Coordinated Task Scheduling in Virtualized Systems: Evaluation and Implementation Details

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 low latency and high responsiveness to applications. In this paper, we demonstrate that specific problems can occur in virtualized environments, where virtual core scheduling on the host can negatively affect process scheduling in the guest. More precisely, there is a need to implement a communication channel between the host and guest task scheduler, particularly when full-virtualization techniques are used, in order to avoid 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 performed with real world and benchmarking applications. In this work we detail possible solutions to solve the issue previously highlighted, and describe the proposed implementation, which is based on a coordinated scheduling mechanism between the host and guest systems. For testing, an embedded ARMv7 Linux-based platform and two different task schedulers were used, with a benchmark suite specifically designed for virtualized environments, with which application responsiveness and latency are measured and compared.**

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

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 showed that latency issues can occur with task schedulers under some conditions [1]. We have already experimented with this objective in mind, for storage-I/O, which led us to the implementation of Virtual-BFQ [2][3], a Linux I/O scheduler based on the Budget Fair Queuing (BFQ) scheduler [4]. The work described in this paper, instead targets process scheduling, so that it could be used as a complementary approach. The solution described in this paper is based on a coordinated scheduling mechanism. This type of solution has already been implemented specifically to make real-time hypervisors [5][6], while in this work we extend coordinated scheduling also to non real-time tasks.

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 can be 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.

A solution, described in this paper, based on a coordinated mechanism, solves the problem highlighted previously, it is based on the default Linux scheduler, CFS, which stands for *Completely Fair Scheduler*, the implementation is described in detail. Experimental results are reported for this extended version of CFS, which includes the coordinated scheduling mechanism.

An ARM-based embedded system was used to run the experiments, it aims to be representative of modern embedded systems and consumer devices, which have relatively small amount of CPU resources. Finally, the virtualization platform selected for this implementation is Kernel-based Virtual Machine (KVM) of Linux together with Quick EMUlator (QEMU), which are 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. In Section VI, possible solutions are detailed in order to solve the issue highlighted. Then, in Section VII the description of a solution based on a coordinated scheduling mechanism for the CFS scheduler is provided. Finally, in Section VIII we report our results for the same experiments,

but with the coordinated scheduling mechanism developed for CFS.

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 [7], which stands for *Completely Fair Scheduler* and is the default scheduler of the Linux kernel, and BFS [8] (see the acronym in [8]), which 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 the included scheduling policies of these schedulers, as well as the entirety of their implementation.

BFS, which is not part of the Linux mainline kernel [9], could be considered as an alternative, it is designed for desktop interactivity on machines with few cores [8], and its source code has a smaller footprint and is by design simpler. For these reasons, BFS was also selected in the experimental results as a comparison to CFS, in case where different behavior is observed.

A. The Completely Fair Scheduler

The default Linux kernel scheduler, named *Completely Fair Scheduler* [7], 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). Additionally 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, while 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 that 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 does not 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 [9].

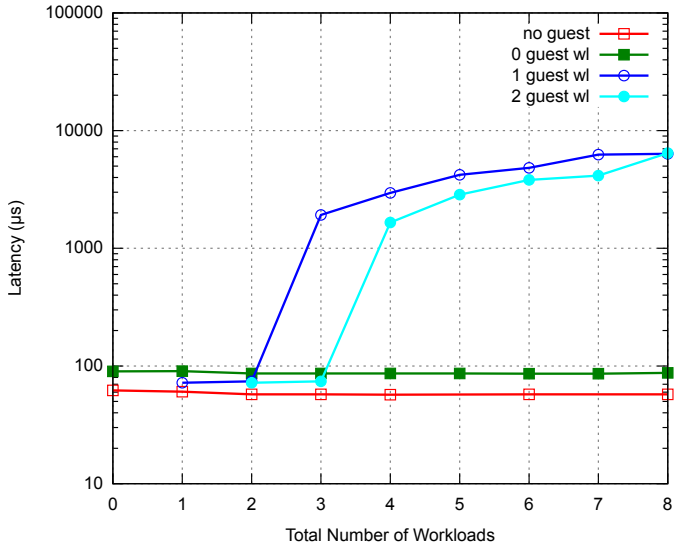
BFS focuses on a simplistic design (about 2.5 times fewer lines of code than CFS) and aims for excellent desktop interactivity and responsiveness on personal computers with a reasonable amount of cores [8]. It uses a single work-queue, *O(n)* look-up for all cores unlike CFS, and implements the *earliest 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.

