Towards Coordinated Task Scheduling in Virtualized Systems

Jérémy Fanguède, Alexander Spyridakis and Daniel Raho
Virtual Open Systems
Grenoble - France
Email: {j.fanguede, a.spyridakis, s.raho}@virtualopensystems.com

***Abstract*—Task scheduling is one of the key subsystems of an operating system. Generally, by providing fairness in terms of processor time allocated to tasks, the task scheduler can guarantee a low latency and high responsiveness to applications. In this paper, we demonstrate that some problems can occur in virtualized environments, in relation to standard task scheduler implementations and the way that tasks and virtual cores are scheduled. More precisely, there is a need to implement a communication channel between virtualized schedulers in the virtual machines and the host task scheduler, particularly when full-virtualization techniques are used, which could lead to latency issues and loss of responsiveness in virtual machines, especially when processors execute excessive workloads. After having analyzed the potential problems in virtual machines, experiments were done with real world and benchmarking applications. For testing, a Linux-based system and two different task schedulers were used, with a benchmark suite especially designed for virtualized environments where application responsiveness and latency can be measured. As an experimental platform, an ARM embedded system was used; this system is almost equivalent to general-purpose systems in terms of task scheduling.**

***Keywords*—KVM/ARM; embedded virtualization; coordinated scheduling; embedded systems; task scheduling; CFS; BFS**

I. INTRODUCTION

Virtualization technology offers a way to increase efficiency and adaptability both in general purpose and embedded systems, but to get an efficient virtualization solution, latency of virtual machines and responsiveness of applications should be guaranteed at a reasonable level. For instance, an interactive application launched in a virtual machine should not have much worse performance in terms of responsiveness and latency than one executed in a host machine in the same conditions.

In previous work, we have already experimented with this objective in mind, specifically for storage-I/O, and the implementation of Virtual-BFQ [1] [2], a Linux I/O scheduler based on the BFQ scheduler [3]. The work described in this paper, instead targets process scheduling, so that it could be used as a complementary approach.

In this paper, we provide the following contributions:

A. Contributions of this paper

We highlight that in virtualized environments there are latency problems with task scheduling, where a missing link between the guest and the host scheduler can affect performance negatively. In fact, there is a need to implement a coordinated communication channel between schedulers in virtual machines and the host task scheduler. As a consequence, latency of a guest operating system is higher, especially in a system with many CPU-bound tasks. This results in degraded responsiveness of applications in virtual machines, compared to similar conditions for non-virtualized systems. To show this problem, through experimentation, we use two different Linux task schedulers.

Then, experimental results are reported, these results confirm that, in virtualized environments, when a process requires a high portion of the processor's time in both the guest and host system, the latency and the responsiveness of the guest application is not guaranteed.

An ARM-based embedded system was used to run the experiments, Kernel-based Virtual Machine (KVM) and Quick EMUlator (QEMU) is the virtualization solution used, which is among the most popular solutions in embedded virtualization.

B. Organization of this paper

The paper is organized as follows. In section II, a description of the two task schedulers used is provided. Then in Section III latency problems and the lack of responsiveness is highlighted. After describing the benchmark suite and the experimentation methods in section IV, the results are reported in Section V. Finally, in SectionVI, possible solutions are detailed in order to solve the issue highlighted.

II. LINUX TASK SCHEDULERS

The task scheduler, also named process or CPU scheduler, is the part of an operating system that decides which task runs when, and on which core. The job of a scheduler is to share the CPU time between processes that require CPU resources, to pick a suitable task to run next if required, and to balance processes between the different CPUs in a multi-core system.

Two Linux task schedulers were used, CFS [4], which stands for *Completely Fair Scheduler* and is the default scheduler of the Linux kernel, and BFS [5], which stands for *Brain Fuck Scheduler* and is a popular alternative.

By default, Linux can handle real-time and non real-time policies, which are implemented by the selected scheduler. Both CFS and BFS schedulers implement their own non real-time and share the same real-time policies. By extension, with the term CFS or BFS we refer to both scheduling policies of these schedulers, as well as the whole of their implementation.

