J Grid Computing (2017) 15:107–125
DOI 10.1007/s10723-017-9392-4

Analysis and Evaluation of Scheduling Policies for Consolidated I/O Operations

Konstantinos Kontodimas ·
Panagiotis Kokkinos · **Yossi Kuperman** ·
Athanasios Houbavlis · **Emmanouel Varvarigos**

Received: 21 August 2015 / Accepted: 2 January 2017 / Published online: 20 January 2017

Abstract Hypervisors' smooth operation and efficient performance has an immediate effect in the supported Cloud services. We investigate scheduling algorithms that match I/O requests originated from virtual resources, to the physical CPUs that do the actual processing. We envisage a new paradigm of virtualized resource consolidation, where I/O resources required by several Virtual Machines (VMs) in different physical hosts, are provided by one (or more) external powerful dedicated appliance(s), namely the I/O Hypervisor (IOH). For this reason I/O operations are transferred from the VMs to the IOH, where they are executed. We propose and evaluate a number of scheduling algorithms for this hypervisor model, concentrating on providing guaranteed fairness among the virtual resources. A simulator has been built that describes this model and is used for the implementation and the evaluation of the algorithms. We also analyze the performance of the different hypervisor models and highlight the importance of fair scheduling.

Keywords I/O hypervisor · Cloud computing · Virtualization · Scheduling

K. Kontodimas (✉) · P. Kokkinos
Department of Computer Engineering and Informatics, University Campus, Patras, Greece
e-mail: kontodimas@ceid.upatras.gr

K. Kontodimas · P. Kokkinos · A. Houbavlis · E. Varvarigos
Computer Technology Institute & Press "Diophantus", University Campus, Patras, Greece

Y. Kuperman
IBM Research and Development, Haifa, Israel
e-mail: yossiku@il.ibm.com

E. Varvarigos
School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece
e-mail: vmanos@central.ntua.gr

1 Introduction

In recent years, there has been a rapid development of Clouds, in various forms and shapes. In their essence Clouds provide an abstraction between what is offered, as a service or as a resource, and what it is actually required (both in hardware and in software) for that offering. This abstraction reduces complexity, cost and increases efficiency since the actual resources can be highly utilized. Additionally, this abstraction eases the provision of tolerance against faults. Hypervisors, the software necessary for creating and running Virtual Machines (VMs), are the basic building blocks that support this abstraction. This is why their operation and performance has a direct effect on the provided Cloud-based services.

Traditionally, there is one physical machine, one hypervisor and multiple guest Virtual Machines (VMs), running on top of them. Clouds are supported by data centers that consist of several thousands of physical machines. The computational, storage and network resources of a data center are orchestrated using Cloud software like OpenStack [1], which among others interacts with the hypervisor running on each physical machine for managing the VMs. Scheduling is apparent at various levels: at a higher level the orchestration software is equipped with scheduling logic in order to decide the physical host where a guest VM will be initiated, while at a lower level the hypervisor's scheduling mechanism matches the requests coming from virtual resources to the hosting machine's physical cores.

In our work, we propose and evaluate scheduling algorithms for the lower level, on a, somewhat, different model of operation from the classical hypervisor. This architectural model stems from the *"ORBIT: Business Continuity as a Service"* project [2], with the main aim to address the needs of mission-critical services, in terms of enabling advanced high performance fault-tolerance and disaster recovery.

In this new paradigm of virtualized resource consolidation, I/O resources used by several Virtual Machines (VM), running on top of multiple physical hosts, are provided by one (or more) external powerful dedicated appliance(s), namely the I/O Hypervisor (IOH). Externalization (from the perspective of the guest VM) and consolidation of virtualized I/O are carried out by transferring I/O operations requested by the VMs to the IOH, where they are executed. In this way I/O services, like firewall, Deep Packet Inspection (DPI) and block-level encryption that consume a lot of CPU resources, can be consolidated in a dedicated server, increasing CPU utilization and accommodating changes in load conditions where demand from different hosts fluctuates.

The way I/O operations, originating from (or destined to) multiple VMs in several physical hosts, are scheduled for execution in the I/O Hypervisor's (IOH) available cores, is critical for the efficiency of I/O consolidation. This efficiency can be translated to the number of VMs an IOH can concurrently serve, to the serving throughput and other parameters. The I/O operations are packet structures that are sent between virtual and physical devices in either direction, and the objective is the selection of the core in the IOH that will process each one.

In a more technical level, the traditional I/O virtualization techniques mainly suffer from two problems a) reduced performance due to context switches between guest VMs and the hypervisor and b) reduced throughput and increased latency due competition between I/O-intensive guests regarding CPU resource allocation. These problems are dealt with by offloading I/O processing tasks to a set of dedicated cores. However, the problem with the dedicated cores is that either they are underutilized when there is not enough I/O traffic for them to process, or they become the bottleneck when there is too much I/O traffic for their available processing power. Rather than partitioning the CPU cores of a single host, the considered model of I/O Hypervisor partitions entirely the hosts to the ones that deal with the VM processing and to the ones that deal with the I/O processing; the I/O-dedicated cores are migrated to remote dedicated hosts.

The model we are referring to describes the newly developed virtual I/O technology of IBM, namely *Split I/O* (or Paravirtual Remote I/O – vRIO) [3]. The concept of this particular work might not deal with the actual technical details of the paravirtualization model, yet it is worth a brief reference to be made to this field. Split I/O is based on the existing technology of *sidecores*, which in particular are CPU cores dedicated to process the requests and responses generated from the paravirtualized network and storage devices. With Split I/O the hosts composing a data center environment are split into *VMhosts* and *IOhosts*, i.e. into the hosts onto which VMs are residing and the hosts that are responsible to handle I/O traffic. In this case, in contrast to the previous virtual I/O models, the sidecores are residing onto the dedicated IOhosts, rather than the same hosts with the VMs; the communication is achieved via an I/O dedicated Ethernet channel. The local VMhost hypervisor is ignorant to the I/O traffic generated by its VMs. The I/O Hypervisor has a set of *workers* each of which runs on a specific sidecore. The workers directly intercept to the local IOhost's NICs without needing to go through TCP/IP stack. Extended description of all technical details can be found in [3]. In the same work, the authors present an experimental evaluation using the actual implementation, comparing Split I/O with the existing virtual I/O models (namely KVM virtio, SRIOV and Elvis).

In our work, we implement and evaluate a number of online fair scheduling algorithms for the IOH. For this purpose, the IOH (Section 3) is formulated as a multiple-queues and multiple-servers system (Fig. 1a). These queues correspond to the virtual and physical devices, located in the VMs and the IOH respectively, while the servers to the actual CPU cores of the IOH that process the I/O operations. Also, a simulator has been built that describes this I/O paravirtualization model and is used for the implementation and evaluation of the various algorithms'. Fairness is the main criterion considered. In addition, we analyze the performance of the different hypervisor models and show that the considered I/O consolidation model achieves smaller queuing delays.

The remainder of this paper is organized as follows. In Section 2 we report on previous work. The I/O Hypervisor problem is formulated in Section 3. Also, I/O Hypervisor's performance is theoretically analyzed in Section 4. Section 5 describes the implemented scheduling algorithms. Sections 6 and 7 present the simulator used for the evaluations and the results. Finally, the paper is concluded in Section 8.

2 Previous Work

Three major techniques are common for hosts to virtualize I/O services for their guests: emulation [4], where a familiar device (e.g., a common network card) is emulated, paravirtualization [5], which emulates a new device, designed not to resemble any existing device but to be as efficient as possible when used across the guest-host boundary, and device assignment [6, 7], where the host gives a guest (mostly) direct access to a portion of a certain physical device.

Despite the proven performance advantage of device assignment, paravirtualization is preferred or even required because device assignment does not support I/O interposition and thus cannot be used when the hypervisor needs to intercept all the I/O channels to implement fault tolerance capabilities. Device assignment also requires more expensive hardware. For these and other reasons, most real-world applications of virtualization today choose to use paravirtual I/O. Using paravirtual I/O, the hypervisor running on each server is responsible for interposing on the I/O of each of its guests. The hypervisor requires and consumes physical resources such as CPU, RAM, and SSDs from each of the servers, which could otherwise be assigned to the running guests or be used to run additional guests. In addition, this model degrades the performance of I/O intensive guests and limits the scalability of the system [8]. In the I/O paravirtualization model we consider in our work, we combine SR-IOV [9] that provides near native I/O performance, together with paravirtual I/O technologies [4, 5], which enable I/O interposition (required for fault tolerance).

The scheduling of the VMs' operations on physical processing units (e.g., cores) is an important task for any kind of hypervisor. Credit scheduler, which is the default scheduler in Xen hypervisor, manages CPU allocation for VMs based on *credit* value set by predefined weight of each VM [10]. The calculated credit is assigned to each VM periodically and is consumed proportional to the processing time of the VM; this consumption is conducted at the granularity of a tick interval. Cherkasova et al. [11, 12] analyzed three CPU of Xen (BVT, sEDF and credit) by measuring I/O throughput for different scheduling parameters. Their experiments demonstrate the performance impact of CPU allocation for host domain, which hosts I/O on behalf of guest domains. They show that frequent interventions of host domain degrade I/O throughput because this incurs several domain switches and prevents guest domains from batching I/O requests. This work illustrates challenging issues related to VM scheduling mechanism for varied workloads. In [10], the authors evaluate the Credit and the sEDF schedulers within Xen, which are able to do a good job of fairly sharing processor resources among computing-intensive domains. However, when bandwidth intensive and latency-sensitive domains were introduced both schedulers showed mixed results.

