vFair: Latency-Aware Fair Storage Scheduling via Per-IO Cost-Based Differentiation

Hui Lu[†], Brendan Saltaformaggio[†], Ramana Kompella[†‡] *, Dongyan Xu[†]

[†]Department of Computer Science, Purdue University, [‡]Google Inc.

Abstract

In virtualized data centers, multiple VMs are consolidated to access a shared storage system. Effective storage resource management, however, turns out to be challenging, as VM workloads exhibit various IO patterns and diverse loads. To multiplex the underlying hardware resources among VMs, providing **fairness and isolation** while maintaining high resource **utilization** becomes imperative for effective storage resource management. Existing schedulers such as Linux CFQ or SFQ can provide some fairness, but it has been observed that synchronous IO tends to lose fair shares significantly when competing with aggressive VMs.

In this paper, we introduce vFair, a novel scheduling framework that achieves IO resource sharing fairness among VMs, regardless of their IO patterns and workloads. The design of vFair takes per-IO cost into consideration and strikes a balance between fairness and storage resource utilization. We have developed a Xen-based prototype of vFair and evaluated it with a wide range of storage workloads. Our results from both micro-benchmarks and real-world applications demonstrate the effectiveness of vFair, with significantly improved fairness and high resource utilization.

Categories and Subject Descriptors D.4.4 [*Operating Systems*]: Communications Management—Input/Output

General Terms Design, Measurement, Performance

Keywords Virtualization, Cloud Computing, Scheduling, I/O

* Contributed to the work while at Purdue University.

SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA.
© 2015 ACM. ISBN 978-1-4503-3651-2/15/08. . . $15.00.
DOI:http://dx.doi.org/10.1145/2806777.2806943

1. Introduction

Virtualization-based server consolidation has become a common practice in today's cloud data centers. Sharing physical hardware among many virtual machines (VMs) offers several benefits such as improved utilization (hence, better return-on-investment) and agility in resource provisioning required for modern applications. Further, typical cloud data centers rely on a high degree of resource sharing (e.g., 40 VMs/physical server [13]), which is critically hinged on providing isolation as well as fairness in resource allocation across individual VMs.

Resource sharing in virtualized cloud environments is largely based on *proportional sharing*. For instance, several existing schedulers [19, 21, 35] focus on proportionally sharing available storage IO resources via allocations of IO throughput (bytes/s), IOPS (IO operations/s), or IO queue depth, and are typically work-conserving in nature.

On the surface, all of these approaches appear to provide fair access to storage resources. Indeed, as long as contending VM workloads exhibit high IO-concurrency (e.g., when IO requests are *asynchronous* in nature), such schedulers can provide reasonable fairness across VMs. However, if a VM's workload exhibits low IO-concurrency (e.g., Web or database servers with *synchronous* IO requests), the work-conserving property can cause synchronous requests to be arbitrarily delayed by interleaved batches of asynchronous IO requests from competing VMs — thereby causing significant unfairness. This situation becomes worse as asynchronous workloads become more aggressive either with increased IO-concurrency and/or IO-size. In public clouds, malicious VMs can even exploit this fact to starve other VMs with synchronous IO workloads.

One possible mitigation is to provide *strict isolation* via reservations and limits [22] or throttling [28]. Unfortunately, static reservations and limits compromise the work-conserving property, and thus do not fully utilize the available resources. Even ignoring the low utilization, it is not clear how to set correct reservations and limits, as performance is dependent on several confounding factors such as whether the IO accesses are sequential or random, synchronous or asynchronous.

To overcome these challenges we present **vFair**, a novel *block-level* proportional share scheduling framework. vFair aims to provide both strict proportional sharing of storage IO among multiple VMs and high storage utilization, regardless of IO patterns (asynchronous, synchronous, or a hybrid). The design of vFair is based on the following key idea: Intuitively, each VM's workload can be modeled as an IO pattern (i.e., combinations of read/write, sequential/random, synchronous/asynchronous). Further, given a particular IO pattern, we can calculate the saturation throughput P_i which would fully utilize the storage subsystem. Essentially, P_i represents the storage system's upper bound on the throughput of a distinct IO pattern *regardless of whether the VM's workload can reach this throughput*. Thus, if we know the P_i for each VM's IO pattern, it makes intuitive sense to allocate storage resources based on their P_i and proportional share percentages (i.e., each VM receives a portion of its *theoretical* peak isolation throughput).

For example, consider two VMs with saturation throughputs P_1 and P_2 in isolation on a given storage device. By assigning 50% : 50% shares, the VMs should ideally receive $P_1/2$ and $P_2/2$ throughputs respectively when sharing the same storage device. More generally, vFair defines *fair allocations* as a guaranteed portion of each VM's P_i based on their share percentages (in Equation 2).

While the above objective makes conceptual sense, enforcing such sharing fairness is not easy in practice. We cannot know P_i *a priori* for each VM, and in fact P_i changes continuously depending on the VM's workload. This is precisely why existing schedulers employ indirect metrics such as IOPS and IO throughput, which unfortunately do not work for all workloads as discussed before. vFair solves this by defining a practical *per-IO cost allocation model* based on a VM's observed IO pattern. vFair builds each IO pattern from several basic IO types (e.g., sequential/random; reads/writes) and uses a simple approximation function to estimate the saturation throughput (P_i) for combinations of these IO types. Based on such a baseline model, vFair approximates the "ideal" fairness share target for each VM, and then at runtime vFair uses an adaptive feedback controller to dynamically adjust the model based on system utilization. Further, vFair ensures isolation for each VM's IO requests — preventing performance interference. To accomplish this, vFair uses a credit-based rate controller to regulate incoming IO requests to meet the predefined IO allocation. vFair then uses a fair queuing scheduler for scheduling the admitted IOs from different VMs.

To the best of our knowledge, vFair is among the first to explicitly consider saturation throughput for individual VMs to ensure fair sharing and high utilization of the storage subsystem. We have implemented a full vFair prototype in the driver domain of Xen, with portability to other hypervisors. Our evaluation with application benchmarks and real-world workloads shows that vFair significantly improves fairness. For instance, we will show that, compared to CFQ [3], vFair can improve fairness of storage IO scheduling by *an order of magnitude* (e.g., the normalized proportional share ratio increases from 0.03 to 0.92 for the Postmark application) while keeping storage fully utilized.

2. Motivation

2.1 Background

The basic aspects to describe storage access patterns are direction (read or write), location (sequential or random), parallelism (synchronous or asynchronous), and block size (I/O request size). Specifically, storage IO (a read or write operation) is broadly classified as sequential or random. Sequential IO is typical of large file reads/writes and involves operating on one block immediately after its neighbor. Conversely, random IO involves large numbers of seeks and rotations (for spinning devices) and is often much slower.

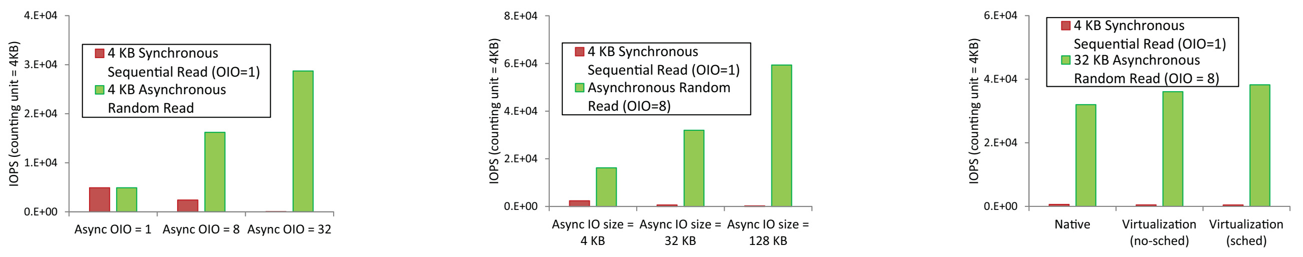
Orthogonally, there are two types of synchronization: synchronous IO (cannot be backlogged) and asynchronous IO (can be backlogged). In synchronous IO, a thread starts an IO operation and enters a wait state until the IO request has completed — such workloads are said to have "low IO-concurrency". While in asynchronous IO, a thread sends an IO request to the kernel and continues processing another job, hence "high IO-concurrency".

