Analysis and Evaluation of I/O Hypervisor Scheduling

Konstantinos Kontodimas*, Panagiotis Kokkinos*†, Yossi Kuperman‡ and Emmanouel Varvarigos*†
*Computer Engineering and Informatics Department, University of Patras, Greece
Email: {kontodimas, kokkinop, manos}@ceid.upatras.gr
†Computer Technology Institute & Press "Diophantus", University Campus, Patras, Greece
‡IBM Research and Development, Haifa, Israel
Email: yossiku@il.ibm.com

***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.**

I. 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 block for creating 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 datacenters that consist of several thousands of physical machines. The computational, storage and network resources of a datacenter are orchestrated using Cloud software like OpenStack [1], which among others interacts with the hypervisor running on each physical machine for managing the VMs. Task 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 processes each one.

In our work, we implement and evaluate a number of online fair scheduling algorithms for the IOH. For this purpose, the IOH (Section III) is formulated as a multiple-queues and multiple-servers system (Fig. 1). 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 build 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 II we report on previous work. The I/O Hypervisor

978-0-7695-5697-0/15 $31.00 © 2015 IEEE
DOI 10.1109/UCC.2015.19

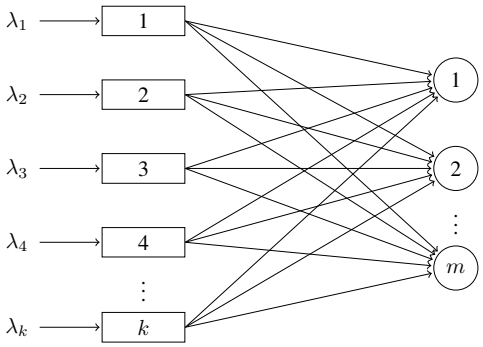


Fig. 1: A graphical representation of a multiple-queues and multiple-servers system.

problem is formulated in Section III. Also, I/O Hypervisor's performance is theoretically analyzed in Section IV. Section V describes the implemented scheduling algorithms. Sections VI and VII present the simulator used for the evaluations and the results. Finally, the paper is concluded in Section VIII.

II. Previous work

Three major techniques are common for hosts to virtualize I/O services for their guests: emulation [3], where a familiar device (e.g., a common network card) is emulated, paravirtualization [4], 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 [5], [6], 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 [7]. In the I/O paravirtualization model we consider in our work, we combine SR-IOV [8] that provides near native I/O performance, together with paravirtual I/O technologies [3], [4], 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 [9]. 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.* [10][11] 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 [9], 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 [12][13], WF^{2}Q [14], SFQ [15], BVT [16], DRR [17]) 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 [18]. Multi-resource contention also arises 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 III) as a multiple-queues and multiple-servers system (Fig. 1), which correspond to the various virtual and physical devices and the cores that serve their I/O operations. A major issue in the above is how requests from different flows 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 [19], [20], [21]. 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 [19].

A number of multi-resource extensions of single-resource fair scheduling algorithms have been proposed over the years. In [22] GPS scheduling was extended to the case of multiple nodes. In [23], the authors extended Weighted 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.

The work in [24] focuses on the allocation of resources of different types and proposes Dominant 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) [25] 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 [26], 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 [27], a greedy algorithm for the fair allocations of resources based on bottlenecks is defined.

A number of works extend DRR for the multi-resource scenario [28] [29]. In [29], 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.

III. Problem Formulation

A. 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. 2.

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

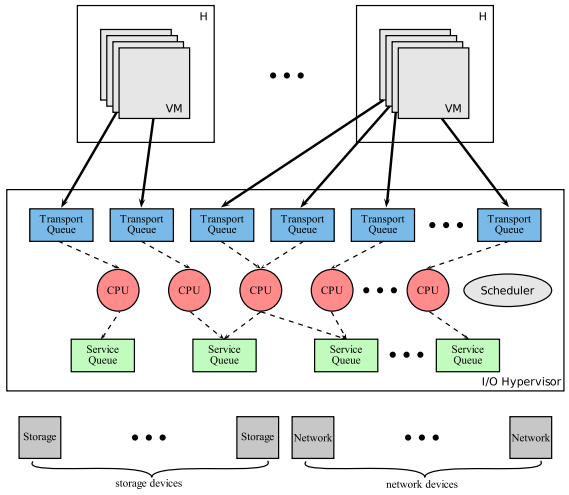


Fig. 2: The I/O Hypervisor's architecture.

time according to scheduling. 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.

