A survey on shared disk I/O management in virtualized environments under real time constraints

Ignacio Sañudo
University of Modena
and Reggio Emilia
ignacio.sanudoolmedo
@unimore.it

Roberto Cavicchioli
University of Modena
and Reggio Emilia
roberto.cavicchioli@unimore.it

Nicola Capodieci
University of Modena
and Reggio Emilia
nicola.capodieci
@unimore.it

Paolo Valente
University of Modena
and Reggio Emilia
paolo.valente@unimore.it

Marko Bertogna
University of Modena
and Reggio Emilia
marko.bertogna@unimore.it

ABSTRACT

In the embedded systems domain, hypervisors are increasingly being adopted to guarantee timing isolation and appropriate hardware resource sharing among different software components. However, managing concurrent and parallel requests to shared hardware resources in a predictable way still represents an open issue. We argue that hypervisors can be an effective means to achieve an efficient and predictable arbitration of competing requests to shared devices in order to satisfy real-time requirements. As a representative example, we consider the case for mass storage (I/O) devices like Hard Disk Drives (HDD) and Solid State Disks (SSD), whose access times are orders of magnitude higher than those of central memory and CPU caches, therefore having a greater impact on overall task delays. We provide a comprehensive and up-to-date survey of the literature on I/O management within virtualized environments, focusing on software solutions proposed in the open source community, and discussing their main limitations in terms of real-time performance. Then, we discuss how the research in this subject may evolve in the future, highlighting the importance of techniques that are focused on scheduling not uniquely the processing bandwidth, but also the access to other important shared resources, like I/O devices.

1. INTRODUCTION

A hypervisor, also called Virtual Machine Manager (VMM), is a combination of software and hardware components that allow emulating the execution of multiple virtual machines upon the same computing platform by properly arbitrating the concurrent access to shared hardware resources. Most of the available open source hypervisors are specifically tailored to server applications and cloud computing. In these areas, hypervisors are mainly designed to provide isolation, load balancing, server consolidation and desktop virtualization within the managed virtual machines. However, the emerging of new potential areas for VMMs, such as automotive applications and other embedded systems, and the possibility to exploit multi-core embedded processors are posing new challenges to real-time systems engineers. This is the case of next-generation automotive architectures, where cost-effective solutions ever more require sharing an on-board computing platform among different applications with heterogeneous safety and criticality levels, e.g., the infotainment part on one side, and a safety-critical image processing module on the other side. These domains are independent, with different period, deadline, safety and criticality requirements. However, they need to be properly isolated with no mutual interference, or a misbehaving module may endanger the timely execution of a high-criticality domain, affecting safety qualification.

In order to provide real-time guarantees, hypervisors either dynamically schedule virtual machines according to a given on-line policy, or they statically partition virtual machines to the available hardware resources. An example of the first category is RT-Xen [23] (now merged into mainline Xen [1]), which implements a hierarchical virtual machine scheduler managing both real-time and non-real-time workloads using the Global Earliest Deadline First (G-EDF) algorithm. On the other hand, statically partitioned solutions tend to isolate virtual machines onto dedicated cores, with an exclusive assignment of hardware resources. An example of this approach is given by Jailhouse [18], which does not allow multiple virtual machines to share the same core. An advantage of this latter approach is that the resulting hypervisors have a typically smaller code footprint, implying much lower certification costs. Indeed, reducing the code size is a prominent characteristic of other recent VMMs, like NOVA [19] and bhyve[1]).

No matter which virtualization approach is taken, most of the current literature on resource access arbitration for virtualized environments mainly focuses on CPU scheduling (see for example surveys [8] and [21]), neglecting the huge impact that the access to other shared hardware resources, like Hard Disk Drives (HDD) and Solid State Disks (SSD), may have on time-critical tasks. In view of this consideration, this paper provides a survey on the state-of-the-art on I/O virtualization and concurrent HDD/SSD read/write operations. We will discuss the applicability of previously introduced solutions to I/O arbitration for enhancing the real-time guarantees that may be provided in a virtualized environment. Main limitations of classic fair provisioning

EWiLi'16, October 6th, 2016, Pittsburgh, USA. Copyright retained by the authors.