IO request size is a generic term to describe the amount of data an application reads from or writes to a storage device. Typically, it is more efficient to read fewer large records than many small ones. However, a large IO request is slower than a small one due to the device's data transfer rate.

2.2 Illustration of Unfairness

To illustrate the unfairness incurred by various IO access patterns, Figure 1 compares the throughput in terms of IOPS observed in several experiments. We use two micro-workloads: Fio [8] issuing synchronous read requests (simulating low IO-concurrency) and aio-stress [2] issuing asynchronous random read requests (simulating high IO-concurrency). We use a single SSD which is *more favorable* to traditional schedulers than HDD as random seeks are much more expensive on the latter. Both workloads run in a single thread, each with the same share weight — Ideally, they should obtain nearly the same throughput in a fair share manner.

In Figure 1(a), we fix the IO request size for both workloads (4 KB) and only vary the outstanding IO (OIO) number of the asynchronous workload from 1 to 32 (i.e., increasing the IO concurrency). Intuitively, the OIO number of the synchronous workload is always 1. The workloads are run on native Linux with the default CFQ time-slice based (non-work-conserving) IO scheduler — the "anticipating" scheduler (i.e., adds idle time at the end of the synchronous IO) favors synchronous requests. Figure 1(a) shows that, as the OIO number of the asynchronous work-



(a) Varying IO concurrency (OIO #). (b) Varying IO size. (c) Native vs virtualization setup.

Figure 1. IO performance unfairness due to various IO parameters. As IO concurrency (a) or IO size (b) increases, asynchronous workloads dominate synchronous workloads. Further, virtualization compounds the unfairness (c).

load increases, the throughput of the synchronous workload degrades significantly from ∼5,000 to less than 100 IOPS. In contrast, the throughput of the asynchronous workload increases from ∼5,000 to ∼28,730. This clearly indicates that as asynchronous workloads become more aggressive (i.e., with larger OIO) the synchronous workloads receive increasingly unfair service times.

In Figure 1(b), using the *same* system setup as before, we instead fix the IO concurrency (8 for the asynchronous workload) and only vary the IO size of the asynchronous workload from 4 KB to 128 KB. Again, because synchronous workloads wait for the previous request to complete, the synchronous workload's OIO is always 1. These results show that as the IO size of the asynchronous workload increases, the throughput of the synchronous workload subsequently degrades from ∼2,400 to ∼200. In sharp contrast, the throughput of the asynchronous workload increases from ∼16,200 to ∼59,360. This indicates that even non-aggressive asynchronous workloads with large request sizes can cause synchronous workloads' throughput to suffer. Note that, although CFQ fairly allocates time slices between the synchronous queue and the asynchronous queue [3], it still cannot guarantee both queues obtain *the same storage service time* resulting in unfairness.

Figure 1(c) shows the impact of only virtualization overhead when we fix both the OIO number and IO size. Three basic scenarios are compared: (1) the native non-virtualized case, (2) the virtualized case without CPU scheduling impact, and (3) with CPU scheduling impact (the most typical scenario in practice [38]). In the virtualized experiments, we run the workloads in 2 VMs separately, with the same configurations as the native case. For the case without CPU scheduling impact, we pin each vCPU to a dedicated pCPU, while for the case with scheduling impact we let all vCPUs share all available pCPUs. Figure 1(c) shows that virtualization overhead *further penalizes* synchronous workloads. The synchronous workload achieves 20% and 30% fewer IOPS compared to the native synchronous workload without and with scheduling impact, respectively. On the contrary, the asynchronous workload receives more IOPS compared to the native one. Further, as more VMs compete for the pCPUs the throughput penalty for the synchronous workloads will become even more significant.

2.3 Investigating the Cause of Unfairness

Next, we take a deeper look into the cause of unfairness in Figure 2. Here we consider two VMs sharing the same storage device — VM1 issuing synchronous IO requests and VM2 issuing asynchronous requests. Since the IO requests from VM1 are synchronous (i.e., requests come one by one), the arrival of each request, req_{i+1}, is delayed by the processing time of the preceding request, req_i. Note that this processing time, $t(req_i)$, includes the amounts of time $t1(req_i)$, consumed across the software IO stack (e.g., guest IO subsystem, kernel IO subsystem, IO scheduler, etc.), and $t2(req_i)$ the latency at the device (e.g., HDD, SSD, RAID or the virtualized software devices). On the other hand, the arrival of asynchronous IO requests from VM2 are not delayed since the thread(s) may continue to work. Hence, the asynchronous requests accumulate much faster than the synchronous requests in the IO scheduler's request queues.

Work-conserving schedulers (e.g., FCFS, deadline, and SFQ(D)) are idle only when there is no IO traffic to send. Such schedulers frequently switch to process the asynchronous requests during the *deceptive idleness period*[1] of the synchronous requests. Interestingly, we observe that $t2(req_i)$ is proportional to $t1(req_i)$. Specifically, as the value of $t1(req_i)$ increases, more asynchronous requests are dispatched to the storage devices by the work-conserving scheduler, causing increased delays in the processing time of the synchronous requests (i.e., $t2(req_i)$). In consequence, the work-conserving property causes VM1 to issue requests at a *deceptively low rate*. This causes the arrival of a synchronous request to be arbitrarily delayed by a batch of asynchronous IO requests from competing VMs.

Even with non-work-conserving schedulers (e.g., CFQ [3] and other anticipatory schedulers) that allow for idle time at the end of each synchronous IO in anticipation of subsequent requests, this unfairness will still occur when $t1(req_i)$ + $t2(req_i)$ is larger than the idle time (which cannot be set too large for storage efficiency). Yet $t1(req_i)$ may become arbitrarily large due to additional latency incurred by: (1) the virtualized IO stack in the guest and (2) scheduling contention at the hypervisor — worsening the delay situation.

[1] A condition where work-conserving schedulers incorrectly assume that the last request issuing process has no further requests.

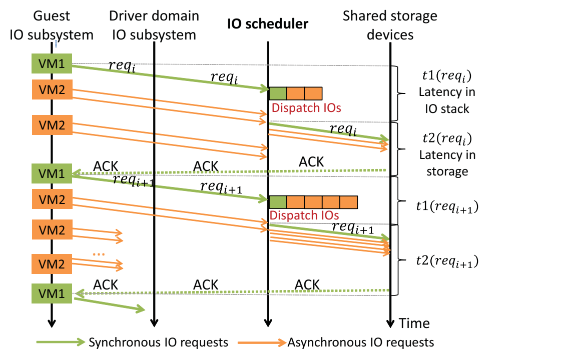


Figure 2. An illustration of unfairness.

The latency from (1) is well known and has been optimized with every new version of a hypervisor, but beyond a certain point this cannot be further reduced. Typically, the virtualized IO stack (KVM [11]) adds 10% more overhead than native. Latency from (2) was observed in related work [37] to be on the order of tens of milliseconds, depending on the CPU quantum allocated to individual VMs. This cannot be easily avoided as long as CPU cores are shared across VMs.

To sum up, the unfairness problem is mainly due to two factors. **(1) Existing IO schedulers do not take various IO access patterns into account**. Yet as our observations in Figure 1 show, the resource costs of different IO access patterns diverge dramatically and cause unfair sharing. Further, simply using IOPS (as most proportional share schedulers do) to divide up storage service time hardly guarantees resource fair allocation among multiple VMs. For example, to distinguish IO requests with different sizes, Amazon EC2 [7] designates a reference IO size of 256 KB — IO with a different size is counted in 256 KB capacity units (e.g., a 1024 KB IO request counts as 4 IOPS). Such a method is far from sufficient, as other characteristics (i.e., synchronous, asynchronous, random, sequential, etc.) are also important factors in determining IO cost. **(2) Non-work-conserving IO schedulers become less effective in virtualized cloud environments.** In particular, the IO latency under virtualization is much larger than that in native. As a result, the idle time added by the non-work-conserving scheduler (e.g., CFQ) would further hurt overall storage efficiency; such idle time will be wasted if the interval of two continuous synchronous IOs is larger than the idle time. In Section 4, we will show how vFair overcomes the unfairness problem while keeping high efficiency of the overall storage system.