B. 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.

Requests r are generated for each transport queue t with rate λ_t requests/sec (for each service queue s with λ_s responses/sec). Each request r belongs to a particular type (read, write, storage, network) with probability $p(r)$ and carries B_r bits on average. The load on a transport queue t is $\lambda_t \sum_{r \in Q_t} p(r)\mathbb{E}(X_r)$, where Q_t are the types of requests (network, storage). The load on a service queue s is $\lambda_s \sum_{r \in Q_s} p(r)\mathbb{E}(X_r)$. The conditions (1), (2) and (3) determine the maximum (type-dependent) load that can be handled by the I/O Hypervisor, without having any drops;

$$\lambda_t \sum_{q \in Q_t} p(q)\mathbb{E}(X_q) < 1 \quad (1)$$

$$\lambda_s \sum_{q \in Q_s} p(q)\mathbb{E}(X_q) < 1 \quad (2)$$

$$\sum_t \lambda_t \sum_{q \in Q_t} p(q)\mathbb{E}(X_q) + \sum_s \lambda_s \sum_{q \in Q_s} p(q)\mathbb{E}(X_q) < K \quad (3)$$

C. 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.

IV. I/O Hypervisor Performance

A. Analysis

As mentioned, the I/O Hypervisor (IOH) suggests a different architectural model of operation from the classical hypervisor, in which memory and I/O resources used by a guest Virtual Machine (VM) are provided by an external host, instead by the local one. This also means that 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. 3, 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 [30]. 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, Poisson arrival rate λ and service rate μ is an acceptable assumption. 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 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

$$T = \frac{1}{\mu} + \frac{P_Q}{c\mu - \lambda} \quad (4)$$

$$\hat{T} = \frac{1}{\mu} + \frac{\hat{P}_Q}{k\mu - N\lambda} \quad (5)$$

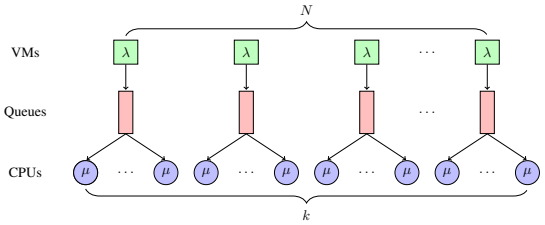
We should note that for the IOH, the load of the system is equal to the accumulated load ($N\lambda$), since we assumed for our analysis that all I/O operations are placed in a single queue.

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 [30]), 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

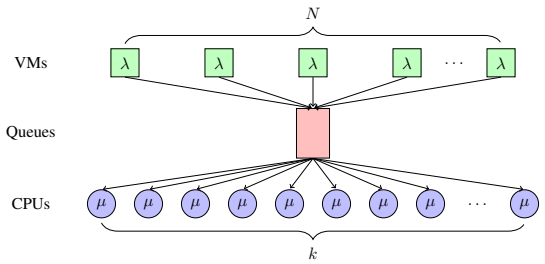
$$\frac{T}{\hat{T}} \cong \frac{k\mu - N\lambda}{c\mu - \lambda} \Rightarrow \frac{T}{\hat{T}} \cong N \quad (6)$$

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



(a) Classical model



(b) I/O Hypervisor model

Fig. 3: The classical and the I/O Hypervisor models.

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

$$\frac{\tilde{T}}{\hat{T}} \cong \frac{k\mu - N\lambda}{\mu - \lambda} = k + \frac{(k-N)\lambda}{\mu - \lambda} \Rightarrow \frac{\tilde{T}}{\hat{T}} \gg N \qquad (7)$$

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.

