Grant Agreement number: 288574

Project acronym: **vIrtical**

Project title: SW/HW extensions for virtualized heterogeneous multicore platforms

Seventh Framework Programme

Funding Scheme: Collaborative project

FP7 -ICT -2011-7

Objective ICT-2011.3.4 Computing Systems

Start date of project: 15/07/2011 Duration: 36 months

D 3.6 *I/O Scheduler for virtualized systems*

Due date of deliverable: M30

Actual submission date: M31

Organization name of lead beneficiary and contributors for this deliverable: VOSYS
Work package contributing to the Deliverable: VOSYS

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

APPROVED BY:

Partners	**Date**
All	**6th February 2014**

INDEX: 1

Abstract

Preserving responsiveness is an enabling condition for running interactive applications effectively in virtual machines. This goal is the focus of the research activity whose steps and results are reported in this document. In particular, in this deliverable we provide:

- An introduction to the concept
- A short description of the BFQ storage-I/O scheduler and of its properties
- Experimental results of two different schedulers in normal and virtualized systems
- Overview of the coordinated scheduling concept and extension
- Experimental results highlighting the advantage of coordinated scheduling mechanisms

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, it is necessary to guarantee for them a high, or at least acceptable, responsiveness. In other words, it is necessary to guarantee that interactive 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. Expectedly, there is interest and active research in preserving a low latency in virtualized environments [5], [6], [7], [8], [9], especially in soft and hard real-time contexts. In particular, some virtualization solutions provide more or less sophisticated QoS mechanisms also for storage I/O [8], [9]. However, several important elements are missing in the picture:

- The first, critical element needed to provide latency guarantees in storage-I/O is an accurate storage-I/O scheduler. Yet a very accurate scheduler for virtualized environments is apparently missing.
- A thorough investigation on application responsiveness, as related to storage-I/O latency. In this deliverable we address this issue by integrating an effective disk scheduler in guaranteeing a low I/O latency in QEMU/KVM on ARM.

1.1 Contributions provided in this deliverable

For brevity, when not otherwise specified, in this document we use the generic term *disk* to refer to both a rotational and a non-rotational storage device. An effective disk scheduler in guaranteeing a low I/O latency is available in the literature: Budget Fair Queuing (BFQ) [1]. In particular, BFQ has been shown to outperform research and production schedulers in terms of both latency and throughput [1], [2]. A production-quality implementation of this scheduler, for the Linux operating system, is available as well [3].

BFQ has been however defined and tested only for non-virtualized environments. Hence, to provide low-latency I/O guarantees also in a virtualized environment, it is necessary to integrate BFQ in a Linux guest OS running in a VM emulated by QEMU/KVM on ARM. In particular, as detailed in this document, we implemented through the following main steps:

- Simple application of the patches that introduce BFQ [3] to the kernel sources for the guest OS. In particular, we worked on Linux 3.12, and used a patch series that introduced a BFQ-v7 candidate release.
- Improvements of the original version of BFQ to let it work correctly in a guest OS inside QEMU/KVM on ARM. These improvements have been integrated in the final v7 release of BFQ.
- Extension of an I/O benchmark suite for virtualized environments
- Experimental results of I/O scheduling

Using this improved version of BFQ and the current default Linux disk scheduler as a reference, we show an important problem: In virtualized environments there is a missing link exactly in the chain of actions performed to guarantee a low storage-I/O latency. In other words, a low latency may not be guaranteed even if an accurate disk scheduler like BFQ is used. Finally, the lessons learned from the analysis carried out are the starting point for extending BFQ so as to preserve I/O latency guarantees also in virtualized environments implemented through QEMU/KVM on ARM.

2 Budget Fair Queuing

In this section we describe the main characteristics of BFQ. In the investigation described in this work, we focus mostly on application-responsiveness issues as related to storage I/O. Budget Fair Queuing (BFQ) is a proportional-share disk scheduling algorithm that provides the following service properties.

- **Low latency for interactive applications:** Whatever the background load is, for interactive tasks the disk is virtually as responsive as if it was idle. For example, even if one or more large files are being read or written, or a tree of source files is being compiled, or else one or more virtual machines are performing I/O, starting a command/application or loading a file from within an application takes about the same time as if the disk was idle. As a comparison, with CFQ, NOOP, DEADLINE or SIO, and under the same conditions, applications experience high latencies, or even become unresponsive until the background workload terminates (especially on SSDs).
- **Low latency for soft real-time applications:** Also soft real-time applications, such as audio and video players, or audio- and video-streaming applications, enjoy about the same latencies regardless of the disk load. As a consequence, the presence of a background workload does not cause almost any glitch to these applications.
- **High throughput:** BFQ achieves up to 30% higher throughput than CFQ on hard disks with most parallel workloads, and about the same throughput with the rest of the workloads we have considered. BFQ achieves the same throughput as CFQ, NOOP, DEADLINE and SIO on SSDs.
- **Strong fairness guarantees:** As for long-term guarantees, BFQ distributes the disk throughput as desired to disk-bound applications (and not just the disk time), with any workload and independently of the disk parameters.

In the next section we show how BFQ works, and, in particular, how BFQ guarantees a low latency for both interactive and soft real-time applications. Understanding how BFQ achieves the latter goal, is key to understand why BFQ may *fail* to provide comparable low-latency guarantees also in a virtual environment. And to devise solutions that re-enable BFQ to provide those guarantees.

2.1 System model and terminology

We consider a *storage system* made of: a storage device, which hereafter we call just disk for brevity, a set of N applications to serve, and the BFQ scheduler in-between. The disk device contains one disk, modeled as a sequence of contiguous, fixed-size *sectors*, each identified by its *position* in the sequence.

The disk serves two types of *disk requests*: reading and writing a set of contiguous sectors. We say that a request is *sequential/random* with respect to another request, if the first sector (to read or write) of the request is/is not located just after the last sector of the other request.

Requests are issued by the N applications, which represent the possible entities that can compete for disk access in a real system, as, e.g., *threads* or *processes*. We define the set of pending requests for an application as the *backlog* of the application. We say that an application is *backlogged* if its backlog is not empty, and *idle* otherwise. For brevity, we denote an application as *sequential* or *random* if most times the next request it issues is sequential or random with respect to the previous one, respectively. We say that an application is *receiving service* from the storage system if one of its requests is currently being served.

2.2 Main algorithm

In this section we outline the main BFQ algorithm, leaving aside low-latency heuristics, which we describe in the following sections. A detailed description of BFQ is available in [1] (where the latest version of BFQ is named BFQ+). The logical scheme of BFQ is shown in the following figure used below.

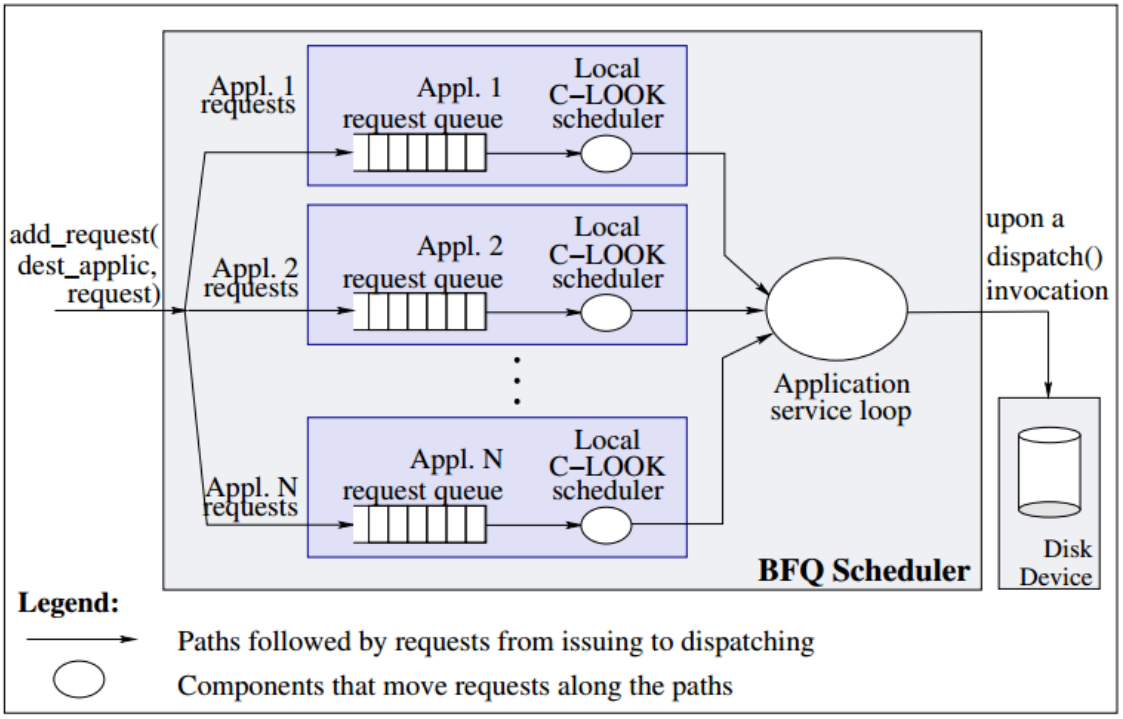

Figure 1. BFQ internals

BFQ grants exclusive access to the disk to each application for a while, and implements this service model by associating every application with a *budget*, measured in number of sectors. After an application is selected for service (in the *application service loop* depicted in the figure), its requests are dispatched to the disk one after the other, in C-LOOK order, and the budget of the application is decremented by the size of each request dispatched. The application is *deactivated*, i.e., its service is suspended, only if one of the following events occurs: 1) the application finishes its budget, 2) a special *budget timeout* fires, 3) the application becomes idle. Actually, an additional condition is needed for the third event to cause the suspension of the service of an application. For brevity in this document we neglect this detail, related to throughput and service-guarantee issues.