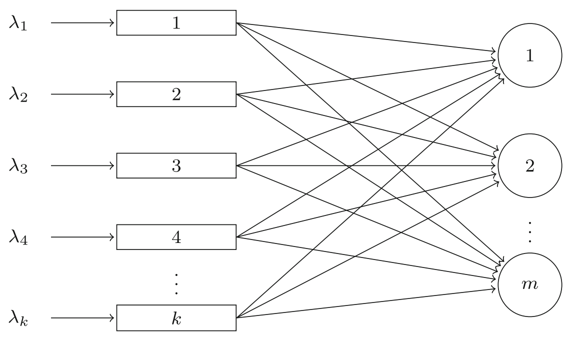
Fairness is one of the most important objectives in any kind of scheduling. Most fair scheduling policies (GPS [13, 14], WF^{2}Q [15], SFQ [16], BVT [17] DRR [18]) target single-resource systems. However, multi-resource (network, storage, memory, computation) systems arise in a number of cases. For example, end-hosts are increasingly equipped with multiple network interfaces, ranging from smartphones with multiple radios, to servers with multi-homing. Also, in many cases these interfaces are diverse; some are expensive to use (e.g., 4G), some are free (e.g WiFi) and they have different rates and reliability. Similarly, the parallel subcarriers of OFDM systems in wireless networks can be considered as multiple servers [19]. Multi-resource contention also arise in systems hosting VMs, which require access to physical resources (computing, storage and memory) for their operations. In our work, the I/O Hypervisor is formulated (Section 3) as a multiple-queues and multiple-servers system (Fig. 1a), which correspond to the virtual and physical devices that create the I/O operations and the cores that process them. A major issue in the above is how requests from different flows (queues, etc) are scheduled to the available resources.

Fair scheduling mechanisms, like GPS-based and round-robin algorithms, for the single resource (link, core etc) do not trivially extend to the multi-resource case [20–22]. Another idea is to employ a GPS-based scheduler for each processor and partition the set of threads among processors such that each processor is load balanced. While such an approach can provide strong fairness guarantees on a per processor basis, it has certain limitations [20].

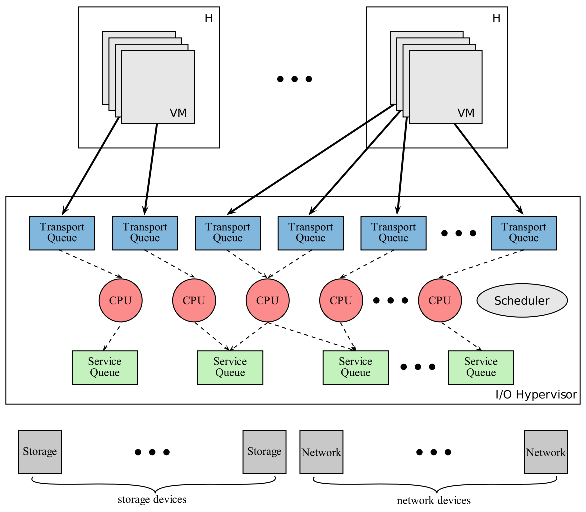
A number of multi-resource extensions of single-resource fair scheduling algorithms have been proposed over the years. In [23] GPS scheduling was extended to the case of multiple nodes. In [24], the authors extended Weight Fair Queuing (WFQ) and WF^{2}Q to multi-link scheduling, proposing the MSFQ algorithm. When a server is idle and there is a packet waiting for service, MSFQ schedules the "next" packet, defined as the first packet that would complete service in the GPS system with one server and equal total capacity if no more packets were to arrive. MSFQ does not provide bounded fairness, in terms of the independence of the amount of service any flow receives, in relation to the set of flows. The authors also propose MSF^{2}Q that provides bounded fairness and prove it.

Surplus Fair Scheduling (SFS) [20] is a proportional-share CPU virtual time based scheduler designed for symmetric multiprocessors with a number of running threads, extending SFQ algorithm [16]. The authors describe a generalized multiprocessor sharing (GMS) – an idealized algorithm for fair, proportionate bandwidth allocation that is an analogue of GPS in the multiprocessor domain. They use the insights provided by GMS to design the SFS algorithm. They experimentally showed that under certain workloads, their multiprocessor scheduling algorithms displays similar properties to their new reference system.

Some works consider variations of the problem of scheduling focusing on perspectives others than fairness. In particular, in [25], the authors explicitly consider the case of the cloud environment as they devise a new cost-effective workflow scheduling algorithm,



(a) A graphical representation of a multiple-queues and multiple-servers system.



(b) A brief representation of The I/O Hypervisor's architecture.

Fig. 1 Graphical representations of an arbitrary multiple-queues and multiple-servers system (Fig. 1a) and of the actual architecture of IOH-Hypervisor (Fig. 1b)

namely, BDHEFT which is an extension of HEFT algorithm. The comparison of proposed algorithm is done with BHEFT heuristic, under same deadline and budget constraint and pricing. Their results show that their proposed algorithm outperforms BHEFT algorithm in terms of monetary cost while producing the makespan as good as produced by BHEFT algorithm. Another approach is followed by Tang et al. in [26], where they focus on the reduction of energy consumption in cloud centers. They provide an energy-saving scheduler DEWTS based on dynamic voltage/frequency scaling algorithm. DEWTS is applicable to the scheduling system of most data centers consist of DVFS-enabled processors. The algorithm can obtain significant energy reduction as well as maintaining the quality of service by meeting the preset deadlines. A relevant work that deals with the problem of optimal load distribution is provided by Li, in [27]. In particular, the author addresses the problem of optimal load distribution of generic tasks on multiple heterogeneous blade servers preloaded with special tasks in a cloud computing environment. The problem is formulated as a multivariable optimization problem based on a queuing model. The author also developed algorithms to find the numerical solution of an optimal load distribution and the minimum average response time of generic tasks.

Grouped Distributed Queues (GDQ) [28] is the first proportional share scheduler for multiprocessor systems that scales well with a large number of processors and processes. GDQ uses a simple grouping strategy to organize processes into groups based on similar processor time allocation rights, and then assigns processors to groups based on aggregate group shares. The work in [29] focuses on the allocation of resources of different types and proposes Dominand Resource Fair (DRF) algorithm. The allocation of a user (flow, etc.) is determined by the user's dominant share, which is the maximum share that the user has been allocated of any resource type. Dominant Resource Fair Queuing (DRFQ) [30] generalizes the concept of virtual time from classical fair queuing to multiple resources that are consumed at different rates over time and implements DRF allocations in the time domain. In [31], the authors show a simple scheduling scheme for packet-by-packet GPS over multiple interfaces, and prove that it can provide bounded delay and rate guarantees. They present again the idea of clustering flows with the same service rate in the same servers, which may better approximate the GPS. In [32], a greedy algorithm for the fair allocations of resources based on bottlenecks is defined.

Regarding the service model, there are various works dealing with the concept of I/O virtualization. The authors of [33] investigate the I/O performance interference through the experimental study of consolidated n-tier applications leveraging the same I/O resources. They claim that by specifying a custom disk allocation results in significantly lower performance than fully sharing disk across VMs. The system considered by the authors applies a per VM isolation of the same CPU resources which are responsible to handle both computation and I/O load and thus the performance degrades due to resource underutilization. In [34] the authors deal with the difficulty induced to resource allocation process, due to unpredictability of the type of the workload by the hypervisor; they propose a mechanism that provides distinction between I/O-bound and CPU-bound workloads which results to improvement of the performance and complete CPU fairness among virtual machines. Similarly, the authors of [35] present the design, implementation and evaluation of a system that can fairly allocate disk I/O bandwidth in virtualized environment. This system can mitigate the interference between multiple DomUs by introducing separated V-Channel and I/O queue for each DomU. These works assume that computational and I/O tasks are both handled by their local hosts' CPUs. However, the architectural model of IOH provides splitting and isolation between the consolidated CPU resources that specifically handle the I/O load and the local CPU resources of the underlying host machine. In fact the I/O-dedicated cores are migrated to a remote dedicated host. This, kind of isolation eliminates the interference between I/O and computational tasks while keeping the resource utilization high, due to consolidation to a single dedicated machine [3].

A number of works extend Deficit Round Robin (DRR) for the multi-resource scenario [36, 37]. In [37], a DRR-based scheduling algorithm is presented, called miDRR, generalized for multiple interfaces. The algorithm is designed for sharing multiple interfaces while respecting user preferences about how they should be used and by which application and rate preferences. miDRR algorithm strives wherever possible to give each flow its weighted fair share of capacity (defined by the rate preference), while guaranteeing that it will never violate the interface preference, and remaining work-conserving on all interfaces. The algorithm remains fair even in the case that the queues prefer using particular CPU cores (e.g., for CPU affinity reasons). The authors prove that miDRR finds the correct max-min allocation. The provided methodology permits building a practical packet scheduler.