Of course 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. Also, in addition to the queuing delay there is the transfer delay of each I/O operation from the original physical machine to the IOH that also hinders the performance. However, this delay is relatively insignificant compared to the processing time of an I/O operation, and we expect that emerging fast interconnects with lower latency and higher throughput will make it even more negligible.

B. 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 datacenter environment this delay is expected to be low, even under heavy I/O workloads. Also, the Layer 2 communication used between the VMs and the IOH induces low additional control overhead, in comparison to Layer 3 communication. 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 which cores will process which 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 serve the IOH queues in a fair manner, providing in this way indirectly Quality of Service (QoS) to the corresponding virtual devices.

The main objectives of this work are described below:

- fairness on service capacity allocation between queues
- minimization of mean queuing delay fir I/O requests/responses
- capability of giving priorities to requests/responses of a particular type (or from a particular VM)
- 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 [29], 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 capacity available. 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 for discovering the number of CPU cores required to serve a particular number of queues.

V. Implemented Algorithms

A. The miDRR-IOH Algorithm

The *multiple interface Deficit Round Robin-I/O Hypervisor*, or miDRR-IOH weighted max-mix fair algorithm, is a variance of miDRR introduced by Yap *et al.* in [29], a generalized form of the classic Deficit Round Robin [31][32] for the case of multiple servers. miDRR is adapted based on the characteristics of the I/O Hypervisor (Section III), 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. The core serves the 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 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 it is served by another core, the flag is reset. This last step is repeated, 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.

B. Other Algorithms

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 is reduced 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.

Symbol	Description
BL_i	Backlog of flow i
$Size_i$	Size of flow i's head-of-line packet
Q_i	Quantum for flow i
DC_i	Deficit counter for flow i
$\mathcal{F}_j$	Set of flows willing to use interface j
$\mathcal{C}_j$	Current flow interface j is serving
$\mathcal{B}$	Set of backlogged flows
SF_{ij}	Interface j's service flag for flow i (Service flags for new flows are initiated to zero.)

Algorithm 1 miDRR-IOH(j)

1: **if** $\mathcal{F}_j \cap \mathcal{B} = \emptyset$ **then**
2: **return**
3: **end if**
4: $i \leftarrow \mathcal{C}_j$
5: **if** $Size_i \leq DC_i$ **then**
6: Send $Size_i$ bytes
7: $DC_i \leftarrow DC_i - Size_i$
8: **end if**
9: **if** $BL_i = 0$ **then**
10: $DC_i \leftarrow 0$
11: Remove i from $\mathcal{B}$
12: **end if**
13: **if** $BL_i = 0$ **or** $Size_i \geq DC_i$ **then**
14: $C_j \leftarrow$ miDRR-IOH-Check-Next(i, j)
15: $i \leftarrow C_j$
16: $DC_i \leftarrow DC_i + Q_i$
17: **end if**

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

1: $(s, d) \leftarrow$ Next device pair with s being backlogged
2: $i \leftarrow s$
3: $C_j \leftarrow i$
4: **while** $SF_{ij} \neq 0$ **do**
5: $SF_{ij} \leftarrow 0$
6: $(s, d) \leftarrow$ Next device pair with s being backlogged
7: $i \leftarrow s$
8: $C_j \leftarrow i$
9: **end while**
10: $SF_{ik} \leftarrow 1, \forall k \neq j$
11: **return** (i)

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 run queues of other cores 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.

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 isn't any available core, 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 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.

VI. I/O Hypervisor Simulator

A. 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 III). It also allows generation of customized workload traffic from the I/O devices (storage and network). Using the simulator we have implemented the algorithms described in Section V that take decisions on the I/O operations processing priorities.

B. Simulation process

The user interacts with the simulator via an internal console, or an input script, using the built-in command set. Initially, the user sets up the parameters of the system, by creating VMs, transport queues (that correspond to virtual devices), 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 I. The workload of each device is also defined as a 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 multi-level logging subsystem that enables viewing of the internal operations of the simulator.