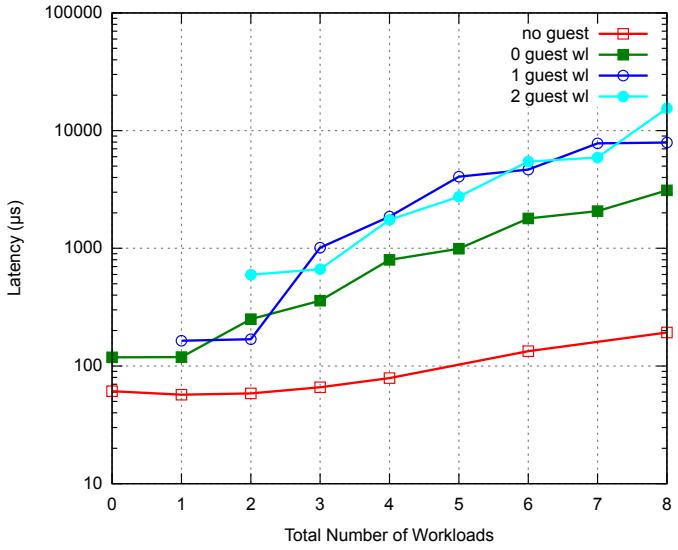
The design of BFS makes it efficient when the number of running processes is small (inferior than the number of CPUs), which is normally, according to its author [8], a common use case for a desktop computer.

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



(a) CFS



(b) BFS

Figure 1. Latency results

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, this 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 workload 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 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 used the tool, *cyclictest*, which is usually used to measure latency on a *Real Time Linux* (i.e, patched with *rt* patches) [10]. Generally, *cyclictest* is used to measure the latency of real-time thread/process (schedule with SCHED_FIFO or SCHED_RR), but it can also be used 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 measured 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 used 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 employed 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 ranging 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, latency increases significantly for the critical curves, which are *1 guest wl* and *2 guest wl*) and with at least one workload in the guest. 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.

Also, it worth noting 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 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), the 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 [11], 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 [8].

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.

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.