3 Problem Formulation

3.1 I/O Hypervisor

The I/O Hypervisor (IOH) externalizes storage and network I/O resources from a total of N Virtual Machines (VMs) residing on H different physical hosts, and consolidates all of them in a single dedicated machine M. The IOH has D physical storage devices (disks) and C physical network devices (communication devices or NICs), for a total of $S = D + C$ physical devices. The IOH's architecture is briefly presented in Fig. 1b.

Each virtual device d of V_i (where $i = 1, 2, ..., N$) is served by a pair of transport $t(d)$ and service $s(d)$ FIFO queues, belonging to the I/O Hypervisor (IOH). A transport queue maps to a particular virtual device (network or storage of a VM), connected via Layer 2 connectivity. A service queue maps to a physical device (network card or disk of M) that can be shared by many VMs (and corresponding virtual devices). We assume that there is a total of T transport queues (virtual devices) and S service queues (physical devices). The mapping of the T queues to the S queues determines which physical devices serve the VMs' virtual ones. Though this mapping may be interesting, performance-wise, we assume it is decided when a VM is set up, and is considered fixed. The IOH is also equipped with a number of Processing queues, where I/O data are stored for processing operations like: deep packet inspection - DPI, block-level encryption etc.

The I/O Hypervisor (IOH) is a multi-core machine, with K CPU cores that can be utilized in parallel. The (t, s) or (s, t) pair each CPU core is dedicated to, changes over time according to the scheduling decisions. It is evident that the maximum number of pairs served simultaneously (in parallel) at any given time is equal to the number K of I/O Hypervisor CPUs. Also, the number of VMs' virtual devices concurrently served cannot be larger than T, the total number of transport queues at the IOH M.

3.2 I/O Operations

An I/O operation r uses a (t, s) pair, e.g., when a VM's virtual device is writing data to the storage or the network, and a (s, t) pair when a VM's device is reading data from the storage or the network (a network packet arrives).

In particular, we define an I/O request r as an I/O packet of size B_r bits that originates from a virtual device and is stored in a transport queue. Indicatively, it can be

(i) a block write;
(ii) a block read request;
(iii) or a packet send.

An I/O response r is an I/O packet of size B_r bits that is stored in the service queue and is destined for a particular virtual device of a VM. Indicatively, it can be one of the following:

i) a block that was read from a physical disk
ii) a notification that a write request has finished
iii) a packet received from the network

Usually, an I/O request such as read request from a VM's device to a disk, results in an I/O response, with the data, from the disk back to the VM.

Generally, a CPU core is engaged twice for processing an I/O request:

(i) for passing data from the transport to the processing queue and for performing the processing operation and
(ii) for passing the processed data from the processing queue to the service queue and for writing data from the service queue to the actual physical device.

Similarly, a CPU core is engaged twice for processing an I/O request:

(i) for passing data from the service to the processing queue and for performing the processing operation and
(ii) for passing the processed data from the processing queue to the transport queue and for writing data from the transport queue to corresponding VM's device.

A different core may be used to serve (i) and (ii). Also, since a response can be created “long” after the initial request (on the order of milliseconds for a disk), a different CPU core may be used for handling it, than the one used for the original request.

The execution time X_r for a request or a response r depends mainly on its size, on a constant time penalty describing the latency of a physical device, on a constant initiation delay penalty (incurred each time the serving CPU is changed) and on the bandwidth (bits/sec) of the corresponding device.

We assume that the requests are generated by the virtual devices while the responses are generated by the physical devices. V is used to map each device i to its appropriate type (virtual or physical). In particular, it returns the set of the potential types of requests (or responses) that are compatible to the type of device. For example, if a device i is a virtual storage device, $V(i)$ maps the latter to the set of request types that are compatible to virtual storage devices. For each request/response instance, $p(q)$ is used to define its probability to belong to type q. $\mathbb{E}(X_q)$ is the average time required for a request (or response) that belongs to type q in order for it to be processed by a core, after it is selected from the head of the queue. Also, λ_i denotes the request (or response) generation rate of a device i while T and S are used to respectively denote the sets of virtual and physical devices.

Provided the above, the theoretical precondition for system stability, given that the scheduling is ideal and without taking into account the various constraints of the actual implementation, is given by (1). In fact, the left hand side of the inequality defines the utilization factor of the system, which should be lower than 1 in order for the aggregated I/O Hypervisor (IOH) system to be stable. By this constraint we can derive the maximum (type-dependent) load that can be handled by the I/O Hypervisor; when the condition is not met, the system becomes unstable.

$$\frac{1}{K}\sum_{i\in T\cup S}\lambda_i\sum_{q\in V(i)}p(q)\mathbb{E}(X_q)<1 \qquad (1)$$

3.3 Scheduling

The I/O Hypervisor (IOH) scheduler is responsible for selecting the CPU core that will serve an I/O operation from a particular (t, s) or (s, t) pair. The scheduling procedure runs periodically so as to serve new I/O requests/responses or assigned CPU cores that completed their assigned I/O operation. The scheduler uses an online algorithm, making decisions in a bounded short time (deadline-based scheduling). This impacts not only the frequency at which we can run the scheduling algorithm, but also impacts the performance of the IOH itself. For example, if we were to run the scheduling algorithm once per second, the decision algorithm would have to finish within the one second period, but would also use 100 % a CPU core during that time, reducing the CPU power available for I/O processing.

4 I/O Hypervisor Performance

4.1 Analysis

As mentioned, the I/O Hypervisor (IOH) suggests a different architectural model of operation from the classical hypervisor, in which I/O resources used by a guest Virtual Machine (VM) are provided by an external host, instead by the local one. In this way the I/O operations generated by the VMs in a host are not carried out by the corresponding local CPU resources but they are all sent and processed by the consolidated CPU resources located in the IOH. The two architectural models are presented in Fig. 2, assuming N physical hosts and corresponding VMs.

In this section, we theoretically compare these two models in terms of the queuing delays of the I/O operations, using standard queuing theory concepts [38]. In the classical hypervisor, we model each physical host as an $M/M/c$ queuing system, assuming that all I/O operations, generated by the guest VMs, are put in a single FIFO queue. When one of the c CPU cores becomes available, it receives the first I/O operation from the queue and processes it. In the IOH model, we do not consider the Transport and Service queues but instead we assume that there is again a single FIFO queue where all operations are placed as they arrive from the VMs located in other machines. In this way, the IOH is again modeled as a $M/M/k$ system, where k the number of CPU cores in the IOH. Also, we assume Poisson arrival rate λ and service rate μ. Additionally, in both cases we do not consider scheduling, but in a way we assume that scheduling is perfect, with all I/O operations treated equally and the CPU resources always working. Based on the assumptions

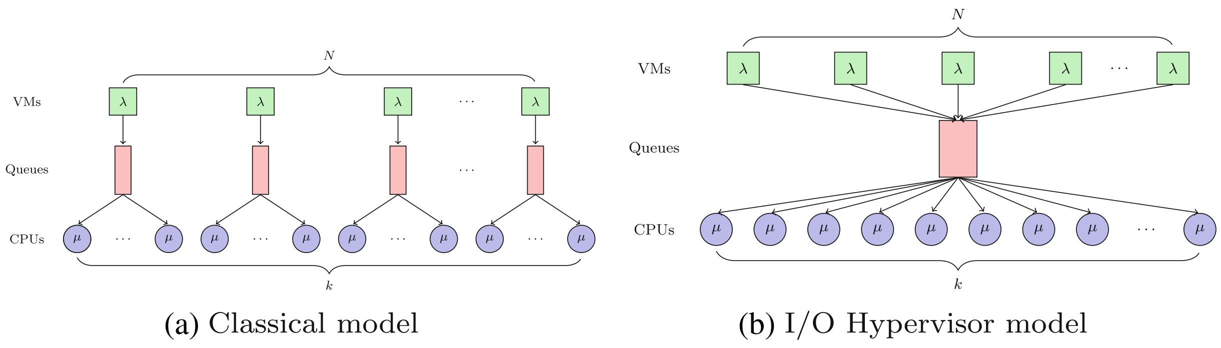


Fig. 2 The classical model (Fig. 2a) analyzed as N independent $M/M/c$ queuing systems and the model of I/O Hypervisor (Fig. 2b) that is analyzed as a single $M/M/k$ queuing system

made, the queuing delay of an I/O operation for the original (T) and the I/O hypervisor ($\hat{T}$) models, in any of the N physical hosts is equal to

$$T = \frac{1}{\mu} + \frac{P_Q}{c\mu - \lambda} \tag{2}$$

$$\hat{T} = \frac{1}{\mu} + \frac{\hat{P}_Q}{k\mu - N\lambda} \tag{3}$$

We should note that for the IOH, the load of the system is equal to the accumulated load ($N\lambda$), since we assumed for our analysis that all I/O operations are placed in a single queue.

