Virtual-BFQ: A Coordinated Scheduler to Minimize Storage Latency and Improve Application Responsiveness in Virtualized Systems

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

***Abstract*—Preserving responsiveness is an enabling condition for running interactive applications effectively in virtual machines. For this condition to be met, low latency usually needs to be guaranteed to storage-Input/Output operations. In contrast, in this paper we show that in virtualized environments, there is a missing link exactly in the chain of actions performed to guarantee low storage-I/O latency. After describing this problem theoretically, we show its practical consequences through a large set of experiments with real world-applications. After detailing the possible solutions to replace this missing connection, we detail our chosen solution based on the I/O scheduler BFQ (Budget Fair Queueing) named Virtual-BFQ. Which it designed to preserve a high application responsiveness in KVM (Kernel-based Virtual Machine) virtual machines on ARM architectures. For the experiments, we used two Linux schedulers, both designed to guarantee a low latency, and a publicly available I/O benchmark suite, extended to be used also in a virtualized environment. As for the experimental testbed, we ran our experiments on the following three devices connected to an ARM embedded system: an ultra-portable rotational disk, a microSDHC (Secure Digital High Capacity) Card and an eMMC (embedded Multimedia card) device. This is an ideal testbed for highlighting latency issues, as it can execute applications with about the same I/O demand as a general-purpose system, but for power-consumption and mobility issues. According to the experimental results reported in this paper, even in the presence of a heavy background workload on the guest virtual disk, plus a heavy additional background workload on the physical storage device corresponding to that virtual disk, Virtual-BFQ does preserve in the guest a high application responsiveness.**

Keywords–KVM/ARM; virtualization; responsiveness and soft-real time guarantees; coordinated scheduling; embedded systems; Virtual-BFQ.

I. INTRODUCTION

Virtualization is an increasingly successful solution to achieve both flexibility and efficiency in general-purpose and embedded systems. However, for virtualization to be effective also with interactive applications, the latter must be guaranteed a high, or at least acceptable responsiveness. In other words, it is necessary to guarantee that these applications take a reasonably short time to start, and that the tasks requested by these applications, such as, e.g., opening a file, are completed in a reasonable time.

To guarantee responsiveness to an application, it is necessary to guarantee that both the code of the application and the I/O requests issued by the applications get executed with a low latency. In virtualized systems the responsiveness is not always guarantees [1], and expectedly, there is interest and active research in preserving a low latency in virtualized environments [2][3][4][5][6][7][8], especially in soft and hard real-time contexts [9][10][11]. In particular, some virtualization solutions provide more or less sophisticated Quality of Service mechanisms also for storage I/O [2][3][12][13]. However, even just a thorough investigation on application responsiveness, as related to storage-I/O latency, seems to be missing. In this paper, we address this issue by providing the following contributions.

A. Contributions of this paper

First, we show, through a concrete example, that in a virtualized environment there is apparently a missing link in the chain of actions performed to guarantee a sufficiently low I/O latency when an application is to be loaded, or, in general, when any interactive task is to be performed. To this purpose, we use as a reference two effective schedulers in guaranteeing a high responsiveness: Budget Fair Queuing [14] and Completely Fair Queuing [15]. They are two production-quality storage-I/O schedulers for Linux.

Then, we report experimental results with real-world applications. These results confirm that, if some applications are competing for the storage device in a host, then the applications running in a virtual machine executed in the same host may become from not much responsive to completely unresponsive. To carry out these experiments, we extended a publicly available I/O benchmark suite for Linux [16], to let it comply also with a virtualized environment.

The solution described in this paper solves the problem highlighted previously, it is an extension of the BFQ storage I/O scheduler [17]. Such an extension can be implemented in several ways. So first, we provide an analysis of the solution space. From this analysis, we highlight the solution that seems to provide most benefits. We did implement such a solution and named Virtual-BFQ (V-BFQ) [18] the resulting extended version of BFQ. We describe its implementation in detail. And we report our experimental results with this scheduler. The results obtained confirm that V-BFQ does preserve a high application responsiveness in a virtualized environment even with the presence of heavy background workloads.

As an experimental testbed, we opted for an ARM embedded system, based on the following considerations. On one

hand, modern embedded systems and consumer-electronics devices can execute applications with about the same I/O demand as general-purpose systems. On the other hand, for mobility and energy-consumption issues, the preferred storage devices in the former systems are (ultra) portable and low-power ones. These devices are necessarily slower than their typical counterparts for general-purpose systems. Being the amount of I/O the same, the lower the speed of a storage device is, the more I/O-latency issues are amplified. Finally, as a virtualization solution we used the pair QEMU (Quick EMUlator) and KVM, one of the most popular and efficient solutions in ARM embedded systems.

B. Organization of this paper

In Section II, we describe the schedulers that we use as a reference in this paper. Then, in Section III we show the important I/O-latency problem on which this paper is focused. After that, in Section IV, we describe how we modified the benchmark suite to execute our experiments. And in Section V, we report our experimental results that highlight the latency problem. Then, in Section VI we provide an analysis of the possible solution space, and highlight the solution that apparently provides the best trade-off between pros and cons. After that, we describe V-BFQ in detail in Section VII. And then in Section VIII, we report our results with V-BFQ for the same experiments that we executed for BFQ and CFQ. Finally, in Section IX we compared the results obtained with V-BFQ with two other standard I/O schedulers for Linux : *Deadline* and *NOOP*.

II. Reference schedulers

To show the application-responsiveness problem that is the focus of this paper, we use the following two storage-I/O schedulers as a reference: BFQ [17] and CFQ [15]. We opted for these two schedulers because, they, both guarantee a high throughput and low latency. In particular, BFQ achieves even up to 30% higher throughput than CFQ on hard disks with parallel workloads. Strictly speaking, only the second feature is related to the focus of this paper, but the first feature is however important, because a scheduler achieving only a small fraction of the maximum possible throughput may be, in general, of little practical interest, even if it guarantees a high responsiveness. The second reason why we opted for these schedulers is that up-to-date and production-quality Linux implementations are available for both. In particular, CFQ is the default Linux I/O scheduler, whereas BFQ is being maintained separately [16]. In addition to the extended tests for BFQ and CFQ, we also identified similar behaviour with the Noop and Deadline schedulers. In the next two sections, we briefly describe the main differences between the two schedulers, focusing especially on I/O latency and responsiveness. For brevity, when not otherwise specified, in the rest of this paper we use the generic term *disk* to refer to both a hard disk and a solid-state disk.

A. BFQ

BFQ achieves a high responsiveness basically by providing a high fraction of the disk throughput to an application that is being loaded, or whose tasks must be executed quickly. In this respect, BFQ benefits from the strong fairness guarantees it provides: BFQ distributes the disk throughput (and not just the disk time) as desired to disk-bound applications, with any workload, *independently of* the disk parameters and even if the disk throughput fluctuates. Thanks to this strong fairness property, BFQ does succeed in providing an application requiring a high responsiveness with the needed fraction of the disk throughput in any condition. The ultimate consequence of this fact is that, regardless of the disk background workload, BFQ guarantees to applications about the same responsiveness as if the disk was idle [17].