 ISBN: 978-1-61208-419-0

BFS, which is not part of the Linux mainline kernel [6], could be considered as an alternative, it is designed for desktop interactivity on machines with few cores [5], and its source code has a smaller footprint and is by design simpler. For these reasons, this scheduler was also selected for investigation.

A. The Completely Fair Scheduler

The default Linux kernel scheduler, named *Completely Fair Scheduler* [4], is modular and permits to use different policies for different tasks. Linux has two main types of scheduling policies: a real-time one for real-time task and a normal one named *fair policy* for all other tasks.

Among the real time scheduling, Linux distinguishes three policies: SCHED_FIFO, a first-in, first-out policy; SCHED_RR, a round robin policy; and SCHED_DEADLINE, a policy implementing the earliest deadline first algorithm (since kernel v3.14).

And within the fair scheduling policies: SCHED_NORMAL, the default Linux time-sharing policy, and SCHED_BATCH, a policy for "batch" processes.

Linux defines the static priority of a task by a value, which ranges from 0 to 99, and the real-time scheduling class uses values from 1 (lowest priority) to 99 (highest priority). Processes using the fair scheduling class have necessarily a static priority of 0. In order to determine which thread (or process) should be run next, the Linux scheduler maintains a list of runnable processes for each possible static priority, and it selects the head of the list with the highest static priority. In other words, a thread, with a higher static priority than the current running thread which becomes runnable, will necessarily preempt the current process.

For the fair scheduling class, the kernel uses a priority called dynamic priority, which from a user's point of view is also better known as the *nice value*, and it ranges from -20 (highest priority) to +19.

CFS is used as the default Linux scheduler since kernel version v2.6.23, it replaced the old scheduler: *O(1)*. And implements a completely fair algorithm (hence the name). The algorithm is based on the concept of an ideal multi-tasking processor. With such a processor, each runnable task would run at the same time, sharing the processor power. Of course this behavior is not possible, but an equivalent behavior, would be to run each runnable task for an infinitesimal amount of time with full processing power. Due to task switching cost, CFS, only approximates this behavior.

For that purpose, CFS stores the runtime value of each task in a variable called *vruntime* (stands for virtual runtime) and tries to keep all *vruntime* values the closer to each other. So the runnable task which has the lower *vruntime* value is chosen to be the next task to run. The priority of a task (the dynamic priority, i.e., the nice value) influences the way vruntime is increased.

To handle interactive tasks, CFS doesn't use complex heuristics. In fact, the concept of fair scheduling is enough to maximize interface performance. For example, consider a processor-bound task (e.g., an encryption calculation, a video encoder, etc.) and a I/O-bound task (e.g., a terminal, a text editor, etc.), which will be the interactive task. In that situation, the scheduler should give to the interactive task a larger share of the processor time to enhance the user experience. In fact, this is what CFS will do: CFS wants to be fair, so each time the interactive task become runnable, CFS will see that this task consumed significantly less processor time than the CPU-bound task. So the interactive task will preempt the other, and will be executed until its runtime reaches the value of the processor-bound task or be blocked from an I/O request.

B. BFS - The Alternative

BFS is an alternative to CFS, it was written by *Con Kolivas*. It is not in the mainline kernel and is available as source code patches [6].

BFS focuses on a simplistic design (about 2.5 times fewer lines of code than CFS) and aims for excellent desktop interactivity and responsiveness on personal computers with a reasonable amount of cores [5]. It uses a single work-queue, *O(n)* look-up for all cores unlike CFS, and implements the *earliest eligible virtual deadline first* algorithm for non real-time policies.

BFS, like CFS, provides real-time task policies: SCHED_FIFO and SCHED_RR, and also two others policies for normal tasks: SCHED_ISO and SCHED_IDLEPRIO. The first, SCHED_ISO (for isochronous) is designed to provide "near real-time" performance to unprivileged users. And SCHED_IDLEPRIO scheduling policy can be used to run tasks only when the CPU would be idle otherwise.