3. Related Work

Proportional Share Fair queuing algorithms have previously been applied to storage IO constraints [1, 3, 16, 20, 21, 27, 29]. Most of these focus on a single resource metric such as IO throughput (bytes/s), IOPS (IO operations/s), or array queue length, and provide mechanisms to fairly allocate according to that metric. However, this relatively simple allocation mechanism cannot guarantee fair allocation in terms of storage service time. Current service time based proportional share schedulers treat IO response time as an approximation [19]. These approaches are too coarse-grained as response time will diverge from service time when the array has large queue depths. The rewarding scheduler [18] adopts a dynamic share allocation to reward clients that efficiently use storage resources. However, due to the delay problem discussed in Section 2.3, the rewarded shares do not benefit synchronous workloads. In contrast, vFair contributes a novel fine-grained service time based allocation to achieve fairer resource sharing.

Reservation and limitation Control groups (cgroups) [4] sets resource limits for a number of resources including IO bandwidth. Further, mClock [22] uses reservation and limit controls for shared storage to mitigate interference between workloads, and thus may protect the performance of sequential workloads. However, the static allocation method requires allocations of absolute VM service rates and are not flexible enough for a pure proportional fair share scheduler. In addition, mClock applies “worst case IOPS” as an upper bound, which is reasonable in reservation; but the non-work-conserving nature of mClock causes the storage capacity to not always be fully utilized. vFair shows up to 85% throughput improvement for low IO-concurrency workloads with better proportional share ratio and high utilization.

To isolate sequential IOs from random IOs, time-quanta-based IO allocations have been proposed [35, 36]. However, because of latency jitter, workloads must wait for others to finish their quantum before each time slice. Further, time-quanta based allocations are not work-conserving. Other workload placement methods [17, 30, 34] avoid IO interference by scheduling competing VMs on different servers. DRF [24] introduces a generalization of max-min fairness to multiple resource types. By applying DRF, Pisces [34] provides per-tenant fairness and isolation for key-value storage, including partition placement, weight allocation, replica selection, and weighted fair queuing. Libra [33], a multi-tenant IO scheduler, further provides application-request throughput reservations while preserving high-utilization for SSD-based key-value storage. To complement these high-level, object-based solutions, vFair focuses on block-level storage scheduling that operates at a lower level of the storage software stack.

IO path improvements The IO latency brought by the virtualized IO stack has been reduced via several software techniques, such as para-virtualization and skipping the guest-level IO scheduler, but this latency is still non-trivial compared to the native scenario. vTurbo [37] accelerates IO processing for VMs by offloading IO processing to a designated core. Though such methods reduce VM scheduling delay, they can hardly eliminate IO scheduling latency in the hypervisor. Vanguard [32] tried to eliminate performance interference by provisioning VMs with dedicated IO resources. However, this approach requires SSD devices for caching, and focuses only on high efficiency. vFair instead optimizes the IO scheduling for efficiency and fairness, though our solutions are complementary to any of these methods.

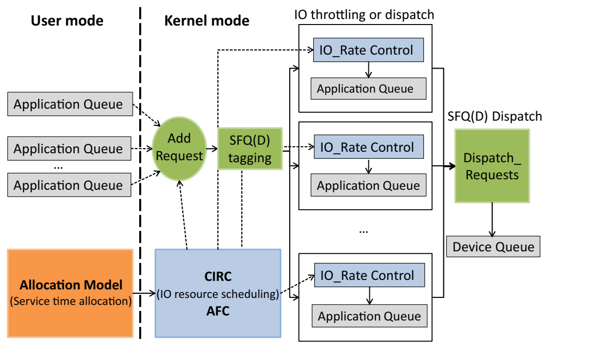


Figure 3. Architecture of vFair.

4. Design

To overcome the challenges highlighted in Section 2.3, vFair incorporates two key components, shown in Figure 3. First, to address the need for *fine-grained IO allocation*, we propose a novel service-time based allocation model (Section 4.1) and a practical way of realizing it using *prior knowledge* from an offline performance model and *posterior knowledge* from the running VMs. Based on this model, three scheduling strategies are proposed considering both fairness and efficiency (Section 4.2). Second, to avoid the limitations of non-work-conserving schedulers, vFair employs a two-level scheduling architecture (Section 4.3) for arbitrating IO resources across different VMs based on the proposed allocation model.

4.1 Per-IO Cost Allocation Model

As demonstrated in Section 2, to fairly allocate storage performance vFair must consider the amount of time each IO request will need to complete. However, it's not possible to accurately measure the service time of each IO operation without help from storage vendors, particularly for large storage arrays with hundreds of IOs concurrently. On the other hand, simply dividing up the number of IOs across VMs is too coarse-grained.

Given this challenge, let us first consider the *theoretical* IO throughput that a VM could receive in *isolation*. Intuitively, each VM's workload can be modeled as an IO pattern (i.e., combinations of read/write, sequential/random, synchronous/asynchronous). Given a particular IO pattern, we aim to calculate the saturation throughput P_i which would fully utilize the storage subsystem. In other words, P_i represents the storage system's upper bound on the throughput of a distinct IO pattern.

Therefore, a fair allocation of storage resources for two equally weighted VMs with saturation performance P_1 and P_2 would result in achieving $P_1/2$ and $P_2/2$, respectively. Intuitively, this allocation strategy will result in each VM obtaining half utilization of the shared storage; thus over an interval of time T, VMs 1 and 2 will obtain roughly T/2 service time, over which they would have achieved $P_1/2$ and $P_2/2$ IOPS, respectively. An example is shown in Figure 4.

Recall that proportional fair share is defined as providing total storage service (denoted by T) to hosts in proportion to

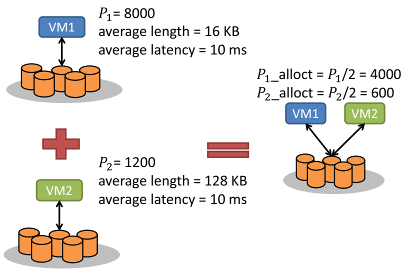


Figure 4. VM1 and VM2 should each obtain 50% of the shared storage throughput, thus the fair share allocation for VM1 and VM2 would be 4000 and 600, respectively.

their shares (denoted by w_i). Hence the service time allocation units T_i to VM_i can be expressed as:

$$T_i = \frac{w_i}{\sum_k w_k} \cdot T \tag{1}$$

Since P_i is directly related to T (P_i refers to full utilization which is T), we can proportionally allocate P_i using:

$$T_i = f\left(\frac{w_i}{\sum_k w_k} \cdot P_i\right) \tag{2}$$

Here, f is a simple mapping function from the saturation performance P to total service time T allocation. For example, if we allocate 100 IOPS to one VM and 200 IOPS to another (assume each IO has the same cost), we correspondingly allocate 1 relative unit of service time to the first and 2 relative units to the other. The function f captures this relationship. The total storage service T is hard to obtain directly in general, mainly confounded by the queue depth with competitive IO requests from various workloads. Our approach, instead, estimates T from distinct P_i for different VMs. Each P_i is obtained using that VM's specific IO pattern *in isolation*, eliminating the influence of other competitive IO requests in the storage queue.

4.1.1 Model Approximation

Our Per-IO Cost Allocation Model allows vFair to estimate the total storage service time T using P_i, however, obtaining P_i for each VM is still challenging in practice. First, since P_i depends on the mix of IO types, it differs across VMs. Further, for the same VM, it also changes continuously, depending on the characteristics of the workloads.

To overcome this, vFair uses an approximate method to calculate an initial P_i from *prior knowledge* (saturation performance profiling done in advance and only **once**). From this, vFair builds a model based on four basic IO types: sequential read, sequential write, random read, and random write. As shown in Figure 5, the saturation IOPS curves of such four basic types over various request sizes remain stable and can be easily obtained by conducting performance characteristics using micro-benchmarks [25][29] or a performance model [23]. Note that the performance specifications of disk arrays in a datacenter/cloud change continuously due to *transient connectivity* errors: network congestion, equipment malfunction, upgrades, or equipment fatigue

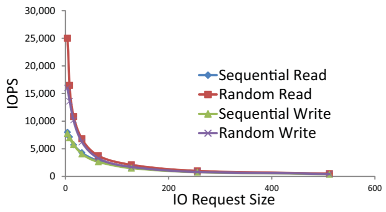


Figure 5. Saturated IOPS performance vs. IO request sizes for a single SSD.

from long-term use. But even in the face of such unstable conditions, a mirco-benchmark test will give a valid P_i. This is because these errors always exist and should therefore be considered in P_i. To account for these errors, P_i should be calculated a few times and averaged within a single profiling [2].