B. CFQ

CFQ grants disk access to each application for a fixed *time slice*, and schedules slices in a round-robin fashion. Unfortunately, as shown by Valente and Andreolini [17], this service scheme may suffer from both unfairness in throughput distribution and high worst-case delay in request completion time with respect to an ideal, perfectly fair system. In particular, because of these issues and of how the low-latency heuristics work in CFQ, the latter happens to guarantee a worse responsiveness than BFQ [17]. This fact is highlighted also by the results reported in this paper.

III. Missing link for preserving responsiveness

We highlight the problem through a simple example. Consider a system running a guest operating system, say guest G, in a virtual machine, and suppose that either BFQ or CFQ is the default I/O scheduler both in the host and in guest G. Suppose now that a new application, say application A, is being started (loaded) in guest G while other applications are already performing I/O without interruption in the same guest. In these conditions, the cumulative I/O request pattern of guest G, as seen from the host side, may exhibit no special property that allows the BFQ or CFQ scheduler in the host to realize that an application is being loaded in the guest.

Hence, the scheduler in the host may have no reason for privileging the I/O requests coming from guest G. In the end, if also other guests or applications of any other kind are performing I/O in the host—and for the same storage device as guest G—then guest G may receive *no help* to get a high-enough fraction of the disk throughput to start application A quickly. As a conclusion, the start-up time of the application may be high. This is exactly the scenario that we investigate in our experiments. Finally, it is also important to note that our focus has been in local disk/storage, as scheduling of network-based storage systems is not always under the direct control of the Linux scheduling policies.

IV. Extension of the benchmark suite

To implement our experiments we used a publicly available benchmark suite [16] for the Linux operating system. This suite is designed to measure the performance of a disk scheduler with real-world applications. Among the figures of merit measured by the suite, the following two performance indexes are of interest for our experiments:

Aggregate disk throughput. To be of practical interest, a scheduler must guarantee, whenever possible, a high (aggregate) disk throughput. The suite contains a benchmark that allows the disk throughput to be measured

TABLE I. Storage devices used in the experiments

Type	Name	Size	Read peak rate
1.8-inch Hard Disk	Toshiba MK6006GAH	60 GB	10.0 MB/s
microSDHC Card	Transcend SDHC Class 6	8 GB	16 MB/s
eMMC	SanDisk SEM16G	16 GB	70 MB/s

while executing workloads made of the reading and/or the writing of multiple files at the same time.

Responsiveness. Another benchmark of the suite measures the *start-up* time of an application—i.e., how long it takes from when an application is launched to when the application is ready for input—with cold caches and in presence of additional heavy workloads. This time is, in general, a measure of the responsiveness that can be guaranteed to applications in the worst conditions.

Being this benchmark suite designed only for non-virtualized environments, we enabled the above two benchmarks to work correctly also inside a virtual machine, by providing them with the following extensions:

Choice of the disk scheduler in the host. Not only the active disk scheduler in a guest operating system, hereafter abbreviated as just guest OS, is relevant for the I/O performance in the guest itself, but, of course, also the active disk scheduler in the host OS. We extended the benchmarks so as to choose also the latter scheduler.

Host-cache flushing. As a further subtlety, even if the disk cache of the guest OS is empty, the throughput may be however extremely high, and latencies may be extremely low, in the guest OS, if the zone of the guest virtual disk interested by the I/O corresponds to a zone of the host disk already cached in the host OS. To address this issue, and avoid deceptive measurements, we extended both benchmarks to flush caches at the beginning of their execution and, for the responsiveness benchmark, also (just before) each time the application at hand is started. In fact the application is started for a configurable number of times, see Section V.

Workload start and stop in the host. Of course, responsiveness results now depend also on the workload in execution in the host. Actually, the scenario where the responsiveness in a Virtual Machine (VM) is to be carefully evaluated, is exactly the one where the host disk is serving not only the I/O requests arriving from the VM, but also other requests (in fact this is the case that differs most from executing an OS in a non-virtualized environment). We extended the benchmarks to start the desired number of file reads and/or writes also in the host OS. Of course, the benchmarks also automatically shut down the host workload when they finish.

V. Experimental results

We executed our experiments on a Samsung Chromebook, equipped with an ARMv7-A Cortex-A15 (dual-core, 1.7 GHz), 2 GB of RAM and the devices reported in Table I. There was only one VM in execution, hereafter denoted as just *the* VM, emulated using QEMU/KVM. Both the host and the guest OSes were Linux 3.12.

A. Scenarios and measured quantities

We measured, first, the aggregate throughput in the VM while one of the following combinations of workloads was being served.

In the guest. One of the following six workloads, where the tag **type** can be either **seq** or **rand**, with **seq/rand** meaning that files are read or written sequentially/at random positions:

1r-type one reader (i.e., one file being read);
5r-type five parallel readers;
2r2w-type two parallel readers, plus two parallel writers.

In the host. One of the following three workloads (in addition to that generated, in the host, by the VM):

no-host_workload no additional workload in the host;
1r-on_host one sequential file reader in the host;
5r-on_host five sequential parallel readers in the host.

We considered only sequential readers as additional workload in the host, because it was enough to cause the important responsiveness problems shown in our results. In addition, for each workload combination, we repeated the experiments with each of the four possible combinations of active schedulers, choosing between BFQ and CFQ, in the host and in the guest.

The main purpose of the throughput experiments was to verify that in a virtualized environment both schedulers achieved a high-enough throughput to be of practical interest. Both schedulers did achieve, in the guest, about the same (good) performance as in the host. For space limitations, we do not report these results, and focus instead on the main quantity of interest for this paper. In this regard, we measured the start-up time of three popular interactive applications of different sizes, inside the VM and while one of the above combinations of workloads was being served.

The applications were, in increasing-size order: *bash*, the Bourne Again shell, *xterm*, the standard terminal emulator for the X Window System, and *konsole*, the terminal emulator for the K Desktop Environment. As shown by Valente and Andreolini [17], these applications allow their start-up time to be easily computed. In particular, to get worst-case start-up times, we dropped caches both in the guest and in the host before each invocation (Section IV). Finally, just before each invocation a timer was started: if more than 60 seconds elapsed before the application start-up was completed, then the experiment was aborted (as 60 seconds is evidently an unbearable waiting time for an interactive application).

We found that the problem that we want to show, i.e., that responsiveness guarantees are violated in a VM, occurs regardless of which scheduler is used in the host. Besides, in presence of file writers, results are dominated by fluctuations and anomalies caused by the Linux write-back mechanism. These anomalies are almost completely out of the control of the disk schedulers, and not related with the problem that we want to highlight. In the end, we report our detailed results **only with file readers**, **only with BFQ** as the active disk scheduler **in the host**, and **for *xterm***.