[1]https://wiki.freebsd.org/bhyve

schemes to resource sharing will be highlighted.

We are interested in software-based solutions that do not require customized device controllers and hardware mechanisms to obtain the desired behavior. Therefore, most of the addressed works deal with virtualized approaches that schedule the access to storage devices by means of a hypervisor or similar mechanisms, shaping the I/O requests to guarantee a given I/O bandwidth to multiple partitions/cores. For each of the presented works, we will highlight the main weaknesses and limitations, in order to stimulate the real-time research community to undertake a more rigorous and structured effort towards achieving the required predictability guarantees.

Contributions are divided by contexts. In this respect, a first coarse-grained distinction is made considering the technology used for data storage: rotational or non rotational. HDDs and flash-based devices, such as SSDs, may have similar issues when it comes to arbitration of concurrent accesses, but their radically different operating principles entail different problems to solve in order to ensure a predictable behavior. A finer-grained distinction is related to arbitration policies, examining how different I/O scheduling algorithms behave in a virtualized environment and whether they are able to satisfy hard/soft real-time guarantees.

The rest of the paper is organized as follows. The next section introduces the motivation behind the presented survey. Section 3 describes the existing solution based on the Xen hypervisor for I/O management. Section 4 discusses statically partitioned solutions for multi-core platforms. Section 5 highlights performance, predictability and security issues related to the layered scheduling systems implied by many virtualization techniques. Existing works introducing real-time parameters within the I/O scheduler are summarized in Section 6, while Section 7 discusses the additional predictability problems incurred with current SSD devices. A final discussion is provided in Section 8 showing promising research lines to improve the predictable management of shared hardware resources by means of properly designed hypervisor mechanisms.

2. MOTIVATION

There are multiple motivations under this document. The initial reason triggering this study relates to the problems encountered when trying to guarantee bounded shared resource access times to tasks concurrently executing on a multi-core platform. Even if often neglected by theoretical works on real-time scheduling, a great share of the predictability problems of modern multi-core platforms is due to potentially conflicting requests to shared hardware resources like caches, bus, main memory, I/O devices, network controllers, acceleration engines, etc. The arbitration of the access to the mentioned shared resources is often hardwired and cannot be easily controlled via software. The enforced policies are mostly tailored to improve average case performance and throughput, often conflicting with the predictability requirements of real-time applications. Finally, low level details on the arbitration policies are difficult to obtain and may significantly vary on different architectures. This makes it extremely difficult to develop a tight timing analysis even for simpler platforms. To sidestep these problems, we are studying scheduling solutions that aim at avoiding conflicts on the device arbiter, by properly shaping the device requests from the different cores. Hypervisors are natural candidates in this sense, providing a centralized decision point with a global view of the requests from the various partitions. This would allow taking the unpredictable arbiters out of the scheduling loop, leaving the resource management at hypervisor level. Before implementing such a solution, we examined the existing related approaches for managing shared resources in virtualized environments, taking storage devices as a representative example.

This choice is mainly due to the large interest in I/O scheduling within the open source community. Modifications to the current Linux schedulers are constantly being proposed and evaluated. For example, at the time of writing, a new scheduler denoted as BFQ (Budget Fair Queuing) [22] is undergoing evaluation for being merged into mainline Linux. This I/O scheduler is based on CFQ, the default I/O scheduler in most Linux systems. Among other goals, BFQ is designed to outperform CFQ in terms of the soft real-time requirements that can be guaranteed to multimedia applications. However, it remains unclear how the proposed modifications can deal with harder real-time constraints, given the unpredictable technical characteristics of storage devices.

A second motivation descends from the consideration that sub-optimal arbitration policies of an I/O storage device can be the primary cause of blocking delays and performance drops. This is due to the considerably worse latencies and bandwidths of storage devices with respect to other shared resources, such as central memory or CPU caches. As an example, the random access times to L1, L2, L3 and DDR main memory on an Intel® i7 architecture are in the order of 1ns, 10ns, 50ns and 100ns, respectively. In contrast, the random access times to SSDs and HDDS are considerably higher, in the order of 100us and 10ms, respectively. Despite the cost of SSDs random accesses is predicted to drop to 10/50us in the next years, the gap from the main memory access times would still be of at least two orders of magnitude. Due to this difference, it is of paramount importance to properly schedule and coordinate the access to storage peripherals.