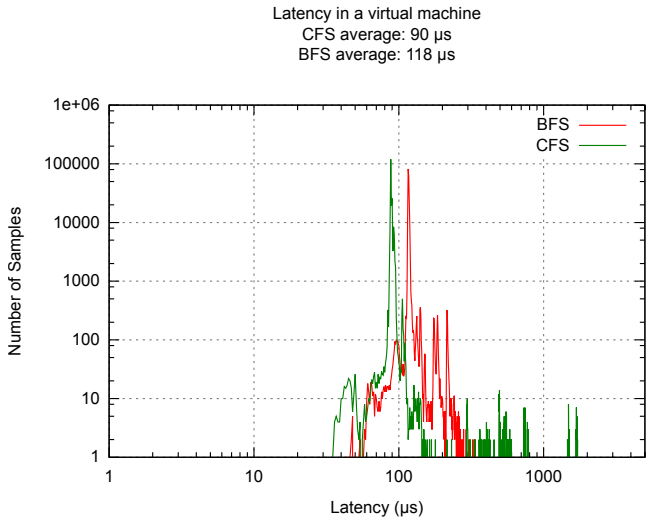


Figure 2. Guest Latency compared between BFS and CFS

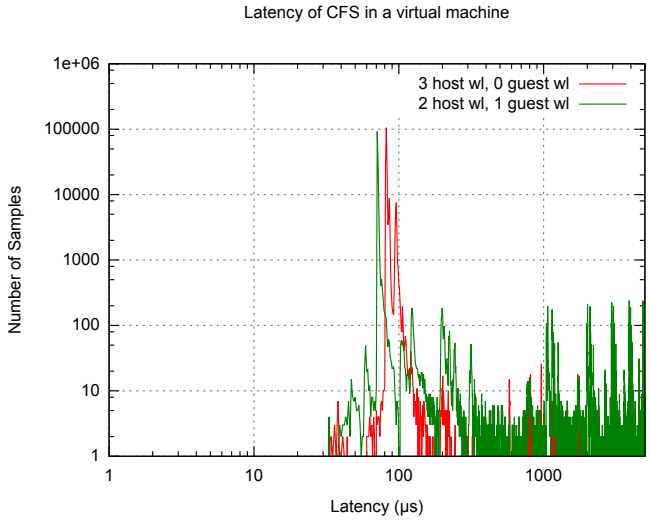
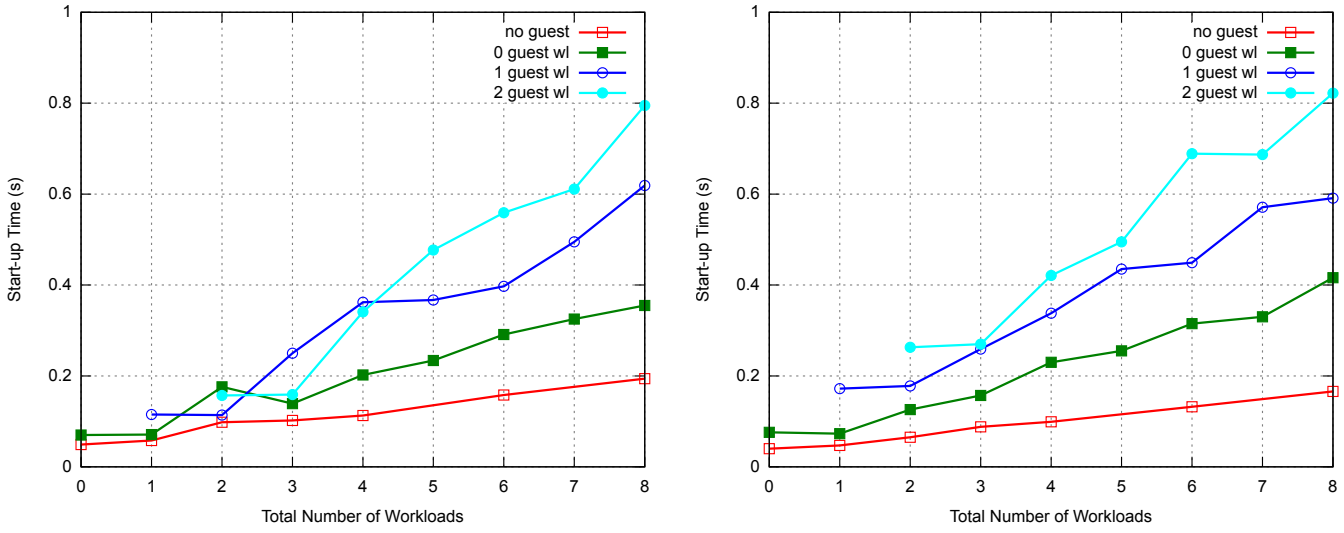


Figure 3. CFS latency in a virtual machine, compared between 3 host workloads and 2 host and 1 guest workloads

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 [2] [3] [4].

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.



(a) CFS (b) BFS

Figure 4. Start-up time results

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 scenarios where a workload is present in both the guest and host, the responsiveness of an application in the guest can not be guaranteed.

VI. Solutions

For the scheduling problem described in the previous chapters, there are two possible solutions that are proposed below. First, scheduling via a simple static prioritization policy and second, a coordinated solution that enables communication between the schedulers of the host and guest systems.

A. Static prioritizing

A straightforward solution could be a static prioritization 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/KVM to avoid interference from 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 [6]. With this solution though, the guest is always privileged even when it does not 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. But in more demanding use cases, where efficiency is required, statically raising the priority of a vCPU is not an option.

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 [3], therefore, the communication mechanism could be equivalent to the one developed for that storage I/O scheduler.

This type of solution has already been implemented and evaluated, especially to make KVM a real-time hypervisor [5] [12] on the *x86* architecture. 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 applications 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 [3] 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 (for an ARM Samsung Chromebook), is not very expensive and can fit the requirement of a task scheduling coordinated mechanism.