When an application is deactivated, BFQ performs two actions: it assigns a new budget to the application and chooses the next application to serve through an internal weighted fair-queuing scheduler, called B-WF2Q+. We provide a few details first on B-WF2Q+, and then on how BFQ computes the new budget of a just-deactivated application.

Under BFQ each application is associated with a fixed *weight*. If an application is not assigned a weight explicitly, then BFQ sets the weight of the application to a common, system-wide value. B-WF2Q+ schedules applications in such a way that each application receives, in the long term, a fraction of the disk throughput equal to the weight of the application, divided by the sum of the weights of the other applications competing for the disk.

B-WF2Q+ guarantees to each application the above fraction of the disk throughput *regardless of the budgets assigned to the application*. This guarantee may seem counterintuitive at a first glance, because the larger the budget assigned to an application is, the longer the application will use the disk once granted access to it. But B-WF2Q+ balances this fact by serving each application with a frequency inversely proportional to the budgets assigned to the application (see the original paper on BFQ [2] for full details).

This is a very important property, for two reasons. First, with any budget-by-budget fair-queuing scheduler, the deviation (request-service delay, service lag, etc.) with respect to an ideal, perfectly fair service is proportional to the maximum budget assigned to applications. In this respect, if the fraction of the throughput guaranteed to each application was proportional to the

budget assigned to the application (as it happens, e.g., with *quanta* in a round-robin algorithm) then an application should be assigned a large budget to get a high fraction of the throughput. At the end, the higher the fraction of the throughput assigned to some applications is, the larger should be the budget to assign to the application, and hence the aforementioned deviation. BFQ does not suffer from this problem, because there is no need to assign a large budget to an application to let the application receive a high fraction of the disk throughput.

Another benefit deriving from the independence between throughput distribution and budgets, it is the freedom to choose any application budget in BFQ that enables the optimization of other properties of the target applications. In this respect, a simple feedback-loop algorithm allows BFQ to achieve a high throughput while providing, tight short-term guarantees to time-sensitive applications.

2.3 Low latency for interactive applications

A system is responsive if it starts applications quickly and performs the tasks requested by interactive applications just as quickly. The first condition motivates the first step of the *low-latency* heuristic present in BFQ from v1: the weight of any newly-created application is raised to load the application promptly. In particular, the weight of the application is kept high for a time interval that: 1) depends on the disk speed and type (rotational or non-rotational), and 2) is equal to the time needed to load (startup) a large-size application on that disk, with cold caches and with no additional workload.

As for the second condition, i.e., executing the tasks requested by an interactive application quickly, if an application is interactive, then it blocks and waits for user input both after starting up and after executing a task. After a while, the user may trigger new operations, after which the application stops again, and so on. Accordingly, the low-latency heuristic raises again the weight of an application in case the application issues new requests after being idle for a sufficiently long (configurable) time. The weight is kept high for the same time interval as for a just-created application.

2.4 Low latency for soft real-time applications

To guarantee a low latency to soft real-time applications, BFQ uses a further heuristic, based on the same principle as the previous heuristic: raising the weight of the applications of interest. The first important difference with respect to the previous heuristic is that now the applications of interest, namely soft real-time applications, are detected as a function of their bandwidth requirements.

To be deemed as soft real-time, an application must meet two requirements. The first is that the application must not require an average bandwidth higher than the approximate bandwidth required to playback or record a compressed high-definition video. The second requirement is that the request pattern of the application is isochronous, i.e., that, after issuing a request or a batch of requests, the application stops for a while, then issues a new batch, and so on.

Unfortunately, even a greedy application may happen to behave in an isochronous way if several processes are competing for the CPUs. In fact, in this scenario the application stops issuing requests while the CPUs are busy serving other processes, then restarts, then stops again for a while, and so on. In addition, if the disk achieves a low enough throughput with the request pattern issued by the application, then the above bandwidth requirement may happen to be met too. As a consequence, to prevent such a greedy application to be deemed as soft real-time, a further rule is used: after becoming idle, the application must not issue a new request before the following time has elapsed: the maximum time for which the arrival of a request is waited for when a sync queue becomes idle. This filters out greedy applications, as the latter issue instead their next request as soon as possible after the last one has been completed (in contrast, when a batch of requests is completed, a soft real-time application spends some time processing data).

3 Integration of BFQ in KVM-ARM

Our first step was to apply the patch series introducing the candidate BFQ-v7 to a 3.12 Linux kernel suitable for execution in a QEMU/KVM virtual machine on ARM. Hereafter we refer to such a virtual machine as just *the VM*. The patch series is applied cleanly, and the resulting guest operating system, hereafter referred to as just *the guest OS*, ran without problems on a Versatile Express board equipped with a Toshiba MK6006GA 1.8-inch hard disk.

Then we proceeded with the following, initial sanity checks about throughput and latency. We executed these tests in the above hardware as well.

3.1 Sanity checks

We measured, first, the average aggregate throughput achieved by BFQ in the VM while: 1) one of the following four workloads was being served inside the VM itself for 60 seconds, and 2) there was no additional workload in the host:

- **1r-seq**: One sequential reader (i.e., one file being read sequentially)
- **5r-seq**: Five parallel readers
- **1r-rand**: One random reader (i.e., one file being read at random positions)
- **5r-rand:** Five parallel random readers

As a term of comparison, for each workload we measured also the throughput achieved by CFQ, the Linux default scheduler. The results, reported in Figure 2, show that BFQ achieves the same performance, or even better performance than CFQ with all workloads. In particular, just an **X** is reported for the results of CFQ with random workloads. With this symbolism, we indicate that the test failed because, with that workload, the system became unresponsive and had only to be switched off. It was then impossible to correctly compute the average throughput at the (natural) end of test.

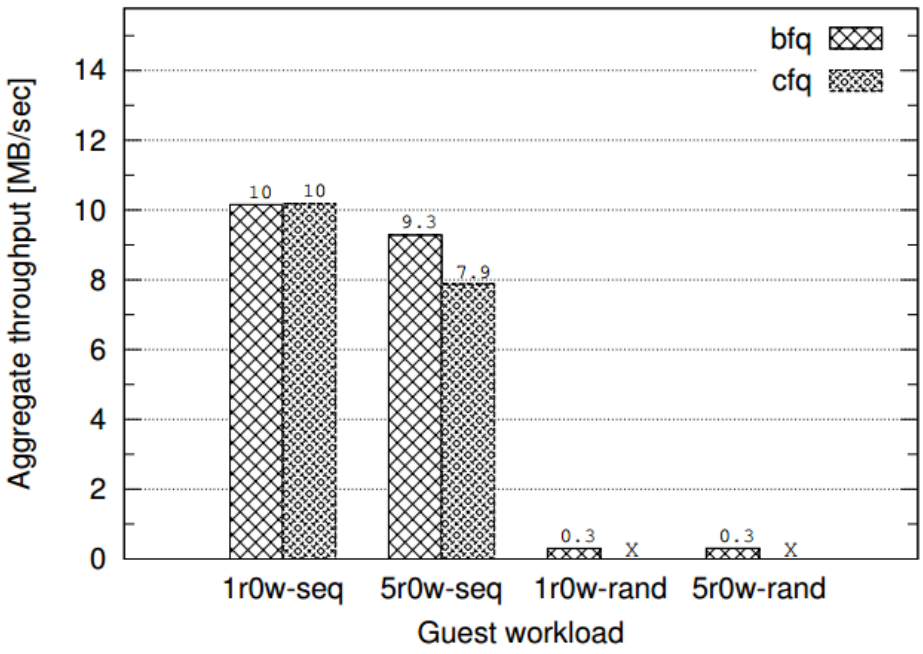

Figure 2. Throughput (higher is better)

Hence, in terms of throughput, BFQ behaved correctly. The next test was about application responsiveness. To this purpose we measured the average time needed to start the *xterm* terminal emulator, with cold caches, in the guest OS and while one of the above workloads was being served. We will provide full details about how we carried out these experiments and how we computed statistics in the following chapters. As for these preliminary tests, we are only

interested in checking for the presence of evident anomalies. Actually, as shown in Figure 3, BFQ does not behave correctly.

The reference line in Figure 3 shows the time needed to start up *xterm* in the guest in case the disk is idle. As can be seen, with sequential workloads, BFQ guarantees about the same start-up time also in presence of any of the two sequential workloads (as expected according to [1]). The performance of CFQ is instead much worse, as already shown and discussed in detail in [1].

