Efficient Virtualisation of Real-Time Activities

Luca Abeni
Dipartimento di Scienza e Ingegneria dell'Informazione
University of Trento, Trento, Italy
luca.abeni@disi.unitn.it

Tommaso Cucinotta
ReTis Lab
Scuola Sup. S. Anna, Pisa, Italy
cucinotta@sssup.it

***Abstract*—Reservation-based scheduling has been proved to be an effective solution for serving virtual machines when some kind of real-time guarantees are required. However, the virtualisation mechanism and the algorithm used for implementing CPU reservations might have a large impact on the guarantees provided to tasks running inside the VMs. This paper presents an experimental evaluation of some different solutions, showing the different trade-offs and the advantages of using more advanced scheduling algorithms.**

I. Introduction

In the last few years, one of the most remarkable trends in computer science has been the move towards virtual environments, in which traditional applications do not directly run on bare hardware, but execute inside Virtual Machines (VMs). This allows to host multiple Operating Systems (OSs) or applications (the guests) on the same physical machine (the host) to better exploit its processing power.

With the recent improvements in virtualisation technologies, even running real-time activities inside VMs is beginning to become possible [1], [2], [3]. Of course, since such activities are characterised by temporal constraints that should be respected (often expressed in the form of deadlines), special care is needed to achieve predictable performance when scheduling the VMs. Such a predictability in VM executions can be achieved by modelling the execution of multiple VMs containing real-time tasks as an instance of the *hierarchical scheduling* problem, which has been extensively studied in real-time literature [4], [5], [6], [7], [8]: real-time tasks are scheduled according to a two-level hierarchy, where the host kernel scheduler (scheduling the various VMs) acts as a *global scheduler* (also called *root scheduler*), and the guest kernels' schedulers (running inside the VMs) act as *local (or second-level) schedulers* (also called *leaf schedulers*).

Reservation-based schedulers are often used as root schedulers, because they can be easily modelled using the periodic resource allocation model [7]. This paper presents additional experiences with reservation-based scheduling of VMs, showing how the scheduling algorithm used to implement CPU reservations can affect the system performance, if combined with proper VM implementations. In particular, if it is possible to ensure that the execution time reserved to a VM is consumed only by the real-time tasks running in the guest (and not by the guest kernel, or by the VM code), then it is possible to take advantage of some particular scheduling algorithms such as the Constant Bandwidth Server (CBS) [9].

II. Background and Definitions

The host can be modelled as a set $\{VM^1 \dots VM^k\}$ of VMs, with each VM VM^k composed by a set of N^k real-time tasks: $VM^k = \{\tau_i^k : i = 1, \dots N^k\}$. Each real-time task τ_i^k is a stream of jobs $J_{i,j}^k$, with job $J_{i,j}^k$ arriving (becoming ready for execution) at time $r_{i,j}^k$, executing for a time $c_{i,j}^k$, and finishing at time $f_{i,j}^k$. Jobs are also characterised by absolute deadlines $d_{i,j}^k$, which are respected if $f_{i,j}^k \leq d_{i,j}^k$. In general, $d_{i,j}^k = f_{i,j}^k + D_i^k$ (D_i^k is the relative deadline of task τ_i^k), and if $r_{i,j+1}^k = r_{i,j}^k + P_i^k$ then τ_i^k is said to be periodic with period P_i^k.

As previously mentioned, the global scheduler is in charge of selecting the VM to be executed (so, the global scheduler schedules the VMs $\{VM^k\}$). In some previous works, reservation-based algorithms have been proposed for the global scheduler, because this class of scheduling algorithms allows to easily isolate the performance of each single VM. Informally speaking, a reservation-based scheduler allows one to allocate to each VM VM^k a computation budget Q^k in every reservation period T^k. In this way, the schedulability of the real-time tasks $\{\tau_i^k\}$ executing in VM^k only depends on $\{\tau_i^k\}$, Q^k, and T^k and does not depend on the other VMs running in the host (this is also known as *temporal isolation* property).