VII. Coordinated Scheduling Proof of concept for CFS

We choose to implement a coordinated mechanism for the CFS scheduler as a proof of concept for ARM processors. A similar mechanism has been also developed for the BFS scheduler, but for the sake of clarity and simplicity only the CFS implementation and its results are detailed, since they are very similar to BFS. This implementation is based on the *HVC* instruction as a communication mechanism between the guest and the host. The virtual machine is able to inform the host when it wants to be prioritized, depending on the real-time or interactive tasks that are executed, as well as when it does not need any prioritization anymore, i.e., a "deboost". This communication mechanism, using *HVC*, is wrapped around the "paravirt"[13] and "hypercall" infrastructure of Linux. Since this "paravirt" and "hypercall" infrastructure does not exist, yet, for *KVM on ARM*, we had to implement it. This implementation is described in the following sections.

A. Paravirt ops interface for ARM

Linux already provides a way to perform some paravirtual actions through an infrastructure named `paravirt-ops` (`pv-ops` for short)[13]. This API is used to run para-virtualized virtual machines on multiple hypervisors with the same kernel binary. This means that the same kernel binary can run on bare hardware, or on hypervisors such as VMWARE VNI or Xen, and it can be para-virtualized or fully virtualized[14].

This infrastructure exists for multiple architectures and hypervisors, but not for *KVM on ARM*, the virtualization solution we use. Therefore, a basic `paravirt-ops` implementation was developed. It is based on a patch series that enables `paravirt-ops` for Xen on ARM/ARM64[15], thus, only the KVM related part was developed.

The paravirtual functions require a hypercall implementation, to be able to send information to the host system. Therefore, hypercall functions specific to KVM have been implemented into the KVM code base of the Linux kernel. These functions use the *HVC* instruction of the ARM architecture, with the immediate argument of the HVC instruction being a constant integer used to recognize a *paravirt* call (from a PSCI call, for instance, which can also use an *HVC* instruction[17]). The parameters of the hypercall are passed through the scratch registers, *r0* contains the identification number of the hypercall and registers *r1* to *r3* represent the potential arguments for this hypercall. Figure 5 details the implementation of the `kvm_hypercall1`, which is the hypercall implementation for hypercalls with one parameter.

Those hypercalls are called from the `paravirt-ops` implementation of each paravirtualized subsystem. In our case, it simply consist of pointers to functions, stored in a structure that represent the paravirt subsystem. Those functions are called if the `paravirt-ops` infrastructure is enabled for the hypervisor on which the virtual machine is running.

```
static inline int kvm_hypercall1(u32 num, u32 arg1)
{
    register u32 n asm("r0"); /* Hypercall ID */
    register u32 r asm("r0"); /* Returned value */
    register u32 a1 asm("r1"); /* First argument */

    n = num; /* Hypercall ID is stored in r0 */
    a1 = arg1; /* The first argument is stored
         in r1 */
    __asm__ __volatile__(
        __HVC(KVM_IMM)
        : "=r" (r) : "r" (n), "r" (a1) : "memory"
        ); /* Inline assembly to call HVC
            instruction */

    return r;
}
```

Figure 5. Source code for the hypercall "1" of KVM on ARM

For our needs a *paravirt-ops* interface named `pv_cosched_ops` was added. Along with a new hypercall named `KVM_HC_COSCHED`. The `pv_cosched_ops` paravirt interface contains three functions:

- **New task**, `new_task()`
  Called each time a new process is created. We use this function to implement a heuristic mechanism to detect which are the tasks that need to be prioritized. This function is called from `wake_up_new_task()` in the Linux kernel code (kernel/sched/core.c)[16].
- **Activate task**, `activate_task()`
  Called each time a task becomes runnable. That is to say, each time a task that was waiting voluntary or due to an I/O wait becomes runnable again. We also use this function for the detection mechanism of the task to prioritize. This function is called from the function `activate_task()` in the Linux kernel (kernel/sched/core.c).
- **Schedule**, `schedule()`
  Called each time a new task is scheduled. It is in this paravirt function that the hypercall `KVM_HC_COSCHED` is performed; to request a *boost* or a *deboost*. This function is called from `__schedule()` in the Linux kernel code (kernel/sched/core.c).

On the host side, *HVC* instructions executed by the guest are trapped by KVM (in function `handle_hvc()` in arch/arm/kvm/handle_exit.c), and thus, can be handled correctly, the immediate argument of the *HVC* instruction is also checked to be sure that it is a hypercall and not something else. Then, KVM can perform the corresponding action to this hypercall according to the value retrieved from `r0`.