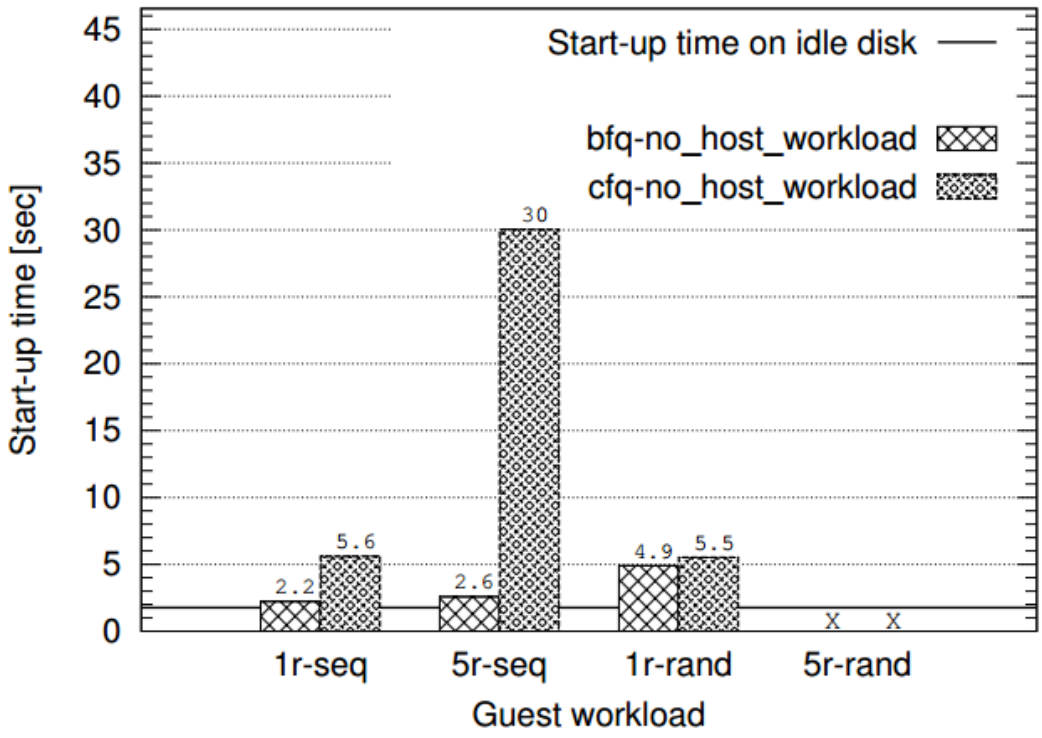

Figure 3. Start-up time of xterm (lower is better)

Problems arise with *1r-rand*, because in this scenario the start-up time guaranteed by BFQ becomes much higher than with an idle disk. With *5r-rand* an **X** is even reported in the figure for BFQ, which indicates again a failure of the test. In this case, a failure means that the start-up of the application is not completed within a 60-second timeout. In the end, with random workloads, BFQ has a performance similar to that of CFQ. This conflicts with the expected performance of BFQ according to [1], where BFQ has been shown to guarantee, also in presence of random workloads, a start-up time comparable to that with an idle disk. After some investigations we found both the cause of the problem and a solution, as explained in the next section.

3.2 Modifications needed for a correct integration of BFQ with QEMU/KVM on ARM

We verified the following facts:

- On an ARM system, the **HZ** parameter, i.e., the frequency of the reference clock ticks for the kernel, is typically set to 100 (the same parameter is typically set to 1000 on Intel systems).
- Clock ticks happen not always updated at regular time intervals, i.e., every 1/HZ seconds. In fact, they may be updated haltingly, with *jumps* of even several ticks.

These issues confuse BFQ heuristics for detecting soft real-time applications. In fact, because of the coarse time granularity and of the uneven tick updating, a greedy application generating

random I/O may happen to not be correctly deemed greedy. In more detail, according to the time measurements performed in the soft real-time heuristics, a larger time interval may seem to elapse before the application issues a new I/O request after becoming idle. This time interval may happen to be (deceptively) large enough to let BFQ not deem the application greedy.

As a consequence, if the application also meets the other requirements for soft real-time applications, then BFQ necessarily concludes that the application is a soft real-time one, and privileges it accordingly. Of course this results in a reduction of the fraction of the throughput that BFQ can then devote to other applications that should instead be correctly privileged. This is exactly what happens to *xterm* in the results with random workloads shown in Figure 3.

To contrast these unavoidably deceptive measures, we modified the heuristics by increasing the reference minimum time interval for not considering an application as greedy. In particular, after some trial and error, we found that increasing this time interval by four ticks was enough to reduce to a negligible value the probability to erroneously consider a greedy application as soft real-time. Figure 4 shows the resulting new performance of BFQ.

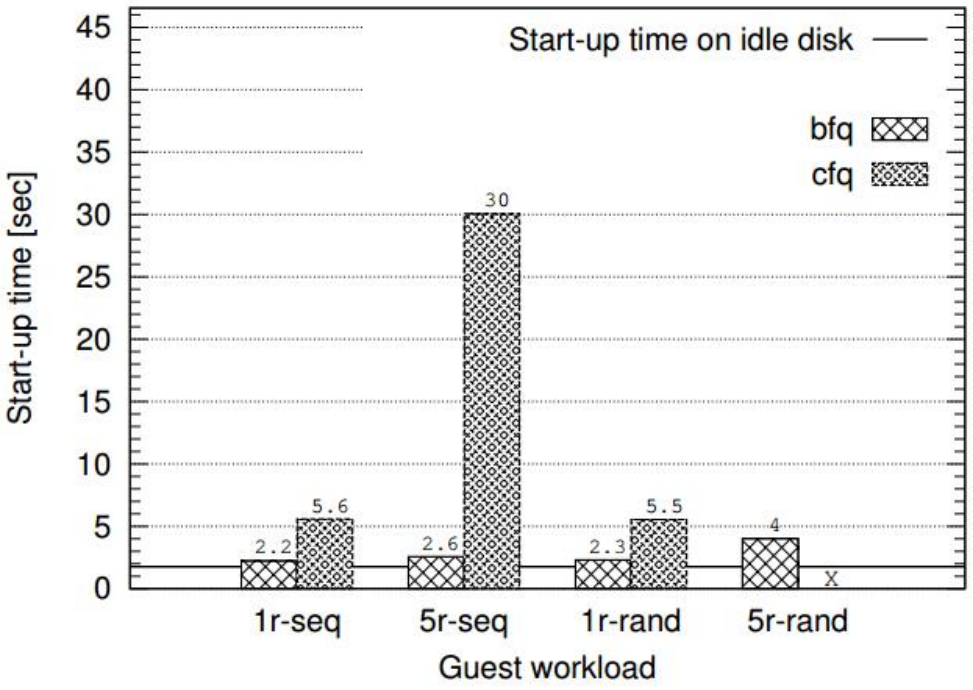

Figure 4. Start-up time of xterm (lower is better)

This special tuning to let BFQ operate correctly also inside kernels running in QEMU/KVM virtual machines on ARM is now included in the v7 version of BFQ. Using this last version of BFQ as a reference, in the next chapters we will report our investigations about whether a high responsiveness can actually be guaranteed in a virtualized environment. In particular, we will focus on less trivial scenarios than the one considered in this deliverable just to perform sanity checks.

4 Preserving low latency for storage-I/O in virtual environments

In order to provide a solution capable to guarantee a low storage-I/O latency in a virtualized environment, we start from integrating the BFQ disk scheduler in a Ubuntu distribution that can run as guest OS in a virtual machine emulated through QEMU/KVM on ARM (contribution described in the previous chapters). We remind that in this deliverable we use the generic term *disk* to refer, in general, to both rotational and non-rotational storage devices.

Using QEMU/KVM as the reference virtualization solution we carry out a detailed performance investigation aimed at assessing application responsiveness related to how storage I/O. Disk schedulers play a critical role in preserving such a service guarantee. In this investigation we use as a reference the improved version of BFQ, and, the current Linux default scheduler: CFQ [4]. Both solutions provide a relevant term of comparison, since they are two of the most effective schedulers in guaranteeing a high responsiveness, and production-quality implementations are available for both.

The purpose of the investigation first of all, shows that there is an important storage-I/O latency scheduling issue in virtualized environments. Second, we are going to describe the causes of such an issue. We carry out the investigation on ARM-based systems and KVM, one of the most popular and efficient virtualization solutions for ARM embedded systems. Modern embedded systems and consumer-electronics devices are based on ARM, and can execute applications with about the same I/O demand as general-purpose systems. In contrast, the storage devices used in former systems are necessarily slower than their typical counterparts for latter systems. Because of the lower speed of storage devices for embedded systems, I/O-latency issues are amplified.

To highlight these issues, we show first, 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. As for the experimental part, 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 may become from little to non-responsive at all. To carry out these experiments, we extended a publicly available I/O benchmark suite for Linux [3], to let it comply also with a virtualized environment.

5 Reference schedulers

The main problem shown in this deliverable is the loss of application-responsiveness guarantees, because of storage-I/O latency issues. To highlight this problem, we use the following two storage-I/O schedulers as a reference: BFQ [1] (more precisely, in the experiments we use the v7 version of BFQ, which we properly tuned to operate correctly also in a kernel running in a QEMU/KVM virtual machine on ARM) and CFQ [3]. We opted for these two schedulers because, first, both guarantee a high throughput and a low latency (in particular, as also shown by the results of our preliminary tests, BFQ may even achieve up to 30% higher throughput than CFQ on a hard disk). Strictly speaking, only the second feature is related to the focus of this deliverable, 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 [3]. In the next two sections we briefly describe both schedulers, focusing especially on the main differences between them in terms of I/O latency and responsiveness.

5.1 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 its strong fairness guarantees. BFQ distributes the disk throughput as requested 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 [1] as if the disk was idle.