A third motivation is related to the abundant presence of I/O scheduling research in cloud and server virtualized scenarios. The major concerns in these fields are performance and fairness, rather than real-time constraints. Also, the concept of fairness is mainly applied to CPU scheduling, rather than on the access to shared resources. Consider the widely known Xen hypervisor [1]. Xen allows the system administrator to specify the policies that regulate how guests are scheduled on the various cores. We can specify that a VM can be scheduled with RTDS and a different CPU with a Credit scheduler. By doing so, we are not going to arbitrate access to the CPU (as they are scheduled in different cores) but we want to specify requirements for the first VM and a non real-time domain for the credit scheduled VM. However, a Credit-scheduled virtual machine that runs an I/O intensive task can cause a priority inversion towards other RTDS-scheduled machines, even if the latter require much less I/O bandwidth. In order to further prove the validity of this motivation, we reproduced this priority inversion with a simple experiment in a Xen virtualized environment. The experiment involved a workstation managed with Xen 4.5.0 and equipped with a quad-core Intel® i7 processor, disabling *hyperthreading*. We setup two virtualized disk partitions using LVM (Logical Volume Manager) on a rotative

HDD. The read peak rate of the HDD was ~130MB/s. The device was paravirtualized. We created two guests pv1 and pv2, pinned to two different cores, each accessing one of the two partitions. The workload executed by these two virtual machines is as follows:

- pv1 is a Credit-scheduled virtual machine, associated with the Xen default scheduling weight (see section 3 for a brief introduction of the Xen Credit scheduler and the description of its parameters). pv1 executes a non-critical, non-real-time, I/O-intensive application, sequentially reading a single 1GB-file. Such an application acts as an interfering workload to other real-time tasks on a different domain.

- pv2 is an RTDS-scheduled virtual machine that runs a single task with a 50ms period. The end of the period is assumed to coincide with its relative deadline. Within its period, this guest has to read a memory-page-sized (4KB) chunk of data, randomly chosen out of a 1GB-file in its partition. This setup allows reproducing the worst-case HDD access latency, which, for random reads, has a bandwidth of ~0.63 MB/s, corresponding to an average latency of around 5ms for a 4KB memory-page read.

In order to rule out any performance bottlenecks into the privileged domain we assign the remaining memory and cores to Dom0. Despite the large slack of the RTDS guest (pv2) to complete its memory read and its higher priority, the I/O interference causes pv2 to experience a large number of deadline misses. Figure 1 shows the results of the experiment sampling 40 periodic I/O read accesses (x axis) by the RTDS domain. The y axis represents the time taken by each request in μs. Each bar represents the actual I/O request time. The horizontal black line indicates the period between subsequent requests, while the vertical green line corresponds to the instant when the interference operated by the Credit scheduled domain (pv1) is over. As can be easily seen, pv2's requests starve during the read process of the Credit guest, which almost monopolizes the access to the HDD. In contrast, when the interference created by pv1 is over, pv2 does not experience any deadline miss. In retrospect, this behavior is not surprising, as the higher priority provided to an RDTS guest affects only the scheduling on different CPUS, but it has no effect on I/O scheduling. In other words, the priority is not transferred from CPU to I/O.

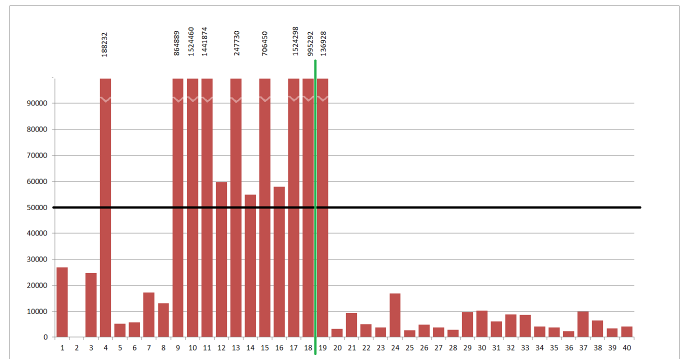

Figure 1: Summary of I/O experiments in Xen.