P_Q and $\hat{P}_Q$ represent the queuing probability in each case that relates to the system load. In case of lightly loaded systems, where few I/O operations are generated, both probabilities can be considered almost equal to 0, since these operations are immediately served by the waiting CPU resources. In this case, the average queuing delay in both models is equal to $T = \hat{T} = 1/\mu$. On the other hand, in case of heavily loaded systems, P_Q and $\hat{P}_Q$ can be considered almost equal to 1 (the same approach has been presented in [38]), meaning that any I/O operation will be queued for some time before a CPU resource becomes available for its processing. In this case, assuming also that $\frac{1}{\mu} \ll \frac{P_Q}{c\mu-\lambda}$, $\frac{1}{\mu} \ll \frac{P_Q}{k\mu-\lambda}$ and $k = Nc$ we have that

$$\frac{T}{\hat{T}} \cong \frac{k\mu - N\lambda}{c\mu - \lambda} \Rightarrow \frac{T}{\hat{T}} \cong N \tag{4}$$

which means that in heavy loads the queuing delay of an I/O operation in the classical hypervisor model can be N times larger than that in the I/O Hypervisor (IOH) model. This performance difference is due to the fact that the IOH can better serve unbalanced, between the various VMs and guest hosts, I/O traffic than the classical model. In particular, in the classical model the I/O operations generated in a heavy loaded machine cannot take advantage of the CPU resources available, in other possible lightly loaded physical machines. On the other hand, in the IOH all I/O operations are served from the same set of resources.

In the previous analysis, we assumed in the classical model that all machines have the same number of CPUs. In case we do not make this assumption and the CPU cardinality varies among machines, then for heavily loaded systems the average queuing delay of an I/O operation on the classical model increases. This scenario highlights even more the advantage of IOH model, since the average operation delay ratio of the two models, increases as well. In this scenario, the machine with the smallest number of CPUs (e.g., one) becomes a bottleneck for the average queuing delay. In particular, the average delay of an I/O operation generated in the bottleneck machine is $\tilde{T} = \frac{1}{\mu} + \frac{\tilde{P}_Q}{\mu-\lambda}$. In case we have lightly loaded systems ($\tilde{P}_Q \cong 0$ and $\hat{P}_Q \cong 0$) the average waiting time, as in the previous case, is equal. In case of having heavily loaded systems ($\tilde{P}_Q \cong 1$, $\hat{P}_Q \cong 1$), we have that

$$\frac{\tilde{T}}{\hat{T}} \cong \frac{k\mu - N\lambda}{\mu - \lambda} = k + \frac{(k-N)\lambda}{\mu - \lambda} \Rightarrow \frac{\tilde{T}}{\hat{T}} \gg N \tag{5}$$

since $k \gg N$ by definition. A similar analysis can be carried out assuming different arrival rates for the various machines, where the machine with the largest rate becomes the bottleneck.

In practice, it is expected that the number of CPU resources of the I/O Hypervisor (IOH) to be smaller than the accumulated count of CPU resources from all the physical host (of the first model): $k < Nc$. In this case the performance difference of the two models is reduced.

The selection of these particular queuing systems – a $M/M/c$ queuing system for every single discrete host of the case of the classical model and a single $M/M/k$ system for the case of the aggregated host of the IOH – is made under the assumptions that there are no delays induced by the scheduling process and no interference occurs between different VMs sharing the same physical host. The latter may not be totally realistic, but the interference is in fact expected to be low; this derives from the architectural model of the system which eliminates interference between I/O and computational load, since IOH provides perfect isolation between the consolidated CPU core resources that specifically handle the I/O load and the local CPU resources of the underlying host machine that handle the computational load.

Some works in the literature, including [33] and [35], consider the latter assumption as a part of their theoretical analyses. More specifically, the distribution of service times in our theoretical analysis is considered to be exponential; this derives from the fact that the operations, on average, require a service time related to the average operation length, but a level of uncertainty, due to physical system complexity, induces an additional stochastic amount to their service times. Under our assumptions, since all devices are mutually independent on their decisions about the usage of the system, the classical model and the IOH model are described well by the $M/M/c$ and $M/M/k$ queuing systems, respectively.

4.2 Criteria

The main performance criterion for the I/O Hypervisor (IOH) is the number of Virtual Machines (VMs) that it can efficiently serve. This actually translates to the number of virtual devices that it can serve assuming a particular average I/O operation generation rate and data size. The efficiency of the IOH has mainly to do with the time required for an I/O operation to complete.

One can relatively easily identify the main bottlenecks, that is communication delay between the physical machines hosting the VMs and the IOH and queuing delays inside the IOH. Communication delay can limit the I/O operation service rate and the corresponding completion time. However, inside a data center environment this delay is expected to be low, even under heavy I/O workloads. In addition, it is expected that emerging fast interconnects with lower latency and higher throughput will make it even more negligible. Also, the Layer 2 communication used between the VMs and the IOH induces low additional control overhead, in comparison to Layer 3 communication. The latter is also supported by the experimentation on the actual system, as shown in the [3], where the results indicated that the latency due to network connectivity is in fact low, though slowly increasing with the the number of serving devices. The latter is a result of the contention over the remote cores. Queuing delays inside the IOH relate to the way transport and service queues are matched and the various queues (transport, service, processing) are served by the available cores of the IOH. For example, matching heavily loaded transport queues to a single service queue of small service rate (e.g., disk write rate) increases queuing delay. Similarly, a poor scheduling mechanism that decides the way the available cores will serve the various queues, may lead to I/O requests waiting for too long in their queues or to VMs' virtual devices that are unfairly handled by the IOH. In this work, we consider the matching between the transport and service queues to be static and we concentrate on the implementation and evaluation of scheduling algorithms that can handle the IOH queues workload in a fair manner, providing in this way indirectly Quality of Service (QoS) to the corresponding virtual devices.

The main objectives of these algorithms are the following:

- (weighted or not) fairness on service capacity allocation between queues
- minimization of mean queuing delay for I/O requests/responses
- maximization of resource utilization
- maximization of I/O throughput
- maximization of the number of VMs that can be served

Among the objectives described above, we mainly target in providing fairness between queues. So, we propose an algorithm that essentially achieves this objective and provides a good performance for the other objectives as well. In particular, we propose the miDRR-IOH fair scheduling algorithm, which is based on [37], miDRR algorithm. The minimization of the mean I/O queuing delay is a natural consequence of fairness in capacity allocation, and is explained in the following sections. Maximum resource utilization is also achieved through the specifications of the algorithm, as it is work conserving and does not waste any available capacity. In addition, maximum resource utilization depends on the operational frequency of the scheduler, as there may be capacity waste if the scheduler operates slowly. Furthermore, we conduct experiments in order to identify the minimum number of CPU cores required to serve particular VMs' virtual devices.

5 Implemented Algorithms

5.1 The miDRR-IOH Algorithm

The *multiple interface Deficit Round Robin-I/O Hypervisor*, or miDRR-IOH, weighted max-mix fair algortihm, is a variance of miDRR introduced by Yap et al. in [37], which is actually a generalized form of the classic Deficit Round Robin [39, 40] scheduling algorithm for the case of multiple servers. miDRR is adapted based on the characteristics of the I/O Hypervisor 3, where multiple I/O virtual devices' queues are served by the available cores. This algorithm, unlike most fair queuing algorithms available, allows both rate and interface preferences to be met, i.e. preferences on service rate of each queue, and the capability a subset of queues to be served by particular interfaces (or CPU resources in the case of miDRR-IOH). Furthermore, the algorithm is also work-conserving, which means that capacity is not wasted when there is traffic available in the queues, and remains max-min fair, even with the existence of interface preferences. This indeed is achieved very efficiently, using a list of flags, called *service flags*, while no other coordination between the queues is required. Each flag corresponds to a device-core pair, indicating whether the device is currently served by the core.

For each queue, the user can set a weight value (defaults to 1.0). Also, each queue keeps a *deficit counter* value that is initialized to 0 and a *quantum* size value that is determined from its weight. The quantum size value Q_i of queue i is derived by $Q_i = w_i \cdot Q_{\min}$, where Q_i and w_i are the quantum size and weight of queue i, respectively. $Q_{\min}$ is the minimum quantum size of all queues. The minimum quantum size, in our case, is derived by the length of the smallest (in bytes) request of the system. A device's queue is designated as "backlogged", at a given time, if there are queued requests.

In each scheduling round, the algorithm selects the first available (not busy) core, in a round-robin manner. If there are no cores available, the algorithm returns. After selecting the core, the algorithm selects the next, again in round-robin, transport-service pair (t, s) with a backlogged queue. If there are no transport-service pairs with backlogged queues left, the algorithm returns. If the length of the selected queue's head-of-line request is less than, or equal, to the deficit counter, then the request is processed. The deficit counter is then decreased by the request's length (in bytes). In addition, the service flags of all other cores that correspond to the selected queue are set. Any core serves a particular transport-service pair (t, s) until the corresponding queue is empty or the deficit counter is zero. When a transport or service queue is empty, the value of the deficit counter is reset, indicating that the it is no longer considered backlogged. In the following round the core selects the next, in round-robin, pair for processing. Also, if the service flag corresponding to a pair is set, which means that is served by another core, the flag is reset. This last step repeats, while iterating cores, until an already reset service flag is found. After that the deficit counter for the selected queue is increased by its quantum. The miDRR-IOH algorithm is described as in Algorithm 1.

Symbol	Description
BL_i	Backlog of the source queue of device queue pair i
$Size_i$	Size of device queue pair i's head-of-line packet
Q_i	Quantum for device queue pair i
DC_i	Deficit counter for device queue pair i
$\mathcal{F}_j$	Set of device queue pairs willing to use core j
$\mathcal{C}_j$	Current queue pair that core j is serving
$\mathcal{B}$	Set of device queue pairs with a backlogged source
SF_{ij}	Core j's service flag for device queue pair i (Service flags for new device queue pairs are initiated to zero.)