5.2 CFQ

CFQ grants disk access to each application for a fixed *time slice*, and schedules slices in a round-robin fashion. Unfortunately, as shown in [1], 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 [1].

6 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.

To deal with this 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. Evidently, BFQ itself has to be better integrated in the host-guest system to achieve such a goal. This critical improvement is the purpose of the activity described in this deliverable.

7 Extension of the benchmark suite

To implement our experiments we used a publicly available benchmark suite [3] 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 while executing workloads made of the reading and/or the writing of multiple files at the same time, with each file accessed sequentially or randomly (i.e., with read/write requests scattered at random positions within each file). These workloads represent both typical disk I/O micro-benchmarks, and worst-case (heavy-load) scenarios for file download, upload and sharing. Hereafter, we refer to workloads like these when we say *heavy workloads*.

- **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. Besides, as for BFQ, according to how the low-latency heuristics of BFQ, under BFQ this quantity is in general a measure of the worst-case latency experienced by an interactive application every time it performs some I/O.

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 the extensions described in following sections. The resulting extended version of the benchmark suite is available here [10]. This new version of the suite also contains the general scripts that we used for executing the experiments reported in this deliverable (all these experiments can then be repeated easily).

7.1 Choice of the disk scheduler in the host

Not only the active disk scheduler for a given virtual disk in a guest operating system, hereafter abbreviated as just guest OS, is relevant for the I/O performance in the guest itself. In fact, also the disk scheduler in the host OS influences the actual order in which I/O requests are served on the disk corresponding to the guest virtual disk. We extended the benchmarks so as to allow the user to choose also the host disk scheduler when the benchmarks are executed in a guest OS.

7.2 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 the section about the experimental results.

7.3 Automatic 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 this additional host workload when they finish.

7.4 Appropriate naming scheme

The suite allows each benchmark to be repeated several times, and overall statistics across repetitions to be easily computed. In particular, the suite provides a script that automatically computes separate overall statistics for each benchmark and scenario. In more detail, the script takes as input a directory that is assumed to contain the output files produced by the benchmarks. If the directory contains results generated by different benchmarks, and/or for different scenarios, then, for each benchmark and each scenario, the script automatically computes overall statistics for all the repetitions executed for that benchmark and scenario.

To distinguish between results related to different benchmarks and scenarios, the script relies on the names of the output files generated by the benchmarks. In particular, for each possible scenario, benchmarks compose output-file names as a function of the values of the parameters

for that scenario, according to a well-defined scheme. In a virtualized environment, scenarios become more complex because also the active scheduler and the additional workload in execution in the host must be taken into account.

We addressed this issue in two steps. First, we extended the output-file naming scheme in such a way to take into account also the parameters related to these more complex scenarios and, at the same, to be backward-compatible with the previous naming scheme (which complies only with a non-virtualized environment). Second, we extended the benchmarks to generate correct output-file names according to this new scheme.

7.5 Appropriate overall statistics computation

Also the output of the script that computes overall statistics had to be improved. In fact, in this output, the overall statistics for each scenario are preceded by a header line describing the scenario itself. For a benchmark executed in a virtualized environment, this header line had to be integrated with more information, namely the name of the active disk scheduler and of description of the workload executed in the host.

8 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 1. 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.

Table 1. Storage devices used in the experiments.

Type	Name	Size – Read peak rate
1.8-inch Hard Disk	Toshiba MK6006GAH	60 GB – 10 MB/s
microSDHC Card	Transcend SDHC Class 6	8 GB – 16 MB/s
eMMC	SanDisk SEM16G	16GB – 70 MB/s

We performed two types of experiments: the first type was aimed at measuring the disk throughput, whereas the second type was aimed at measuring the start-up time of popular applications. In both types of experiments, one of a set of possible background-workload combinations in the guest and the host was executed (details below).

For each workload combination, we carried out experiments and computed statistics as follows. As for the throughput experiments, we computed, after 60 seconds of workload execution, the following statistics on throughput samples taken every 2 seconds: minimum, maximum, average, standard deviation and 95% confidence interval. As for the responsiveness test, we started the application at hand five times, and computed the same statistics as above over the measured start-up times. We denote as a *single run* any execution of one of these two types of experiments for some workload, i.e., either any 60-second execution of the throughput experiment, or any sequence of five invocations of the application at hand.

We repeated each single run ten times, and computed the same five statistics as above also across the average throughput or 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).

8.1 Disk throughput

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. For this benchmark, we considered only scenarios where the only disk I/O workload was generated by the VM. In fact, in a scenario where additional applications would compete for the disk in the host, the throughput in the guest would of course be limited, in ways not dependent on the behavior of the guest disk schedulers, by the fact that the host scheduler would have to provide part of the disk throughput also to the additional applications competing for the disk.
We measured the aggregate throughput in the VM while one of the following six workloads was being served in the guest, 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 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.

We found that:

- For each of the two schedulers, the throughput achieved in the guest while that scheduler was used in the guest was independent of which scheduler was used in the host.
- In presence of file writers, results were 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. In particular they depend on the fact that, when a certain threshold of dirty pages is reached in the Linux virtual memory system, pages start to be flushed to the disk at a high rate. This causes the limited pool of available I/O requests inside the kernel to be quickly saturated (only a limited number of I/O requests may be outstanding at the same time). From this point on, and until there is some process that continues to *dirty* memory pages, the actual I/O service guarantees experienced by an application may depend basically on how lucky that application is in grabbing available I/O requests from the saturated pool. The problem is further exacerbated by the fact that, due to metadata-updating delays, serialization issues may show up at the file-system level and cause some applications to become almost completely unresponsive.

In the end, given the above issues, we report our throughput results only with file readers, and only with BFQ as disk scheduler in the host. For completeness, we show however an example of the consequences of executing file writes in our results on start-up times.

As can be seen from Figures 5, and already commented in previous chapters, BFQ achieves the same performance as CFQ, or even a higher performance than CFQ with a hard disk. The **X** for random workloads with CFQ indicates that it was not possible to compute the average throughput because the VM became unresponsive.

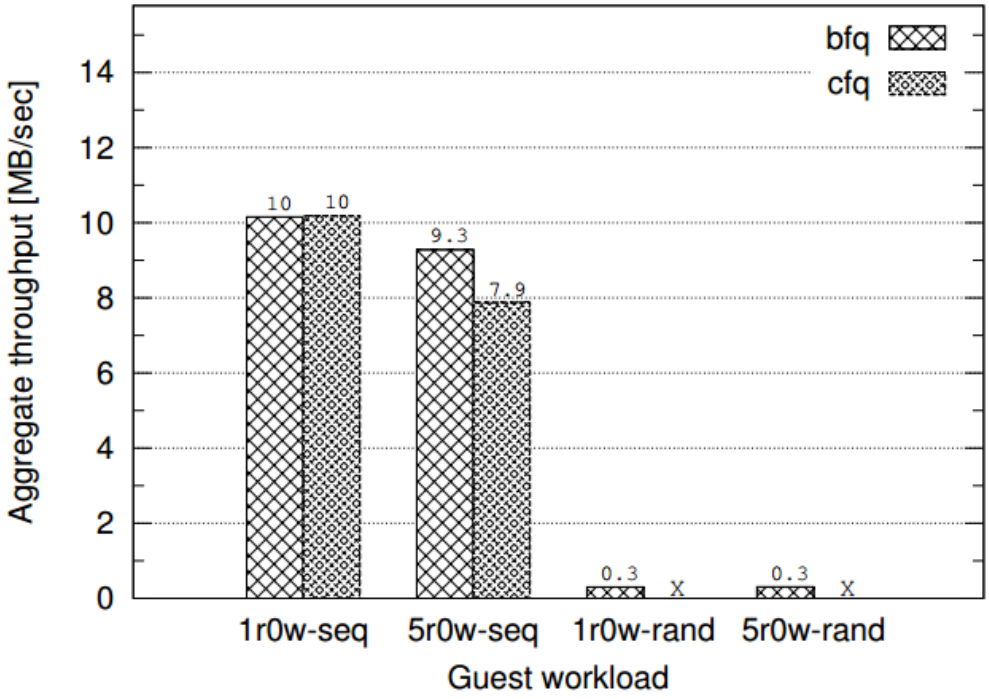

Figure 5. Throughput achieved on hard disk (higher is better)

Figures 6 and 7 show instead our results on the flash-based devices. With these devices both schedulers achieve about the same performance.