3. I/O SCHEDULING IN VIRTUALIZED ENVIRONMENTS

A significant number of contributions addressed I/O virtualization issues by modifying the existing Xen Credit scheduler. The Credit scheduler is the default CPU scheduler, and it works by distributing *credits* among virtual machines in proportion to their *weights*. Weights can be freely set on guest creation. A virtual machine consumes its credits while running on a physical CPU, and is in an *over* or *under* priority status depending on whether it has exceeded or not its share of CPU resource within a considered time window. Credits are redistributed for each virtual machine by a specifically designed system-wide thread.

The part of the credit scheduler that relates to I/O scheduling is connected to what is known as *boosting mechanism*, i.e., an additional *boost* priority status that allows performance improvements for I/O-intensive guests [16]. A very demanding I/O task running on a guest causes the virtual machine to get blocked often, leading to a very limited credit consumption, with the guest always in the *under* state. When waking up after completing an I/O request, the VMM will grant the guest a boost priority, allowing it to preempt other running virtual machines to process the requested data.

Different works tried to improve I/O scheduling in virtualized environments by acting on the mechanisms used by Xen to assign priority statuses within guests and on the above described boosting mechanism. In [12], the authors developed a solution that extends the mechanisms of the Xen Credit scheduler. They introduced the notion of *task-aware (I/O) scheduling* arguing that a task-aware model is beneficial for scheduling purposes, especially in situations where mixed resource usage and I/O-intensive tasks are concentrated in specific domains. Once the VMM has knowledge of which guest has higher I/O bandwidth requirements or specific latency-related constraints, the hypervisor will use this information to decide how and when to assign the boost priority to those I/O-bound guests. In [7], the authors followed a somehow similar, but more complex, approach. They developed a technique for speeding up I/O virtualization using direct I/O with hardware IOMMU. To support a real-time response for high quality I/O virtualization, a new `REAL_TIME` priority state is added to the Xen Credit scheduler supporting preemption. Consider a latency-sensitive application running inside a guest to which the associated latency-critical pass-through device is assigned. Whenever the pass-through device fires an interrupt, the associated virtual machine is automatically promoted to the `REAL_TIME` state, triggering a preemption of any non `REAL_TIME` guest to schedule that particular machine right away. While the first contribution [12] mainly focuses on achieving a fair behavior among domains, the latter [7] shows promising results in terms of I/O throughput and latencies. Due to the low latency values obtained, the authors in [7] claim to have designed a real-time virtualized environment, but no experimental or analytical evidence has been provided to support these claims using classic real-time metrics, such as schedulability ratio, worst-case response times, deadline misses, etc.

Another interesting approach is presented in [3] and [11]. Both works are focused on adaptive time-slice sizing in Xen.

In the first contribution, the authors modified the Xen Credit scheduler to guarantee Quality of Service (QoS) requirements for streaming audio applications in virtualized environments. They designed an adaptive modifier of the Xen Credit scheduler that allows flexible time-slices and real-time priority flags to be dynamically assigned to guests. According to their presented results, the authors were able to improve the responsiveness of latency-sensitive applications, achieving some kind of soft real-time guarantee. They tested their implementation by pinning multiple virtual CPUs (vCPUs) to the same physical core, hence testing concurrent I/O requests rather than parallel I/O operations. In [11], the authors adopt a similar mechanism for an on-the-fly adaptation of the time slices within the I/O scheduling policies (mainly CFQ and Anticipatory) of the Linux kernels that are executing within the Xen unprivileged domains. Here, parallel HDD requests are explicitly considered, showing an improved latency. However, even if improving latencies is an important step towards predictability, a system that dynamically adapts scheduling constraints, such as time slices, makes it very difficult to identify worst-case scheduling settings where to build a tight timing and schedulability analysis.