B. Statistics details

For each workload combination, we started the application at hand five times, and computed the following statistics over

the measured start-up times: minimum, maximum, average, standard deviation and 95% confidence interval (actually we measured also several other interesting quantities, but in this paper we focus only on application responsiveness). We denote as a *single run* any of these sequences of five invocations. We repeated each single run ten times, and computed the same five statistics as above also across the average start-up times computed for each repetition. We did not find any relevant outlier, hence, for brevity and ease of presentation, in the next plots we show only averages across runs (i.e., averages of the averages computed in each run).

C. Results

Figure 1 shows our results with the hard disk (Table I). The reference line represents the time needed to start *xterm* if the disk is idle, i.e., the minimum possible time that it takes to start *xterm* (a little less than 2 seconds). Comparing this value with the start-up time guaranteed by BFQ with no host workload, and with any of the first three workloads in the guest (first bar for any of the *1r-seq*, *5r-seq* and *1r-rand* guest workloads), we see that, with all these workloads, BFQ guarantees about the same responsiveness as if the disk was idle. The start-up time guaranteed by BFQ is slightly higher with *5r-rand*, for issues related, mainly, to the slightly coarse time granularity guaranteed to scheduled events in the kernel in an ARM embedded system, and to the fact that the reference time itself may advance haltingly in a QEMU VM.

In contrast, again with no host workload, the start-up time guaranteed by CFQ with *1r-seq* or *1r-rand* on the guest is 3 times as high than on an idle disk, whereas with *5r-seq* the start-up time becomes about 17 times as high. With *5r-seq* the figure reports instead an **X** for the start-up time of CFQ: we use this symbolism to indicate that the experiment failed, i.e., that the application did not succeed at all in starting before the 60-second timeout.

In view of the problem highlighted in Section III, the critical scenarios are however the ones with some additional workload in the host; in particular, *1r_on_host* and *5r_on_host* in our experiments. In these scenarios, both schedulers unavoidably fail to preserve a low start-up time. Even with just *1r_on_host*, the start-up time, with BFQ, ranges from 3 to 5.5 times as high than on an idle disk. The start-up time with CFQ is much higher than with BFQ with *1r_on_host* and *1r-seq* on the guest, and, still with *1r_on_host* (and CFQ), is even higher than 60 seconds with *5r-seq* or *5r-rand* on the guest. With *5r_on_host* the start-up time is instead basically unbearable, or even higher than 60 seconds, with both schedulers. Finally, with *1r-rand* all start-up times are lower and more even than with the other guest workloads, because both schedulers do not privilege much random readers, and the background workload is generated by only one reader.

Figures 2 and 3 show our results with the two flash-based devices. At different scales, the patterns are still about the same as with the hard disk. The most notable differences are related to CFQ: on one side, with no additional host workload, CFQ achieves a slightly better performance than on the hard disk, whereas, on the opposite side, CFQ suffers from a much higher degradation of the performance, again with respect to the hard-disk case, in presence of additional host workloads.

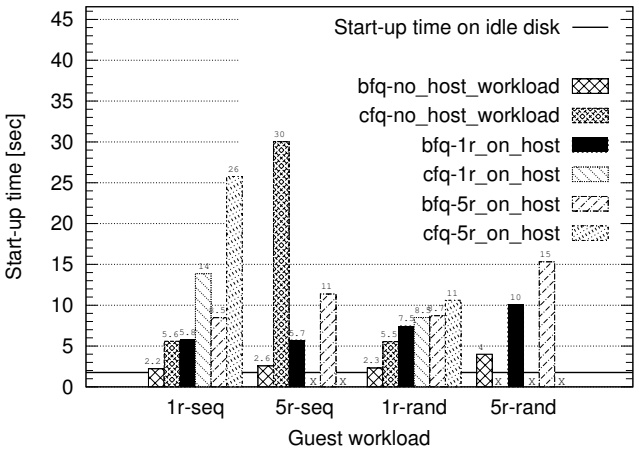

Figure 1. Results with the hard disk (lower is better).

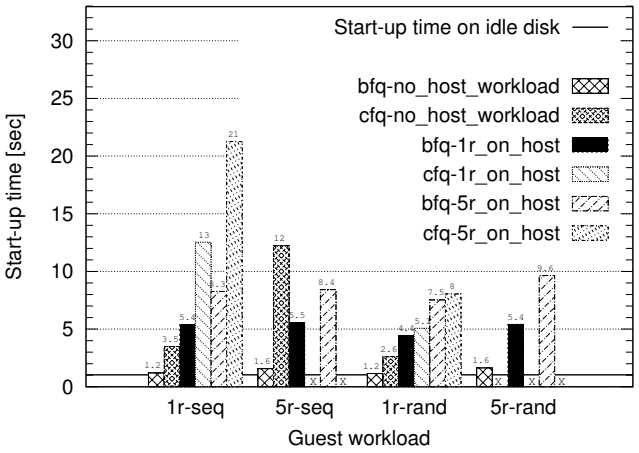

Figure 2. Results with the microSDHC CARD (lower is better).

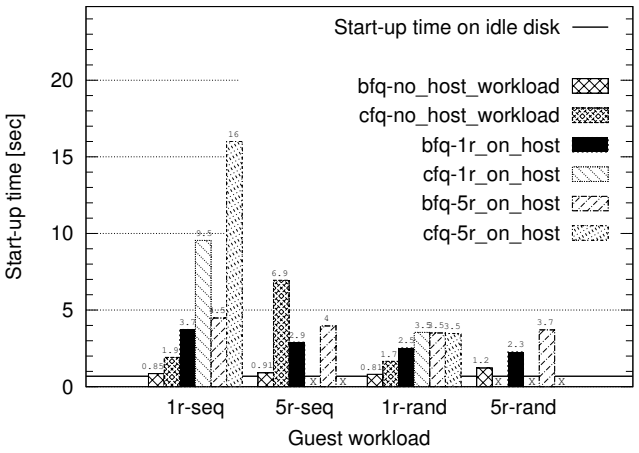

Figure 3. Results with the eMMC (lower is better).

To sum up, our results confirm that, with any of the devices considered, responsiveness guarantees are lost when there is some additional I/O workload in the host.

VI. Possible solutions

The key idea to recover responsiveness is the coordination between schedulers, in order to create the missing link described in Section III. That is why, to deal with the above problem, the guest disk scheduler should somehow inform the host disk scheduler that the I/O requests of the guest should be privileged to preserve a low latency. On the opposite side, the host disk scheduler should properly privilege a guest asking for an urgent and high-throughput access to the disk. In other words, the guest and the host disk schedulers should somehow coordinate with each other to achieve the desired latency goals. As shown in the preceding section, BFQ guarantees a much higher responsiveness than CFQ. For this reason we choose it as the candidate scheduler to extend. The idea is then to realize a coordinated version of BFQ, which we named Virtual-BFQ (V-BFQ). We can describe this idea as follows:

Guest side: when the guest V-BFQ scheduler detects, through its internal heuristics, that some application needs urgent service from the virtual disk, it communicates this need to the V-BFQ scheduler in the host. On the other hand, the V-BFQ guest scheduler also communicates to the host when there are no more applications needing a quick service.