The design of BFS makes it efficient when the number of running processes is small (inferior than the number of CPUs), which is normally, according to its author [5], a common use case for a desktop computer.

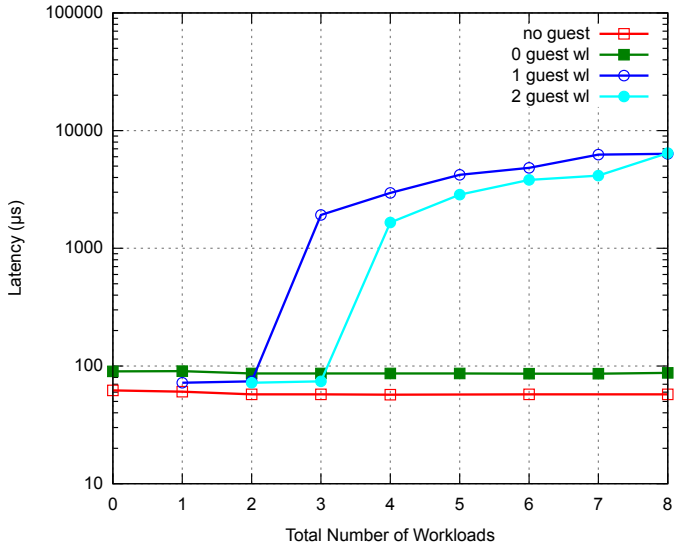
III. Potential Problems in Virtualized Environments

In a virtualized environment a guest system is seen, from the host scheduler, as just one, or more additional jobs to schedule, without any awareness from the host of the fine-grained requirements of the corresponding guest scheduler. For example, a new spawned task in the guest system could be scheduled in a different way by the guest scheduler, but this information is not visible on the host side. Under certain conditions, that could lead to undesired behavior.

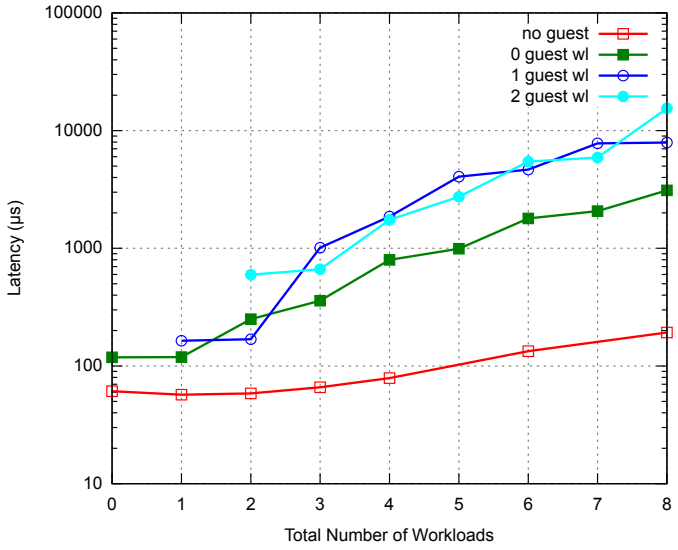
To highlight the problem we can consider a system, with two physical CPUs and a guest with one virtual CPU. Two CPU-bound workloads are launched in the host (one per CPU) and one in the guest (one per virtual-CPU). In this situation, the task scheduler will share fairly the processor time between the vCPU thread (which runs a workload) and the two workloads in the host, since these three tasks are quite similar in terms of CPU time demand.

When an interactive task is started in the guest system, the guest scheduler will *detect* this new task and assign a substantial amount of the vCPU time compared to the work-load running in the same guest. On the host side though, the scheduler sees only three processes that request a large amount of CPU time for only two CPUs. So, the host scheduler has absolutely no reasons to privilege the vCPU thread compared

 ISBN: 978-1-61208-419-0



(a) CFS



(b) BFS

Figure 1. Latency results

to other processes (workloads). Additionally, the latency of this interactive task will probably be higher than in a host system with the same number of workloads (aside from the constant overhead of KVM/QEMU). This problem persists for whatever value the priority of the interactive task in the guest is set to (could be a real-time one), since the priorities and policies are not made aware to the host system.