4. MULTI-CORE PARTITIONING AND VIRTUALIZATION

Another promising direction to obtain a predictable behavior is to exploit the multi-core nature of today's CPUs, assigning specific I/O handling functions to specific cores. This can be accomplished with CPU hardware extensions and/or a different virtualization paradigm using partitioning-based hypervisors. The work in [9] proposes a Xen implementation monitoring runtime information of the bandwidth requirements of the different guests. Specific functions related to I/O handling are pinned to specific cores, e.g., one core is used for driver-related aspects, another one to handle I/O events, another one for generic computations. Performance improvements are claimed in terms of bandwidth and latencies with a slight drop in the performance of compute-intensive tasks.

Another interesting contribution that relates to core specialization is presented in [13]. A VMM based on hardware resource partitioning is taken into account, proposing a hypervisor (SplitX) that resembles the operating mechanisms of Jailhouse[2]. Specialized cores can handle I/O related interrupt and hypervisor instructions. The authors claim that I/O level performance is expected to reach near bare-metal performance, by means of hardware extensions to allow directed inter-core signals for events notification but also for managing resources belonging to other cores. An example of this latter feature may allow a core to assign specific values to the registers of a different core. Unfortunately, this latter batch of related works mainly deals with performance improvements. Even if a considerable drop on latency values is a promising start for achieving real-time guarantees, these approaches are not concerned with obtaining worst-case delay bounds and a tight timing analysis.

In a recent publication [20], a scratch pad centric non-virtualized architecture is presented in which real-time requirements are explicitly taken into considerations. Similarly to the other approaches presented in this section, a specific core is delegated for I/O operations exploiting a dedicated I/O bus. Task executions are decoupled from instruction and data loading using a Time Division Multiplexing (TDM) approach. I/O operations are included in the same time slice used for task loading/unloading. While this latter contribution present a very rigorous and sound timing analysis, it does not explore I/O intrinsics threats to predictability in virtualized environments, nor it addresses the problems of having multiple I/O tasks hogging the dedicated core.

[2]Jailhouse does not yet support any mechanism to predictably manage the concurrent access to shared resources like I/O devices, each of which is statically pinned to a given partition/core having exclusive access to it.

5. PERFOMANCE AND SECURITY ISSUES INTRODUCED BY I/O VIRTUALIZATION

It is straightforward to observe that a hypervisor allowing its guests to run entire operating systems can easily introduce noticeable overheads due to the local CPU and I/O schedulers. Virtualized platforms, such as Xen, have their own CPU arbitration policies for scheduling guests, but also privileged domains have to go through their own block layer, while each guest runs its own kernel with different local scheduling policies for both CPU and I/O, hence providing an added level of complexity when accessing the storage device. This hierarchical structure is known to cause performance drops compared to bare-metal systems, but it also exposes a more complex architecture that dramatically increases the difficulties of deriving a sound timing analysis.

The performance overhead issue has been studied in different works [26, 16, 5]. In a recent paper [25] the authors measured the overhead of I/O stack duplication between host and virtual machines running KVM as VMM. It also provided a very complete survey on previous tests on different VMMs that eliminated a layer of the IO scheduler. A simple QEMU + virtIO solution is shown to outperform almost all tested scenarios.

The hierarchical organization of these kinds of hypervisors also poses significant security threats. In [24], an untamed I/O intensive task running within a compromised/malicious guest is used to slow down and interfere with other supposedly separated domains. For this reasons, the authors recommend to adopt a separate and unique I/O scheduler for virtualized environments.

We believe that such an I/O scheduler should be designed with the same guidelines considered when implementing efficient CPU real-time schedulers, ensuring a proper isolation among tasks that require disk access, while allowing them to complete their workload within given deadlines. On this latter consideration, it has to be pointed out that the Linux kernel provides features such as control groups (cgroups) that can be used to isolate, limit and control disk (rotational or SSD storage device) accesses of sets of processes. For example, cgroups can be set within privileged domains to limit resource usage by unprivileged guests. However, this feature does not provide for specific scheduling policies, rather it relies on the underlying I/O scheduler, and on its policy, for enforcement. In this respect, current Linux I/O schedulers implement too coarse policies for typical real-time requirements.