Therefore, given a certain VM, P_i can be calculated from the four basic IO types by fitting the following linear formula:

$$\begin{aligned} 1/P_i = {} & \alpha \cdot \beta / P_{seq\_rd} + \alpha \cdot (1-\beta)/P_{rand\_rd} \\ & + (1-\alpha)\cdot \gamma / P_{seq\_wr} \\ & + (1-\alpha)\cdot(1-\gamma)/P_{rand\_wr} \end{aligned} \tag{3}$$

where α is the read ratio of all IO operations, β is the sequential read ratio of the total read operations, and γ represents the sequential write ratio of all write operations. P_{seq_rd},P_{rand_rd},P_{seq_wr}, and P_{rand_wr} refer to the points on the curves of Figure 5 for a given IO request size. Each item on the right side of Equation 3 represents the percentage of a specific IO type. For example, $\alpha \cdot \beta \cdot P_i / P_{seq_{rd}}$ represents the percentage of the sequential read IO.

Using Equation 3, vFair can derive P_i for a certain VM by observing α, β, γ, and average IO sizes. The values for these parameters are acquired by a sampling of the VM's IO access pattern (*posterior knowledge*). Note that many applications have time-varying IO patterns. For example during query processing a database system may initially start with heavy sequential reads (e.g., during scans), switch to random writes (e.g., while building hash tables), and then sequential writes (e.g., writing the results). To account for such dynamic behavior, vFair performs periodic parameter updates using Equation 3 every 30 ms. Additionally, at runtime an Adaptive Feedback Control (Section 4.3) is used to refine this estimation closer to the actual throughput dynamically.

Notably, some IO workloads could be very complex in practice (e.g., with multithreading). Yet, they can be easily modeled as a specific IO pattern by examining per-VM IO behavior at the hypervisor level. Multiple synchronous threads will generate multiple outstanding IOs, and therefore behave more similarly to asynchronous IO workloads.

[2] In our model, we use the aio-stress micro-benchmark tool inside a native Linux box to measure raw storage performance and minimize software latency (e.g., bypassing file systems and IO schedulers). Though data profiling requires up to several hours (with ∼300 sampling points), this offline profiling (only done once) will not add any overhead to the runtime system.

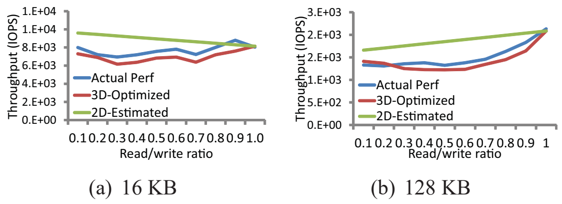


Figure 6. Estimation vs Actual Throughput with IO size of (a) 16 KB and (b) 128 KB.

For instance, given a 16-thread synchronous workload, there could be at most 16 outstanding IO requests; given an asynchronous workload with an IO depth of 16, there can be at most 16 outstanding IO requests as well. For both workloads, only when one of the 16 outstanding IO requests completes, can the next IO request be issued. Since Equation 3 captures the IO pattern of both synchronous and asynchronous workloads using saturation performance, it can handle multi-threaded workloads naturally. In our evaluation (Section 5), we choose various multithreaded IO workloads to test the effectiveness of vFair.

4.1.2 Model Accuracy and Improvement

The effectiveness of vFair's allocation mechanism closely depends on the accuracy of the approximation model. To show the estimation error between the actual throughput and estimated allocation, we run Sysbench [15] fileio test on a native Linux system to generate workloads with read percentage from 0 to 100% and IO sizes ranging from 16 to 128 KB. For each combination, we measure the actual throughput (in IOPS) and compare it to the estimated allocation from Equation 3.

Over all workloads, the sequential cases (both read and write) produce the minimum estimation error (within 10%), while the random cases produce estimation errors ranging from 2% to just over 20%. Figure 6 plots the estimation errors for sizes of 16 KB and 128 KB, and shows that the estimation forms a straight line and tends to be above the actual throughput curve.

To further reduce the estimation error, we introduce an enhanced model by considering varying read/write ratios in the offline profiling — instead of only the linear combination of peak performance reads and writes. We call the original estimation model 2D-estimation and the enhanced model 3D-estimation. Accordingly, for Equation 3, instead of using the percentage of reads/writes, we use the joint distribution to estimate peak performance in 3D-estimation. As shown in Figure 6, the average estimation error of 16 KB and 128 KB reduce from 17% and 26% to 9% and 8%, respectively. In general, the 3-D estimation curve in Figure 6 follows closely below the actual throughput curve. We can further increase the accuracy by varying the sequential ratio in addition to the read/write ratio. For now, we choose the 3D-estimation model to obtain a relatively accurate estimated P_i with reasonable offline data profiling.

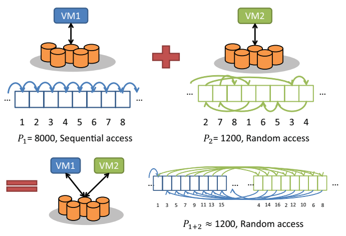

VM1
VM2
1 2 3 4 5 6 7 8
P_1= 8000, Sequential access
2 7 8 1 6 5 3 4
P_2= 1200, Random access
VM1
VM2
1 3 5 7 9 11 13 15
4 14 16 2 12 10 6 8
$P_{1+2} \approx 1200$, Random access

Figure 7. The sequential IO of VM1 becomes random when consecutive accesses are interleaved with those of VM2. As a result, the saturation performance becomes $P_{1+2} = 1200$ (the worst case).

4.2 IO Resource Scheduling Strategy

The Per-IO Cost Allocation Model allocates IO resources using P_i, and thus provides the *ideal* goal of both fairness and storage efficiency. However, in practice vFair must account for the IO-blender effect [10], where for example two VMs with sequential IO patterns interleave, resulting in a random IO pattern and greatly reducing the overall storage utilization. Due to this, using static P_i to estimate storage capacity would become imprecise over time. Hence we have developed three scheduling strategies that vary depending on the amount of correction they incorporate for the blender effect. One of these scheduling policies can be specified before vFair has started running.

Blender-oblivious proportional share (BOPS) The simplest scheme does not take blender effect into account and only allocates according to the VM's workload characteristics. This results in overestimating the P_i values since sequential reads/writes can become random and random IOs cost more than sequential, resulting poor throughput. Figure 7 shows an example in which, due to IO-blender, VM1 cannot achieve its fair share IOPS (i.e., the half of P_1, 4000, by Equation 2). A naive way to solve this problem is to use the worst case IOPS as an upper bound for allocation[22] (e.g., 1200 in Figure 7). However, such a conservative method is not efficient and cannot fully utilize the capacity of the underlying storage devices. For empirical comparison, we include BOPS as a candidate algorithm.

Blender-aware proportional share (BAPS) To solve the inaccuracy in BOPS, we propose adjusting the allocations *on the fly*. Given a VM_i with IO pattern t without blender and IO pattern t' with blender, the capacity change can be calculated by $P_i - P_i'$. By summing all the weighted changes, we obtain the overall capacity change $\triangle P$. The allocation of each VM should be adjusted (deducted) by the corresponding weighted $\triangle P$. By doing so, the penalty brought by the IO-blender distributes to every VM.

For the example in Figure 7, assume the IO-blender converts all of VM1's sequential IO to random IO. According to BAPS, we first calculate $\triangle P$ as an 85% reduction — 4000 IOPS promised to VM1 (VM1 and VM2 are equally weighted) reduces to 600 IOPS after the blending and no IOPS change for VM2. Subsequently, we reduce the VMs' allocations by $85\%/2$ and allocate 2300 IOPS to VM1 and 345 IOPS to VM2.

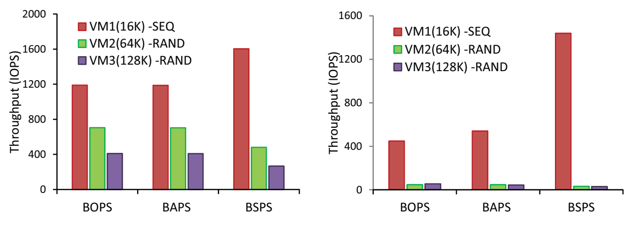
Of course, since the overall IOPS will reduce to 1200 in the **worst case**, BAPS may only achieve 855 IOPS for VM1 and 345 IOPS for VM2. In the general case though, BAPS ensures that VM1 receives between 855 to 4000 IOPS depending on the severity of the blender effect's interleaving. Regardless, by scheduling more IOPS for VM1, the storage system is better utilized and the performance penalty caused by IO-blender distributes to both VMs instead of only those with sequential IO patterns.