Algorithm 1 miDRR-IOH(j)

```
1:  {If no device queue pairs willing to use core $j$ exist, return.}
2:  if $\mathcal{F}_j \cap \mathcal{B} = \emptyset$ then
3:      return
4:  end if
5:  {Set serving device queue pair to the one selected for service by core $j$.}
6:  $i \leftarrow \mathcal{C}_j$
7:  {If the selected queue's head-of-line request is lower than its deficit counter, serve it.}
8:  if $Size_i \leq DC_i$ then
9:      Send $Size_i$ bytes
10:     $DC_i \leftarrow DC_i - Size_i$
11: end if
12: {If $i$ is no longer backlogged, remove it from the set of backlogged queues.}
13: if $BL_i = 0$ then
14:     $DC_i \leftarrow 0$
15:     Remove $i$ from $\mathcal{B}$
16: end if
17: {If $i$ is no longer backlogged, or its deficit counter if lower than its head-of-line request, select the next queue that $j$ will handle the next time it is triggered; update $i$'s deficit counter by "quantum" size.}
18: if $BL_i = 0$ or $Size_i \geq DC_i$ then
19:     $C_j \leftarrow$ miDRR-IOH-Check-Next($i$, $j$)
20:     $i \leftarrow C_j$
21:     $DC_i \leftarrow DC_i + Q_i$
22: end if
```

Algorithm 2 miDRR-IOH-Check-Next(i, j)

```
1:  {Find the first available backlogged device queue pair for core $j$, with respect to the constraints induced by the service flags of core $j$.}
2:  $(s, d) \leftarrow$ Next device queue pair with $s$ being backlogged
3:  $i \leftarrow s$
4:  $C_j \leftarrow i$
5:  while $SF_{ij} \neq 0$ do
6:      $SF_{ij} \leftarrow 0$
7:      $(s, d) \leftarrow$ Next device queue pair with $s$ being backlogged
8:      $i \leftarrow s$
9:      $C_j \leftarrow i$
10: end while
11: $SF_{ik} \leftarrow 1, \forall k \neq j$
12: return ($i$)
```

5.2 Other Algorithms

5.2.1 Credit Algorithm

The Credit algorithm is based on the Xen's Credit Scheduler, which is a proportional fair share CPU scheduler built from the ground up to be work conserving on Symmetric multiprocessing (SMP) hosts. We have implemented and experimented on a packetized variation of the algorithm for the case of I/O operation scheduling. We have reduced all Xen components to their corresponding components of an I/O Hypervisor. In particular, a domain reduces to a VM, a VCPU is reduced to a transport-service pair and a PCPU to a core. Each core must be assigned to at least one virtual I/O device, while a virtual I/O device may be served by multiple cores, depending on the user specifications.

We associate a *weight* to an I/O device, either virtual that corresponds to a transport-service pair (t, s) or physical that corresponds to a service-transport pair (s, t). A weight value may range between 1 and 65536, defaulting to 256, determining the core capacity assigned to the device, i.e. a device with a weight of 512 takes twice as much core capacity as a device with a weight of 256.

Each I/O device pair is also associated to a priority value, `under` or `over` that indicates if there is a positive or negative amount of credits left. Each core manages a local run queue of I/O device pairs that can be scheduled. This queue is sorted by I/O devices' priorities, which means that the devices labeled as `under` have priority to those that are labeled as `over`. When inserting an I/O device pair to a core's run queue, it is placed after all other pairs of equal priority to it. Devices of the same priority are scheduled in a round robin manner. When a core doesn't find an I/O device of priority `under`, it looks on other cores' run queues to find one. If no backlogged `under` devices found, it looks in the same manner for devices with priority `over`. This technique guarantees that the system remains work conserving and that each I/O device pair receives its fair share of core capacity system-wide. As an I/O device pair is served, credits equal to its served bytes are consumed. After a device has been served, it is placed in the end of the core's local queue. Periodically, a system-wide accounting process recomputes how many credits each active I/O device pair has.

5.2.2 First Come First Served Algorithm

The First Come First Served Algorithm (FCFS) algorithm, is a variation of the classical round robin algorithm and is used in this work for comparison with miDRR-IOH. In summary, the basic idea of the algorithm is the following:

1. A core is selected, in a cyclical repetitive manner. The core which is selected, is the first available found while iterating, starting from the last selected core. If there is not any available cores, the algorithm returns.
2. We select the first available transport-service pair whose source is backlogged and can be served by the selected core. The selection is done in a cyclical repetitive manner. If there are no available backlogged pairs, the algorithm returns.
3. We serve the selected pair, by sending the head-of-line queued request to the selected core, for service.

We should note that the quantum (described in Section 5.1) is equal to the minimum request size, so if all requests are equally sized miDRR-IOH and FCFS behave identically, assigning an (unweighted) fair-share to transport-service pairs.

6 I/O Hypervisor Simulator

6.1 Overview

In order to conduct our experiments on I/O operation scheduling for the I/O Hypervisor, we have developed a state-of-the-art simulator, written in Python. The simulator allows the user to set up an infrastructure that includes Virtual Machines (VMs), I/O devices, and the I/O Hypervisor (described in Section 3). It also allows generation of customized workload traffic from the I/O devices (storage and network). Using the simulator, we implemented the algorithms described in Section 5 that take decisions on the I/O operations processing

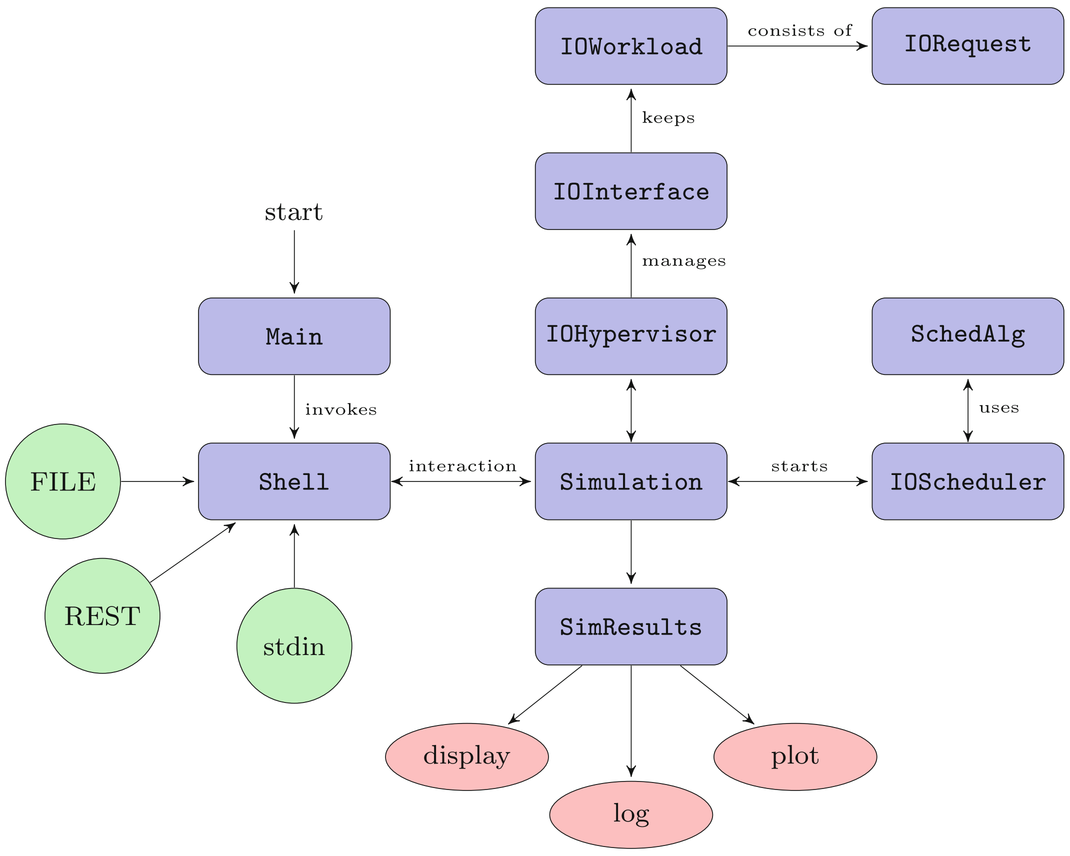


Fig. 3 An abstract representation of the basic modules of the simulator, and the corresponding interconnections between them

priorities and CPU cores assignments. Figure 3 illustrates the basic modules of the simulator.

6.2 Simulator Subsystems

In this subsection, we give a brief description of the modules composing our simulator. A graphical representation of the interconnections between them, is provided in Fig. 3.

`Main`
: is the input module that starts the Shell module.

`Shell`
: is the module that handles the user input, given by console or through an input script.

`Simulation`
: is the main module that keeps all simulation process details. It uses the `Shell` module for receiving input commands and interconnects with the other basic modules (`IOHypervisor`, `IOScheduler`, `SimResults`).

`IOHypervisor`
: creates, configures and manages the deployed `IOInterface` instances, i.e. physical and virtual storage/network devices (service and transport queues), cores and VMs. It also creates associations between devices, based on the user input.