The particular reservation algorithm used in this work as a global scheduler is the Constant Bandwidth Server (CBS) [9], which implements CPU reservations building on the top of an Earliest Deadline First (EDF) scheduler. When a VM VM^k is served, the amount of CPU time consumed by the tasks executing inside it is accounted by decreasing a variable q^k called *budget*, and VMs are scheduled based on *scheduling deadlines* d^k assigned by the CBS (the VM having the earliest scheduling deadline is scheduled).

When VM^k is started, q^k and d^k are initialised to 0. When a job $J_{i,j}^k$ arrives at time $r_{i,j}^k$, VM^k becomes ready for execution, and the CBS has to assign a scheduling deadline to it: if $r_{i,j}^k \leq d^k - \frac{q^k}{Q^k} T^k$ (the CBS is said to be *active*), then the latest scheduling deadline d^k (and the latest budget q^k) can be used; if $r_{i,j}^k > d^k - \frac{q^k}{Q^k} T^k$ (the CBS is said to be *idle*), a new scheduling deadline $d^k = r_{i,j}^k + T^k$ is generated and the budget q^k is replenished to Q^k.

When q^k arrives to 0, the CBS serving VM^k is said to be depleted, and according to the original CBS algorithm the budget q^k is immediately replenished to Q^k and the

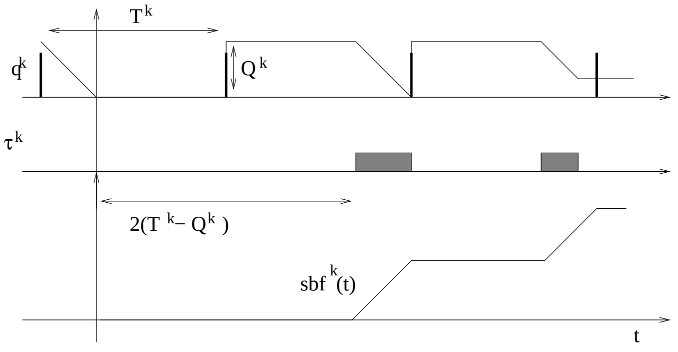


Figure 1. Worst-case arrival pattern for a periodic allocation model, showing the server budget, the task execution, and the supply bound function.

scheduling deadline is postponed to $d^k + T^k$ (so, VM^k can be immediately scheduled). As an alternative, the budget will be replenished and the deadline will be postponed (as above) only at time d^k. Until such a time, VM^k is not schedulable (so, tasks τ_i^k hosted inside VM^k cannot execute until d^k). This is known as *hard reservation behaviour* [10]. When using the CBS to schedule concurrent VMs, the hard behaviour is preferred, because it simplifies the schedulability analysis. Hence, this paper will only consider the hard CBS algorithm.

When using a hard CBS, if $\sum_k \frac{Q^k}{T^k} \leq 1$ (global schedulability condition) then VM^k is guaranteed to execute for Q^k time units every T^k. Hence, the local schedulability of tasks $\{\tau_i^k\}$ can be checked by using time-demand analysis and assuming the so called periodic resource allocation model with the worst-case arrival pattern.

In particular, the worst-case arrival pattern for task τ_i^k (i.e., the arrival pattern that maximises the response time for τ_i^k) happens when the arrival time $r_{i,j}^k$ of a a job $J_{i,j}^k$ of such a task coincides with the CBS depletion time (this means that q^k becomes 0 at time $r_{i,j}^k$). In this case, $J_{i,j}^k$ can have to wait for a time $2(T^k - Q^k)$ before being able to execute (see Figure 1).