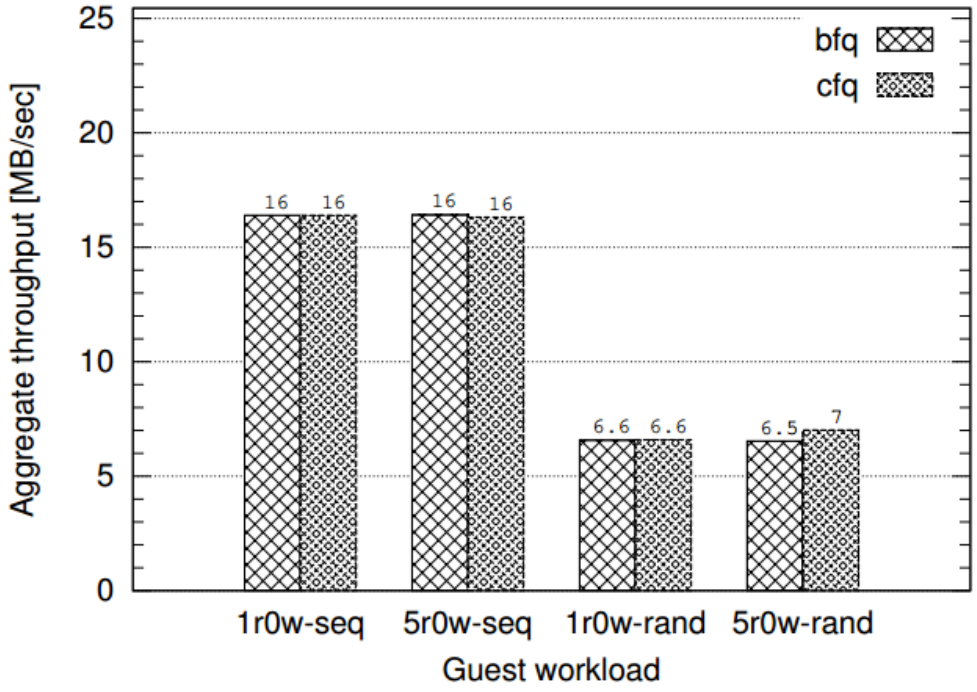

Figure 6. Throughput achieved on the microSDHC Card (higher is better)

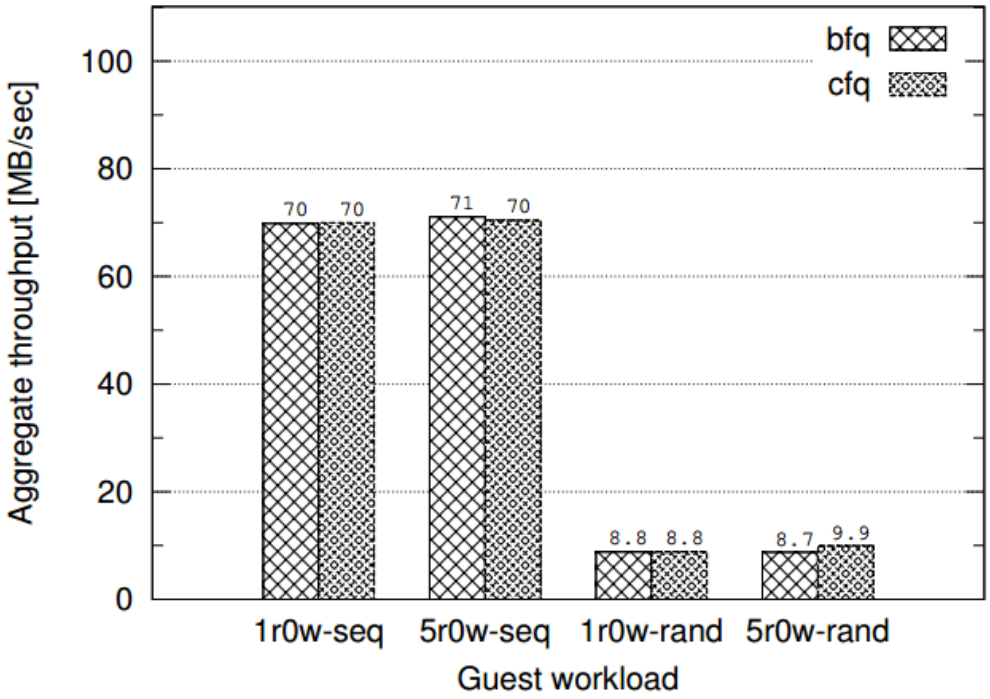

Figure 7. Throughput achieved on the eMMC (higher is better)

8.2 Application start-up time

We measured the start-up time of three popular interactive applications of different sizes, inside the VM and while one of the following combinations of workloads was being served.

- **In the guest.** One of the following six workloads (the same as for the throughput experiments), 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. As for the throughput experiments, 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.

Finally, 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 in [1], these applications allow their start-up time to be easily computed. For example, for **xterm** and **konsole**, the start-up time can be obtained by measuring the time elapsed since the invocation of these applications till when the applications contact the graphical server to have their window rendered.

To get worst-case start-up times, we dropped caches both in the guest and in the host before each invocation (using the new extended capabilities of the benchmark). 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 main problem is that responsiveness guarantees may be violated in a VM and occur regardless of which scheduler is used in the host. Besides, in presence of file writers, as it happened with the throughput experiments, results are dominated by fluctuations and anomalies caused by the Linux write-back mechanism. To show, separately and clearly: 1) the responsiveness problems that occur in any scenario where there is an additional workload in the host as well as the one generated by the VM, 2) the effects of changing the host scheduler, and 3) the anomalies occurring in presence of file writers, we report, separately, our results for the following subsets of workload combinations:

- Only file readers and only BFQ as disk scheduler in the host
- Only file readers and only CFQ as disk scheduler in the host
- Also file writers and only BFQ as disk scheduler in the host

Finally, for brevity we report our results only for **xterm**.

8.3 Only file readers and only BFQ as disk scheduler in the host

Figure 8 shows our results with the hard disk (Table 1). 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 KVM/QEMU VM.

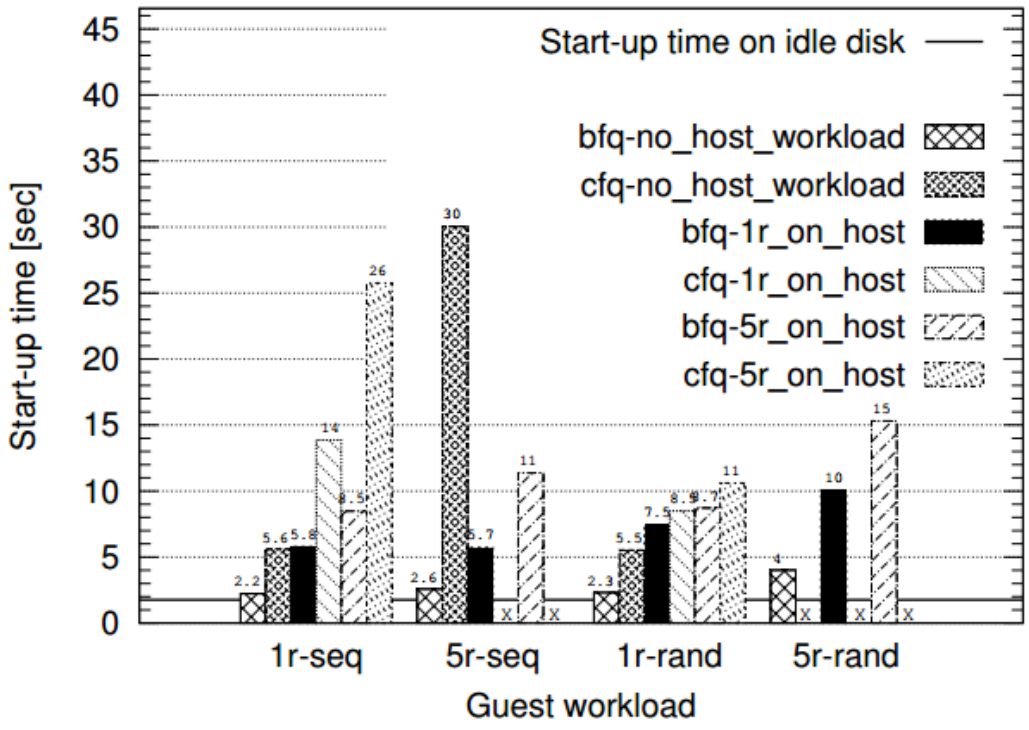

Figure 8. Results with the hard disk and BFQ as a disk scheduler on the host

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 already highlighted, 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 9 and 10 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. 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.

