the one of the kvm-based one, because of the para-virtualised nature of the container-based scheduler (as previously mentioned): when a virtual CPU is throttled, the scheduler can migrate its tasks to a different virtual CPU that still has some runtime to be exploited.

[7]Other virtualisation solutions use different kinds of hypervisors - for example, Xen uses a bare-metal hypervisor - but the results do not change.

Table 1: JACK experiment parameters

JACK clients	C_i	$290\mu s$
$(i = 1, 2)$	P_i	$1319.32\mu s$
jackd	C_3	$58.05\mu s$
	P_3	$263.86\mu s$
Interference	$C_4 = Q_4$	$6667\mu s$
	P_4	$16667\mu s$

4.2 Audio Pipeline

The third experiment shows the advantages of using the container-based hierarchical scheduler for the management of a real-time JACK audio processing workflow, with respect to the approach of isolating each activity in a separate CPU reservation by scheduling all the JACK threads as SCHED_DEADLINE [10]. In this experiment, 2 JACK clients are sequentially chained. The first one takes as input stream the input data of the audio device, performs some computations and forwards it to the next client, until the last one. Actual audio I/O through ALSA is performed by jackd, the JACK audio server, which constitutes an implicit 3^{rd} element in our sample audio processing pipeline. JACK has been configured with a buffer size of 128 audio frames with a sample rate of 44100 frames/s, resulting in a total audio latency of $1000/44100 * 128 = 2.9025ms$ (corresponding to the period of the audio pipeline). The JACK clients performed a synthetic audio processing activity with nearly constant execution time. The overall JACK real-time workload added up to a 22% CPU load on the CPU. Also, an additional *interference load* has been added, as an additional 4^{th} real-time activity using a 40% CPU reservation. All real-time tasks have been pinned down on the same CPU, in order to reproduce the assumptions behind the calculations in Section 3.3. All the parameters for all real-time tasks and associated reservations have been summarised in Table 1.

The performance of JACK in terms of experienced xruns have been measured when the JACK threads are scheduled under the mainline SCHED_DEADLINE policy (indicated as "DL" in the figures) and when the hierarchical control group scheduler (indicated as "HCBS" in the figures) is used. First, the 3 real-time threads belonging to the JACK audio processing pipeline from the 2 JACK clients and jackd itself, have been scheduled with SCHED_DEADLINE using the parameters obtained from the deadline splitting approach in from Eq. (1). As expected, using these parameters no xruns were experienced. Since Eq. (1) can end up in an over-allocation of CPU

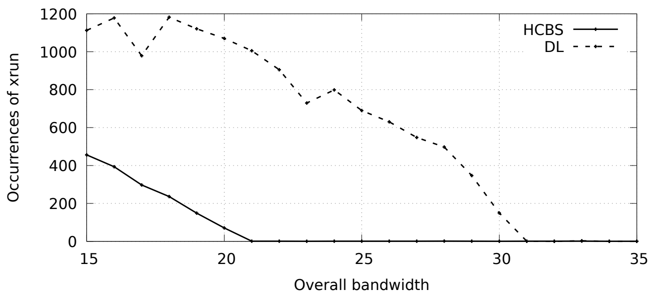

Figure 2: Number of xruns reported by JACK throughout all the 10 repetitions of each configuration. The proposed control group scheduler (HCBS) requires a smaller percentage of CPU time than standard `SCHED_DEADLINE` (DL) to avoid xruns.

time, the test has been repeated reducing the `SCHED_DEADLINE` runtimes. In practice, the Q_i parameters have been rescaled proportionally, so that the fraction of CPU time reserved to the three threads ranged from 15% to 35%. The DL line in Figure 2 reports the results, highlighting that the system starts behaving correctly when the percentage of CPU time reserved to the three JACK threads is about 31%. This is well below the conservative theoretical bound in Eq. (2), as expected from the arguments at the end of Section 3.3.

After using the standard `SCHED_DEADLINE` policy to schedule the JACK threads, the test was repeated running JACK within an LXC container, so the whole set of JACK real-time threads have been scheduled within a hierarchical reservation. The period of the reservation was set to $2.9025ms$, and runtime varied around the ideal value $Q = \sum_{i=1}^{3} C_i = 638.05\mu s$ (21.98% of real-time bandwidth utilisation), so as to match a real-time utilisation from 15% to 35%. As visible from the HCBS line in Figure 2, there are no xruns for a real-time utilisation of 21%, closely matching theoretical expectations from Eq. (3). This value is significantly below the experimental threshold found above when using the original `SCHED_DEADLINE`.