For the hypercall we added, `KVM_HC_COSCHED`. It takes only one argument, which is an integer set to `1` if the guest needs to be prioritized and `0` if it does not need this anymore.

B. Host side

The modifications done in the host side are located in the KVM and scheduler code base of Linux. We had to implement the "backend" of the `KVM_HC_COSCHED` hypercall, which retrieves the argument of the hypercall and performs the corresponding actions. Thus, according to the argument, which could be `0` or `1` the hypercall handler will finally invoke the functions `coshed_boost_task()` or `coshed_deboost_task()` on task `current`. The task `current` is always a vCPU thread in that case.

The added function `cosched_boost_task()` lives in the scheduler code base of Linux (kernel/sched/core.c), it takes a `struct task_struct` as an argument, which is the task to prioritize (although in our case this function is always called with `current` as an argument). It boosts the priority of all threads associated to this task, i.e., the potential other vCPU threads and the I/O threads. We choose to prioritize those processes with a `SHED_RR` policy of priority 1. For this purpose, it invokes the function `sched_setscheduler_nocheck()` to change the scheduling policy of these tasks.

The function `cosched_deboost_task()` does the reverse operation, that is to say it *deboosts* all the threads related the virtual machine, so that the policy of the processes is re-set to `SCHED_NORMAL`.

C. Guest side

On the guest side the modifications consist of calling the hypercall to request a *boost* or a *deboost* at the right time. Therefore, the `schedule()` paravirt function of `pv_coshed_ops` is called from the core `__schedule()` function (in kernel/sched/core.c)[16] equipped with the next task to schedule as a parameter. A test on this future process to run is performed to determine if this process needs to be prioritized or not, according to this information the hypercall is executed with the correct argument (*boost* or *deboost*).

Importantly enough, the time needed to perform a hypercall is not negligible, especially because of the *HVC* instruction, trapped by KVM. We estimate that the guest to host plus host to guest context switch, is around 2.1K cycles for the *Samsung ARM Chromebook* with Linux kernel v3.17. Thus, if the number of hypercalls is too frequent the performance will be worse than without the co-scheduling mechanism due to this overhead. So, in order to solve this problem, the guest will request a boost for a process, for at least a minimal period of time, i.e., the guest guarantees that it will not require a prioritization period inferior of the minimal boost time. The pseudo-code of this paravirtual `schedule()` function is detailed in Figure 6.

Function `need_to_be_boosted()` determines whether a task deserves to be prioritized or not. All tasks managed by a real time policy (i.e., `SCHED_FIFO`, `SCHED_RR` and `SCHED_DEADLINE`) are qualified for being "boosted", it corresponds to all the tasks that have the `prio` field of the `struct task_struct` strictly inferior to `100`. For tasks managed by the fair policies (i.e., `SCHED_NORMAL` and `SCHED_BATCH`), a linked list of all tasks to prioritize is maintained, this is where the two other paravirt functions are useful: *New task* and *Activate task*.

```
function pv_cosched_ops.schedule(next_task):
 static start_time /* Time on which a task
     needed a boost */
 static boosted = false /* Static variable that
     stored the state of the guest */
    if need_to_be_boosted(next_task):
       start_time = current_time() /* Time is
            updated */
       if not boosted:
          /* Ask for a boost */
          kvm_hypercall1(KVM_HC_COSCHED, 1)
          boosted = true
    else if boosted:
       now = current_time()
       /* Check if enough time has been spent
            on boost */
       if (start_time + MIN_BOOST_TIME) <= now:
          /* Ask for a deboost */
          kvm_hypercall1(KVM_HC_COSCHED, 0)
          boosted = false
```

Figure 6. Pseudo-code of the paravirtual "schedule" function

Each time a new task is created the paravirt function `new_task()` adds this task to the *prioritized list of tasks* and each task has a counter associated and initialized to a positive value. This paravirt function is called form `wake_up_new_task()` in Linux (kernel/sched/core.c). Each time a task of this list is scheduled, its counter is decremented (in the `schedule()` paravirt function), and when it reaches 0 the task is removed from the list. The counter is incremented each time a task is woke-up from a voluntary sleep, that is to say, a sleep caused by the task itself, e.g., a wait for a I/O job or a timer, this is done in paravirt `activate_task()`, which is called from `activate_task()` in Linux (kernel/sched/core.c).