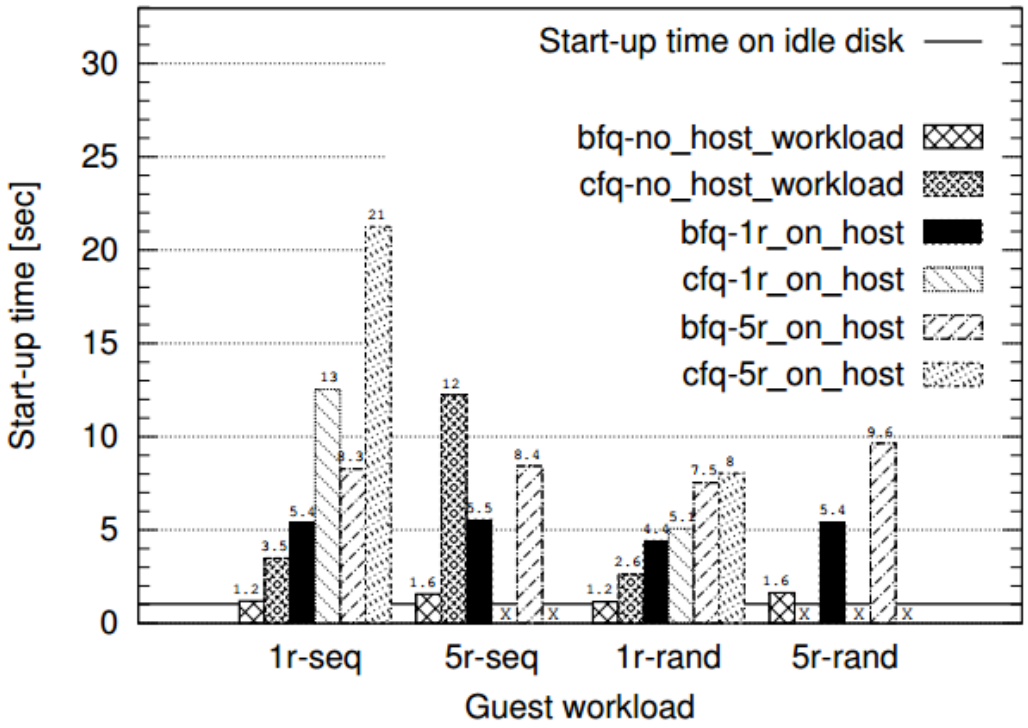

Figure 9. Results with the microSDHC Card and BFQ as disk scheduler on the host

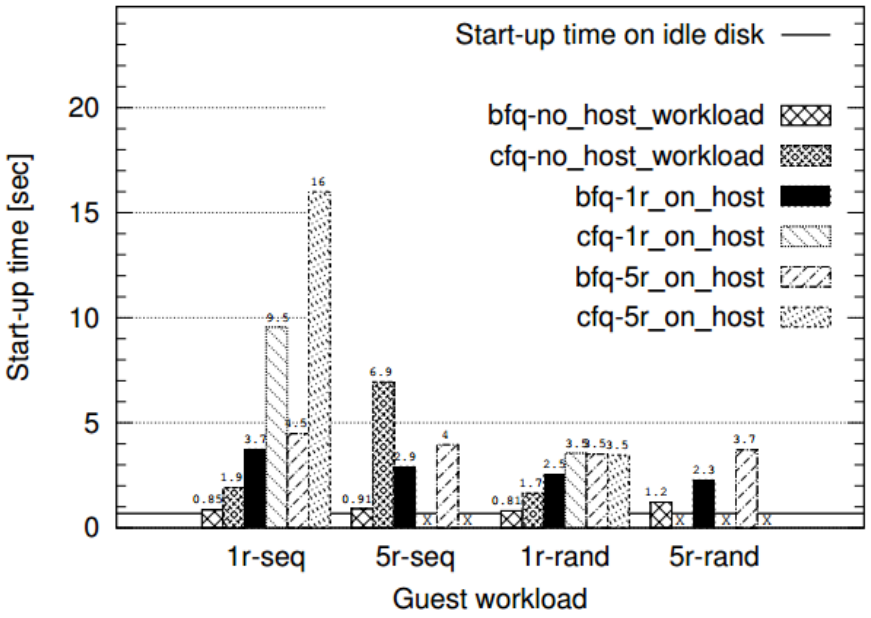

Figure 10. Results with the eMMC and BFQ as disk scheduler on the host

8.4 Also file writers and only BFQ as disk scheduler in the host

Finally, to show the anomalies occurring in presence of file writers, we report only our results with eMMC and with sequential file writers. In fact, with the other two slower devices or with random writers, it is difficult to appreciate any difference among the various scenarios, and between the two schedulers, because **xterm** just does not succeed in starting up before the 60-second timeout in almost any case (actually, most of the times the whole system becomes unresponsive as well).

Figure 11 shows our results with the eMMC and *2r2w-seq*. As a term of comparison, the figure shows also the start-up times (already shown in Figure 10) with *1r-seq* and *5r-seq*, i.e., with the workloads for which the schedulers achieve their best performance, as well as the results with *5r-rand*, i.e., the workload for which both schedulers achieve instead their worse performance. As can be seen, for each scenario, with *2r2w-seq* start-up times are extremely higher than with all the other workloads.

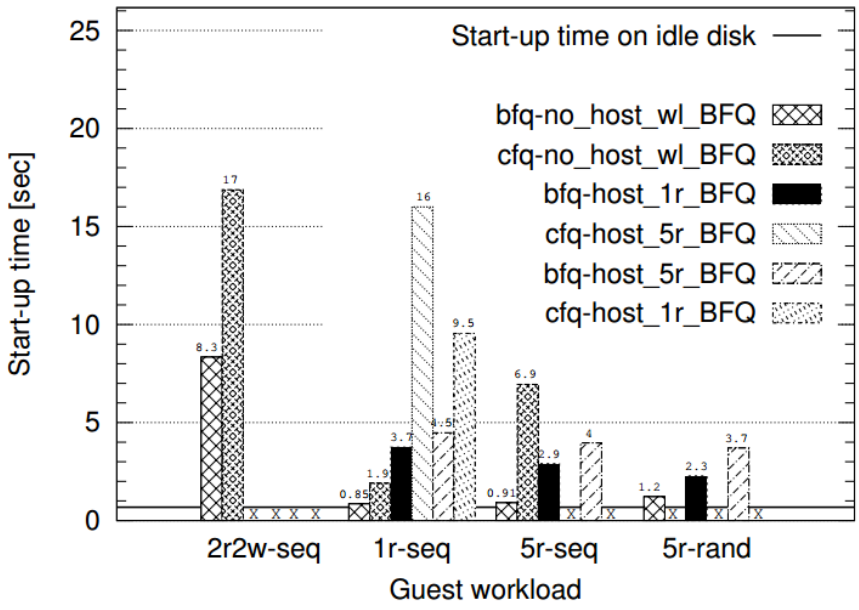

Figure 11. Results with the eMMC, file writers and BFQ as disk scheduler on the host

9 Extending BFQ with a coordinated scheduling mechanism

In this chapter we describe an extension of the BFQ storage I/O scheduler, which enables it to preserve a high application responsiveness in KVM virtual machines on ARM architectures. As shown in detail in this document, such an extension is the result of the combination of several effective and accurate design choices. According to the experimental results reported in this deliverable, 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, we manage to preserve in the guest a high application responsiveness.

Besides, the mechanisms that BFQ uses to guarantee a high responsiveness are exactly the same it uses to guarantee a low latency to soft real-time applications. Hence this kind of coordinated scheduling mechanism most certainly preserves also this additional important service guarantee in a virtualized environment.

9.1 BFQ host/guest extensions

As highlighted in previous chapters, preserving responsiveness is an enabling condition for running interactive applications effectively in virtual machines. For this enabling condition to be met, a low latency usually needs to be guaranteed to storage-I/O operations. In contrast, we showed that, in virtualized environments, there is a missing link exactly in the chain of actions performed to guarantee a low storage-I/O latency.

The solution described, fills this gap, by extending the BFQ storage I/O scheduler [1] as follows, on both the guest and the host side:

- **Guest side:** when the guest BFQ scheduler detects, through its internal heuristics, that some application needs urgent service from the virtual storage device, it communicates this need to the BFQ scheduler in the host. On the other end, the BFQ guest scheduler also communicates to the host scheduler when there are no more applications needing a quick service.
- **Host side:** when the host BFQ scheduler receives the above **help** request from one VM, it privileges the I/O requests coming from that VM, until the same VM tells the host BFQ scheduler that no help is needed anymore.

Such an extension can be implemented in several ways. In this document we provide, the solution that seems to provide most benefits. In this deliverable we describe the concept behind such extensions and experimental results in reducing I/O latency in virtualized systems.

We repeated, the same experiments that we performed for BFQ and CFQ. The gist of our results with an ultra-portable hard disk, a microSDHC Card and an eMMC device, is that, even in the presence of a heavy background workload on a guest virtual disk, plus a heavy additional background workload on the physical storage device corresponding to that virtual disk, BFQ coordinated scheduling does preserve a high application responsiveness in a KVM/QEMU guest. In particular, with, on one hand, the heaviest total workloads, and, on the other hand, the other total workloads we considered, the responsiveness achieved, respectively, are comparable to or about the same as if both the guest virtual disk and the host storage device were idle. Besides, we can note that BFQ guarantees a low latency to soft real-time applications [2] using exactly the same mechanisms that it uses to guarantee a high responsiveness.

9.2 QEMU I/O requests

We restart from the same simple example we reported previous chapters. 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.

In particular, QEMU handles I/O requests coming from the guest OS as follows. It first translates these I/O requests into filesystem-level read/write operations, for the files containing the images of the virtual disks of the VM interested by the I/O. These read/write operations are queued in a dedicated I/O-service queue, and a set of (host-side) I/O threads is dynamically created. I/O threads run in parallel, with each thread iteratively taking on serving the next read or write operation. In the end, not only the overall I/O generated by guest G may conceal the presence of the component related to application A being loaded, but the final pattern of I/O requests coming from the set of I/O threads may further modify the property of the original I/O pattern generated by guest G.

As a conclusion, the scheduler in the host has in general no hint to know that it would be better to privilege the I/O requests of the QEMU threads serving 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. The start-up time of the application may therefore be high. This is exactly the scenario that we investigate in our experiments.

9.3 Coordination between schedulers

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 previously, 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. We can describe this idea as follows:

- **Guest side:** when the guest scheduler detects, through its internal heuristics, that some application needs urgent service from the virtual disk, it communicates this need to the scheduler in the host. On the other hand, the guest scheduler also communicates to the host when there are no more applications needing a quick service.
- **Host side:** when the host scheduler receives the above *help* request from one VM, it privileges that VM until the same VM tells the host scheduler that no help is needed anymore.

From this scheme we can easily deduce that the communication between the guest and host extended 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.