Therefore, our experimental results confirm the increased ease of use and efficiency in the use of the real-time CBS computational bandwidth when using our proposed control group scheduler, compared to the original `SCHED_DEADLINE` scheduler.

5 CONCLUSIONS

This paper presented a new hierarchical scheduler for Linux, designed to support container-based virtualisation so that it fits in the CSF, that can be conveniently used with LXC containers having multiple virtual CPUs. The presented scheduler is implemented by modifying the real-time control groups mechanism so that the `SCHED_DEADLINE` policy is used underneath to schedule the real-time runqueues of each cgroup.

Experimental results show that a real-time application scheduled within an LXC container under the new scheduler behaves as predicted by existing theoretical CSF analysis. Also, in a realistic use-case involving the use of reservation-based CPU scheduling for guaranteeing CPU time to an audio processing workflow, our control groups scheduler proved to be easier to configure and achieved better results, with the potential of occupying a lower real-time computational bandwidth within the system, for preventing the occurrence of xruns.

In the future, we plan to make a more in-depth experimentation of the proposed scheduler in the context of parallel real-time activities deployed in multi-CPU containers.

REFERENCES

[1] L. Abeni and G. Buttazzo. 1998. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. of the IEEE Real-Time Systems Symp.* Madrid, Spain.
[2] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *OSDI*, Vol. 99. 45–58.
[3] E. Bini, M. Bertogna, and S. Baruah. 2009. Virtual Multiprocessor Platforms: Specification and Use. In *2009 30th IEEE Real-Time Systems Symposium*. 437–446. https://doi.org/10.1109/RTSS.2009.35
[4] E. Bini, G. Buttazzo, and M. Bertogna. 2009. The Multi Supply Function Abstraction for Multiprocessors. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 294–302. https://doi.org/10.1109/RTCSA.2009.39
[5] A. Burmyakov, E. Bini, and E. Tovar. 2014. Compositional multiprocessor scheduling: the GMPR interface. *Real-Time Systems* 50, 3 (01 May 2014), 342–376. https://doi.org/10.1007/s11241-013-9199-8
[6] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari. 2009. Hierarchical Multiprocessor CPU Reservations for the Linux Kernel. In *5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009)*.
[7] T. Cucinotta, G. Anastasi, and L. Abeni. 2009. Respecting Temporal Constraints in Virtualised Services. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 2. https://doi.org/10.1109/COMPSAC.2009.118
[8] A. Easwaran, I. Shin, and I. Lee. 2009. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems* 43, 1 (01 Sep 2009), 25–59.
[9] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky. 2012. Realizing Compositional Scheduling through Virtualization. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. 13–22. https://doi.org/10.1109/RTAS.2012.20
[10] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. 2016. Deadline Scheduling in the Linux Kernel. *Software: Practice and Experience* 46, 6 (2016), 821–839. https://doi.org/10.1002/spe.2335 spe.2335.
[11] G. Lipari and E. Bini. 2010. A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation. In *Proc. of 31st IEEE Real-Time Systems Symposium*. 249–258.
[12] L. T. X. Phan, J. Lee, A. Easwaran, V. Ramaswamy, S. Chen, I. Lee, and O. Sokolsky. 2011. CARTS: A Tool for Compositional Analysis of Real-time Systems. *SIGBED Review* 8, 1 (Mar 2011), 62–63.
[13] G. J Popek and R. P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7 (1974), 412–421.
[14] I. Shin and I. Lee. 2004. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*. 57–67.
[15] I. Shin and I. Lee. 2008. Compositional Real-time Scheduling Framework with Periodic Model. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 30 (May 2008), 30:1–30:39 pages.
[16] J. K. Strosnider, J. P. Lehoczky, and L. Sha. 1995. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.* 44, 1 (1995), 73–91.
[17] S. Xi, J. Wilson, C. Lu, and C. Gill. 2011. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *2011 Ninth ACM International Conference on Embedded Software (EMSOFT)*.
[18] K. Yang and J. H. Anderson. 2016. On the Dominance of Minimum-Parallelism Multiprocessor Supply. In *2016 IEEE Real-Time Systems Symposium (RTSS)*.