Host side: when the host V-BFQ scheduler receives the above help request from one VM, it privileges that VM until the same VM tells the host V-BFQ scheduler that no help is needed anymore.

From this scheme we can easily deduce that the communication between the guest and host V-BFQ schedulers is a critical issue. Actually, the possible solution space stems mainly from the choices we can make in terms of communication between the schedulers.

We can consider two main alternative approaches to let the guest scheduler communicate to the host scheduler when it needs to be privileged:

A. Augment metadata in guest storage-I/O requests.

We could add some additional boolean field the description of an I/O request, in both the host and the guest kernels. This flag could be set by the guest scheduler for each request coming from an application to be served quickly. The information that this flag is set for a given guest I/O request should then somehow flow through the chain of components that translate I/O requests coming from the guest into corresponding host-side I/O requests. Then the same flag should be set also in the latter requests. Finally, when the host scheduler would see this flag set for an I/O request, it could privilege the host process that generated that requests.

The main benefit of this solution is that it is very simple and little invasive in terms of in-kernel modifications: just the declaration of a data structure should be modified.

There are however two main disadvantages:

Higher latency: host I/O threads would not be privileged immediately (i.e., right after a guest starts to need help), but only when the flagged I/O requests would be eventually issued by these threads. This may be a serious problem with synchronous I/O threads, which issue their next request or batch of requests only after the last pending one has been completed. Until the pending requests of a non-yet-privileged I/O thread are completed, that thread may not issue the flagged one. And, exactly because the thread is not yet privileged, its pending requests may wait for a long time before being served.

User-space modifications would be needed, to let the flag percolate from the I/O requests in the guest kernel to the I/O requests in the host kernel. In this regard, it is worth highlighting an additional important issue: guest I/O requests are currently turned into just simple read-/write operations on the image file of the virtual disk, and not directly into I/O requests. Hence, the chain of components involved in carrying information about this flag, and eventually flagging I/O requests on the host side may be rather long.

B. Immediately signal the need for help to the host with some form of direct communication

For this approach, we can consider two alternative solutions:

B.1 User-space solution: host-side I/O threads serving guest I/O requests of (in-guest) applications needing urgent service may directly ask for their weight to be increased. The main benefit of this solution is that the interaction scheme would be very simple: no modification would be needed in the host scheduler to guess what processes/threads need to be privileged. But there is a significant drawback : it is user-space invasive, QEMU code would need to be modified.

B.2 In-kernel solution: the guest disk scheduler could just tell directly to the host disk scheduler that it needs help to guarantee a high responsiveness to some application. So, no user-space modification needed. But a non-trivial logic would be needed in the host disk scheduler to retrieve the IDs of all the I/O threads to privilege after receiving the help request from the guest. In fact, in QEMU, the threads handling virtual CPUs in a VM differ from the I/O threads that take care of serving I/O requests for that VM. Even worse, I/O threads are dynamically created and destroyed as needed during the lifetime of a VM.

Analyzing the above solution space, we concluded that solution B.2 is the one with the better trade-offs between advantages and disadvantages, and hence decided to extend BFQ accordingly. We provide full details on the resulting implementation, named V-BFQ, in the following section.

VII. The Virtual-BFQ solution

In this section we describe V-BFQ in detail, using pseudo-code. In particular, we focus mostly on the logical aspects.

The first issue to address was how to let the guest V-BFQ communicate directly with the host V-BFQ. As a simple solution, we opted for the Hypervisor-Call instruction available on ARM architectures, hereafter abbreviated as just *hvc*. The execution of an *hvc* generates a Hypervisor Call exception. In particular, if the *hvc* is executed by a KVM/QEMU guest, then a dedicated KVM handler gets called.

Finally, *hvc* has an integer number as an immediate argument. We used two possible values for the immediate argument to let the guest scheduler tell the host scheduler whether it

```
// just after any point in the code where
     *raised_busy_queues* is increased
if (raised_busy_queues == 1) // transition from
     0 to 1
   hvc #1 ; // notify the host that this guest
        needs to be privileged
// just after any point in the code where
     *raised_busy_queues* is decreased
if (raised_busy_queues == 0) // transition from
     1 to 0
   hvc #0 ; // notify the host that this guest
        does not need help anymore
// at the exit from the scheduler
if (raised_busy_queues > 0) // the guest is
     still being privileged
   hvc #0 ; // notify the host that this guest
        does not need help anymore
```

Figure 4. HVC call in the guest

needs to be prioritized or not. In particular, we decided to use the following two values:

1 to indicate the guest needs to be privileged.
0 to indicate the guest does not need to be privileged anymore.

We can now describe in detail both the guest and the host extensions that we integrated in BFQ to implement V-BFQ.

A. Guest extensions

Under Linux, and, in particular, from the BFQ standpoint, a thread is just a process. BFQ basically associates a queue to one or more processes/threads, and raises the weight of the queues associated to processes/threads to be privileged. As a consequence, a guest needs to be privileged if and only if the number of weight-raised and backlogged (i.e., non-empty) queues in the guest BFQ scheduler is higher than 0. In fact, even if there are weight-raised queues, but they are all empty, there is no urgent pending I/O request, and hence there is no need to privilege the guest for the moment.

Fortunately, BFQ maintains a variable that contains exactly the number of backlogged and weight-raised queues. This variable is called *raised_busy_queues* in the code [16]. Hence, to decide when it is time to either ask the host V-BFQ scheduler for a higher fraction of the disk throughput or inform the V-BFQ scheduler that no special treatment is needed anymore, it is enough to track the transitions of this variable, respectively, from 0 to 1 and from 1 to 0. The guest extension of BFQ does exactly that, by invoking an *hvc* with the right argument for each of the two cases. We describe this extension in more detail in the pseudo-code snippet in Figure 4.

B. Host extensions

The service of the I/O requests generated from a guest is delegated by QEMU to a pool of I/O threads created on demand. It follows that:

The V-BFQ scheduler in a host must raise the weights of the queues associated to the I/O threads of a VM whose guest is requesting to be privileged. As a consequence, every time the host V-BFQ scheduler has the opportunity to raise the weight of some queue, it must know what is the set of I/O threads to privilege, so as to check whether that queue is actually associated to one of such threads (and hence must be weight-raised).

As for knowing, every time this information is needed, what is the set of I/O threads to privilege, we need to consider the following important issues.

Although QEMU creates and destroys *supporting* threads, such as I/O threads, dynamically, the group leader of these threads *never changes* for a given VM. Since QEMU creates and kills I/O threads dynamically, when an *hvc #1* is received from a guest, the I/O thread that will handle the guest I/O request that caused the guest to issue that *hvc* may even not yet exist. And the I/O thread group for a VM may change over time, without the V-BFQ scheduler in the host receiving any notification about changes of in set of I/O threads for any VM.