6. DEADLINE-AWARE I/O SCHEDULING

The need to provide tighter real-time guarantees to tasks accessing disk I/O has been a problem addressed since the early 90's. In [17], Reddy et al. presented a scheduling algorithm called SCAN-EDF that combined the Earliest Deadline First (EDF) [15] and SCAN schedulers for minimizing request latency in HDD while serving deadline constrained tasks. During the years, this algorithm has also been modified and improved by means of heuristics, such as batching and delaying requests, or aggregating multiple queues of requests. In [10, 14] and [4], the Xen I/O architecture has been modified to include deadline-based scheduling for the storage in a virtualized environment. In [10, 14], a two level scheduler called Flubber is introduced. The first level defines the throughput using a credit-rate controller to ensure performance isolation, while the second level applies Batch and Delay EDF (BD-EDF) to manage the request queues from the different guests. Even if the authors claim that Flubber improves Xen performance and allows the system administrator to specify deadlines, no results are provided to evaluate the worst-case delays and blocking times needed to establish a sound timing analysis. In [4], a similar approach is used, where the first level accumulates the amount of I/O requests in a fixed time slice while analyzing the disk bandwidth, and the second level exploits the deadline-modified SCAN algorithm reordering the requests according to the deadline group and to their location on the disk. While there is a performance enhancement for I/O intensive workloads, neither this work appears to lend itself to the analytical guarantees required in a real-time setting.

7. REAL-TIME ISSUES IN SSDS

Solid State Drive based storage devices deliver from 5 to 10 times the bandwidth of a HDD, while maintaining a low power consumption and a much stronger resistance to shocks and vibrations. These features make SSDs the primary choice for applications in the automotive and avionics sectors, in which embedded platforms have to sustain prolonged vibrations while still delivering high performance. This makes it particularly interesting to understand whether the previous considerations coming from experiments executed on HDDs equivalently hold for guests sharing access to a SSD. In this respect, it has to be pointed out that the *intrinsic operating mechanisms* of SSDs pose significant problems towards the design of predictable hard real-time systems. This is due to the fact that flash memories are a write-once and bulk-erase medium, that implies that a flash translation layer (FTL) and a Garbage Collection (GC) mechanism are needed to provide applications a transparent storage service. A naïve best effort GC policy can unpredictably start segmentation operations causing tasks to wait for potentially long blocking times. The authors in [2] focused on providing hard real-time guarantees for the GC phase in small NAND flash devices by proposing a token based garbage collection system. The presented results showed that the implemented system is predictable and robust to interferences introduced by non real-time tasks. A prototype is tested in a 16MB NAND-flash drive with two real-time tasks and one non real-time task in a manufacturing system scenario, with no deadline violations until high CPU utilization. A more recent contribution [6] observed that the previous solution does not scale well, making it impossible to apply to modern SSDs having a much larger storage capacity. An FTL implementation (KAST) is then proposed to allow the user to control the worst case blocking time by tuning some GC parameters.

8. CONCLUSIONS

Hypervisors represent a possible solution to bypass unpredictable scheduling policies enforced by off-the-shelf arbiters for the access to shared hardware resources. By taking informed decisions on the scheduling of the different requests coming from multiple tasks, a hypervisor may provide stronger timing guarantees to real-time tasks, predictably limiting the delays due to interfering requests on the shared devices. Taking I/O scheduling as a representative case for resource sharing, we highlighted the main results concerned with improving the delays due to competing accesses to storage devices in virtualized environments. We showed how I/O intensive tasks within non-critical virtual machines can easily cause more critical partitions to experience high blocking delays, leading to repeated deadline misses. This was the case with the Xen hypervisor, whose critical partitions are "privileged" only when assigning processing bandwidth, but not when arbitrating the access to other shared resources. We argued that smarter scheduling policies are needed, that take into account the timing requirements of the different tasks/partitions also when arbitrating the access to shared devices.