10 Experimental results of coordinated scheduling

We repeated the same experiments as in Chapter 8. In particular, as for throughput, our solution trivially achieved the same performance as BFQ, which, in its turn, achieved optimal performance according the results reported. Hence, for brevity, in this document we do not report throughput results. Along the same line, we do 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. We reported and thoroughly commented on these results in Chapter 8.

10.1 Results with the hard disk

Figure 12 shows the start-up time recorded in case of he extended scheduler where it 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 is evident with *1r-seq*, *5r-seq* and *1r-rand*: regardless of the host workload, with *1r-seq* we observe 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 instead sensitive to the host workloads with *5r-rand*. In fact, with this workload the issues already highlighted in Chapter 8 interfere with the correct operation of the heuristics in both the host and the guest extended schedulers. These issues are basically the slightly coarse time granularity guaranteed to scheduled events in the kernel in an ARM embedded system, and the fact that the reference time itself may advance haltingly in a QEMU/KVM VM.

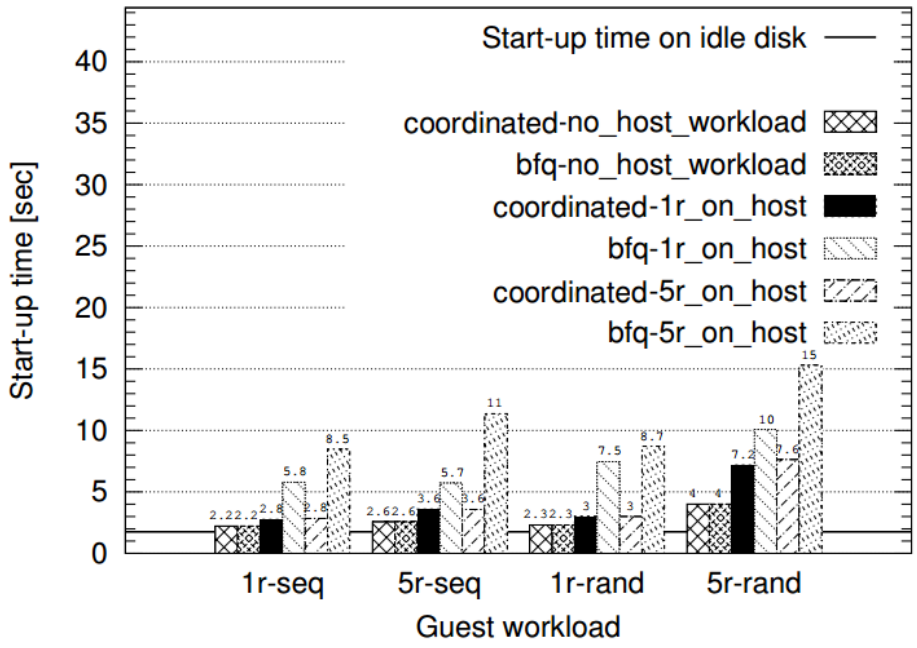

Figure 12. Hard disk results - Coordinated scheduling compared to BFQ

To compare the responsiveness achieved, against the one experienced with a typical Linux disk-scheduling configuration, in Figure 13 we compare the start-up times (i.e., the same values already reported in Figure 12) against the ones recorded when CFQ, i.e., the default Linux I/O scheduler, is used as disk scheduler in the guest. As in Chapter 8, 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 benefits provided by our implementation.

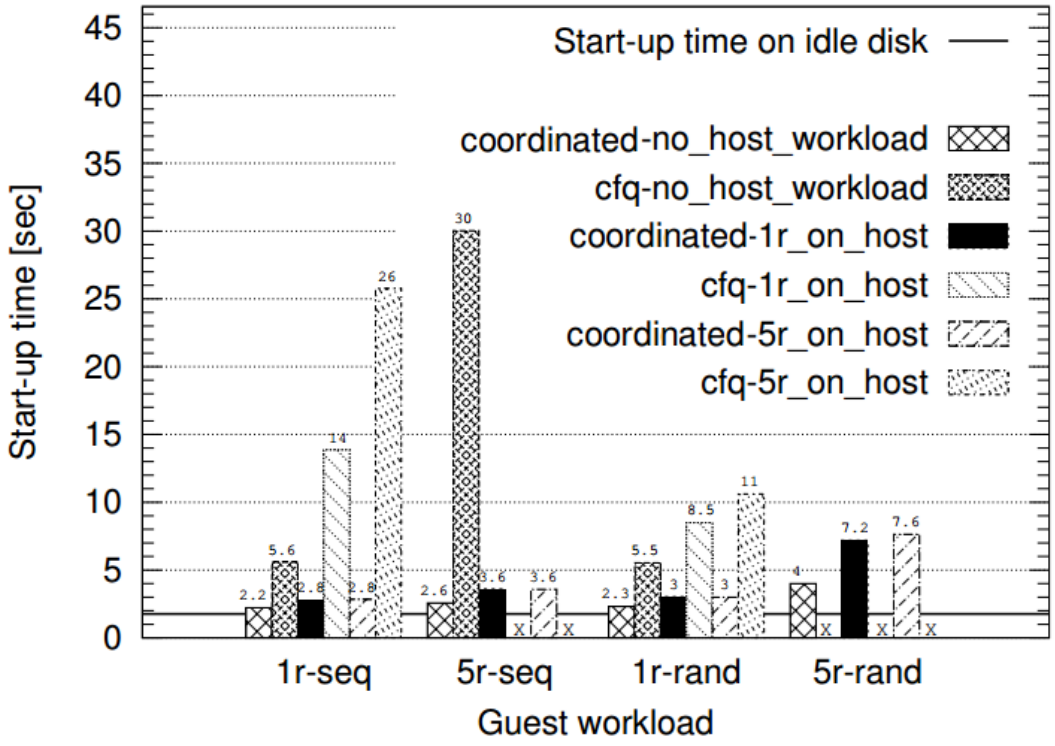

Figure 13. Hard disk results - Coordinated scheduling compared to CFQ

10.2 Results with the microSDHC Card

As shown in Figures 14 and 15, with the microSDHC Card results are along the same line as with the hard disk.