In view of the above issues related to the dynamic creation/destruction of I/O threads, and exploiting the fact that the group leader for a VM is however constant during the lifetime of the VM, we use the following strategy to allow the V-BFQ host scheduler to correctly weight-raise the right queues.

As a basic step, the V-BFQ scheduler maintains a list of (only) the leaders of the groups of I/O threads to be privileged. In more detail, V-BFQ maintains the list of the Process Identifiers (PID) of these leaders. In this respect, it is worth recalling that a thread basically coincides with a process under Linux. From the PID of a group leader it is then extremely simple to scan the list of its current child threads. In particular, the latter list is trivially kept up-to-date by the kernel itself. Hence, when V-BFQ has to decide whether a given queue is to be weight-raised, it consults this list to reconstruct the list of all the threads to privilege.

We describe the host extension of BFQ by describing in detail each of the above two points.

1) Manipulating the list of group leaders: This list of group leaders is updated according to the *hvc #1* or *hvc #0* received from active guests (and also automatically pruned when some group leader is discovered to be non-existing anymore, as shown in detail in Section VII-B2). To achieve this goal, we had first to modify the KVM handler of *hvc* exceptions in the host kernel. The modification are described, using pseudo-code, in Figure 5.

The last function invoked by the *hvc* handler is the V-BFQ hook that handles the update of the list of the PIDs of the leaders of the threads to privilege. Of course, we deduce that, with respect to BFQ, V-BFQ must contain this additional hook. The exact steps made by this hook are described in the Figure 6 where the list of the PIDs of the leaders of the groups of threads to be privileged is named *leader_pid_list*.

As shown in the snippet Figure 6, the hook does not only update the list of group-leader PIDs: if the mode is add the hook also immediately raises, by calling the function *weight_raise_queue()* described in the next section, the weight of all the backlogged queues associated either to the group leader being added or to any of its children. We describe this part and the rest of the host extensions to BFQ in the next

```
// input: data structure describing the qemu
    virtual cpu on which the hvc is executed
HVC-handler(in: vcpu) {
  // get the descriptor of the host-side qemu
      process/thread implementing the vcpu
  qemu_task = get_pid_task(vcpu->pid);
  leader = qemu_task->group_leader; // pid of
      the thread-group leader

  // value passed to hvc when invoked in the
      guest
  arg_value = vcpu->arch.fault.hsr & 0x000000ff;

  if (arg_value == 1)
    mode = add;
  else
    mode = remove;

  // update the list of the leaders of the
      threads to privilege
  V-BFQ-VM_threads_update_hook(leader, mode);
}
```

Figure 5. Pseudo-code of the HVC handler

```
V-BFQ-VM_threads_update_hook(leader, mode) { //
    mode can be either add or remove
  if (mode == add) {
    if (look_for_pid(leader_pid_list) == NULL)
        // pid not present
      add_to_list(leader_pid_list, leader->pid);
  } else {// mode == remove
    pid_entry = look_for_pid(leader_pid_list,
        leader->pid);
    if (pid_entry != NULL) // pid in list
      rm_from_list(leader_pid_list, pid_entry);
  }
  // additional code to achieve maximum
      responsiveness (see below)
  for_each_child_thread(leader)
  // at each iteration child is set to one of
      the child threads
    for_each_backlogged_bfq_queue()
    // at each iteration, bfqq is one of the
        backlogged queue
    if (child->pid == bfqq->pid)
      // to achieve maximum responsiveness,
          immediately raise the weight of
      // the queue and reschedule the queue
          (see below)
      weight_raise_queue(bfqq);
}
```

Figure 6. Pseudo-code of the V-BFQ hook function

section.

2) Raising the weight of the queues associated with the threads to privilege: In the V-BFQ hook, the weights of all the backlogged queues associated either to the group leader being added or to any of its children are raised immediately, because this step is crucial for starting to serve as soon as possible the

```
weight_raise_queue(bfqq) {
  list_entry pid_entry =
      look_for_pid(leader_pid_list, tmp->pid);
  if (pid_entry != NULL) { // pid in list
    bool need_reposition = bfqq !=
        in_service_queue && !bfqq_is_idle;
    if (is_already_raised(bfqq)) {
      move_forward_raising_start_time(bfqq) ;
          // see below
      return; // nothing else to do
    }

  if (need_reposition)
    deactivate(bfqq) ; // remove queue from
        schedule
  perform_core_weight_raising_operations(bfqq);
  if (need_reposition)
    activate(bfqq); // reschedule in the right
        position for the new weight
  }
}

perform_core_weight_raising_operations(bfqq) {
  raise_weight_coeff(bfqq); // raise queue
      weight
  set_raising_start(); // (re)set the start
      time of the weight-raising (see below)
  set_raising_duration(); // (re)set the
      duration of the raising period (see below)
}
```

Figure 7. Pseudo-code of function *weight_raise_queue()*

I/O requests related to a guest asking to be privileged. This step is however effective only provided that the following issue is properly addressed.

Unless it is currently in service, a backlogged queue is of course scheduled for service. In this respect, if the weight of an already-scheduled queue is raised, but the schedule is not changed immediately, then the queue will wait to be served according to its old, low weight. Only after being served, and if still backlogged, the queue will be rescheduled according to its new high weight, and hence, only from that moment on, the queue will get a high fraction of the disk throughput (until its weight is lowered again). In view of this important issue, in the hook, the queues whose weights are raised are also immediately *rescheduled* according to their new weights. This guarantees the minimum possible latency for a guest asking to be privileged. The experimental results in Section VIII clearly show the benefits of this immediate reschedule.

The exact steps taken by the function *weight_raise_queue()* are described in the Figure 7.

As for the functions *move_forward_raising_start_time()*, *set_raising_start()*, and *set_raising_duration()*, these functions are related to how the weight-raising heuristics work in BFQ, and hence in V-BFQ: when weight-raising starts for a queue, BFQ stores the time instant when it happens in a variable that we call *raising_start_time* hereafter. BFQ also sets the duration for the weight-raising: if the queue is constantly backlogged for all this time period, then its weight is lowered again.

In particular, if the queue is already being weight-raised, then V-BFQ just moves forwards its *raising_start_time* , as if the weight-raising period for the queue just (re)started. In fact, differently from the physiological BFQ behavior, for a queue associated to a thread to be privileged, weight-raising is never stopped. To correctly guarantee this special treatment, V-BFQ also contains the following modification with respect to BFQ: at any point in the code where raising might finish for a queue, V-BFQ controls whether that queue is associated to a thread to be privileged, and, in that case, *does not* stop weight-raising for the queue.

To sum up, the part of the host extension of V-BFQ shown so far guarantees that backlogged queues associated to threads to be privileged are immediately weight-raised and, if needed, rescheduled to guarantee minimum latency.