Considering this worst case situation, it is possible to define the *supply bound function* $sbf^k(t)$ for VM VM^k, indicating the minimum amount of execution time that VM^k is guaranteed to receive in the time interval $(r_{i,j}^k, r_{i,j}^k + t)$. By looking at Figure 1 it is possible to notice that $sbf^k(t)$ has a large "hole" of size $2(T^k - Q^k)$ at the beginning. The function is formally described by the following equation:

$$sbf^k(t) = \left\lfloor \frac{t - (T^k - Q^k)}{T^k} \right\rfloor Q^k +$$
$$+ \max\{0, t - 2(T^k - Q^k) - \left\lfloor \frac{t - (T^k - Q^k)}{T^k} \right\rfloor T^k\} \qquad (1)$$

The local schedulability of VM^k can then be tested by checking if each task τ_i^k is schedulable, and such a check can be performed by comparing $sbf^k(t)$ with the demanded time for τ_i^k. For example, in case of fixed priority scheduling inside the guest and periodic real-time tasks, the demanded time for task τ_i^k in time interval $(r_{i,j}^k, r_{i,j}^k + t)$ is smaller than $C_i^k + \sum_{P(\tau_h^k) > P(\tau_i^k)} \left\lceil \frac{t}{T_h^k} \right\rceil C_h$ where $C_i^k = \max_j\{c_{i,j}^k\}$ is the Worst Case Execution Time (WCET) of task τ_i^k. Note that this analysis is very pessimistic because of the large "hole" of size $2(T^k - Q^k)$ at the beginning of the supply bound function.

III. Using KVM as Virtualisation Mechanism

The pessimism in traditional hierarchical scheduling analysis (highlighted in Section II) can be reduced in some particular situations. For example, if all the real-time tasks executing in VM^k are periodic, and have offset equal to 0 then it is possible to take advantage of the CBS properties to reduce the pessimism in the analysis. This is possible due to a property of the hard CBS algorithm:

Lemma 1: When scheduling task τ_i with a hard CBS, if τ_i^k is ready for execution at time t, then $d^k - T^k \leq t < d^k$ [10].

Based on the previous property, it is possible to prove the following theorem:

Theorem 2: If the hard CBS algorithm is used for scheduling VM^k and all the real-time tasks in VM^k are released simultaneously ($\forall i, r_{i,0}^k = r_0^k$) and all the tasks periods are integer multiples of T^k ($\forall i, P_i^k \% T^k = 0$), then the maximum time t_0 for which $\forall t < t_0, sbf^k(t) = 0$ is $T^k - Q^k$ (and not $2(T^k - Q^k)$ as in the original analysis).

Proof: Since $sbf^k(t)$ represents the minimum execution time received by VM^k in an interval $(r_{i,j}^k, r_{i,j}^k + t)$, t_0 can be larger than $T^k - Q^k$ only if a job $J_{i,j}^k$ arrives when the budget q^k is 0 (hence, it must wait until d^k to replenish the budget; then, an additional delay $T^k - Q^k$ can be added due to interference from higher priority CBSs). Hence, to prove that $t_0 \leq T^k - Q^k$ it is sufficient to prove that when a new job $J_{i,j}^k$ arrives, d^k is always equal to $r_{i,j}^k + T^k$ (this implies that $q^k = Q^k$).

If $J_{i,j}^k$ arrives when the CBS is idle, then $d^k = r_{i,j}^k + T^k$ by definition.

If, instead, $J_{i,j}^k$ arrives when the CBS is active, then $d^k - T^k \leq r_{i,j}^k < d^k$ (see Lemma 1). Moreover, d^k is equal to $r_{i,j0}^k + zT^k$, where z is an integer number and $r_{i,j0}^k$ is the arrival time of the latest job arrived when the CBS was idle. Since $P_i^k \% T^k = 0$, $r_{i,j}^k = r_0^k + nT^k$ and $r_{i,j0}^k = r_0^k + mT^k$, with n and m integer numbers. Hence, $d^k - r_{i,j}^k = r_{i,j0}^k + zT^k - r_{i,j}^k = r_0^k + mT^k + zT^k - (r_0^k + nT^k) = (m + z - n)T^k$ is an integer multiple of T^k. Combining $d^k - T^k \leq r_{i,j}^k < d^k$ and $d^k - r_{i,j}^k = (m + z - n)T^k \Rightarrow d^k - (m + z - n)T^k = r_{i,j}^k$ the only possible result is $d^k = r_{i,j}^k + T^k$. ■

Note that Theorem 2 is only valid if the hard CBS algorithm is used for VM scheduling (it relies on the fact that a CBS assigns scheduling deadlines as $r_{i,j}^k + T^k$ when the server is idle), and when the reservation budget is consumed only by the real-time tasks τ_i^k running inside the VM. In particular, if the reservation budget is consumed by the host kernel, or by the VM code (to emulate the various hardware devices), or by non real-time tasks running inside the VM, then the property cannot be applied and the more pessimistic analysis must be used.