`IOInterface`
: implements all component devices that can be deployed in the system, including I/O devices, cores or VMs. Each instance of these devices, when initiated, defaults to predefined values and can be individually customized by the user. Furthermore, each device keeps its respective `IOWorkload`.

`IOWorkload`
: is a sequence of `IORequest` instances, which define the workload generated for a particular time period by a virtual device.

`IORequest`
: implements an I/O request. It keeps all the useful timestamps, concerning itself (generation, processing etc.) and its length in bytes.

`IOScheduler`
: is a wrapper module around the actual scheduling algorithm of the I/O Hypervisor. It operates periodically, and in each iteration it detects all I/O devices with backlogged queues and all available cores, and gives them as input to the actual scheduling algorithm.

`SchedAlg`
: is a package that keeps the implementations of the various scheduling algorithms.

`SimResults`
: records the output of a scheduling process and provides output formatting capabilities.

6.3 Simulation Process

The user interacts with the simulator via an internal console, or an input script, using the build in command set. Initially, the user set up the system's parameters, by creating VMs, transport (that correspond to virtual devices) and service queues (that correspond to physical devices) and CPU cores. Also, each VM is assigned to virtual I/O devices and each virtual I/O device to a physical one (it must be the same type, storage or network). The attributes of each component are set to to their default values, expect if otherwise specified. The types of entities that are used in the system and their corresponding attributes, are shown in Table 1. The workload of each device is also defined as sequence of requests (following a Poisson process) generated by each device for a given finite time duration. The size of each I/O request is uniformly distributed from a predefined range. In particular, the default sizes of the requests from storage devices (block requests) range between 4 KB and 128 KB, while the ones from network devices (packet requests) range between 64 B and 64 KB. Of course other ranges can also be setup. Next, the scheduling algorithm is set and we can start the simulation for a time duration, given as input (in ms). The simulator schedules workload, using the selected algorithm. The simulation lasts for the duration given in the input, regardless of the duration for which requests are generated (Poisson process duration). In the end the simulator returns the results, and resets the workload to its initial state, in order for another simulation process, with alternate parameters, to be ready to start. Also there is a logging

Table 1 Attributes of the Simulator Entities

Device	Attribute	Default Value
VM	weight	1.0
Core	capacity	1500.0 MHz
Virtual Storage	type	hdd
	workload	–
	weight	1.0
	queue	10000
Physical Storage	type	hdd
	weight	1.0
	queue	10000
Virtual Network	workload	–
	weight	1.0
	queue	10000
Physical Network	weight	1.0
	queue	10000

subsystem with different levels that enables viewing the internal operation of the simulator.

7 Experimental Evaluation

We evaluated miDRR-IOH and the other state-of-the-art algorithms by conducting experiments using the I/O Hypervisor Simulator. The simulator is implemented in Python 2.7.6 and the experimental runs took place on a VMware ESXi VM with using Ubuntu 14.04.1 LTS (Linux 3.16.0-41-generic under x86_64 platform) with 4 Intel® Xeon® CPUs E5-2620 @ 2.00 GHz and 4 GiB of RAM.

In what follows, we compare the implemented algorithms with respect to i) the service capacity allocated to each virtual device, defined as the amount of data (in bytes) of each transport-service pair that was processed by any core and ii) the I/O operations' queuing times, defined as the interval between the time a request (or response) was generated and placed in a queue and the time the request was processed by a core. It is also important to mention that the measurements are taken on continuously backlogged queues. Also, when a queue is full of requests, future arrivals are dropped.

Table 2 provides information on the exact simulation settings. In particular, the simulation duration was set to 300 ms in all the experiments conducted. The scheduler runs periodically, with a period of 0.5 μs and on each loop assigns a queued request (or response) to a core for processing. If no assignment is possible, the scheduler skips to the next loop.

Table 2 Simulation Parameters

Parameter	Value
Simulation duration	300 ms
Scheduler loop period	0.5 μs
λ	2000 requests/sec
Number of Virtual storage devices	14
Number of Virtual network devices	14
Number of Physical storage devices	1
Number of Physical network devices	1
Number of Cores	8

7.1 Experiment 1: Basic Evaluation

In the first set of experiments, we evaluated the miDRR-IOH, Credit and FCFS algorithms performance in relation to the I/O request sizes (for block-device and network-packet operations). The actual sizes of block and packet requests used for our experiments, are presented in Table 3, along with the corresponding ratios. The request arrival rate for each virtual device is modeled as a Poisson process, with $\lambda = 2000$ requests/sec (Table 2).

Figure 4a illustrates the standard deviation of capacity share per device pair (s, t), while increasing the differentiation between the size of block and network requests (according to Table 3). We observe that initially when block and packet request sizes are equal, all algorithms seem to allocate a similar service capacity share among devices. However, as the request size ratio grows (Table 3), miDRR-IOH keeps

Table 3 Sizes of Requests and Responses

Ratio	Block Request/ Response Size	Packet Request/ Response Size
1.0	50000	50000
2.0	100000	50000
3.0	150000	50000
4.0	200000	50000
5.0	250000	50000
6.0	300000	50000
7.0	350000	50000

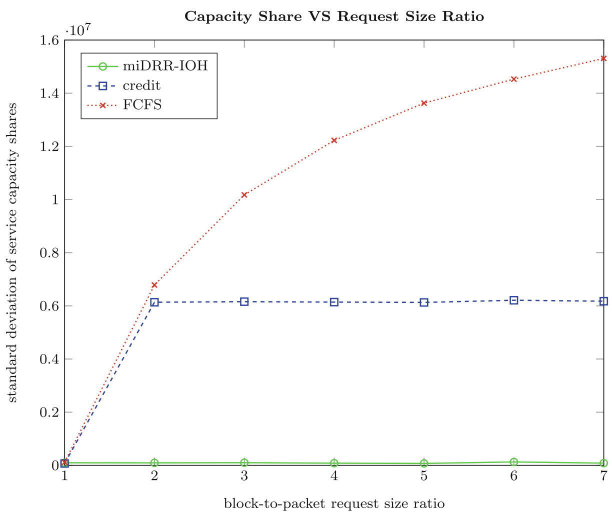
a similar capacity share allocation between the device pairs (s, t), while FCFS deviates proportionally to the request size ratio. Credit scheduling policy has also a deviation on the capacity allocation, thus it does not achieve near-optimal fairness as miDRR. However, this deviation is bounded and is not affected by the request size ratio. We should note that since each (s, t) pair corresponds to a particular virtual device of a VM, in practice miDRR-IOH provides the same service capacity to all virtual devices utilizing the particular I/O Hypervisor. On the other hand, FCFS does not take into account the requests' size and as a result in the end the (t, s) pairs generating larger requests are benefited more than from those that generate smaller ones.

The deviation of capacity share ratio, illustrated in the experiments conducted with Credit policy, is due to the fact that credit updating is taking place in a fixed time-slice, independent from the time required to process a request; this results to a lower granularity than IOH-miDRR's. Nevertheless, the way that Credit policy prioritizes devices, bounds this deviation.

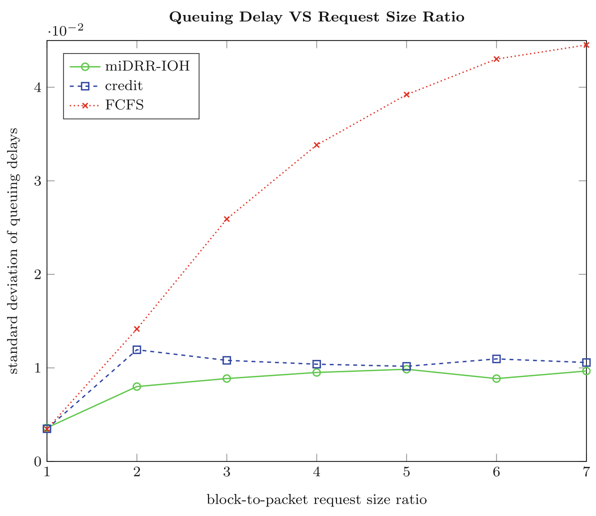
Also, the miDRR-IOH and Credit algorithms achieve shorter, on average, requests' queuing time than FCFS, as it is shown in Fig. 4b. This is due to the fact that these two policies also takes into account the requests sizes, balancing the capacity allocated in terms of bytes, while FCFS balances the number of requests between the devices, disregarding their sizes. As a result, FCFS may send for processing, much more arbitrarily large requests than miDRR-IOH and Credit that result in keeping cores busy, for arbitrarily long time periods, amplifying the average waiting times for the rest of the devices.

It should be noted that in our experimental evaluation, there is no distinction between the latency of physical storage and physical network devices, which would probably incur due to different levels of complexity between disk and network virtualization; this derives from the fact that for the case of disk virtualization, the aspect of bandwidth should be concurrently considered with the latency due to its physical operation. Thus, miDRR-IOH provides standard deviation close to zero. In a more realistic simulation scenario, these latencies could be distinct.

