Virtual I/O Scheduler: A Scheduler of Schedulers for Performance Virtualization

Seetharami R. Seelam *
IBM T. J. Watson Research Center,
Yorktown Heights, NY 10598
sseelam@us.ibm.com

Patricia J. Teller
Department of Computer Science,
The University of Texas at El Paso, El Paso, TX 79968
pteller@utep.edu

Abstract

Virtualized storage systems are required to service concurrently executing workloads, with potentially diverse data delivery requirements, that are running under multiple operating systems. Although a number of algorithms have been developed for I/O performance virtualization among operating system (OS) instances and their applications, none results in absolute performance virtualization. By absolute performance virtualization we mean that the performance experienced by applications of one operating system does not suffer due to variations in the I/O request stream characteristics of applications of other operating systems. Key requirements of I/O performance virtualization are fairness and performance isolation. In this paper, we present a novel virtual I/O scheduler (VIOS) that provides absolute performance virtualization by being fair in sharing I/O system resources among operating systems and their applications, and provides performance isolation in the face of variations in the characteristics of I/O streams. The VIOS controls the coarse-grain allocation of disk time to the different operating system instances and is OS independent; optionally, a set of OS-dependent schedulers may determine the fine-grain interleaving of requests from the corresponding operating systems to the storage system.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling; D.4.8 [*Operating Systems*]: Performance—Measurements; C.4 [*Performance of Systems*]: Performance attributes

General Terms Design, Algorithms, Experimentation, Measurement, Performance.

Keywords I/O Scheduler, virtual I/O, fairness, performance isolation, predictable performance.

1. Introduction

Server consolidation and virtualization of compute and I/O resources [1, 2, 13, 20] are attractive trends for economic reasons. For these trends to be successful, virtualized storage systems are required to service concurrently executing workloads, with potentially diverse data delivery requirements and different I/O request characteristics, that are running under multiple operating systems. For example, today a typical virtualized system services interactive, real-time, and throughput-intensive applications. Each of these applications, potentially running under a different operating system (OS) instance, has different data delivery requirements and different I/O stream characteristics. For example, interactive applications, such as LaTeX editing, require shorter average response times and their requests are sequential; real-time applications, such as podcast audio, require a bounded latency for their requests to be serviced and their requests are sequential; commercial database applications are not time critical but their requests are random; and throughput-intensive applications, such as data transfers and HPC applications, require high bandwidth across multiple requests but their I/O could be random or sequential. Since all of these applications access a shared storage system, an I/O scheduler is needed to arbitrate the service among them. The I/O scheduler must provide service to these applications such that no one application impacts the storage service provided to another application; in other words, the storage system performance must be virtualized among the applications. In the remainder of this paper, we assume that each application is running under a different OS instance.

Although a number of I/O scheduling algorithms have been developed for I/O performance virtualization among applications, none results in absolute performance virtualization. By absolute performance virtualization we mean that the performance experienced by applications of one operating system do not suffer due to variations in the I/O request stream characteristics, such as request size and seek characteristics (random or sequential), of applications of other operating systems. The two key requirements for such absolute I/O performance virtualization are fairness and performance isolation. As we shall see later in the paper, strict performance isolation may result in inefficient use of storage resources especially in storage systems with controllers that support out-of-order servicing of I/O requests, which amortizes the costs associated with disk-head positioning. In such systems, the virtualization algorithm also must be able to share the performance among applications at a finer granularity while ensuring performance isolation at a coarser granularity.

Accordingly, the question we will answer in this paper is: When multiple applications access a storage system, how can we control access to the storage system such that all applications get their fair share of the resource in the face of variations in I/O request stream and storage device characteristics?

In this paper, to address the above question, we present an OS-independent virtual I/O scheduler (VIOS). Under the VIOS, each application is assigned a quantum of disk time and an application is allowed to access the storage system for no more or no less

* The work reported here was done while the author was affiliated with the University of Texas at El Paso.

VEE'07, June 13–15, 2007, San Diego, California, USA.

than the quantum duration, irrespective of the idiosyncrasies of requests and storage devices. Thus, the VIOS controls the coarse-grain allocation of disk time among competing OS instances. In our framework, optionally, an I/O scheduler can be associated with each OS instance; this I/O scheduler would determine the fine-grain interleaving of requests from applications.

We show that, due to the disk-time fairness of the VIOS, (a) it maintains performance isolation in such a way that the I/O characteristics of any application can not impact the I/O system performance of another application; (b) it provides performance virtualization as a result of providing performance isolation and applications are given the impression that a fraction of the I/O system is dedicated to them; (c) it is work-conserving, thus, the I/O system is never idle if there is pending work; and (d) it can be extended to enforce strict Quality of Service (QoS) objectives.