IV. Experiment Method and Benchmark Suite

To highlight the problem described above, we set-up a benchmark suite in order to measure, in particular, the latency of the system. We use the tool, *cyclictest*, which is usually used to measure latency on a *Real Time Linux* (i.e. patched with *rt* patches) [7]. Generally, *cyclictest* is used to measure the latency of real-time thread/process (schedule with SCHED_FIFO or SCHED_RR), but we can also use it with *normal* (SCHED_NORMAL) threads. For each latency measurement *cyclictest* is run twice, each one with a 100000 loop, which means that the latency provided by the benchmark is the average of 200 thousands measurements. The following command line is used "cyclictest -q -n -l 100000 -h 5000" to generate the results, and the latency histogram is also retrieved (*-h* option) in order to analyze in more detail.

The second kind of benchmark measures the *start-up time* of an application. We simply measure how long it takes from when an application is launched to when an application is ready. This benchmark gives an idea of the *responsiveness* of an application. The start-up time is measured with hot caches, to avoid any I/O perturbations. For each configuration (i.e. number of workload in the host and guest), 100 measurement iterations are performed, and the average, as well as the standard deviation are retrieved.

As workload, we use a simple program that does an *infinite loop*, and therefore has a very low memory footprint.

V. Experimental Results

We executed our experiments on a Samsung Chromebook equipped with an ARMv7-A Cortex-A15 processor (dual-core, 1.7 GHz) and 2 GB of RAM. Both the host and the guest run upstream Linux v3.17 with the *PREEMPT* configuration option enabled.

A. Latency

In order to measure latency, we used the *cyclictest* tool and the number of workloads is kept the same as in the start-up time test. The result of this experiment is shown in Figure 1, where latency is measured in microseconds and represented in a logarithmic scale on axis Y.

For the host and guest system we employ up to 8 and 2 workloads respectively. Axis X corresponds to the total number of workloads, i.e., host plus guest workloads. The output of the results are four different curves:

no guest: No virtual machine, serves as reference, the application is launched in the host

N guest wl: With N workloads in the guest, the application is launched in the guest. With N ranges from **0** to **2**.

We can notice that, with the CFS scheduler (Figure 1a), as soon as there are more workloads than physical cores (total of two cores in the system, critical curves are *1 guest wl* and *2 guest wl*) and with at least one workload in the guest, latency increases significantly. By adding more workloads, this behavior persists until values are not suitable for interactive usage. This kind of result confirms the issue highlighted in Section III, where an interactive application in a virtualized system can have an extremely high latency.

One can also point out that the latency is better with *2 guest workloads* than with *1 guest workload* when the total number of workloads is high. This behavior is perfectly explainable

 ISBN: 978-1-61208-419-0

due to the difference in the number of workloads in the host. For instance, in the specific case of 4 total workloads, when we have *1 guest workload* the host system *sees* four main processes requesting a high amount of CPU time for only two CPUs, but when we have *2 guest workloads*, there are only three processes that still share two CPUs. In the latter case, the process corresponding to the vCPU has more CPU time: this could lead, depending on the efficiency of the guest scheduler, to a better latency compared to the former case.

With BFS (Figure 1b), results are less obvious, but we can still notice the difference between virtualized and normal environments, and between the curves of 1 or 2 guest workloads and the curve of 0 guest workloads.

Although our objective is not to purely compare the two schedulers, which has already been done [8], we can remark that even with no virtual machines (curve *no guest*), latency with BFS increases steadily, contrary to CFS. This is probably due to the fact that BFS is not designed to be efficient when the number of running tasks is higher than the number of physical cores [5].

We can also analyze the histogram provided by the *cyclictest* results to compare the distribution of latency. Figure 2 shows the two latency histograms on a virtual machine without any workload. We can notice that even if the average value is slightly lower with CFS, the BFS case exposes more converged values with a lower maximum.