This is a constant time penalty describing the latency of a physical device, on a constant initiation delay penalty (incurred each time the serving CPU is changed) and on the bandwidth (bits/sec) of the corresponding device. These parameters where included in the implemented simulator. However, in the experiments performed we focus on the virtual capacity share and the queuing delays of the virtual devices' requests. These metrics depend on the efficiency of the applied scheduling algorithm and are not directly affected by the latency and the bandwidth of the



(a) Capacity share deviation.



(b) Queuing delay deviation.

Fig. 4 Comparison of the standard deviation of capacity share (Fig. 4a) and standard deviation of average queuing delay (Fig. 4b) per device pair, while increasing the differentiation on the sizes of requests

Table 4 Number of Devices per VM

	# Disks	# Network	Weight (1)	Weight (2)
`vm0`	3	1	2.0	1.0
`vm1`	1	3	2.0	1.0
`vm2`	4	0	1.0	1.0
`vm3`	0	4	1.0	1.0
`vm4`	3	1	1.0	1.0
`vm5`	1	3	1.0	1.0
`vm6`	2	2	1.0	1.0

storage devices. In our revised manuscript, we included the above discussion, while also commenting on the effects of the available storage bandwidth.

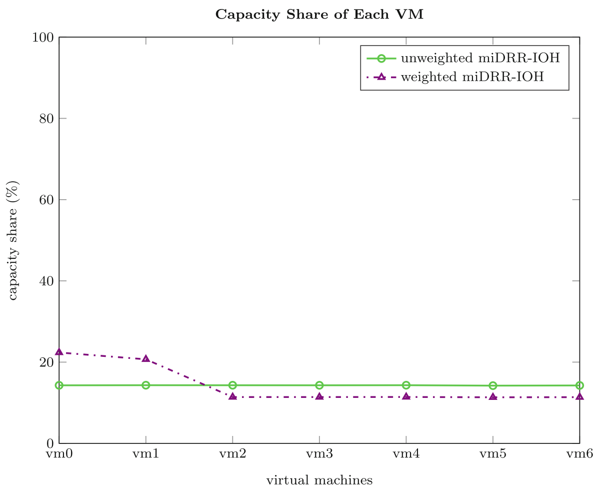
7.2 Experiment 2: Prioritize VMs

Next, we evaluated miDRR-IOH algorithm using different weight values (Table 4) on some of the served VMs and the corresponding (device queues), in order to evaluate the algorithm's capacity share capabilities. It is clear from Fig. 5a that by setting a value of 2.0 on the weights of `vm0` and `vm1` VMs, the miDRR-IOH based scheduler allocates to the corresponding devices twice as much capacity in comparison to the other devices. Additionally, it uniformly reduces the capacity share of the devices belonging to VMs with a weight of 2.0. Furthermore, we can also observe in Fig. 5b that requests generated by the devices that belong to `vm0` and `vm1` are served faster than requests from other devices (and corresponding VMs).

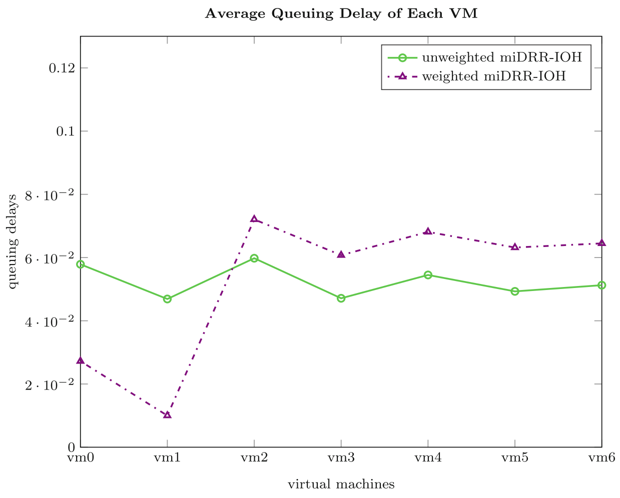
7.3 Experiment 3: Number of IOHs

We also evaluated the number of virtual devices (and corresponding VMs) that can be served by a given IOH configuration (number of IOHs and respective CPU cores), using the miDRR-IOH algorithm, so that no I/O packets are dropped. We conducted two sets of experiments, one for the case of varying the number of operating CPU cores and the other for the case of varying the request arrival rates that are generated by the devices. The results are presented in Figs. 6a and b, respectively.

In each experiment, we also consider three IOH configurations. In the first, there is a single IOH and all CPU cores are available to all devices for serving their requests, scheduled by the IOH. This is the same IOH configuration used in all the previous experiments. In the second and third configuration, we partition the set of virtual devices into two and four, equal in size, subsets/clusters, which are served by two and four respectively IOHs with the same total CPU capacity (number of cores) as the original single IOH. This partitioning is expected to introduce a bound in the maximum traffic capacity that can be served. Also, in all cases, all virtual devices are identical, configured

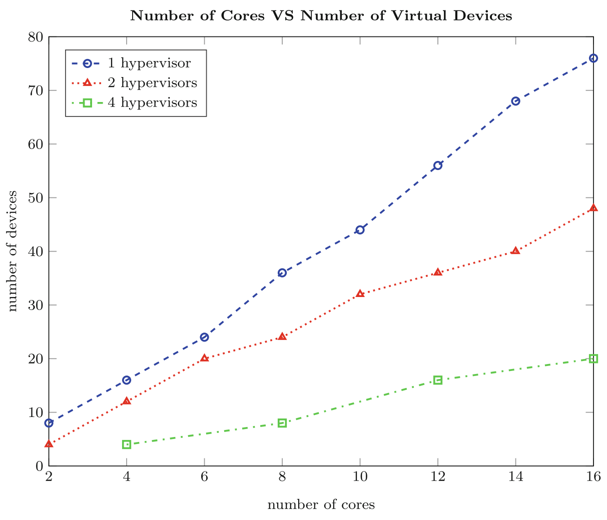


(a) Capacity share deviation.

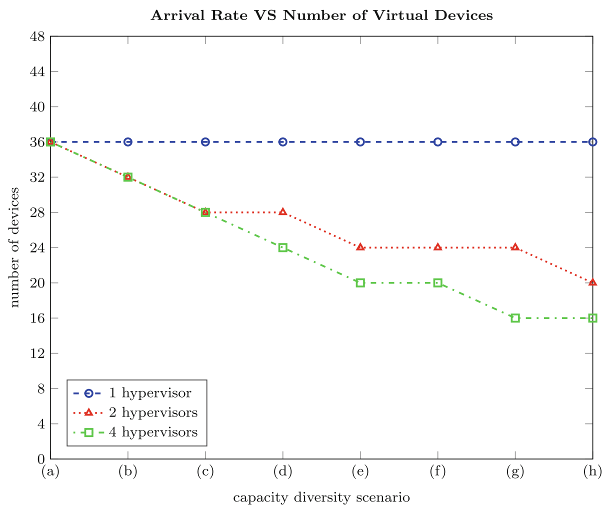


(b) Queuing delay deviation.

Fig. 5 Comparison of capacity share percentage allocated to each VM, while using weights on `vm0` and `vm1` (Fig. 5a); comparison of average queuing delay of each VM, while using weights on `vm0` and `vm1` (Fig. 5b)



(a) Varying the number of cores.



(b) Varying the arrival rates.

Fig. 6 Comparison of the maximum number of virtual devices that can be served without drops, by a varying total number of operating CPU cores (Fig. 6a) and by various traffic arrival rate scenarios (Fig. 6b), for different number of IOHs (1, 2 and 4)

to generate traffic with equal arrival rates (in bps). All cores are identical, operating at 1500 MHz clock frequency.

In the first set of the experiments, we vary the total number of operating CPU cores in the IOH, while keeping the same configuration in the devices of each cluster. The sizes of the requests that are generated by the virtual devices of each cluster are set to be 50000, 25000, 15000 and 10000 bytes respectively. Thus 50 % of the traffic is generated by the first cluster, 25 % is generated by the second one, 10 % by the third and 5 % is generated by the fourth cluster. We assume that in the case of a single or of two IOHs these clusters are appropriately aggregated to one (single) or two respectively clusters. As it is shown in Fig. 6a the maximum number of devices that can be served increases linearly with the number of available cores. We also observe that the single IOH can serve more devices than two or four IOHs of the same CPU capacity, this is due to the fact that the consolidation of CPU resources (in the single IOH) manages to balance the heterogeneous traffic loads, exploiting all the available capacity.

In the second set of the experiments, the number of CPU cores is fixed and equal to 8 and we vary the traffic arrival rates of the devices of each cluster, through the sizes of the generated requests (Table 5). The results are presented in Fig. 6b and clearly show that the introduction of heterogeneous traffic loads, when multiple IOHs are used, reduces the number of devices that can be efficiently (without I/O operations drops) served. This reduction is intensified with the increase of traffic load diversity between different clusters. The slope of the reduction becomes steeper when the number of IOHs increases. In all cases, a single IOH that schedules to a consolidated pool of resources produces the best results.