TABLE I: Simulator Entities Attributes

Device	Attribute	Default Value
VM	weight	1.0
Core	capacity	1500.0
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

VII. 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 has been implemented in Python 2.7.6 and the experimental runs took place on a VMware ESXi VM with 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 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 some 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 II 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 iteratively, 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 II: Simulation Parameters

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

A. Experiment 1

In the first set of experiments, we evaluated the miDRR-IOH, Credit and FCFS algorithms performance as the request sizes (for block-device and network-packet operations) scale. The actual sizes of block and packet requests used for our experiments, are given in TABLE III, 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 II).

TABLE III: 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

Fig. 4 illustrates the standard deviation of capacity share per device pair, while increasing the differentiation between the size of block and network requests (according to TABLE III). 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 request size ratio grows (TABLE III), miDRR-IOH keeps a similar capacity share allocation between the device pairs (s, t), while FCFS deviates proportionally to the request size ratio. Credit scheduling policy has a deviation on the capacity allocation, thus it does not achieve near-optimal fairness as miDRR-IOH does. However, this deviation is bounded and is not affected by the request size ratio. We should remind that each (t, s) pair corresponds to a particular virtual device of a VM, so 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' sizes and as a result in the end the (t, s) pairs that generate larger requests are more benefited than those that generate smaller ones.

The deviation of capacity share ratio, introduced in the experiments conducted with credit policy, is due to the fact that credit updating is taking place in a fixed time-slice,

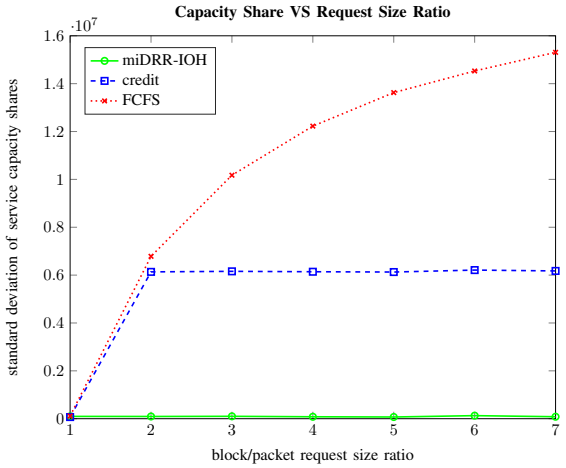


Fig. 4: Standard deviation of capacity share per device pair, while increasing the differentiation on the sizes of requests.

TABLE IV: Devices Cardinality of VMs

	# 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

independent from the time required a request to be processed. This results to a lower granularity than miDRR-IOH's which is reflected as a relaxation in the accuracy of the results. Therefore the existence of requests of diverse lengths, results to this deviation of capacity share allocation. However, the way that credit policy prioritizes devices, bounds this deviation.

Also, the miDRR-IOH and Credit algorithms achieve shorter, on average, queuing time for the requests than FCFS does, as it is shown in Fig. 5. This is due to the fact that these two policies also take into account the sizes of the requests, 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 do that results in keeping cores busy, for arbitrarily long time periods, amplifying the average waiting times for the rest of the devices.

B. Experiment 2

Next, we evaluated miDRR-IOH algorithm using different weight values (TABLE IV) 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. 6 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 as the capacity allocated

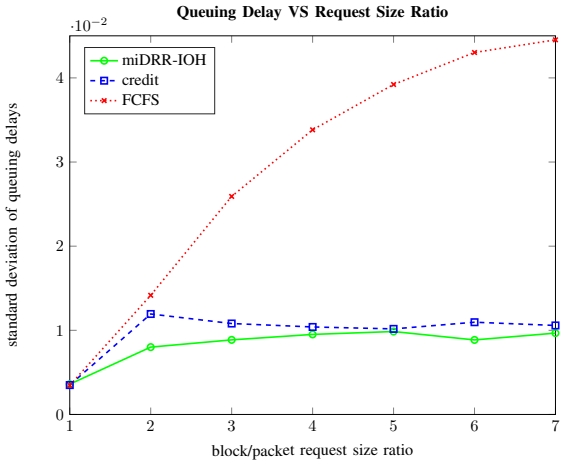


Fig. 5: Standard deviation of average queuing delay per device, while increasing the differentiation on the sizes of requests.