We also present an extension to VIOS that provides virtualization among applications that access more complex storage systems, such as out-of-order devices, where the performance is both virtualized and shared at different granularities. These characteristics make our framework suitable for contemporary and next-generation operating systems and storage systems.

The remainder of the paper is organized as follows: In Section 2, we describe the related work, while in Section 3, we describe the framework and the algorithm used by the VIOS. In Section 4 and 5, we present analytical and experimental evaluation of our framework, respectively. The framework is extended to handle complex I/O systems in Section 6 and the experimental evaluation of the extensions are presented in Section 7. Finally, we present our concluding remarks in Section 8.

2. Related Work

Schedulers developed for fair sharing and virtualization of processor and network resources cannot be adopted for disk resources because of the fundamental differences between them. For instance, the relative order of request accesses to disk resources impacts disk utilization and fairness. However, such relative order has no impact on either processor or network resource utilization or fairness. While fair queuing and round-robin scheduling have been used extensively in sharing network resources [6, 8, 9, 16, 19, 21], only a handful of papers explore it with respect to disk I/O schedulers [3, 5, 20, 29] and none of them achieve absolute performance virtualization.

Fair queue disk schedulers presented in [5, 20] achieve fairness to an extent but they require detailed performance models of the disk system and/or models to estimate the seek and rotation delays for each I/O request. Constructing such models is difficult for a single drive, let alone multi-drive disk arrays that are commonplace, and it is not clear how they handle devices with controllers that support out-of-order service. In addition, although these schedulers are purported to be fair, they fail to provide performance isolation. There are various studies on Quality of Service (QoS) management of disk I/O [12, 13, 24, 25] using fair queuing, which also claim to achieve performance virtualization: they all require similar performance models. Furthermore, as we will show, even though request size variability and seek characteristics have profound impacts on performance isolation, these studies ignore either one or both of these aspects.

There are four studies that are somewhat closely related to our work: two of them associated with disk scheduling [3, 29] and the other two are associated with network scheduling [19, 21]. The Completely Fair Queuing (CFQ) Scheduler is loosely based on the ideas of Stochastic Fair Queueing [3, 16]. It enqueues requests from different applications in different queues. It dequeues and dispatches a set s of requests from a busy queue and cycles through the list of busy queues using the round-robin algorithm. As we have shown in our earlier work [18], because this round-robin scheduling ignores the request size and head-positioning delays associated with requests, and only considers the number of requests, it results in unfair disk resource allocation and fails to provide performance isolation. In this round-robin scheduling, applications with relatively large size requests and those with random seeks will impact the performance of other applications with different I/O seek and request size characteristics.

To eliminate the impact of request size characteristics on performance, Time Sliced CFQ (TCFQ) [29] (independently) in disk scheduling and Tan and Guttag [21] in wireless LANs use resource time as the sharing notion in round-robin scheduling, as we do in our scheduler. However, there are two substantial differences between our work and the work reported in [29]: (1) Although, in [29], application classes are assigned a quantum of disk time based on their priorities, any excess disk time consumed by an application class is never accounted for, thus, TCFQ does not provide a strong degree of fairness, as does the VIOS, and applications can consume more disk time than their quanta; and (2) our algorithm nicely lends itself to the exploitation of concurrency at the disk controller with out-of-order request service capability, without losing its fairness properties. In contrast, it is difficult to incorporate parallelism into TCFQ. Adapting the algorithm discussed in [21] essentially leads to TCFQ, hence, it has the same drawbacks.

Fair Queuing with Deficit Round-Robin (DRR), which, in spirit, is similar to our scheduler, was proposed by Shreedhar and Varghese [19] to address bandwidth fairness issues of network resources and the packet processing overhead of contemporary scheduling algorithms. DRR uses fair queuing and assigns a quantum of service to each busy queue. Round-robin scheduling is used for selecting a busy queue and dispatching a packet from it. The quantum of each queue is decremented by the size of the request that was dispatched. A different queue is selected when the quantum available for a queue is smaller than the next request size, the remaining quantum of this queue for this round is added to the quantum for the next round. Thus, the algorithm keeps track of shortchanged queues in preceding rounds and adjusts their quanta in succeeding rounds. It provides near perfect network fairness for competing application classes in terms of amount of data transmitted.

Although, in network scheduling, the assigned quantum can be adjusted using request size for fairness, a similar approach in disk scheduling does not provide similar disk-time fairness and fails to provide disk-performance isolation.

However, following the approach of DRR, the algorithm used in our framework, instead of shortchanging a queue, allows a queue to use "slightly" more disk time than the assigned quantum and adjusts the quantum of subsequent rounds with the excess disk time used in previous rounds.