Blender-aware strict proportional share (BSPS) While BAPS adjusts for performance loss due to blender effect, to ensure a strict proportional fair share, we must also guarantee $P_1'/P_2' = (w_1 \cdot P_1)/(w_2 \cdot P_2)$($w_1$ and w_2 are the weights of VM1 and VM2). To this end, we propose BSPS to provide the best fairness guarantees at the cost of perhaps less efficient utilization of storage throughput.

BSPS adjusts the VMs' IO allocation until $P_1'/P_2' = (w_1 \cdot P_1)/(w_2 \cdot P_2)$ or the storage system becomes idle. Intuitively, the reasons a VM cannot achieve its allocated shares are due to idleness or IO competition. In consequence, the main challenge to implement BSPS is identifying which is the real cause. BSPS uses the **per-VM** average IO latency, from dispatching to the storage to completion of the IO request, as the indicator. Specifically, if the average IO latency is less or around $\frac{\sum_k w_k}{w_i \cdot P_i}$ (i.e., latency calculated from Equation 2), the storage is able to serve such a VM well (i.e., this VM is idle), and vice versa. We fractionally adjust the non-idle VMs, whose average IO latency is larger than $\frac{\sum_k w_k}{w_i \cdot P_i}$, from the VM with the relatively largest IO allocation (converting to the standard size), until $P_1/P_2 = P_1'/P_2'$.

For the two-VM example in Figure 7, BSPS will result in allocating roughly $8000/9200 * 1200$ IOPS for VM1 and $1200/9200 * 1200$ IOPS to VM2, thus being fair even in the worst case. However, by admitting far fewer sequential IOs for VM1 to avoid blender effect changes to the proportional fair share equation, we may drive the storage system sub-optimally, which is a reasonable tradeoff.

In Figure 8, we compare these scheduling strategies using a representative example: VM1 performing 16 KB sequential read, VM2 and VM3 performing 64 KB and 128 KB random IOs (60% read) on both SSD and HDD. We observe that, compared with BOPS, BAPS produces slightly higher throughput for VM1 and lower throughput for VM2 and VM3. BSPS brings more throughput benefit to VM1, since it reduces the shares of VM2 and VM3 resulting in less intensive competition of the shared storage. Specifically, in Figure 8(b), for the HDD case, the throughput of VM1 using BOPS, 448 IOPS, is lower than that of BAPS, 540 IOPS; whereas, with BSPS it reaches 1440 IOPS. On the other hand, in Figure 8(b), with BSPS VM2 and VM3 obtain 32 and 30 IOPS respectively; with BOPS the IOPS for VM2 and VM3 are 47 and 55 IOPS; while with BAPS they



(a) SSD (b) HDD

Figure 8. IOPS throughput of three scheduling schemes (a) for single SSD and (b) for single HDD.

are 47 and 44 IOPS. In addition, the storage system utilization (measured from the Linux kernel) is close to 100% using BAPS, and around 90% using BSPS. We observe a similar trend in Figure 8(a) using the SSD.

Table 1 summarizes the comparisons of different scheduling schemes. BSPS ensures the best fairness guarantees, but in most scenarios the IO scheduler desires high efficiency with best-effort fairness guarantees. Thus, we choose BAPS as the default algorithm for vFair for the rest of the paper.

Scheduling Strategy	Fairness Guarantee	Work-conserving?	High Utilization?
BOPS(vFair)	Good	Yes	Yes
BAPS(vFair)	Better	Yes	Yes
BSPS(vFair)	Best	Yes	No
PS(mClock)	Good	Yes	Yes
CFQ(Linux)	Bad	No	No
Time-Quanta based scheduler	Best	No	No

Table 1. Scheduling strategy comparison.

4.3 Two-level Scheduling Architecture

To realize the Per-IO Cost Allocation Model and scheduling strategies, vFair employs a traditional two-level scheduling architecture. This consists of a credit-based IO rate controller (CIRC), responsible for regulating incoming traffic to meet the rate requirement and ensuring isolation between VMs, and a fair queuing scheduler, for achieving fair scheduling among the isolated classes provided by CIRC. Though we could adjust the tag in SFQ(D) (i.e., work-conserving) to perform credit allocation and accumulation, it is still necessary to use a throttling-based scheduler to throttle aggressive VMs and thus leave the desired time slices for other VMs (e.g., with synchronous IOs).

We propose a simple credit-based IO rate controller (CIRC) inspired by Xen's CPU Credit Scheduler. An IO credit amount is assigned to an *active* VM's IO queue periodically depending on the allocation model and IO scheduling strategies. VMs burn one credit every time they receive one IO request from applications and become throttled when there is no remaining credit.

To handle **bursts**, CIRC adopts a reset mechanism: if an active VM does not use its fair share, it will slowly accumulate credits. Once it reaches a threshold, it is marked as *inactive* and the credits are dropped (set to zero). This VM will be re-activated by a new IO request and marked as *burst*, which has a chance to schedule requests in bursts (up to the bursty threshold). By adjusting the bursty threshold, CIRC easily regulates the bursty degree and minimizes the bursty impact to other workloads. To keep **work-conserving**, CIRC collects unused credits from "under VMs" (unable to consume all credits) and distributes them proportionally to "over VMs" (consumed all credits) to be used during the next running interval. To reduce fluctuation, a moving average method is adopted to smooth total unused credits within several previous intervals.

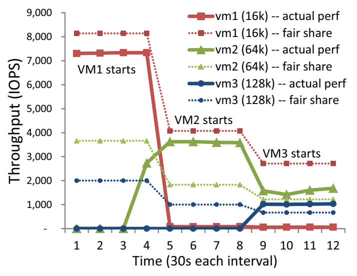
In order to guarantee fair interleaving of IO requests, we use a fair queuing algorithm, SFQ(D) (same algorithm used in Parda [21]), to achieve second-level fairness. Some two-level architectures [26, 39] use real-time scheduler latency goals, but they are unfair (i.e., they do not isolate request flows from unexpected demand surges) and require admission control. SFQ(D) assigns tags to each request when it arrives and dispatches requests in increasing order of the tags. The fairness property results from the way that tags are computed and assigned to requests of different workloads. The tag values represent the time when each request should start and complete according to a virtual time that advances monotonically.

To adjust for inaccuracies in our offline performance model, we employ an adaptive feedback control (AFC) mechanism, as used in prior work [28]. AFC detects the state of the storage system based on unused credits C_i from CIRC and the average system-wide IO latency L_i measured over a fixed time period, and adjusts the estimated P_i for each VM_i correspondingly. A simple exponential moving average method is used to smooth and reduce variability of L_i and C_i. For lack of space, we do not describe this in further detail; we use the same algorithm as described in [28].

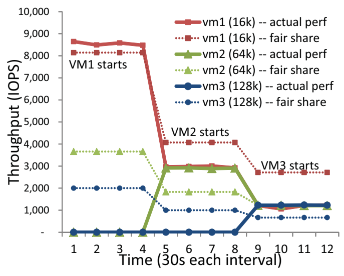
In many environments, storage can be shared across different hosts — requiring a scheduler that can run in a *distributed* fashion. vFair can be extended for these settings similar to [21]. This involves running vFair on each host and exchanging fair share information (VM weights and overall IO latency), with each other. For our implementation in Section 5, we used the NFS file system to exchange information every 30 ms, but vFair can be extended to use any other communication mechanism (e.g., TCP).