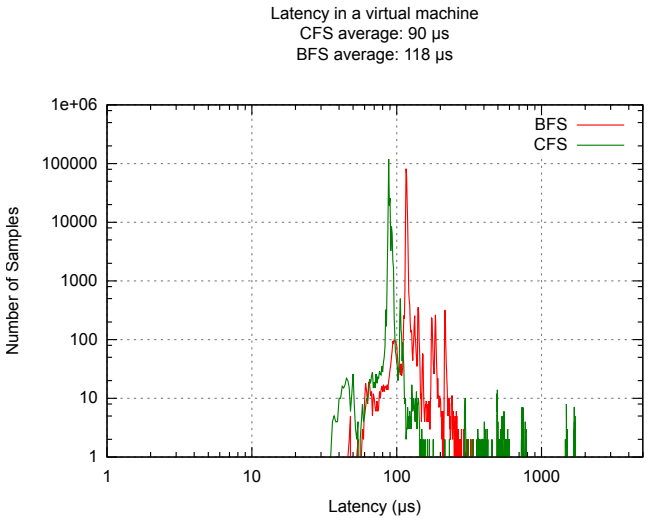


Figure 2. Guest Latency compared between BFS and CFS

In Figure 3, two cases are compared for CFS latency measured in a virtual machine. Both test cases have the same amount of CPU-bound workloads, but distributed in a different manner. In the first case all workloads reside in the host, while in the second, one of the workloads is reserved for the guest. Although the distribution of samples for low latency is quite similar for both cases, in the case where one of the workloads is in the guest, we still observe a significant amount of samples in the range of 200 to 5000 μs. This is different from the first case, where almost all samples are around the 100 μs mark.

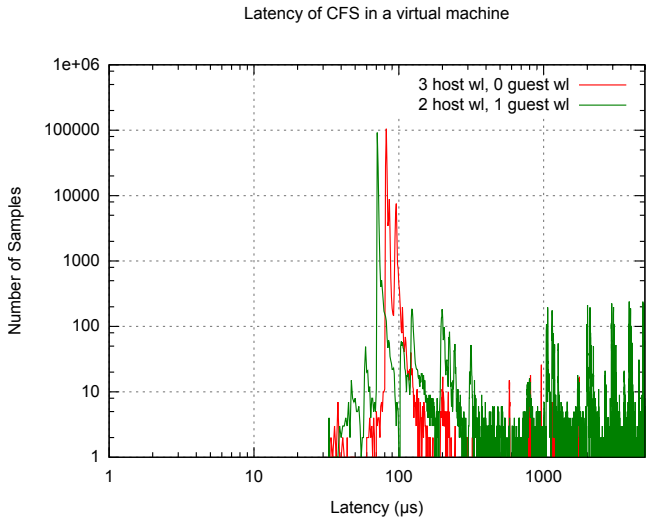


Figure 3. CFS latency in a virtual machine, compared between 3 host workloads and 2 host and 1 guest workloads

B. Start-up Time

Next, we measure the start-up time of an application. We choose the *xterm* application because its start-up time can be easily measured. In addition, this application was also selected to measure performance of the BFQ and Virtual-BFQ I/O scheduler [1] [2] [3].