3. Virtual I/O Scheduling Framework

This section describes our Virtual I/O Scheduling (VIOS) framework. As mentioned above, in this section and in the rest of the paper, we assume that each of the applications is running on a different operating system instance, thus, when we speak of multiple applications, we also are speaking of multiple operating systems.

3.1 Framework Objectives

In our framework, the storage system is accessed by multiple applications, each requiring a fraction of the storage performance. Explicit specification of the fractions is based on Quality of Service (QoS) objectives. In the absence of QoS, all applications equally share the total resource.

The VIOS framework has three objectives: (1) It must provide fairness with respect to the specified fractions of the storage performance. (2) It must provide performance isolation such that one

application cannot impact the I/O performance of another. (3) During its use of the storage system, each application may decide to share the resource with other applications or it may want exclusive rights to the resource, in which case it will have its own scheduling algorithm to satisfy its particular data delivery requirements. Both of these cases are particularly suitable in storage systems that support out-of-order service at their controllers. In the second half of the paper, we present an extension to our VIOS that is targeted at such systems.

3.2 Architectural Principles

Our I/O scheduling framework achieves the above three objectives with two core components: the VIOS and an optional set of application-dependent I/O schedulers. A schematic representation of our framework is shown in Figure 1. At the coarser granularity, the VIOS provides virtual slices of the shared I/O resource to the different applications. Optionally, at the finer granularity, during each of an application's time slices, its scheduler determines the "order" in which I/O requests are dispatched from its request queue. The former provides fairness and performance isolation across the different applications, while the latter, in conjunction with the former, aligns service with application data delivery requirements. Even when applications are willing to simultaneously share the I/O resource, instead of requiring an exclusive slice, each application is guaranteed its fair share of the resource, but its slice is an aggregation of smaller slices it gets during a scheduling interval. Traditionally, the above queuing mechanism is called fair queuing and the scheduling mechanism is called round-robin scheduling.

The key contributions of our I/O scheduling framework are (1) fairness in disk resource sharing, (2) disk performance isolation through the allocation of virtual slices of the storage system to applications, and (3) adaptive scheduling, i.e., provision of different scheduling algorithms that are appropriate for different application data delivery requirements. The remainder of the paper presents definitions needed to understand the design and implementation of the framework, describes the underlying algorithm, and provides analytical and experimental evidence to prove that the framework, indeed, meets its objective, with respect to fairness and performance isolation.

3.3 Queuing Model and Definitions Used in VIOS

Consider a system with n application classes and a queueing system with n I/O queues, $q_1, q_2, ..., q_n$, one queue per application class. Assume that all application classes generate I/O requests to a single storage system. We refer to this queueing discipline as the "Complete Fair Queueing Discipline" or CFQD, a term used in the rest of the paper. Note that we use the word "application" and "application class" interchangeably to mean a group of concurrently executing processes, and that these terms also imply operating system instances.

Definition 1. *A queue q_i ($1 \leq i \leq n$) is busy or active if it has one or more requests waiting that have not yet been dispatched to the disk, otherwise it is idle. A storage system is busy if there is at least one busy queue, otherwise it is idle.*

During any given time interval $[t_1, t_2]$, a queue may transition between the idle and busy states.

Definition 2. *A queue q_i ($1 \leq i \leq n$) is backlogged during any time interval $[t_1, t_2]$ of scheduling if it is busy during the entire interval. In other words, there is at least one request waiting for scheduling in the queue at all times during the interval, i.e., the queue never transitions to the idle state during the interval.*

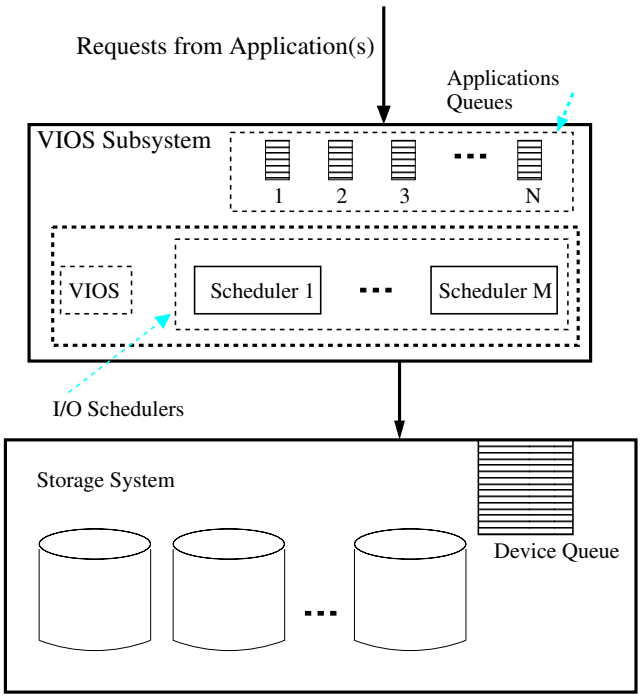

Figure 1. Schematic Representation of the Virtual I/O Scheduling Framework

3.4 Fairness and Performance Isolation by VIOS

We start this section with a widely used formal definition of fairness, adopted from [7]. Next, we describe how the VIOS provides fairness and performance isolation properties.