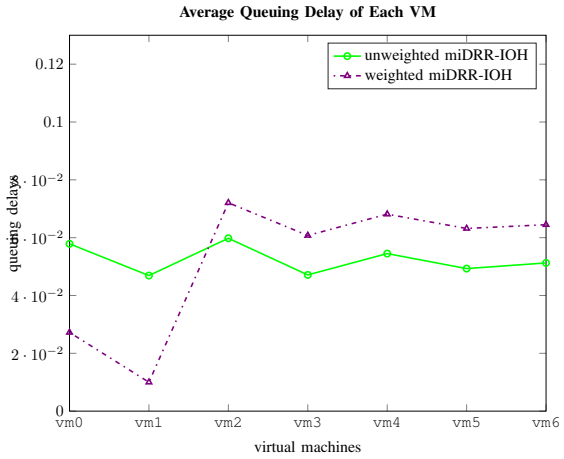


Fig. 7: Average queuing delay of each VM, while using weights on `vm0` and `vm1`.

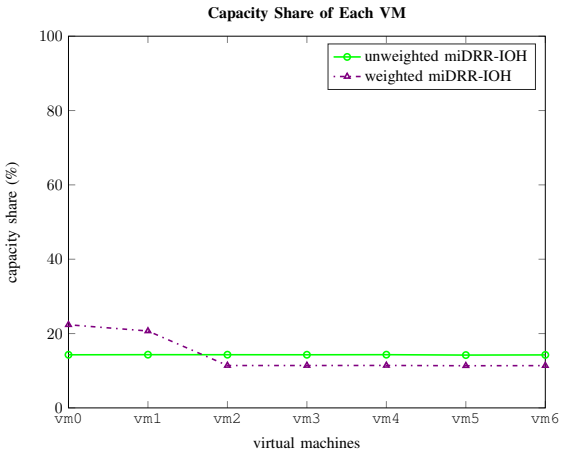


Fig. 6: Capacity share percentage allocated to each VM, while using weights on `vm0` and `vm1`.

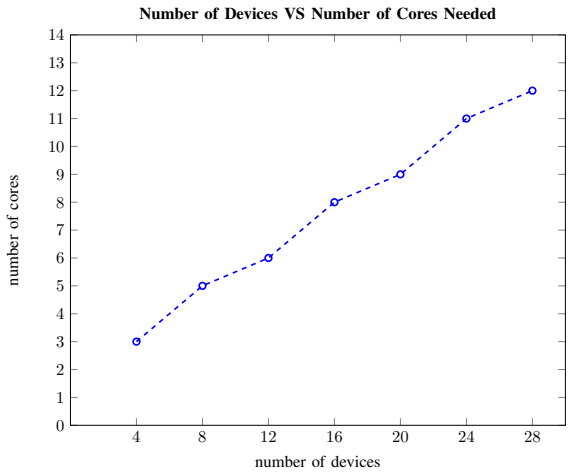


Fig. 8: Minimum number of cores required to serve the traffic generated by a variable number of devices.

to the rest of the 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. 7 that requests generated by the devices that belong to `vm0` and `vm1` are served faster than requests from other devices (and corresponding VMs).

C. Experiment 3

In the third set of experiments, we evaluated the number of cores required to serve a given number of devices (Fig. 8) using the miDRR-IOH algorithm, so as to avoid any dropped requests. All devices used for this experiment were configured with the same characteristics (e.g., generated workload), using the parameters of TABLE II, while both block and packet request sizes are set to 50000 bytes. It is evident that the minimum number of cores required to serve the generated I/O requests increases linearly to the number of devices.

VIII. 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 in comparison 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 similary 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 allocated CPU core capacity, 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. [Online]. Available: https://www.openstack.org/

[2] (2013) ORBIT project. [Online]. Available: http://www.orbitproject.eu/

[3] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on VMware Workstation's hosted virtual machine Monitor." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 1–14.