VIII. Experimental results with coordinated scheduling

We repeated the same experiments as in Section V, but with the co-scheduling mechanism previously described. We report the results for the CFS scheduler, including latency and start-up time tests. The testing platform is once again *Samsung's ARM Chromebook* with version 3.17 of the Linux kernel.

A. Latency

The selected application to measure latency while testing is *cyclictest*. The minimal prioritization time (minimal boost time) selected is *500* μs, which according to our tests corresponds to the best compromise between performance and granularity in the coordination mechanism.

The results are compared to the ones reported in Section V, Figure 1a. The first four curves are kept the same for reference, and the results corresponding to co-scheduling are curves: *0 guest wl with co-sched*, *1 guest wl with co-sched* and *2 guest wl with co-sched*. The same *cyclictest* command line is used (two times, 100000 measurements, with a default interval of 1 ms).

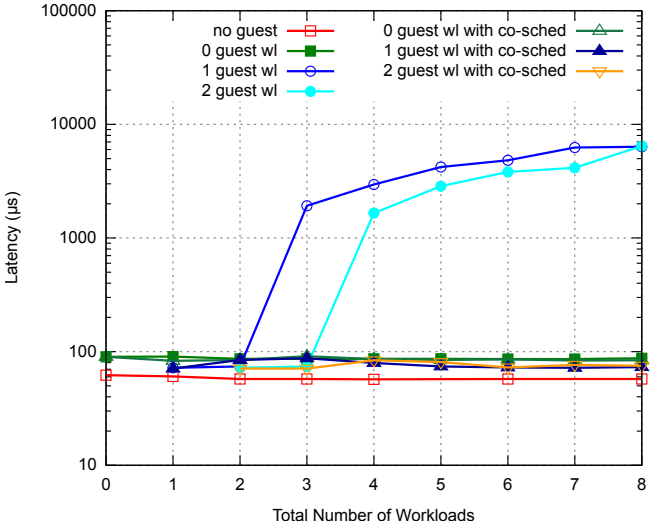


Figure 7. CFS latency results with and without coordinated scheduling mechanism

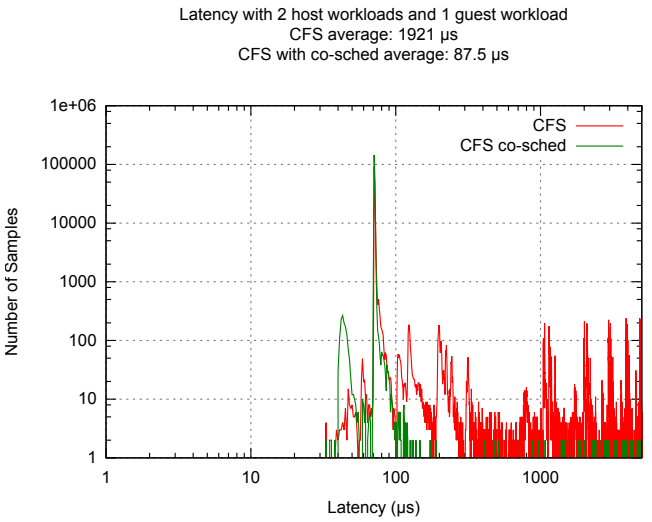


Figure 8. Latency in a virtual machine compared between CFS and CFS with a co-scheduling mechanism

The plot in Figure 7 represents the results in CFS, with and without co-scheduling. As we can see, the curves with the co-scheduling mechanism (the last three) are now almost completely horizontal, and the latency increase of the curves with one and two guest workloads is significantly improved.

Figure 8 represents the histogram of the distribution of the latency measured by *cyclictest* for the case where the system is loaded with two host workloads and one guest workload. First, we notice than the average latency with coordinated scheduling is now close to the average latency of a system without workloads, given the high value for CFS without coordinated scheduling (1921 μs). The distribution of latency is also better since there are less overall values in the high range.