The hierarchical scheduling approach described above has been first tested on a full system emulator, that emulates all the hardware details of a real machine. The CPU can be virtualised by using the KVM technology[1], which uses a Linux kernel module for safely executing the guest code on the host CPU (without having to simulate all the CPU instruction). Implementing the virtual hardware devices is more complex, and requires to emulate in software the behaviour of each device.

Recent versions of the QEMU emulator[2] integrate KVM technology (the final goal of the developers is to integrate the whole KVM codebase in QEMU). For the first set of experiments presented in this paper, QEMU with KVM enabled (referred as QEMU/KVM) has been used. QEMU is based on a multithreaded architecture, using a pool of threads to emulate POSIX asynchronous Input/Output (AIO). In addition, the main QEMU thread is responsible for emulating the hardware devices, and an additional thread is created for each virtual CPU (this is called VCPU thread). All the code of the tasks running in the guest is executed inside the VCPU threads (remember that thanks to the KVM kernel module the guest code can be directly executed in the host).

In order to apply the previously described hierarchical scheduling analysis to a KVM-based VM, the CPU reservation should be attached to the VCPU thread, not to all the threads of the VM. However, in order to control the I/O performance of the VM it might be useful to attach additional CPU reservations to the other QEMU/KVM threads, similarly to what is done in the cooperative scheduling approach [11].

A simple test from [2] (based on two applications $\Gamma^1 = \{(30, 150), (50, 200)\}$ and $\Gamma^2 = \{(30, 120), (40, 240)\}$) is reported as an example (see the original paper for all the details). All the experiments presented in this paper use the CBS implementation provided by the Irmos kernel [12] to schedule the VMs, and the Debian stable distribution installed in all the VMs.

When using hierarchical scheduling and CBS [9] as a root scheduler, by applying Theorem 2 it can be seen that Γ^1 can respect its temporal constraints if it is scheduled through a server $CBS^1 = (27, 50)$, and Γ^2 can respect its temporal constraints if it is scheduled through $CBS^2 = (50, 120)$. Hence, if the overhead of the VMs and of the guest kernel is not considered, the two applications can correctly respect their temporal constraints when executed in two virtual machines VM^1 and VM^2 served by $CBS^1 = (27, 50)$ and $CBS^2 = (50, 120)$. In fact, the original paper shows that when running the tasks in two KVM instances scheduled by two CBSs $CBS'^1 = (28, 50)$ and $CBS'^2 = (52, 120)$ all the deadlines are respected (the maximum budgets have been slightly increased to $Q'^1 = 28ms$ and $Q'^2 = 52ms$ to account for the overheads caused by KVM and by the host kernel).

The original experiments were performed using an old version of KVM and the CBS scheduler provided by AQuoSA [13]. When repeating them with a recent version of QEMU/KVM, it was possible to notice that the execution of the VM code now consumes a larger and less predictable amount of time (this is probably due to the fact that QEMU/KVM now provides a larger set of features and emulates more complex hardware devices). As a result, the probability distributions of the response times had some long tails causing some sporadic deadline violations. This problem can be solved by scheduling only the relevant QEMU/KVM thread (the VCPU thread) through CBS, as explained above: the latest version of QEMU/KVM allows to execute the guest code in a dedicated thread, and if only such a VCPU thread is attached to the CBS then the VM code cannot consume the CBS budget and the VM execution returns to be predictable. When doing this, the response times of the tasks resulted to be similar to the expected ones (the overhead caused by the guest kernel and by the VM code did not affect the results too much).