Hence, to cover all possible cases, we are left with handling the case of the queues that: 1) are still idle or not-yet-existing when the hook is invoked to add a new group leader, but 2) are actually associated or, when they become backlogged, will be associated to one of the threads in the group of the just-added leader.

Each of these queues then moves from idle, or non-existing, to backlogged when its first request eventually arrives. At that time, V-BFQ can easily raise the weight of the queue without even needing complex rescheduling operations, because the queue is of course not yet scheduled for service. In this respect, there is however a last subtlety to consider. I/O threads naturally tend to perform random I/O. In fact, even if the original I/O pattern in a guest is sequential, QEMU spawns several I/O threads and each I/O thread will happen to read or write only a chunk of the whole I/O to perform. The merge of these chunks covers a contiguous portion of the virtual disk, but, served separately by each I/O thread, these chunks happens to be located at random positions on the virtual disk.

To still achieve a high throughput also in the presence of this *fragmented* I/O, BFQ merges queues when it detects that they are associated to I/O threads whose merged I/O pattern would be sequential. In particular, this queue merging is realized by choosing a candidate shared queue and redirecting requests arriving from all the I/O threads to the same shared queue. Such a shared queue preserves its original association with a PID. Hence, it may happen that requests coming from I/O threads with a different PID than that stored in a shared queue are however redirected to the queue. In the end, when a new request arrives, to check whether the destination queue is to be weight-raised, it is the PID of the thread making the request to be checked, and not the PID associated to the destination queue.

The steps needed to perform the above control are reported in the function described with pseudo-code in Figure 8. Note that this function also takes care of properly pruning the *leader_pid_list* if needed.

We have now all the elements we need to describe the proper way to extend the BFQ hook, *insert_request()*, invoked to add a new request to a queue, so as to weight-raise a queue associated to an I/O thread to be privileged, when the queue moves from idle or non-existing to backlogged. This function is described in Figure 9.

```
privileged_thread(pid) {
  for_each_element(leader_pid_list) {
  // at each iteration. leader is set to one of
      the elems
    if (thread_no_more_existing(leader)) //
        thread is dead
    rm_from_list(leader_pid_list, leader); //
        remove from list

  for_each_child_thread(leader)
  // at each iteration child is set to one of
      the child threads
    if (pid == child->pid)
      return true ;
  }
  return false ;
}
```

Figure 8. Pseudo-code of the function *privileged_thread*

```
insert_request(request) {
  // rest of the code of the function
  bfqq = get_destination_queue(request);
  if (was_idle_or_nonexisting(bfqq) &&
      privileged_thread(current_thread->pid))
    if (is_already_raised(bfqq))
      move_forward_raising_start_time(bfqq) ;
    else
      perform_core_weight_raising_operations(bfqq)
          ;

  // rest of the code of the function
}
```

Figure 9. Pseudo-code of the function *insert_request()*

There is a final, important issue to consider: a request arrival can be intercepted even before the function *insert_request()* is invoked. In fact a preliminary BFQ hook is invoked just after a thread has obtained an I/O request from the pool of available requests, and has initialized the fields of the requests. In this hook, BFQ inspects the request, and from this inspection it may discover that: 1) the request comes from a thread whose requests are being redirected to a shared queue, but 2) this redirection is not needed anymore (see the code of BFQ for details [16]). If this happens, the code-path that will then be followed in the function *insert_request()* does not pass through the extension described in the code snippet Figure 9. A special *split* portion of the code of the *insert_request()* function is instead executed, to redirect again the requests coming from that thread to the original queue. In this code-path a *resume_state()* function is called to correctly resume the state of this original queue. Accordingly, to properly handle weight-raising for privileged threads also in this special case, we added the code shown in Figure 9 also to the function *resume_state()*.

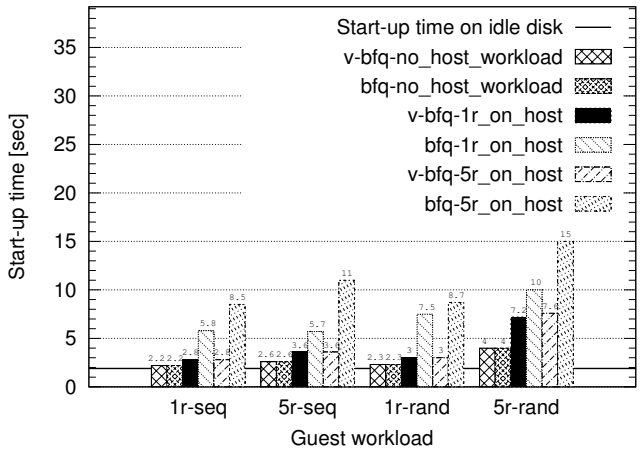

Figure 10. Results with the hard disk and V-BFQ as disk scheduler in both the guest and the host, compared against BFQ as disk scheduler in both the guest and the host (lower is better).

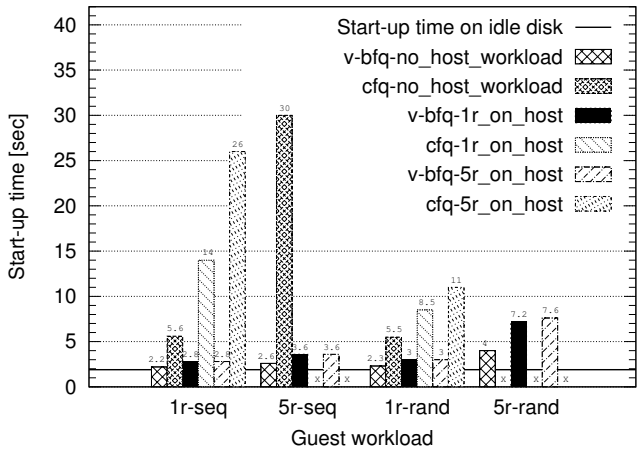

Figure 11. Results with the hard disk and V-BFQ as disk scheduler in both the guest and the host, compared against CFQ as disk scheduler in the guest and BFQ as disk scheduler in the host (lower is better)

VIII. Experimental results with V-BFQ

We repeated the same experiments as in Section V. In particular, as for throughput, V-BFQ trivially achieved the same performance as BFQ, which, in its turn, achieved optimal performance. Hence, for brevity, in this document we do not report throughput results for V-BFQ. Along the same line, we dot not report results for the workloads for which the actual service received by applications has not much to do with the decisions made by the disk schedulers, namely workloads containing greedy writers. And for the scenario where CFQ is used as disk scheduler in the host, because results do not vary significantly depending on whether BFQ or CFQ is used in the host.

A. Results with the hard disk

Figure 10 shows the start-up time recorded in case V-BFQ is used in both the guest and the host. As a reference, in the figure these results are compared against the ones achieved in case BFQ is used in both the guest and the host.

The effectiveness of V-BFQ is evident with *1r-seq*, *5r-seq* and *1r-rand*: regardless of the host workload, with 1r-seq V-BFQ guarantees about the same start-up time as if both the virtual and physical disk were idle. Even with *5r-seq* and *1r-seq*, start-up times are comparable to those recorded when both the virtual and the physical disk are idle.