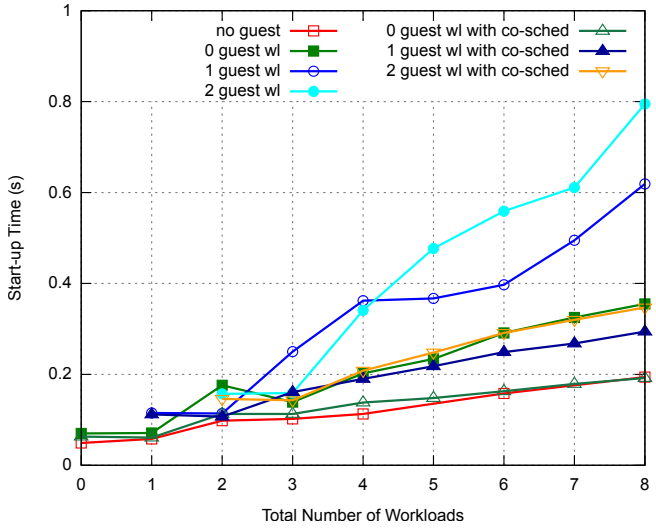


Figure 9. CFS start-up time results with and without coordinated scheduling mechanism

B. *Startup Time*

The start-up time measurements were also tested with the coordinated scheduling mechanism, where latency measurements are compared to the initial tests found in Figure 4a.

The plot in Figure 9 shows the start-up time results of *xterm*, with and without the coordinated scheduling mechanism in CFS. The minimal prioritization time used is also kept at 500 μs. We can observe a significant improvement since the curves with the coordinated scheduling mechanism have, a lower slope, and the difference with the *no_guest* curve is almost constant.

IX. Conclusion and Future Works

In virtualized environments, we highlighted that the task scheduler in the host, can fail to preserve low latency for the guest environment, and thus to maintain 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 ineffective. This issue mostly occurs when the number of CPU-bound processes is higher than the number of physical cores in the system.

Furthermore, from this work, it is shown that a coordinated scheduling mechanism can be used for process scheduling, as a way to achieve lower latency and high responsiveness in an over-committed virtual environment. The target platform used for the implementation and testing of this mechanism was based on an ARMv7 embedded system with the KVM hypervisor. Additionally, a new paravirtual interface for the scheduler was introduced, which makes easier the implementation and deployment of a coordinated scheduler.

The presented implementation of coordinated scheduling is still a proof of concept, and further optimization and regression testing is needed, especially in the area of task

detection heuristics. Finally, an extension for tests with more complex scenarios, including more than one virtual machines and multiple vCPUs, is under way.

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] J. Fanguède, A. Spyridakis, and D. Raho, "Towards Coordinated Task Scheduling in Virtualized Systems," ADVCOMP 2015, The Ninth International Conference on Advanced Engineering Computing and Applications in Sciences, 2015, pp. 106-111.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] "CFS scheduler," [retrieved: February 2016]. Available: http://lwn.net/Articles/230501/

[8] C. Kolivas, "BFS FAQ," [retrieved: February 2016]. Available: http://ck.kolivas.org/patches/bfs/bfs-faq.txt.

[9] C. Kolivas, "BFS Patches," [retrieved: February 2016]. Available: http://ck.kolivas.org/patches/bfs/3.0/.

[10] "Cyclictest," [retrieved: February 2016]. Available: https://rt.wiki.kernel.org/index.php/Cyclictest

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

[12] M. Aichouch, J-C. Prevotet, and F. 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.

[13] "Linux kernel paravirt_ops documentation," [retrieved: February 2016]. Available: http://lxr.free-electrons.com/source/Documentation/virtual/paravirt_ops.txt

[14] "Xen Paravirt ops," [retrieved: February 2016]. Available: http://wiki.xen.org/wiki/XenParavirtOps

[15] S. Stabellini, "Xen ARM/ARM64 CONFIG_PARAVIRT patch series" [retrieved: February 2016]. Available: http://lists.xen.org/archives/html/xen-devel/2014-01/msg00851.html

[16] "Linux process scheduler core code file," [retrieved: February 2016]. Available: http://lxr.free-electrons.com/source/kernel/sched/core.c

[17] "PSCI Linux documentation," [retrieved: February 2016]. Available: http://lxr.free-electrons.com/source/Documentation/devicetree/bindings/arm/psci.txt