5. Evaluation

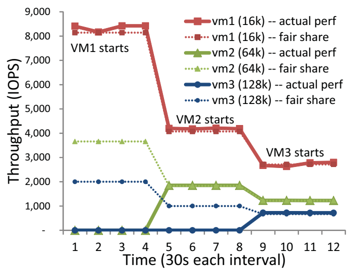
We have implemented a prototype of vFair (∼1500 lines code) as a Linux 3.2.10 loadable module and integrate it with the Xen 4.2 hypervisor. However, vFair is easy to port to other hypervisors (e.g., KVM). In this section we present a detailed evaluation of vFair, which consists of several hosts accessing a shared storage system. Each host is equipped with a quad-core 3.2GHz Intel Xeon CPU, 16GB of RAM, and connected via two 1Gigabit Ethernet links. These hosts run a Xen 4.2 hypervisor and Linux 3.2 driver domain and



(a) CFQ(Linux)

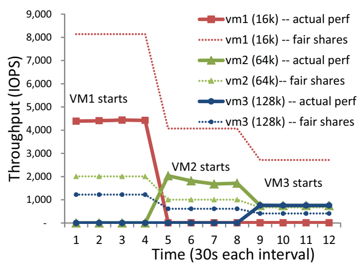


(b) PS(mClock)

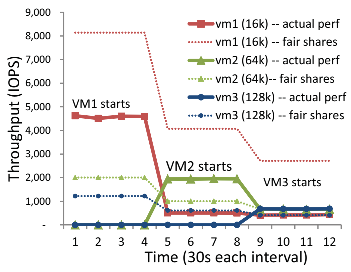


(c) BAPS(vFair)

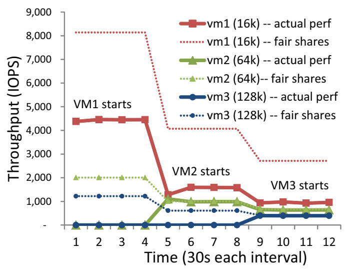
Figure 9. The less-backlogged case (SSD): throughput (IOPS) distribution among three VMs running workloads with random IOs with various IO sizes using CFQ(Linux), PS(mClock) and BAPS(vFair).



(a) CFQ(Linux)



(b) PS(mClock)



(c) BAPS(vFair)

Figure 10. The non-backlogged case (SSD): throughput (IOPS) distribution among three VMs running workloads with sequential or random IOs with various IO sizes using CFQ(Linux), PS(mClock) and BAPS(vFair).

	BAPS(vFair)	CFQ(Linux)	PS(mClock)	Fair shares
vm1 (16k) – 60% random read	48	28	37	51
vm2 (64k) – 60% random read	42	58	46	45
vm3 (128k)– 60% random read	40	48	45	40

Table 2. The less-backlogged case in the three-VM share phases using HDD.

	BAPS(vFair)	CFQ(Linux)	PS(mClock)	Fair shares
vm1 (16k) – sequential read	540	190	256	1940
vm2 (64k) – 60% random read	44	51	48	45
vm3 (128k)– 60% random read	41	46	43	40

Table 3. The non-backlogged case in the three-VM share phases using HDD.

guest VMs. Each VM is assigned sufficient resources to ensure no performance bottlenecks are due to CPU or memory. To include scheduling impact, all vCPUs share all available pCPUs. We have performed our experiments using one 300G solid-state drive (SSD) and one 1TB hard disk drive (HDD) separately as the shared storage device.

Three proportional fair share IO schedulers, CFQ (Linux), PS (mClock) and BAPS(vFair) are compared. CFQ (Linux) comes from Linux kernel 3.2.10. We implemented mClock's proportional share scheduler (as we could not obtain the source code) according to Algorithm 1 in [22] (not including mClock's limits and reservations). By default, mClock handles different sized IOs by converting larger IOs into multiple reference IOs (we use a unit of 16 KB in all experiments). As discussed in Section 2, this technique is too coarse-grained as the cost of an IO is determined by many factors, such as sequential/rand, read/write, size, etc. We choose vFair BAPS over BSPS since it strikes a better trade-off between storage utilization and fairness. The max queue depth of vFair is set to be 32.

5.1 Micro-Benchmark Results

In this section, we evaluate the effectiveness of vFair by examining fairness — whether a VM can achieve its proportional fair shares — and the ability to handle bursty and idle VMs using the micro-benchmark tool Fio.

5.1.1 Fairness

First, we evaluate the fairness of vFair by demonstrating that the measured throughput (in terms of IOPS with a system-wide latency threshold, e.g., 5 ms) meets the proportional fair shares (as defined in Equation 2) [3].

We launch three VMs with each running the micro-benchmark Fio without idle time. We use VM1 to emulate low IO-concurrency workloads with small IO size (16 KB) and VM2 and VM3 to emulate high IO-concurrency workloads with large IO size (64 KB and 128 KB). We cat-

[3] The ground truth of storage service time should be obtained from the storage devices. However, such ground truth is hard to measure accurately. Instead, we use Equation 2 and the 3D-estimation model to approximate the share target (error is within 10% as shown in Figure 6).

egorize VM1's IO access patterns into two groups: (1) the less-backlogged case using asynchronous random IO with 4 concurrent Fio threads and an IO depth of 4; and (2) the non-backlogged case using synchronous sequential IO with only 1 Fio thread and the IO depth being 1 (in fact, this is an extreme case). In contrast, VM2 and VM3 generate asynchronous random IOs (with 8 concurrent Fio threads and IO depth being 32). According to Section 2, the performance penalty of the low IO-concurrency workloads in case (2) should be more severe than case (1).

In each experiment, VM1 is the first to run on the physical host. Later, we start VM2 to share IO bandwidth with VM1 and finally VM3 is joined. We set equal weights to all VMs.

Less-backlogged case In Figure 9, dotted lines show the ideal fair share target calculated from Equation 2, while solid lines depict the actual throughput (measured every 30 seconds). We observe that when three VMs share the same storage, the throughput of VM1 (the low IO-concurrency VM with 16 KB size) is only 60 IOPS using CFQ(Linux) and 1224 IOPS using PS(mClock) — far from the target share 2714 IOPS. In contrast, using BAPS(vFair), the average throughput of VM1 reaches 2721, which is quite close to the target share. On the other hand, using CFQ(Linux) and PS(mClock), VM2 and VM3 (the high IO-concurrency VMs with 64 KB and 128 KB) achieve higher throughput than their target shares. For example, with CFQ(Linux), VM2 and VM3 achieve 1568 and 1019 IOPS, higher than the target 1220 and 667 IOPS respectively. While for BAPS(vFair) the average throughputs of VM2 and VM3 are still close to the target fair shares. The results in Figure 9 indicate that, both CFQ(Linux) and PS(mClock) are not fair and favor VMs with larger IO sizes. On the contrary, with BAPS(vFair) the actual throughput matches the specified fair share — demonstrating the desired fairness requirements. Though the throughput of the HDD is lower than the SSD, we observed similar results on the HDD shown in Table 2 — when three VMs share the same HDD device the actual performance of each VM is close to the fair share target.

More evaluation results obtained by varying read/write ratios are presented in Figure 11 using the SSD. The read ratio varies from 0% to 100% for VM1 and kept at 60% for the other two VMs. The actual (measured) performance curve follows the fair share target closely for all cases with error ranging from 1% to 9%. Figure 11 shows that the largest error (only 9%) happens when the read ratio is near 40%. These results show vFair provides very good fairness for less-backlogged cases as the actual throughput reaches the specified fair share target with less than 10% variation. Besides, the utilization of the storage system is 100% using BAPS(vFair) with the system-wide IO latency around 5 ms.

Non-backlogged case In the non-backlogged case, VM1 suffers greater unfairness when competing with high IO-concurrency VMs as demonstrated in Section 2.

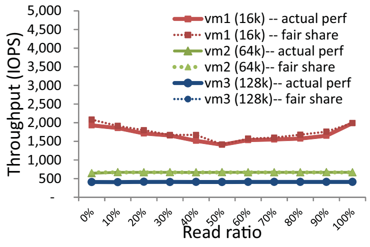


Figure 11. Throughput (IOPS) distribution among three VMs when varying read ratio.

In Figure 10, different from the random scenario, the throughput of the sequential workloads (VM1) is far from the specified fair share allocation even when VM1 is the only running VM. Figure 10 shows the maximum throughput (4413 IOPS), is only half of that (8141 IOPS) in the native system when only VM1 is running. Upon deeper investigation, we found that this is caused by virtualization latency (as discussed in Section 2.3). Moreover, the capacity of the shared storage device changes in the shared phases due to IO-blender effect — partial sequential IOs of VM1 become random accesses. Thus the dotted line of VM1 represents the target share in the best case (i.e., no IO-blender).