Table 5 Scenarios of Traffic Load Share Between Clusters

Label	Percentage of Generated Traffic Load			
	Cluster 1	Cluster 2	Cluster 3	Cluster 4
(a)	25%	25%	25%	25%
(b)	30%	25%	25%	20%
(c)	35%	25%	25%	15%
(d)	40%	25%	25%	10%
(e)	45%	25%	25%	5%
(f)	50%	25%	20%	5%
(g)	55%	25%	15%	5%
(h)	60%	25%	10%	5%

8 Conclusion

We introduced a new paradigm of virtualized resource consolidation, where I/O resources used by several

Virtual Machines (VMs), running on top of multiple physical hosts, are provided by one (or more) external powerful dedicated appliance(s), namely the I/O Hypervisor (IOH). In particular, I/O Hypervisor hosts a number of physical storage and network devices, which are shared between the virtual devices of the VMs. In this way, I/O operations requested by the VMs are transferred to the I/O Hypervisor, where they are executed.

We analyzed theoretically the I/O Hypervisor's model with the classical one, where a separate hypervisor runs on each host, without any kind of device consolidation, and compared their average queuing delays. The theoretical results show that for heavy loads, an I/O operation on the classical model has about N times larger queuing delay (where N is the number of hosts) than a similar I/O operation taking place on the I/O Hypervisor's model. Both models behave similarly on light loads.

We proposed miDRR-IOH algorithm for the I/O Hypervisor's scheduler that is responsible for deciding the way the available CPU resources will serve the aggregated (from multiple VMs) I/O operations. For the evaluations a novel I/O scheduling simulator was implemented, along with the miDRR-IOH and two state-of-the-art algorithms (Xen's Credit and FCFS). A number of experiments were conducted, comparing miDRR-IOH with Credit and FCFS algorithms, measuring the average CPU core capacity allocated, and the average I/O request queuing delay. The results show that miDRR-IOH achieves fair allocation of the CPU capacity between the VMs' virtual devices and smaller average queuing delays than the other algorithms. It is clear from our work that scheduling efficiency plays a major role in the success of the I/O Hypervisor.

Acknowledgment The research leading to the results presented in this paper has received funding from the European Union's seventh framework programme (FP7 2007-2013) project ORBIT under grant agreement number 609828.

References

1. OpenStack, "OpenStack Open Source Cloud Computing Software". https://www.openstack.org/ (2016). Accessed 8 Jan 2017
2. Orbit Project, "ORBIT – Business Continuity as a Service". http://www.orbitproject.eu/ (2016). Accessed 8 Jan 2017
3. Kuperman, Y., Moscovici, E., Nider, J., Ladelsky, R., Gordon, A., Tsafrir, D.: Paravirtual remote I/O. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 49–65. ACM (2016)
4. Sugerman, J., Venkitachalam, G., Lim, B.-H.: Virtualizing I/O devices on VMware Workstation's hosted virtual machine Monitor. In: USENIX Annual Technical Conference, General Track, pp. 1–14 (2001)
5. Russell, R.: virtio: towards a de-facto standard for virtual I/O devices. ACM SIGOPS Operating Systems Review **42**(5), 95–103 (2008)
6. Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., Williamson, M.: Safe hardware access with the Xen virtual machine monitor. In: 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), pp. 1–1 (2004)
7. Yassour, B.-A., Ben-Yehuda, M., Wasserman, O.: Direct device assignment for untrusted fully-virtualized virtual machines. Tech. rep. (2008). IBM Research Report H-0263, Tech. Rep.
8. Gordon, A., Har'El, N., Landau, A., Ben-Yehuda, M., Traeger, A.: Towards exitless and efficient paravirtual I/O. In: Proceedings of the 5th Annual International Systems and Storage Conference, p. 10. ACM (2012)
9. Dong, Y., Yu, Z., Rose, G.: SR-IOV networking in Xen: Architecture, design and implementation. In: Workshop on I/O Virtualization (2008)
10. Ongaro, D., Cox, A.L., Rixner, S.: Scheduling I/O in virtual machine monitors. In: Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp. 1–10. ACM (2008)
11. Cherkasova, L., Gupta, D., Vahdat, A.: Comparison of the three CPU schedulers in Xen. SIGMETRICS Performance Evaluation Review **35**(2), 42–51 (2007)
12. Cherkasova, L., Gupta, D., Vahdat, A.: When virtual is harder than real: Resource allocation challenges in virtual machine based it environments Hewlett. Packard Laboratories, Tech. Rep. HPL-2007-25 (2007)
13. Demers, A., Keshav, S., Shenker, S.: Analysis and simulation of a fair queueing algorithm. ACM SIGCOMM Computer Communication Review **19**(4), 1–12 (1989)
14. Parekh, A.K., Gallager, R.G.: A generalized processor sharing approach to flow control in integrated services networks: the single-node case. IEEE/ACM Transactions on Networking (ToN) **1**(3), 344–357 (1993)
15. Bennett, J.C., Zhang, H.: WF^{2}Q: worst-case fair weighted fair queueing. In: INFOCOM'96. 15th Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE, vol. 1, pp. 120–128. IEEE (1996)
16. Goyal, P., Vin, H.M., Chen, H.: Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In: ACM SIGCOMM Computer Communication Review, vol. 26, pp. 157–168. ACM (1996)
17. Duda, K.J., Cheriton, D.R.: Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In: ACM SIGOPS Operating Systems Review, vol. 33, pp. 261–276. ACM (1999)

18. Shreedhar, M., Varghese, G.: Efficient fair queuing using deficit round-robin. IEEE/ACM Trans. Networking **4**(3), 375–385 (1996)
19. Zhang, Y.J.: A multi-server scheduling framework for resource allocation in wireless multi-carrier networks. IEEE Trans. Wirel. Commun. **6**(11), 3884–3891 (2007)
20. Chandra, A., Adler, M., Goyal, P., Shenoy, P.: Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4, pp. 4–4. USENIX Association (2000)
21. Mohanty, S.R., Bhuyan, L.N.: Fair scheduling over multiple servers with flow-dependent server rate. In: Proceedings 2006 31st IEEE Conference on Local Computer Networks, pp. 73–80. IEEE (2006)
22. Li, T., Baumberger, D., Hahn, S.: Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In: ACM Sigplan Notices, vol. 44, pp. 65–74. ACM (2009)
23. Parekh, A.K., Gallagher, R.G.: A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. IEEE/ACM Transactions on Networking (TON) **2**(2), 137–150 (1994)
24. Blanquer, J.M., Özden, B.: Fair queuing for aggregated multiple links. ACM SIGCOMM Computer Communication Review **31**(4), 189–197 (2001)
25. Verma, A., Kaushal, S.: Cost-time efficient scheduling plan for executing workflows in the cloud. Journal of Grid Computing, 1–12 (2015)
26. Tang, Z., Qi, L., Cheng, Z., Li, K., Khan, S.U., Li, K.: An energy-efficient task scheduling algorithm in dvfs-enabled cloud environment. Journal of Grid Computing, 1–20 (2015)
27. Li, K.: Optimal load distribution for multiple heterogeneous blade servers in a cloud computing environment. Journal of grid computing **11**(1), 27–46 (2013)
28. Caprita, B., Nieh, J., Stein, C.: Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling. In: Proceedings of the 25th annual ACM symposium on Principles of distributed computing, pp. 72–81. ACM (2006)
29. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: NSDI, vol. 11, pp. 24–24 (2011)
30. Ghodsi, A., Sekar, V., Zaharia, M., Stoica, I.: Multi-resource fair queueing for packet processing. ACM SIGCOMM Computer Communication Review **42**(4), 1–12 (2012)
31. Yap, K.-K., McKeown, N., Katti, S.: Multi-server generalized processor sharing. In: Proceedings of the 24th International Teletraffic Congress, p. 29. International Teletraffic Congress (2012)
32. Zeldes, Y., Feitelson, D.G.: On-line fair allocations based on bottlenecks and global priorities. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, pp. 229–240. ACM (2013)
33. Lai, C.A., Wang, Q., Kimball, J., Li, J., Park, J., Pu, C.: IO performance interference among consolidated n-tier applications: Sharing is better than isolation for disks. In: 2014 IEEE 7th International Conference on Cloud Computing, pp. 24–31. IEEE (2014)
34. Kim, H., Lim, H., Jeong, J., Jo, H., Lee, J.: Task-aware virtual machine scheduling for i/o performance. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp. 101–110. ACM (2009)
35. Tan, H., Li, C., He, Z., Li, K., Hwang, K.: VMCD: A virtual multi-channel disk i/o scheduling method for virtual machines (2015)
36. Nikolova, D., Blondia, C.: Bonded deficit round robin scheduling for multi-channel networks. Comput. Netw. **55**(15), 3503–3516 (2011)
37. Yap, K.-K., Huang, T.-Y., Yiakoumis, Y., Chinchali, S., McKeown, N., Katti, S.: Scheduling packets over multiple interfaces while respecting user preferences. In: Proceedings of the ninth ACM conference on Emerging networking experiments and technologies, pp. 109–120. ACM (2013)
38. Bertsekas, D.P., Gallager, R.G., Humblet, P.: Data networks, vol. 2. Prentice-Hall International, New Jersey (1992)
39. Shreedhar, M., Varghese, G.: Efficient fair queueing using deficit round robin. SIGCOMM Comput. Commun. Rev. **25**(4), 231–242 (1995). [Online]. Available: doi:10.1145/217391.217453
40. Shreedhar, M., Varghese, G.: Efficient fair queueing using deficit round robin. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, ser. SIGCOMM '95, pp. 231–242. ACM, New York, NY, USA (1995). [Online]. Available: doi:10.1145/217382.217453