[1]http://www.linux-kvm.org
[2]http://www.qemu.org

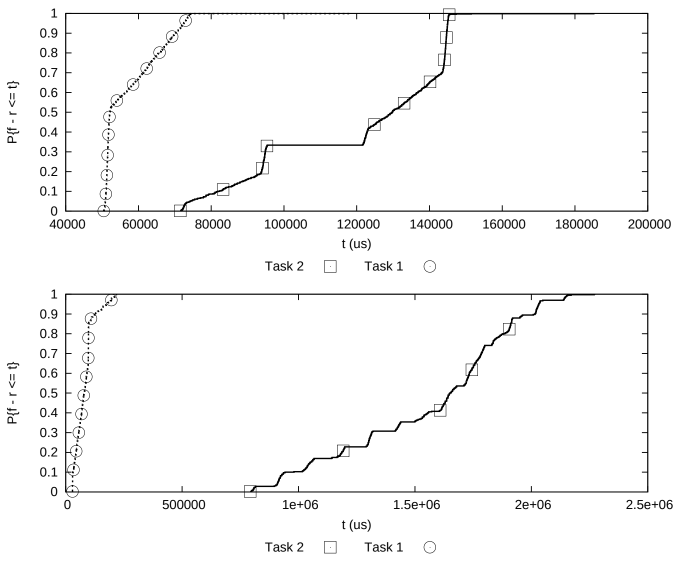
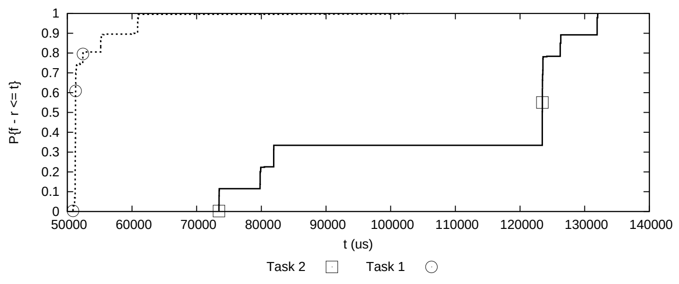


Figure 2. CDFs of the response times of tasks in VM^1 and VM^2 when using correctly dimensioned CBSs, with some non real-time background load.

However, Theorem 2 (which has been used for dimensioning the two reservations) cannot be applied if the VMs contain non real-time tasks that consume the reservations budgets. Hence, when VM^1 and VM^2 host some background (non real-time) tasks consuming the server's budget, the real-time performance of the VMs are out of control. For example, Figure 2 shows the Cumulative Distribution Functions (CDFs) of the response times $f_{i,j} - r_{i,j}$ of the 4 tasks when a CPU hungry non real-time task is executed in background in the VMs. Note that the presence of non real-time tasks in the VM is influencing the predictability of the real-time tasks execution.

IV. Container-Based Virtualisation

The KVM technology used in [2] and Section III emulates real hardware (CPU and I/O devices), based on the real resources provided by the host. In this way, KVM can run an arbitrary guest operating systems without modifications because the code running inside the VM is not aware of the



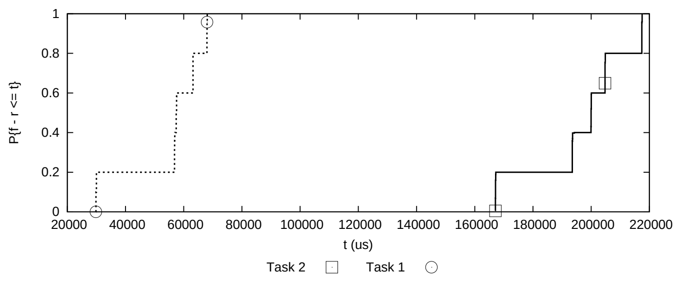


Figure 3. CDFs of the response times of tasks in VM^1 and VM^2 when executed in LXC VMs using correctly dimensioned CBSs with background load. The two VMs are executed simultaneously

fact that it is not running on real hardware.