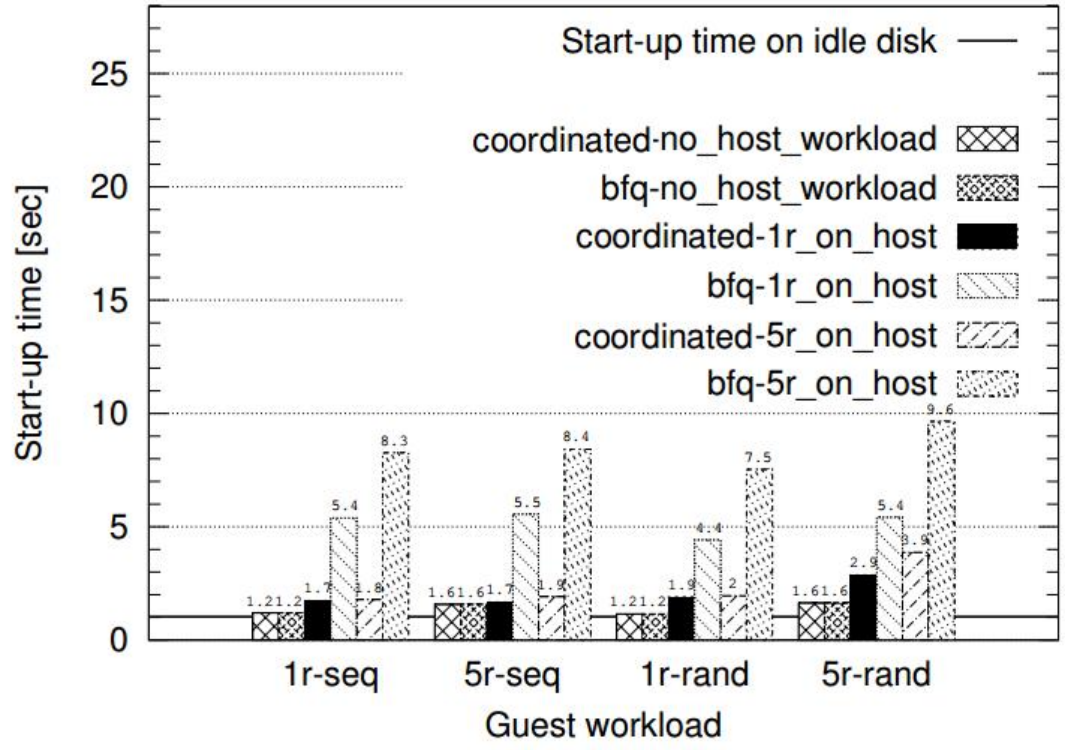

Figure 14. microSDHC results - Coordinated scheduling compared to BFQ

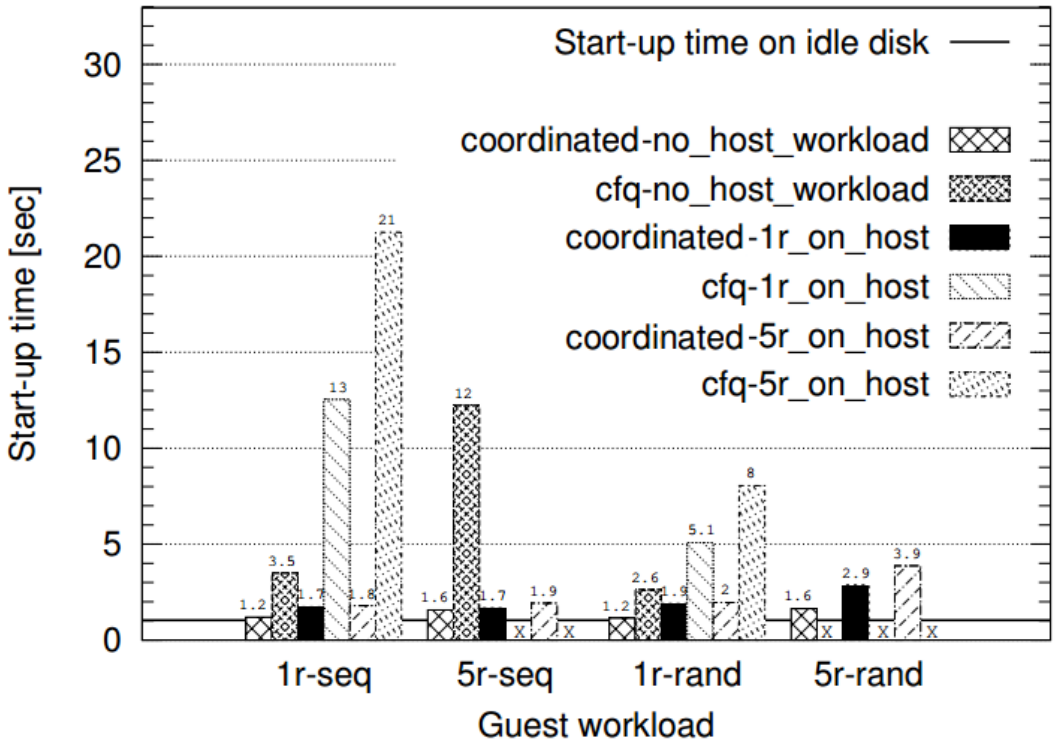

Figure 15. microSDHC results - Coordinated scheduling compared to CFQ

10.3 Results with the eMMC

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

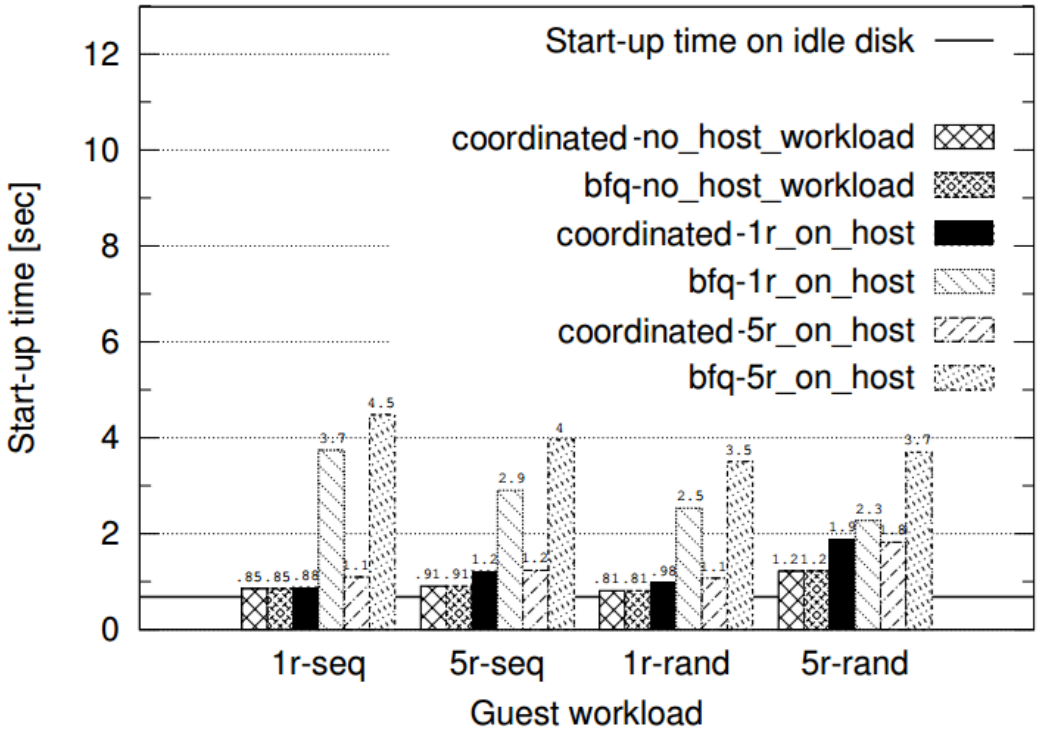

Figure 16. eMMC results - Coordinated scheduling compared to BFQ

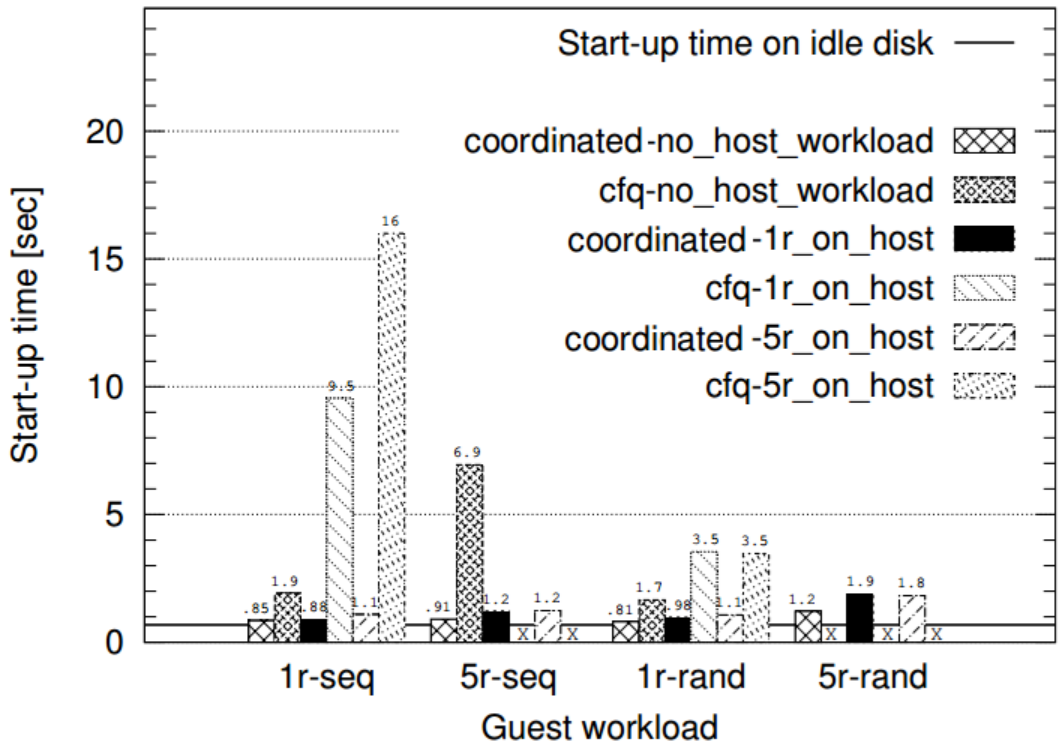

Figure 17. eMMC results - Coordinated scheduling compared to CFQ

11 Conclusion

In this deliverable we described the limitations of current scheduling mechanisms, in relation to preserving low latency on I/O operations in virtualized systems. By extending tools for usage in virtual machines we highlighted the underlying problem where a host cannot identify and schedule properly guest I/O requests. Finally, the concept of *coordinated scheduling* is introduced and a mechanism was implemented to try and resolve I/O latency issues.

The implementation of coordinated scheduling utilized, 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 it from BFQ could probably be extended and applied also to schedulers of other, important resources, such as CPUs and transmission links.

12 Bibliography

1. P. Valente, M. Andreolini, “Improving Application Responsiveness with the BFQ Disk I/O Scheduler”, Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12). ACM, New York, NY, USA, DOI=10.1145/2367589.2367590 http://doi.acm.org/10.1145/2367589.2367590
2. P. Valente, F. Checconi, “High Throughput Disk Scheduling with Fair Bandwidth Distribution”, IEEE Transactions on Computers, vol. 59, no. 9, pp. 1172-1186, September 2010, doi:10.1109/TC.2010.105
3. BFQ homepage. http://algogroup.unimore.it/people/paolo/disk_sched
4. CFQ I/O Scheduler. http://lca2007.linux.org.au/talk/123.html
5. Sandstrom, K.; Vulgarakis, A.; Lindgren, M.; Nolte, T., "Virtualization technologies in embedded real-time systems", \emph{Emerging Technologies \& Factory Automation (ETFA), 2013 IEEE 18th Conference on}, Sept. 2013
6. 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.
7. J. Lee, S. Xi, S. Chen, L.T.X. Phan, et. al., "Realizing Compositional Scheduling through Virtualization", IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12), April 2012.
8. Storage I/O Control Technical Overview and Considerations for Deployment. http://www.vmware.com/files/pdf/techpaper/VMW-vSphere41-SIOC.pdf
9. Virtual disk QoS settings in XenEnterprise. http://docs.vmd.citrix.com/XenServer/4.0.1/reference/ch04s02.html
10. Extended version of the benchmark suite for virtualized environments. http://www.virtualopensystems.com/media/bfq/benchmark-suite-vm-ext.tgz