Under such circumstance, Figure 10 shows the throughput of VM1 using CFQ(Linux) and PS(mClock) is quite far from the line of the best fair share target (2714 IOPS) in the three-VM share phases, only 10 IOPS and 414 IOPS respectively. Whereas using BAPS(vFair) it becomes much higher, 975 IOPS. Though BAPS(vFair) cannot guarantee VM1 will achieve the target throughput in such an extreme case, it improves the performance of sequential workloads greatly, compared to the other two schedulers — the throughput of VM1 using BAPS(vFair) is 100X that of CFQ(Linux) and 2.35X of PS(mClock). The underlying reason for this is vFair's IO-blender compensation. Thus, by adjusting credits among VMs BAPS(vFair) benefits the low IO-concurrency VMs and achieves better fairness, yet with nearly full storage utilization. Specifically, Figure 10(c) shows that with BAPS(vFair) VM1's throughput remains close to the fair share target and VM2 and VM3 exactly reach the target shares. The same observation applies in the HDD setup shown in Table 3. In the three-VM share phases, the throughput of VM1 using BAPS(vFair) is 2.8X of CFQ(Linux) and 2.1X of PS(mClock).

Further, we observed that as the IO *concurrency* increases within VM1, the actual performance of VM1 quickly reaches the fair share target. Table 4 shows the Fio tool's IO depth (i.e., the number of concurrent outstanding IO requests) versus the measured performance (IOPS) of VM1. As Table 4 shows, when Fio's outstanding IO depth = 4, vFair is able to ensure the exact fair share.

Fio IO depth	1	2	4
Actual Perf. of VM1 (IOPS)	975	1920	2690

Table 4. Actual performance vs. Fio IO depth.

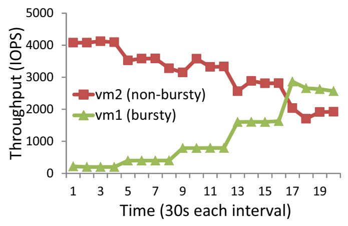


Figure 12. Bursty workloads.

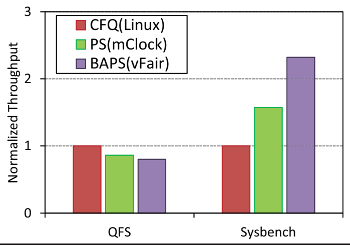


Figure 14. Application-level throughput comparisons of three schedulers between QFS and Sysbench.

5.1.2 Handling Bursts and Work-Conserving

To demonstrate the effectiveness of vFair with handling bursty workloads and keeping work-conserving, we varied workloads with idle time. Recall that bursty VMs are allowed to receive IOs in a burst (up to the bursty threshold) when such VMs have been idle in the past. This ensures that if an application is idle for a while, it is preferred when it becomes activated again.

In this experiment, we run two Fio workloads in two VMs separately. The first VM is bursty, generating 16 KB 60% random read in the rate of 200, 400, 800, 1600 and 3200 IOPS. The second VM is steady, producing 64 KB 60% random read without idle time. Both VMs have the equal shares. The vFair burst threshold was set to twice of the allocated fair share.

Figure 12 shows the average storage utilization is almost 100% with the system-wide latency being around 5ms (using SSD) — indicating the idle bandwidth from VM1 is taken over by VM2. Further, VM1 is able to achieve its bursty loads varying from 200 to 2600 IOPS. As intended, as the bursty rate of VM1 increases from 200 to 3200, the throughput of VM2 decreases correspondingly. At the bursty rate 3200 IOPS, VM1 reaches to the fair share limit, 2600 IOPS, and VM2 drops to its allocated fair share, 1800 IOPS, as well. These results indicate that the vFair credit-based mechanism is able to handle bursty applications and ensure fairness and high utilization.

5.2 Application Workloads

We also test vFair with realistic applications: Sysbench [15], Postmark [12], and DVDstore [6] as low IO-concurrency workloads and secure copy (SCP) [14] and FTP [9] as high IO-concurrency workloads. In Section 5.3, we use QFS [31] (a high-performance distributed file-system) as the high IO-concurrency workload to highlight how vFair can isolate independent throughputs and provide proportional fair shares[4]. The low IO-concurrency applications running inside VM1 generate small random IO requests with size ranging from 5 KB to 23 KB (obtained from measuring the results). VM2 runs an FTP server for clients to upload large files, and VM3 runs secure copy (SCP) to copy large files to clients. The average IO size of these two workloads is about 122 KB. We set the proportional share weight to be 1(VM1):1(VM2):1(VM3).

Further, these experiment use the *distributed version* of vFair as described in Section 4.3. Three VMs run in three individual physical hosts and use the ISCSI protocol to connect to the storage server. vFair is running in each physical host. For all tests, all three VMs are running and servicing requests at peak performance that fully utilize the underlying storage. The 300G SSD is used as the shared storage device due to high performance.

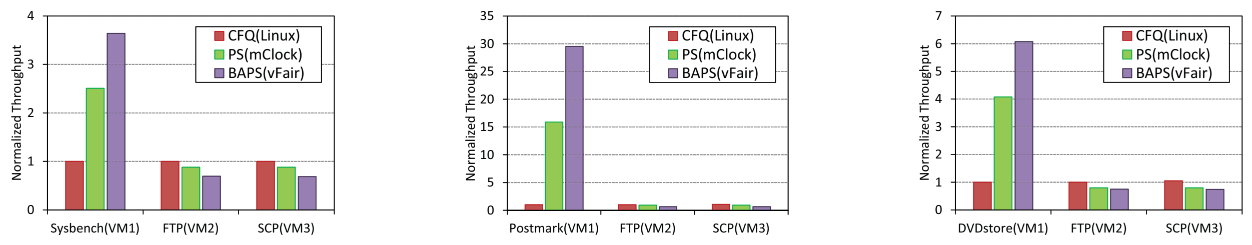
Sysbench First we evaluate Sysbench, the OLTP application benchmark running on top of a MySQL database with the InnoDB storage engine. We create a 20GB MySQL database and run 16 client threads remotely in a client with a 1GB network connection to VM1. Recall that all three VMs are operating simultaneously. In Table 5 Row 1, we observe that the proportional share (PS) ratio using BAPS(vFair) is close to 1:1:1. The PS ratio[5] of Sysbench using CFQ(Linux) is the lowest among all three schedulers, only 0.19:1:1, while the PS ratio using PS(mClock) is 0.69:1:1.

Note that, the underlying storage is observed to be fully utilized for all three schedulers, but because of the delay problem discussed in Section 2.3, both CFQ(Linux) and PS(mClock) favor FTP and SCP, resulting in more performance for them and less performance for Sysbench. In contrast, BAPS(vFair) balances storage capacity between the high IO-concurrency workloads (SCP and FTP) and the low IO-concurrency workload (Sysbench). Figure 13(a) shows that with BAPS(vFair) Sysbench's performance increases by 260% and 45% in comparison with CFQ(Linux) and PS(mClock) respectively, and the file transferring bandwidth (the sum of FTP and SCP) is naturally throttled by 31% and 22%. Consequently, by reducing the performance of FTP and SCP and in turn improving the performance of Sysbench, BAPS(vFair) achieves very good fairness (i.e., PS ratio in Table 5 Row 1 is close to 1:1:1).

Postmark Postmark emulates real world usage of a file system using small file operations accessed by busy mail servers and news servers. We generate 2 million small files with sizes ranging from 1KB to 9KB. The test-bed configurations are the same as the Sysbench testcase. Table 5 Row 2 shows that the PS ratio using BAPS(vFair) is able

[4] Similar to the CloudSuite [5] benchmark, these applications simulate the basic functional services of web serving, web searching, and data serving.

[5] To calculate PS ratio in terms of service time, we need to know the storage utilization for each VM — P_{actual}/P_i, where P_{actual} is the actually measured performance and P_i is the saturation performance. P_i can be obtained using sampled IO pattern and 3D-estimation model.



(a) Sysbench (b) Postmark (c) DVDstore

Figure 13. Application-level throughput (trans/second) comparisons of three schedulers using (a) Sysbench, (b) Postmark and (c) DVDstore.