Start-up times are sensitive to the host workloads with *5r-rand*. In fact, with these workload the issues already highlighted in Section V-C interfere with the correct operation of the heuristics in both the host and the guest V-BFQ schedulers.

To compare the responsiveness achieved by V-BFQ against the one experienced with a typical Linux disk-scheduling configuration, in Figure 11 we compare the start-up times achieved by V-BFQ (i.e., the same values already reported in Figure 10) against the ones recorded when CFQ, i.e., the default Linux I/O scheduler, is used as disk scheduler in the guest. As in Section V, the symbol **X** is used to indicate that the experiment failed because the application did not start within a 60-second timeout. The figure clearly shows the remarkable benefits provided by V-BFQ.

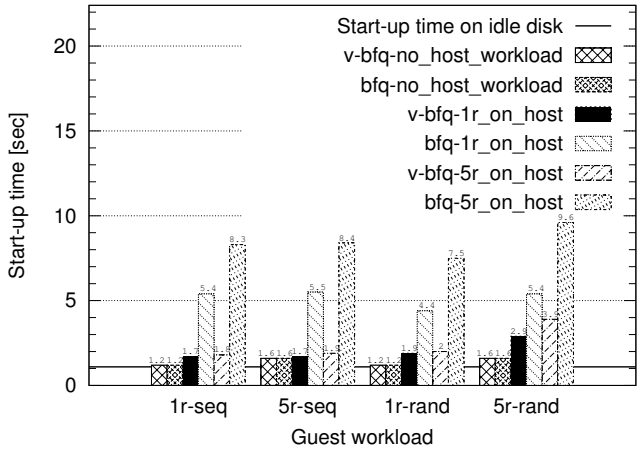

Figure 12. Results with the microSDHC Card and V-BFQ as disk scheduler in both the guest and the host, compared against BFQ as disk scheduler in both the guest and the host (lower is better).

B. Results with the microSDHC Card

As shown in Figures 12 and 13, with the microSDHC Card, results are along the same line as with the hard disk.

C. Results with the eMMC

Finally, also with the eMMC, V-BFQ achieved the same near-optimal performance as with the other two storage devices.

IX. Other schedulers

We also compared V-BFQ with two other schedulers for Linux: Deadline and NOOP in order to point out that V-BFQ is more responsive than all standard I/O schedulers for Linux. The scenario of these experiments is the same in Section VIII and therefore the results can be compared. Only the results for sequential readers are reported.

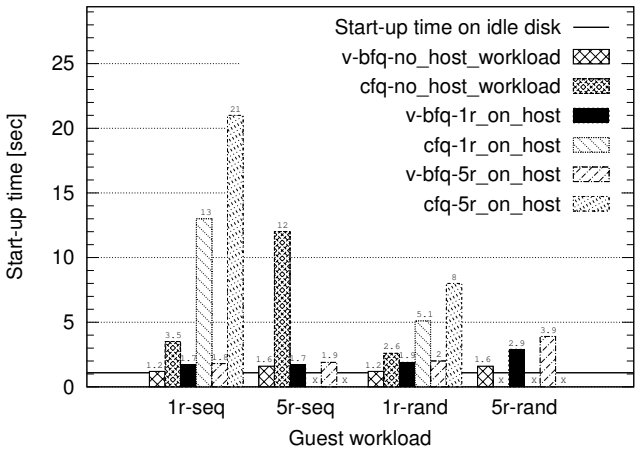

Figure 13. Results with the microSDHC Card and V-BFQ as disk scheduler in both the guest and the host, compared against CFQ as disk scheduler in the guest and BFQ as disk scheduler in the host (lower is better)

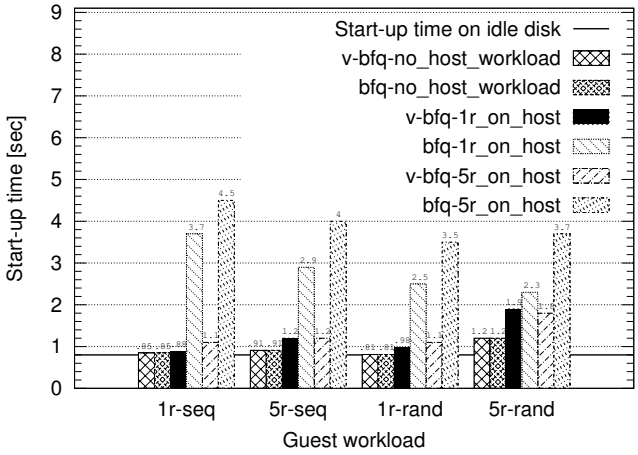

Figure 14. Results with the eMMC and V-BFQ as disk scheduler in both the guest and the host, compared against BFQ as disk scheduler in both the guest and the host (lower is better)

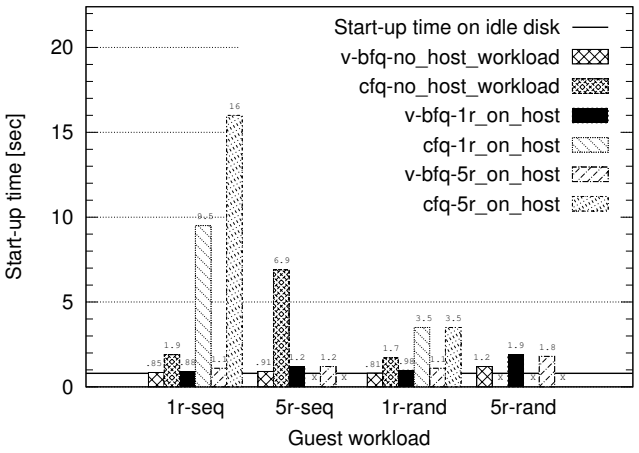

Figure 15. Results with the eMMC and V-BFQ as disk scheduler in both the guest and the host, compared against CFQ as disk scheduler in the guest and BFQ as disk scheduler in the host (lower is better).

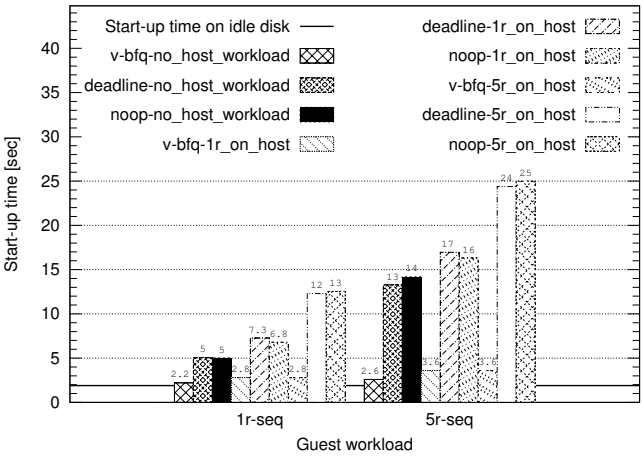

Figure 16. Results with the hard disk and V-BFQ as disk scheduler in both the guest and the host, compared against Deadline and NOOP as disk scheduler in the guest and BFQ as disk scheduler in the host (lower is better).