Intuitively, allocation of a *disk resource* to a set of I/O-generating processes is fair if, during *every* time interval, equal portions of the disk resource are allocated to each busy queue q_i. Our choice of the disk-resource measure is described later. The intuitive fairness notion assumes that all busy queues have equal fractions or weights. However, to generalize the definition, let us assume that according to some policy, the capacity of the resource is assigned to the queues in proportion to some *weights*. A detailed description of the assignment of weights is beyond the scope of the paper, but it suffices to say that such weights can be assigned by a system administrator to satisfy QoS objectives or may be computed based on process priorities at run-time [29]. Let w_i be the weight assigned to queue q_i. The disk resource allocation should be fair according to the weights.

Definition 3: *A scheduling policy is fair if there exists a constant θ such that for all time intervals $[t_1, t_2]$ during which a pair of queues q_i and q_j are backlogged,*

$$\left| \frac{S_i(t_1, t_2)}{w_i} - \frac{S_j(t_1, t_2)}{w_j} \right| \leq \theta,$$

where θ is a measure of fairness of the scheduling policy based on the disk resource measure S. Smaller values of θ indicate better fairness. If θ is a function of the length of the time interval $[t_1, t_2]$, then the scheduler is unfair.

Comment: Given a scheduling policy and a disk-resource measure S, the fairness measure θ assigns a quantitative value to the fairness of the policy with respect to S. Different constant values of θ mean different degrees of fairness, while a variable θ means the policy is unfair.

Now to the choice of the disk-resource measure. The second objective of the VIOS is to provide performance isolation among applications such that one application's `disk time` (and, hence, its execution time) does not depend on the characteristics of any other competing applications, in other words, the objective is to avoid application interference. Therefore, when we say that a scheduler is fair and provides performance isolation, we mean that the algorithm results in constant θ with respect to the disk-time resource measure (S^T). It should be clear by now, however, that the disk time allocated to an application depends on the number of active applications and an increase in the number of active applications reduces the disk time allocated to any particular application.

Below we present the reasons why a general round-robin algorithm fails to provide performance isolation and then we present our algorithm. Our analytical and experimental evaluations show the detailed fairness and performance isolation properties of the algorithm.

3.5 Problems with a Fair Queuing Round-Robin Scheduler (FQ-RR)

The VIOS could use an ordinary Fair Queuing Round-Robin Scheduler (FQ-RR), which assigns virtual slices of disk time to applications and lets applications dispatch their requests for the duration of the slices. More formally, a FQ-RR enqueues requests from different application classes using CFQD. It allows dequeuing and dispatches requests for a Q_i (Q_j) duration from an active queue q_i (q_j), and it round-robins through the list of active queues.

The fairness measure θ with respect to the disk-time resource (S^T) for $Q_i = Q_j = Q$ is given below [18]:

$$\left| \frac{S_i^T(t_1,t_2)}{Q} - \frac{S_j^T(t_1,t_2)}{Q} \right| = 1 + \frac{m}{Q} \cdot (T_{max} - 1), \quad (1)$$

where m is the number of rounds of service in time interval $[t_1, t_2]$.

Notice that θ involves m, which is a function of the length of the time interval $[t_1, t_2]$, i.e., the longer the time interval, the bigger the m and, hence, the larger the θ, which makes this scheduler unfair. It results in unfairness because of the varied sizes and seek characteristics of the requests of different applications. In other words, using this scheduler, even when both backlogged queues q_i and q_j have equal weights, one queue may be allocated more disk time than the other. For example, in each round, one queue's requests take exactly Q disk time but the other queues requests garner as much as $Q + T_{max}$ disk time. Here T_{max} is the time required to service the largest request size possible with the worst case seek and rotational latency [1]. In other words, the disk spends no more than T_{max} to service any request.

Accordingly, a FQ-RR results in disk-time fairness only if I/O requests can be scheduled on a bit-by-bit basis or all requests take the same amount of time to service. But I/O requests are indivisible units and can be scheduled only in chunks of request size. In addition, they are likely to take different amounts of disk time. Therefore, in practice, such an ordinary fair queuing scheduler results in unfairness in disk time and cannot provide performance isolation.