On the other hand, the OS-level virtualisation aims at virtualising the OS kernel, and not the whole real hardware (whereas KVM provides hardware virtualisation). This approach is designed to provide the required isolation and security between different VMs by using one single OS kernel (the host kernel) and by virtualising the services it provides. In this way, it is possible to run multiple distributions based on the same OS kernel (or multiple applications) on the same server (of course, every application or OS distribution will be isolated from the others, having the impression to be the only one running on the kernel). Such an isolation can be implemented by isolating the various kernel resources inside *containers*, and associating each VM to a container. This approach is also known as container-based virtualisation.

When using container-based virtualisation, the host kernel is responsible for scheduling all of the tasks contained in the various VMs, hence it can directly know if a guest is executing a real-time task (using the POSIX `SCHED_FIFO` or `SCHED_RR` policy) or not. As a result, it is quite easy to schedule only real-time tasks through the CBS serving the VM.

All the experiments presented in the previous section have been repeated using a VM based on the standard container-based virtualisation technology which is integrated in the Linux kernel, LXC[3]. The same Irmos kernel and the same Debian stable installation used in the previous experiments has been used[4]. When using LXC, the response times resulted to be perfectly deterministic even in presence of a 100% CPU load (non real-time tasks were not able to affect the response times of real-time tasks). Even when running multiple VMs concurrently with some other tasks scheduled through the CBS algorithm, the response times are under control as shown in Figure 3 (obviously, this is true until the global schedulability condition $\sum_k Q^k/T^k \leq 1$ is respected). By comparing this figure with Figure 2 it is immediately possible to notice the improvements achieved by using LXC and the CBS algorithm.

[3]http://lxc.sf.net

[4]The disk images used for QEMU have been mounted using a loop device and used for the LXC root filesystem.

V. Conclusions

This paper presented some experiments with real-time tasks running in virtual machines scheduled through a reservation-based algorithm. The results show that the VM technology and the scheduling algorithm can affect the real-time performance of the guests. In particular, if the CBS algorithm is used together with a container-based virtual machine (such as LXC) then it can be possible to use a less pessimistic analysis to dimension the VM scheduling parameters. As a future work, the CBS analysis provided in Section III will be extended and improved, and alternative virtualisation technologies will be tested. Moreover, experiments will be performed in order to control the I/O performance by attaching the QEMU I/O threads to additional reservations.

References

[1] J. Kiszka, "Towards linux as a real-time hypervisor," in *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, September 2009.

[2] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constraints in virtualised services," in *Proceedings of the 2^{nd} IEEE International Workshop on Real-Time Service-Oriented Architecture and Applications (RTSOAA 2009)*, Seattle, Washington, July 2009.

[3] S. Xi, J. Wilson, C. Lu, and C. Gil, "Rt-xen: Towards real-time hierarchical scheduling in xen," in *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, Taipei, Taiwan, October 2011.

[4] A. K. Mok and X. A. Feng, "Towards compositionality in real-time resource partitioning based on regularity bounds," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 2001.

[5] A. K. Mok, X. A. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings of the Seventh Real-Time Technology and Applications Symposium*, 2001.

[6] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *Journal of Embedded Computing*, vol. 1, no. 2, 2004.

[7] I. Shih and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. 24^{th} Real-Time Systems Symposium*, 2003.

[8] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *Proceedings of the 25^{th} IEEE International Real-Time Systems Symposium*, December 2004, pp. 57–67.

[9] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[10] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari, "Resource reservations for general purpose applications," *IEEE Transactions on Industrial Informatics*, 2009.

[11] S. Saewong and R. R. Rajkumar, "Cooperative scheduling of multiple resources," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS99)*. Phoenix, AZ: IEEE, 1999, pp. 90–101.

[12] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, "Hierarchical multiprocessor CPU reservations for the linux kernel," in *Proceedings of the 5^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009)*, Dublin, Ireland, June 2009.

[13] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA — adaptive quality of service architecture," *Software – Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.