PS ratio	CFQ(Linux)	PS(mClock)	BAPS(vFair)
Sysbench(VM1):FTP(VM2):SCP(VM3)	0.19:1:1	0.69:1:1	0.99:1:1
Postmark(VM1):FTP(VM2):SCP(VM3)	0.03:1:1	0.49:1:1	0.92:1:1
DVDstore(VM1):FTP(VM2):SCP(VM3)	0.14:1:1	0.56:1:1	0.89:1:1
Sysbench(VM1):QFS(VM2-VM10)	0.64:1	0.81:1	0.99:1

Table 5. Proportional share ratio operating Sysbench, Postmark, DVDstore, and QFS benchmarks simultaneously.

to reach to 0.92:1:1 indicating very good fairness. Like before, the PS ratio using CFQ(Linux) is extremely low, only 0.03:1:1, while the PS ratio using PS(mClock) is 0.49:1:1. Correspondingly, as shown in Figure 13(b) the throughput of Postmark using CFQ(Linux) is extremely low, only 3% of that using BAPS(vFair). The reason for this is that the average IO size of PostMark is quite small ($\sim$ 5KB). As discussed in Section 2, workloads with smaller IO suffer more performance degradation using CFQ(Linux). Compared to PS(mClock), BAPS(vFair) increases the throughput of Postmark by 85%. Correspondingly, with BAPS(vFair) the (greedy) file transferring bandwidth reduces by 35% and 30% from that of CFQ(Linux) and PS(mClock), for the same reason as discussed above.

DVDstore(DS2) Dell DVDstore implements a complete online e-commerce application with a backend database component (MySQL), a web application (Apache) layer and driver programs. We use two clients over a 1GB network connection to stress VM1 (where the web and the database server are running). We generate several DS2 files with 10GB sizes, and again the test-bed configurations are the same as before. Table 5 Row 3 again indicates that BAPS(vFair) provides a very fair PS ratio (0.89:1:1). We also see that the PS ratio of DVDstore using CFQ(Linux) is again extremely low, only 0.14:1:1, while the PS ratio using PS(mClock) is 0.56:1:1. Moreover, in Figure 13(c) the throughput of DVDstore using CFQ(Linux) is quite low, only 16% of that using BAPS(vFair). BAPS(vFair) increases throughput of DVDstore by 50% over that of PS(mClock). The file transferring bandwidth with BAPS(vFair) reduces by 29% and 10% compared to using CFQ(Linux) and PS(mClock), again showing BAPS(vFair) naturally enforces fairness for the (less aggressive) DVDstore application.

5.3 Distributed File System

Quantcast File System (QFS) is a high-performance, fault-tolerant, distributed file system developed to support MapReduce processing or other operations which read and write large files sequentially. It involves a central metadata server that manages the file system's directory structure, mappings, and many chunk servers (the distributed component managing IO to the disks). In our experiment we launched 1 metadata server and 9 chunk servers (to satisfy 6+3 Reed-Solomon Encoding). We use one VM running Sysbench to compete for the shared storage. The sharing weight is set to 1:1 between Sysbench and QFS (all 9 chunk servers are treated as a group). For QFS, we issue read operations and the average IO size is observed to be around 300 KB.

Table 5 Row 4 shows PS ratio for QFS and Sysbench. The PS ratio of BAPS(vFair) is able to reach to 0.99:1, while the ratio of CFQ(Linux) is 0.64:1 and the ratio of PS(mClock) is 0.81:1. In Figure 14, the throughput of Sysbench using BAPS(vFair) improves by 1.32X compared to CFQ(Linux) and 47% compared to PS(mClock). Correspondingly, the throughput of QFS with BAPS(vFair) reduces by 33% and 22% than using CFQ(Linux) and PS(mClock). This testcase highlights that BAPS(vFair) is able to isolate throughput of different tenants running various applications and achieves the specified proportional fair shares.

6. Conclusion

We have presented vFair, a block-level storage scheduling framework with a novel per-IO cost allocation model. For a shared storage system vFair provides fine-grained IO allocation using a practical performance modeling method. vFair achieves fairness among sharing VMs while maintaining high resource utilization, regardless of the VMs' workloads and IO patterns. Our evaluation with micro-benchmarks and realistic applications indicates the effectiveness of vFair, compared with state-of-the-art techniques.

7. Acknowledgments

We thank our shepherd, Ippokratis Pandis, and the anonymous reviewers for their insightful comments. This work was supported in part by NSF under Award 1219004.

References

[1] Self-clocked fair queuing scheduler. `https://www.ee.iitb.ac.in/~prakshep/IBMA_lit/manual/manual239.html`.

[2] AIO-stress. `http://openbenchmarking.org/test/pts/aio-stress`.

[3] CFQ. `https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt`.

[4] Linux control groups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.

[5] CloudSuite. `http://parsa.epfl.ch/cloudsuite/overview.html`.

[6] DVDstore test application. `http://www.dell.com/downloads/global/power/ps3q05-20050217-Jaffe-OE.pdf`.

[7] I/O characteristics. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-io-characteristics.html`.

[8] Fio - flexible I/O tester synthetic benchmark. `http://www.storagereview.com/fio_flexible_i_o_tester_synthetic_benchmark`.

[9] File transfer protocol. `http://en.wikipedia.org/wiki/File_Transfer_Protocol`.

[10] IO-Blender problem. `http://searchvirtualstorage.techtarget.com/definition/I-O-Blender`.

[11] KVM I/O overhead. `http://events.linuxfoundation.org/sites/events/files/slides/CloudOpen2013_Khoa_Huynh_v3.pdf`.

[12] Postmark. `http://www.dartmouth.edu/~davidg/postmark_instructions.html`.

[13] The economics of virtualization: Moving toward an application-based cost model. `http://www.vmware.com/files/pdf/Virtualization-application-based-cost-model-WP-EN.pdf`.

[14] Secure copy. `http://en.wikipedia.org/wiki/Secure_copy`.

[15] Sysbench OLTP benchmark. `http://www.storagereview.com/sysbench_oltp_benchmark`.

[16] J. C. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. In *ACM SIGCOMM Computer Communication Review*. ACM, 1996.

[17] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 2013.

[18] A. Elnably, K. Du, and P. J. Varman. Reward scheduling for qoS in cloud applications. In *CCGRID*. IEEE, 2012.

[19] A. Elnably, H. Wang, A. Gulati, and P. Varman. Efficient qos for multi-tiered storage systems. USENIX Association, 2012.

[20] A. Gulati, A. Merchant, and P. J. Varman. pclock: an arrival curve based approach for qoS guarantees in shared storage systems. In *SIGMETRICS*. ACM, 2007.

[21] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, 2009.

[22] A. Gulati, A. Merchant, and P. J. Varman. mclock: Handling throughput variability for hypervisor IO scheduling. USENIX Association, 2010.

[23] A. Gulati, G. Shanmuganathan, I. Ahmad, C. A. Waldspurger, and M. Uysal. Pesto: online storage performance management in virtualized datacenters. In *SoCC'11*. ACM, 2011.

[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[25] H. H. Huang, S. Li, A. Szalay, and A. Terzis. Performance modeling and analysis of flash-based storage devices. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*. IEEE, 2011.

[26] L. Huang, G. Peng, and T.-c. Chiueh. Multi-dimensional storage virtualization. In *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2004.

[27] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. *ACM SIGMETRICS Performance Evaluation Review*, 2004.

[28] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage (TOS)*, 2005.

[29] T. Kelly, I. Cohen, M. Goldszmidt, and K. Keeton. Inducing models of black-box storage arrays. *HP Laboratories, Palo Alto, CA, Technical Report HPL-2004-108*, 2004.

[30] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*. ACM, 2010.

[31] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 2013.

[32] Y. Sfakianakis, S. Mavridis, A. Papagiannis, S. Papageorgiou, M. Fountoulakis, M. Marazakis, and A. Bilas. Vanguard: Increasing server efficiency via workload isolation in the storage i/o path. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, 2014.

[33] D. Shue and M. J. Freedman. From application requests to virtual iops: Provisioned key-value storage with libra. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014.

[34] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, 2012.

[35] P. Valente and F. Checconi. High throughput disk scheduling with fair bandwidth distribution. *IEEE Transactions on Computers*, 2010.

[36] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. USENIX, 2007.

[37] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. USENIX Association, 2013.

[38] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, 2013.

[39] J. Zhang, A. Sivasubramaniam, A. Riska, Q. Wang, and E. Riedel. An interposed 2-level I/O scheduling framework for performance virtualization. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*. ACM press, 2005.