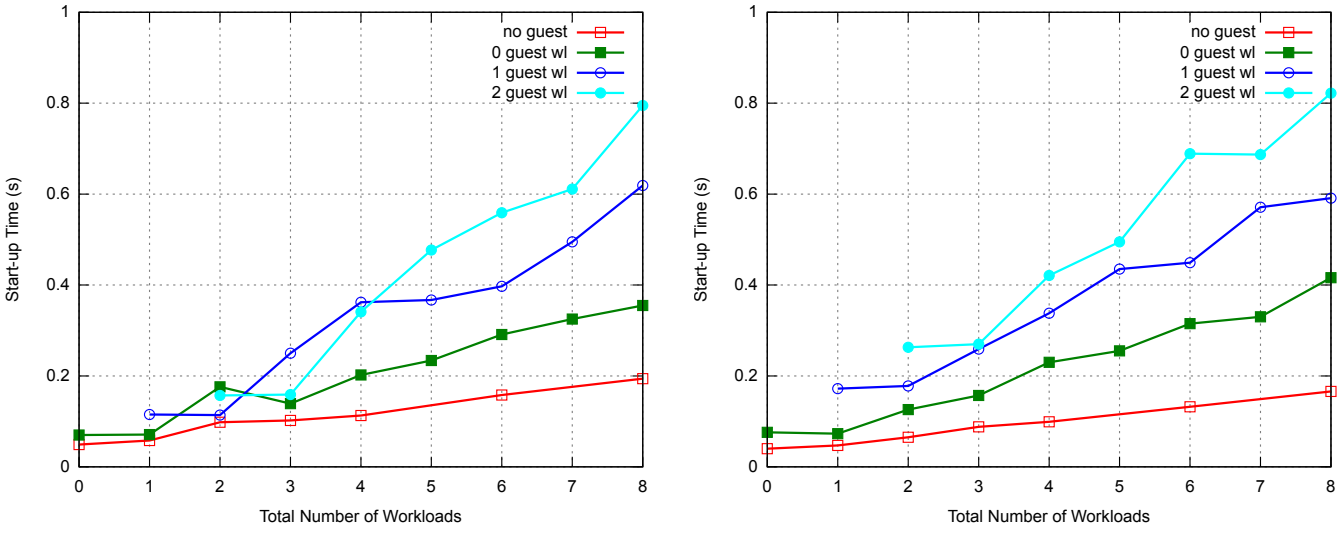
As we can see in this Figure 4a, which represents the startup time measured with the CFS scheduler, the curve corresponding to a measurement in the host (*no guest*) has a slightly positive constant slope. This increase is not unexpected because CFS tries to guarantee only fairness: an increase in the number of CPU-bound can negatively affect the start-up time of a new application. Curve *0 guest wl*, corresponds to the case in which there is no workload in the guest, but only in the host. We can see that this curve almost follows curve *no guest*, where a constant overhead is observed.

In view of the problem highlighted above, the critical scenarios are the ones corresponding to the curves *1 guest wl* and *2 guest wl*, more particularly when the number of workloads in the host is equal or greater than number of physical cores (in our case 2). In fact, when vCPU threads are allowed to use all available cores, the results are acceptable as the start-up time remains quite low (case *1 guest wl* with a total workload of *1* and *2*, and with *2 guest wl* with *2* and *3* total workloads). To summarize our test case results, when the number of workloads in the host is higher than two, the start-up time increases significantly.

With the BFS scheduler (Figure 4b), although the appearance of the curves seems quite different, we have the same behavior: higher start-up times when there are too many workloads.

To sum up, our results are coherent both for start-up times as well as latency. Moreover they clearly prove that, in scenario where a workload is present in both guest and host, the responsiveness of an application in the guest can not be guaranteed.

 ISBN: 978-1-61208-419-0



(a) CFS (b) BFS

Figure 4. Start-up time results

VI. SOLUTIONS

Some solutions, can be developed to address the above problem. Two possible solutions are described below, a simple static prioritizing, and a coordinated solution that enables a communication between scheduler of the host and guests systems.

A. *Static prioritizing*

A straightforward solution could be a static prioritizing scheme, by simply increasing the priority of the QEMU vCPU threads, or by changing the scheduling policy to a real-time one. This solution will allow QEMU to not be interfered by other tasks in the host system (if there are no other real-time threads). This method will result in a better latency, in particular a reduction of the maximum latency [9]. With this solution though, the guest is always privileged even when it doesn't execute an interactive program. This solution can be useful for simple use cases, i.e, when a guest system which executes soft real-time applications needs to be prioritized compared to other guests or applications.

B. *Coordinated scheduling*

Instead of prioritizing QEMU threads statically, another solution could be to *boost* these threads only when it is necessary, i.e, temporary increasing the priority or changing the scheduler policy, when the guest system requests it. It is a sort of dynamic prioritizing with a coordinated scheduling mechanism: the guest kernel detects when it needs higher priority, and informs the host system about it. This *co-scheduling* mechanism was already implemented successfully for Virtual-BFQ [2], therefore the communication mechanism could be equivalent to the one developed for this storage I/O scheduler.