Next, we present an algorithm used in the VIOS that provides disk-time fairness and performance isolation for disk systems with a single disk. The algorithm accounts for heterogenous request sizes as well as differences in disk seek and rotational times associated with different requests. It compensates for excess time allocated to an application class in subsequent rounds. Our algorithm does not require knowledge of the seek and rotational characteristics of the disk system and it does not estimate the time required to satisfy requests of different sizes. It treats the disk system as a black box and uses feedback information from request responses to compensate for associated delays.

3.6 Completely Fair Queuing and Compensating Round-Robin Scheduler (CFQ-CRR) of VIOS

Initially, we provide an intuitive explanation of our scheduling algorithm and then in succeeding sections we delve into the details. At the beginning of a CFQ-CRR round of scheduling, a *quantum* of disk service time is assigned to each of the active queues. Then *requests from a busy queue are scheduled one by one until the queue's quantum is exhausted. When a request is scheduled and serviced [2], its service time is subtracted from the queue's quantum. When the remaining quantum is less than or equal to zero, no more requests are scheduled from the queue. In the next round of scheduling, a negative quantum from the previous round is added to the quantum for this round. In this way, the algorithm keeps track of the extra time taken by requests of any queue and compensates for it in succeeding rounds.*

3.7 CFQ-CRR Algorithm

This section formally presents our CFQ-CRR algorithm to schedule I/O requests to a disk system with a single disk. Let $w_i = Q_i$ be the weight assigned to queue q_i, where Q_i is the corresponding disk time to be consumed by q_i during a round of scheduling. Also, since the algorithm works in cycles, let us call each *cycle* of the algorithm a *round*.

Let the cumulative disk time for the requests dispatched for service from q_i for the k^{th} round be T_i^k. Queue q_i is allowed to dispatch requests in a round as long as its cumulative request service time for the round is less than its assigned quantum, i.e., $T_i^k < Q_i$. Dispatching of requests stops when $T_i^k \geq Q_i$ or when there are no more requests in the queue. If there are no more requests in the queue, then the queue is marked idle. If there are more requests in the queue when dispatching stops, then the cumulative disk time allocated to q_i could be more than the assigned quantum, i.e., $T_i^k > Q_i$. In other words, the last request was dispatched when $T_i^k < Q_i$, but the time to service the request was larger than the remaining time in the quantum. The extra disk time utilized by a request from q_i, over the assigned quantum, is called the *Compensating Quantum* CQ_i, i.e., $CQ_i = T_i^k - Q_i$. In the next round, the assigned Q_i is shortchanged by the CQ_i of the previous round, i.e., $Q_i = Q_i - CQ_i$. Note that, after compensation, if the quantum Q_i for round $k+1$ is negative, then no requests from q_i are scheduled and $T_i^{k+1} = 0$ and $CQ_i = T_i^{k+1} - Q_i$. This process is repeated for all queues and for all subsequent rounds.

4. CFQ-CRR Properties: Analytical Evaluation

In this section, we formally prove that CFQ-CRR is fair in terms of allocated disk time. To prove the fairness properties of CFQ-CRR, we start with Lemmas 1 and 2 below. Lemma 1 proves an invariant that is true after any round of service using the round-robin algorithm. A similar invariant was proven in [19] for network scheduling algorithms. Lemma 2 bounds the disk time allocated to each queue in the time interval $[t_1, t_2]$. Recall that the disk spends no more than T_{max} to service any request. For simplicity in presenting the proof, we assume that $Q_i > T_{max}$, i.e., the quantum allocated to each queue is larger than the time taken by any individual request in the queue; a similar proof can be constructed rather easily by relaxing this assumption.

[1] Often the size of the request is a constraint imposed by the hardware; the size can be restricted by the OS as well.

[2] Note that in simple disk systems, only one request can be serviced at a time. More complex disk systems are treated in the second half of the paper.

Lemma 1: For any queue q_i, where $0 \leq i < n$, the following invariant is true after any round of service by CFQ-CRR:

$$0 \leq CQ_i < T_{max}.$$

Proof: Before the start of any round $CQ_i = 0$. Note that CQ_i is assigned a value only when a round of service of q_i is completed and it is zeroed out after it is used to adjust Q_i. After servicing of q_i is completed in round k, there are two possibilities:

- If the total disk time T_i used by q_i is less than the assigned quantum Q_i, then $CQ_i = 0$. Note that, in this case, the queue does not have enough requests to utilize the entire quantum. A queue that does not have enough backlogged requests forfeits its unused quantum of that round; this makes our scheduling algorithm work conserving.
- If $T_i \geq Q_i$ then the compensating quantum $CQ_i = T_i - Q_i$. Let us assume that the number of requests scheduled in this round is m. Because $Q_i > T_{max}$, at least two requests are scheduled in one round from each queue, i.e., $m \geq 2$. Let us call the aggregate time for the first $m-1$ requests T_i^{m-1}. We know that $T_i^{m-1} < Q_i$, otherwise the m^{th} request would not be scheduled. The aggregate time T_i for m requests is the sum of T_i^{m-1} and the time required for the m^{th} request. We know that the m^{th} request can take no more than T_{max} because that it is the worst-case time required to service a request. Thus, the total aggregate time T_i cannot exceed $T_i^{m-1} + T_{max}$, i.e., $T_i \leq T_i^{m-1} + T_{max}$. Since, $T_i^{m-1} < Q_i$, $T_i < Q_i + T_{max}$. In other words, $T_i - Q_i < T_{max}$, or $CQ_i < T_{max}$.