We showed that existing mechanisms to improve the blocking delays are mainly tailored to obtain better average performance or achieve a fairer behavior, but cannot be leveraged to develop a sound timing and schedulability analysis. While it would be possible to manually tune the bandwidth allocated to each partition when accessing an I/O device, e.g., by playing with cgroups parameters in Xen, such a solution has clear limits in terms of flexibility, efficiency and responsiveness, preventing a tight timing analysis. Moreover, we pointed out that hypervisors like Xen and KVM add further layers of complexity to guest operating systems, with repeated scheduling and block layers coupled with para-virtualized driver architectures, making it very difficult to formalize the I/O scheduling model. Partitioned hypervisors seem more suitable in this sense, especially when the number of cores increases and each domain can be statically assigned to one or more dedicated cores. Still, most of the available partitioned VMMs do not allow for a predictable and concurrent access to shared devices, but they either exclusively pin each resource to a selected domain, preventing tasks running on other partitions to access it, or they implement para-virtualized schemes that are not aware of the different real-time requirements.

To conclude, we believe that guaranteeing hard real-time requirements within embedded virtualized platforms requires the hypervisors to be made aware of the I/O requirements of their guests. Performance-oriented considerations aiming at improving average latencies need to be sacrificed to achieve a more predictable behavior. We hope this paper may stimulate the research on predictable I/O scheduling policies for virtualized environments, paving the way towards simpler and tighter timing analysis.

9. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM*

SIGOPS Operating Systems Review, 37(5):164–177, 2003.

[2] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.*, 3(4):837–863, Nov. 2004.

[3] H. Chen, H. Jin, K. Hu, and M. Yuan. Adaptive audio-aware scheduling in xen virtual environment. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010*, AICCSA '10, pages 1–8, Washington, DC, USA, 2010. IEEE Computer Society.

[4] T.-Y. Chen, H.-W. Wei, Y.-J. Chen, W.-K. Shih, and T.-s. Hsu. Integrating deadline-modification scan algorithm to xen-based cloud platform. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–4. IEEE, 2013.

[5] L. Cherkasova and R. Gardner. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

[6] H. Cho, D. Shin, and Y. I. Eom. Kast: K-associative sector translation for nand flash memory in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 507–512, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[7] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang. Towards high-quality i/o virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 12. ACM, 2009.

[8] 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*, 5(4):227–290, 2012.

[9] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia. I/o scheduling model of virtual machine based on multi-core dynamic partitioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 142–154, New York, NY, USA, 2010. ACM.

[10] H. Jin, X. Ling, S. Ibrahim, W. Cao, S. Wu, and G. Antoniu. Flubber: Two-level disk scheduling in virtualized environment. *Future Generation Computer Systems*, 29(8):2222–2238, 2013.

[11] 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*, pages 6–6. USENIX Association, 2010.

[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*, VEE '09, pages 101–110, New York, NY, USA, 2009. ACM.

[13] A. Landau, M. Ben-Yehuda, and A. Gordon. Splitx: Split guest/hypervisor execution on multi-core. In *WIOV*, 2011.

[14] 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 *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 81–89, Washington, DC, USA, 2012. IEEE Computer Society.

[15] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[16] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[17] A. Reddy and J. Wyllie. Disk scheduling in a multimedia i/o system. In *Proceedings of the first ACM international conference on Multimedia*, pages 225–233. ACM, 1993.

[18] V. Sinitsyn. Jailhouse. *Linux Journal*, 2015(252):2, 2015.

[19] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[20] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[21] G. Taccari, L. Taccari, A. Fioravanti, L. Spalazzi, A. Claudi, and A. B. SA. Embedded real-time virtualization: State of the art and research challenges. 2014.

[22] P. Valente and F. Checconi. High throughput disk scheduling with fair bandwidth distribution. *IEEE Transactions on Computers*, 59(9):1172–1186, Sept 2010.

[23] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in xen. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 27:1–27:10, New York, NY, USA, 2014. ACM.

[24] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang. Understanding the effects of hypervisor i/o scheduling for virtual machine performance interference. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 34–41, Dec 2012.

[25] M. Yi, D. H. Kang, M. Lee, I. Kim, and Y. I. Eom. Performance analyses of duplicated i/o stack in virtualization environment. In *Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication*, page 26. ACM, 2016.

[26] P. Zhao and G. Tan. Evaluating i/o scheduling in virtual machines based on application load. *Engineering Journal*, 17(3):105–112, 2013.