As it can be seen in Figures 16, 17 and 18 with Deadline or NOOP scheduler in the guest, the start-up time (of *xterm* application) is better than with CFQ as guest scheduler (Figures 11, 13, and 15). And the start-up time for Deadline and NOOP scheduler are roughly equivalent whatever the medium used and the workload. But the latency with V-BFQ as guest scheduler is always lower.

X. Conclusion and Future Work

In this paper, we have shown both theoretically and experimentally that responsiveness guarantees, as related to storage I/O, may be violated in virtualized environments. Even with schedulers, which target to achieve low latency through heuristics, the problem of low responsiveness still persists in virtual machines. The host receives a mix of interactive and background workloads from the guest, which can completely contradict per process heuristics by schedulers such as BFQ. That is why, we have devised a solution, based on BFQ, for preserving responsiveness also in virtualized environments, specifically for embedded systems and the KVM on ARM hypervisor: V-BFQ. This solution introduces the concept of coordinated scheduling between the host/guest scheduler and KVM itself. V-BFQ lived up to its expected performance improvements, guaranteeing high application responsiveness in a virtualized environment, also in the presence of heavy background workloads in both the guest and the host virtual and physical storage devices. Besides, the general scheme adopted to define V-BFQ from BFQ shall be extended and applied also to schedulers of other, important resources, such as CPUs and transmission links. We also plan to extend our investigation to latency guarantees for soft real-time applications (such as audio and video players), and to consider more complex scenarios, such as more than one VM competing for the storage device.

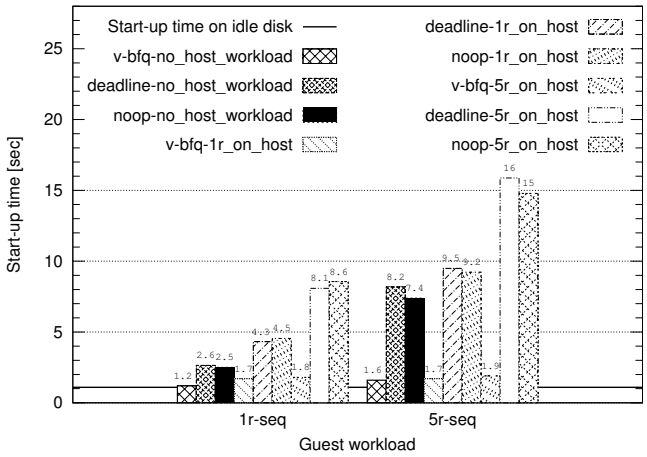

Figure 17. Results with the microSDHC and V-BFQ as disk scheduler in both the guest and the host, compared against Deadline and NOOP as disk scheduler in the guest and BFQ as disk scheduler in the host (lower is better).

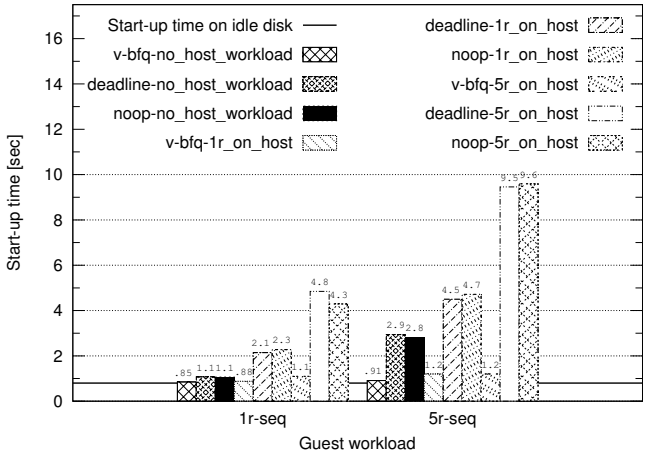

Figure 18. Results with the eMMC and V-BFQ as disk scheduler in both the guest and the host, compared against Deadline and NOOP as disk scheduler in the guest and BFQ as disk scheduler in the host (lower is better).

ACKNOWLEDGMENTS

This research work has been supported by the SeventhFramework Programme (FP7/2007-2013) of the European Community under the grant agreement no. 610640 for the DREAMS project. The authors would like to thank Paolo Valente for providing details about BFQ and his support for the benchmark suite. We also thank the anonymous reviewers for their precious feedback and comments on the manuscript.

REFERENCES

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

[2] Storage I/O Control Technical Overview [retrieved: November, 2014]. http://www.vmware.com/files/pdf/techpaper/VMW-vSphere41-SIOC.pdf

[3] Virtual disk QoS settings in XenEnterprise [retrieved: November, 2014]. http://docs.vmd.citrix.com/XenServer/4.0.1/reference/ch04s02.html

[4] M. Kesavan, A. Gavrilovska, and K. Schwan,"On disk I/O scheduling in virtual machines," in Proceedings of the 2nd conference on I/O virtualization, USENIX Association, 2010, p. 6.

[5] J. Shafer, "I/O virtualization bottlenecks in cloud computing today," Proceedings of the 2nd conference on I/O virtualization. USENIX Association, 2010, p. 5.

[6] D. Boutcher and A. Chandra, "Does virtualization make disk scheduling passé?," ACM SIGOPS Operating Systems Review 44.1, 2010, pp. 20-24.

[7] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, IEEE, 2010, pp. 51-58.

[8] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on, IEEE, 2013, pp. 233-240.

[9] J. Lee et al., "Realizing compositional scheduling through virtualization," IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12), April 2012, pp. 237-246.

[10] K. Sandstrom, A. Vulgarakis, M. Lindgren, and T. Nolte, "Virtualization technologies in embedded real-time systems," Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on, Sept. 2013, pp. 1-8.

[11] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization," Journal of Software Engineering and Applications, Vol. 5 No. 4, 2012, pp. 277-290.

[12] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for I/O performance," in Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM, 2009, pp. 101-110.

[13] X. Ling, H. Jin, S. Ibrahim, W. Cao, and S. Wu, "Efficient disk I/O scheduling with QoS guarantee for xen-based hosting platforms," in Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on, IEEE, 2012, pp. 81-89.

[14] F. Checconi and P. Valente, "High throughput disk scheduling hwith fair bandwidth distribution," IEEE Transactions on Computers, vol. 59, no. 9, May 2010, pp. 1172-1186.

[15] CFQ I/O Scheduler [retrieved: November, 2014]. http://lca2007.linux.org.au/talk/123.html

[16] BFQ homepage [retrieved: November, 2014]. http://algo.ing.unimo.it/people/paolo/disk_sched/

[17] 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.

[18] Virtual-BFQ homepage [retrieved: November, 2014]. http://www.virtualopensystems.com/en/products/virtual-bfq/