This type of solution has already been implemented and evaluated, especially to make KVM a real-time hypervisor [10] [11]. Such attempts mainly focused to run a real-time Linux OS as a guest, thus when a guest executes a real-time thread it informs the host of its current scheduling policy and priority, the host system then has to pass on this policy and priority to the affected QEMU thread.

In order to extend this coordinated scheduling mechanism also to non real-time applications, a mechanism to detect interactive application in the guest system is needed. Heuristic algorithms have to be added for this purpose.

The communication mechanism between the host and guest scheduler, is a crucial part, it needs to be fast or at least not too frequent. The solution chosen in the Virtual-BFQ [2] I/O scheduler is to use, a special ARM instruction, *HVC*, that results in a hypervisor *trap*. Moreover, the cost of calling this instruction, around 2000 CPU cycles, is not very expensive and can fit the requirement of a task scheduling coordinated mechanism.

VII. CONCLUSION AND FUTURE WORKS

In virtualized environments, we highlighted that the host task scheduler could fail to achieve full system low latency, and thus to preserve responsiveness when the system is loaded with CPU-bound programs in certain conditions. The behavior of an interactive application inside a guest will be masked by other processes requiring a lot of CPU time in the host, and the attempts of the guest scheduler to enhance the responsiveness of this application may be useless. This issue mostly occurs when the number of CPU-bound processes is higher than the number of physical cores.

We are currently designing a solution which implements a coordinated scheduling mechanism between the host and guests schedulers, and we have promising results. The target of this approach is ARM embedded systems with the KVM hypervisor. Besides, we also plan to extend tests with more complex scenarios including more than one virtual machines.

 ISBN: 978-1-61208-419-0

ACKNOWLEDGMENT

This research work has been supported by the Seventh Framework Programme (FP7/2007-2013) of the European Community under the grant agreement no. 610640 for the DREAMS project.

REFERENCES

[1] A. Spyridakis, and D. Raho. "On Application Responsiveness and Storage Latency in Virtualized Environments," CLOUD COMPUTING 2014, The Fifth International Conference on Cloud Computing, GRIDs, and Virtualization, 2014, pp. 26-30.

[2] A. Spyridakis, D. Raho, and J. Fanguède. "Virtual-BFQ: A Coordinated Scheduler to Minimize Storage Latency and Improve Application Responsiveness in Virtualized Systems," International Journal on Advances in Software, vol 7 no 3 & 4, 2014, pp. 642-652.

[3] P. Valente and M. Andreolini, "Improving application responsiveness with the BFQ disk I/O scheduler," Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR'12), June 2012, p. 6.

[4] "CFS scheduler," [retrieved: June 2014]. Available: http://lwn.net/Articles/230501/

[5] C. Kolivas, "BFS FAQ," [retrieved: June 2014]. Available: http://ck.kolivas.org/patches/bfs/bfs-faq.txt.

[6] C. Kolivas, "BFS Patches," [retrieved: June 2014]. Available: http://ck.kolivas.org/patches/bfs/3.0/.

[7] "Cyclictest," [retrieved: June 2014]. Available: https://rt.wiki.kernel.org/index.php/Cyclictest

[8] T. Groves, J. Knockel, and E. Schulte. "Bfs vs. cfs scheduler comparison," 2009.

[9] R. Ma, F. Zhou, E. Zhu, and H. Guan, "Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System," Journal of Information Science and Engineering 29.5, 2013, pp. 1021-1035.

[10] J. Kiszka, "Towards linux as a real-time hypervisor," In Proceedings of the 11th Real-Time Linux Workshop, 2009, pp. 215-224.

[11] Aichouch, Mehdi, J-C. Prevotet, and Fabienne Nouvel. "Evaluation of an RTOS on top of a hosted virtual machine system," In Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on. IEEE, 2013, pp. 290-297.

 ISBN: 978-1-61208-419-0