Thus, the compensating quantum CQ_i is bounded by T_{max}, i.e.,

$$0 \leq CQ_i < T_{max}.$$

■

CFQ-CRR dispatches requests from queues using the round-robin algorithm. Therefore, between two rounds of service to backlogged queue q_i there must be one round of service to backlogged queue q_j and vice versa. Lemma 2 bounds the disk time allocated to any backlogged queue q_i during any time period $[t_1, t_2]$. Then, using Lemmas 1 and 2, Theorem 1 proves that CFQ-CRR is fair.

Lemma 2: For all time intervals $[t_1, t_2]$ during which queue q_i, where $0 \leq i < n$, is backlogged and there are m rounds of scheduling, the disk time $S_i^T(t_1, t_2)$ allocated to q_i in the interval is given by

$$m \cdot Q_i - T_{max} < S_i^T(t_1, t_2) < m \cdot Q_i + T_{max}.$$

See appendix for the proof.

Theorem 1: For any time interval $[t_1, t_2]$ during which queue q_i and q_j, where $0 \leq i, j < n$, are backlogged, the difference in the normalized disk time allocated to the two queues q_i, q_j by CFQ-CRR can be expressed as follows:

$$\left| \frac{S_i^T(t_1, t_2)}{Q_i} - \frac{S_j^T(t_1, t_2)}{Q_j} \right| < 1 + T_{max} \cdot \left(\frac{1}{Q_i} + \frac{1}{Q_j} \right).$$

See appendix for the proof. ■

Theorem 1 proves that the fairness measure of allocated disk time to two backlogged queues is independent of the length of the time interval and it is a constant. Thus, it proves that CFQ-CRR is fair in terms of disk-time allocation.

Comment: When $Q_i = Q_j = Q$, where Q is the disk time allocated to each q_i, the fairness measure is $< 1 + 2 \cdot \frac{T_{max}}{Q}$. As $Q \rightarrow \infty$, the fairness of this algorithm approaches 1. Although larger values of Q are desirable, as in process scheduling, smaller values result in better response times, especially for time-critical applications. We adopt a 20ms quantum for Q in the experiments presented below.

5. Experimental Evaluation of VIOS

To experimentally evaluate the fairness and performance isolation properties of the VIOS, we implemented it by extending the Completely Fair Queueing (CFQ) scheduler in the Linux 2.6.10 kernel. We used First-Come-First-Serve (FCFS) as the application-dependent scheduler for dispatching requests at the finer granularity. The VIOS allocates a quantum Q to each queue and uses CFQ-CRR to distribute the disk time across queues, and each of the queues uses FCFS (the default scheduler) to schedule its requests. In our analysis, all applications have equal weights for I/O, i.e., $w_i = w\ (0 \leq i < n) = Q$.

We conducted experiments using microbenchmarks and the `tiobench`[27] benchmark, and used `blktrace`[28] to capture I/O request response time traces that we used to compare the disk time allocated to each of the different applications.

Our experimental methodology is comprised of the following two classes of experiments, each of which demonstrates a different property of the VIOS.

- Experiments to prove that VIOS provides a strong degree of *fairness* in allocating disk time and that VIOS is not sensitive to heterogeneous request sizes and request seek and rotational latency (seek) characteristics of applications.
- Experiments to prove that VIOS provides total *performance isolation* even when applications generate requests of different sizes and when they have different seek characteristics.

5.1 Test Environment, Benchmarks, and Trace Collection

This subsection presents a brief description of our test environment, the `tiobench` benchmark, the two microbenchmarks, and the `blktrace` trace collection tool.

5.1.1 Test Environment

The experimental platform, which runs the Linux 2.6 kernel, consists of a dual-processor (2.28 GHz Pentium 4 Xeon with Hyperthreading) system with 2GB main memory (256MB used as a `ramdisk` to collect traces) and 1MB level-2 cache. For this study, a total of four logical processors are used and the system is attached to a 10,000 RPM 150GB SATA disk. To eliminate the interference of I/O requests from OS log activities, all benchmarks are configured to access the SATA disk configured with an `ext3` file system, which is different from the one hosting the OS.