[4] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "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)*, 2004, pp. 1–1.

[6] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman, "Direct device assignment for untrusted fully-virtualized virtual machines," Tech. rep., IBM Research Report H-0263, Tech. Rep., 2008.

[7] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger, "Towards exitless and efficient paravirtual I/O," in *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 2012, p. 10.

[8] Y. Dong, Z. Yu, and G. Rose, "SR-IOV networking in Xen: Architecture, design and implementation." in *Workshop on I/O Virtualization*, 2008.

[9] 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*. ACM, 2008, pp. 1–10.

[10] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," *SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, 2007.

[11] ——, "When virtual is harder than real: Resource allocation challenges in virtual machine based it environments," *Hewlett Packard Laboratories, Tech. Rep. HPL-2007-25*, 2007.

[12] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4. ACM, 1989, pp. 1–12.

[13] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Transactions on Networking (ToN)*, vol. 1, no. 3, pp. 344–357, 1993.

[14] J. C. Bennett and H. Zhang, "WF^{2}Q: worst-case fair weighted fair queueing," in *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, vol. 1. IEEE, 1996, pp. 120–128.

[15] P. Goyal, H. M. Vin, and H. Chen, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," in *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 4. ACM, 1996, pp. 157–168.

[16] K. J. Duda and D. R. Cheriton, "Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler," in *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5. ACM, 1999, pp. 261–276.

[17] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *Networking, IEEE/ACM Transactions on*, vol. 4, no. 3, pp. 375–385, 1996.

[18] Y. J. Zhang, "A multi-server scheduling framework for resource allocation in wireless multi-carrier networks," *Wireless Communications, IEEE Transactions on*, vol. 6, no. 11, pp. 3884–3891, 2007.

[19] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "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*. USENIX Association, 2000, pp. 4–4.

[20] S. R. Mohanty and L. N. Bhuyan, "Fair scheduling over multiple servers with flow-dependent server rate," in *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*. IEEE, 2006, pp. 73–80.

[21] T. Li, D. Baumberger, and S. Hahn, "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin," in *ACM Sigplan Notices*, vol. 44, no. 4. ACM, 2009, pp. 65–74.

[22] A. K. Parekh and R. G. Gallagher, "A generalized processor sharing approach to flow control in integrated services networks: the multiple node case," *IEEE/ACM Transactions on Networking (TON)*, vol. 2, no. 2, pp. 137–150, 1994.

[23] J. M. Blanquer and B. Özden, "Fair queuing for aggregated multiple links," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 189–197, 2001.

[24] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, vol. 11, 2011, pp. 24–24.

[25] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 1–12, 2012.

[26] K.-K. Yap, N. McKeown, and S. Katti, "Multi-server generalized processor sharing," in *Proceedings of the 24th International Teletraffic Congress*. International Teletraffic Congress, 2012, p. 29.

[27] Y. Zeldes and D. G. Feitelson, "On-line fair allocations based on bottlenecks and global priorities," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 2013, pp. 229–240.

[28] D. Nikolova and C. Blondia, "Bonded deficit round robin scheduling for multi-channel networks," *Computer Networks*, vol. 55, no. 15, pp. 3503–3516, 2011.

[29] K.-K. Yap, T.-Y. Huang, Y. Yiakoumis, S. Chinchali, N. McKeown, and S. Katti, "Scheduling packets over multiple interfaces while respecting user preferences," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 109–120.

[30] D. P. Bertsekas, R. G. Gallager, and P. Humblet, *Data networks*. Prentice-Hall International New Jersey, 1992, vol. 2.

[31] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 4, pp. 231–242, Oct. 1995. [Online]. Available: http://doi.acm.org/10.1145/217391.217453

[32] ——, "Efficient fair queueing using deficit round robin," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '95. New York, NY, USA: ACM, 1995, pp. 231–242. [Online]. Available: http://doi.acm.org/10.1145/217382.217453