5.1.2 Benchmarks

In this study, we use the threaded I/O benchmark (`tiobench`) [27] and two microbenchmarks (described later).

`tiobench`: `tiobench` is a file system benchmark that tests I/O performance using multiple threads that make simultaneous accesses in a sequential or random fashion. Given, as input parameters, the time to read/write, the read/write pattern (random or sequential), the block size, and the number of threads, this benchmark executes the I/O operations and reports the average bandwidth provided by the I/O system, average latency of request responses, and maximum latency over all accessed blocks. We use `tiobench` to quantify the fairness characteristics of the algorithms under study. Using this benchmark, it is very easy to control the request size, but it is difficult to control the exact seek characteristics of the generated requests. We use it to generate requests of a targeted size by generating requests to random locations of a file, thus, eliminating the prefetching effects of the file system, which alter the request size when read requests are sequential.

`microbenchmarks`: We use two microbenchmarks. In all our related experiments two application classes execute concurrently.

`Program 1` generates read requests to three 512MB files, which are contiguous on the disk; three threads each read one third of a different file sequentially. Initially, all blocks of the file are sequentially stored on the disk, thus, the inter-request seek distance, generated by the three threads accessing three files, is approximately the total number of sectors occupied by one file (each file occupies 1 million 512B sectors). Similarly, `Program 2` generates read requests to three 512MB files that are not necessarily contiguous on the disk; three threads read a third of a corresponding file sequentially. In this case, the physical separation between the three files dictates the seek behavior of the program. We have 64 files on the disk, from which we can select different files to simulate different seek distances. In this program, we vary this distance from 1 million (1M) 512B sectors (e.g., by selecting files 5, 6, and 7) to 64M sectors (by selecting files 1, and 64). One should note that the total data and number of requests generated by both programs are the same but the seek characteristics differ.

5.1.3 `blktrace`: Trace Collection Tool

To capture the disk-time allocation for every request, we collected I/O request traces at the block layer using `blktrace`. The OS kernel is instrumented to record I/O traces in kernel space; `blktrace` collects the traces from kernel space and transfers them into user space for long-term storage or makes them available for online processing via the `relayfs` pseudo-file system [26]. This file system reduces perturbation and increases the speed and reliability with which traces are collected. To further reduce the perturbation of copying traces to disk, we use a 256MB partition of main memory as a `ramdisk` to store the traces until the end of an experiment. Furthermore, to reduce the final size of the trace collected and stored in `ramdisk`, we use, in live mode, `blkparse`, a `blktrace` event parsing tool, to extract only queue insertion and completion event traces. For each request, from queue insertion and completion events, `blkparse` captures for each request, its logical block address (LBA), number of sectors accessed, type, time stamp, triggering process, and disk time. From the per-request disk time, the associated LBA, and the time stamp, we compute the disk time allocated per application class during each round. This information is captured for the experiments described in the next section. Note that even though these tools share CPU resources with the benchmarks, they do not effect benchmark behavior since the benchmarks are not CPU intensive – the majority of their execution times is spent doing I/O.

5.2 Effect of Request Size on Fairness

Our first experiment is designed to show the effect of application request size on the fairness measures of our framework in terms of disk-time allocation. For this we use `tiobench` – one application class (hereafter called `Application 1`) generates a fixed number of requests of size 4KB and another (hereafter called `Application 2`) generates the same number of 256KB requests. To ensure that a sufficient number of requests are backlogged, each class consists of eight concurrent threads.

The fraction of disk time allocated to each application is shown in Figure 2. Observe that CFQ-CRR allocates approximately 50% of the disk time to each application. The fraction of the disk bandwidth obtained by the two applications is 44% and 48%, respectively. This bandwidth is equivalent to the corresponding application's individual bandwidth when executed with an application similar to it. For example, the individual bandwidth of `Application 1`, when executed with another instance of `Application 1`, but accessing different data, is about 48%. The sum does not total 100% because of the disk time incurred for inter-application seeks. Also, from the figure, note that `Application 1` finishes first because, although an equal number of 4KB and 256KB requests are accessed, a 4KB request requires less disk time than a 256KB request. This shows the effectiveness of VIOS in providing fairness in terms of disk time. We further explore the effectiveness of VIOS in terms of performance isolation in Section 5.4. Also note that our framework is work conserving in that it allocates 100% of the disk time to `Application 2` when `Application 1` does not need it.

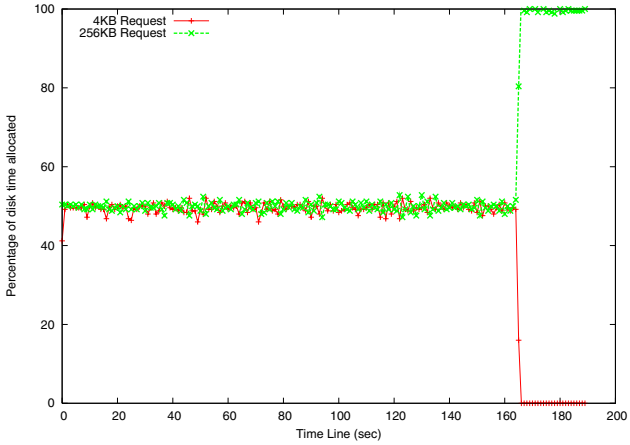

Figure 2. VIOS Disk Access Time Partitioning among Application Classes: Different Request Sizes

5.3 Effect of Application Seek Behavior on Fairness

Our second experiment is designed to show the effect of application seek behavior on the fairness measures of VIOS in terms of disk-time allocation. For this we use `Program 1` and `Program 2`, the microbenchmarks described in Section 5.1.2. `Program 1` and `Program 2` are configured to generate requests with an inter-request seek distance of 1M and 64M sectors, respectively.

The fraction of disk access time allocated to each program by VIOS is shown in Figure 3. Observe that VIOS allocates approximately 50% of the disk time to each of the programs. Also note, as shown in Figure 3, that equal disk time is allocated to both programs, even though they have different seek characteristics. `Program 1` finishes early, at about 170 seconds, whereas `Program 2` executes for about 30 seconds longer because, in comparison to `Program 1`, it spends more of its allocated disk time in seeking. This shows the effectiveness of CFQ-CRR in providing fairness in terms of disk time. The work-conserving nature of VIOS can be observed here as well. In Section 5.4, we further explore the effectiveness of CFQ-CRR in terms of performance isolation.

5.4 Performance Isolation with Request Size and Seek Characteristics

In this section, we investigate the effect of different application request sizes and seek characteristics on the execution times of other competing applications.

To explore the impact of request size on performance isolation behavior, we again use `tiobench` – `Application 1` generates a fixed number of requests of size 4KB and `Application 2` generates the same number of requests, but with request sizes of 4KB, 16KB, 32KB, 64KB, 128KB, 256KB, and 512KB for different experiments. To ensure that a sufficient number of requests are backlogged, each application consists of eight concurrent threads.

Figure 4 shows the execution times of both applications running alone (bottom two lines), and the execution times of each when executed concurrently (top two lines). The figure shows the impact

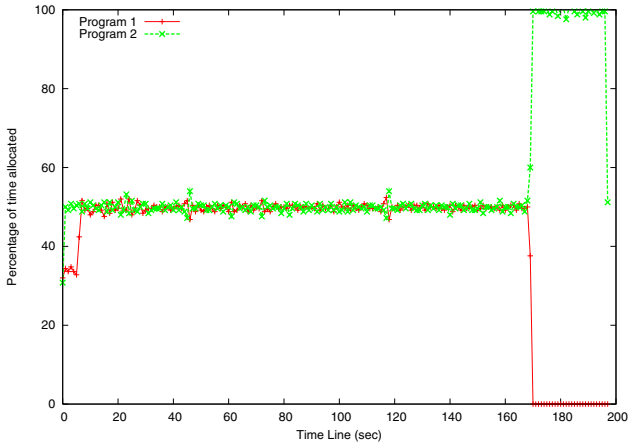

Figure 3. VIOS Disk Access Time Partitioning among Application Classes: Different Seek Characteristics

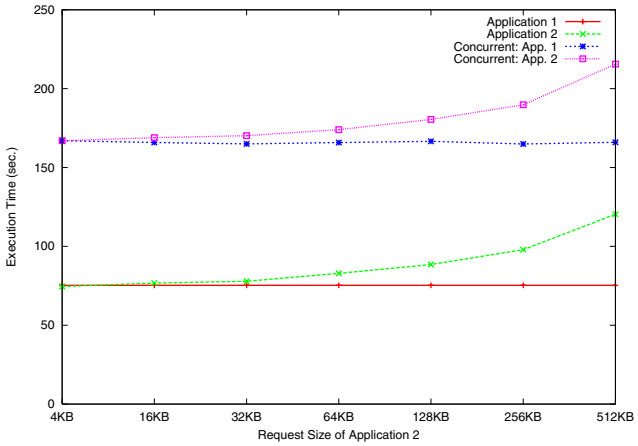

Figure 4. VIOS Application Execution Time: Varying Request Size of `Application 2`

of the different request sizes from `Application 2` on the execution time of `Application 1`. As can be seen in the figure, as the request size increases, only the corresponding application takes a longer time to execute – the increase in request size has no impact on the execution time of the other application. Thus, VIOS provides perfect performance isolation and is insensitive to the request size characteristics of competing applications.

To explore the impact of seek characteristics on performance isolation behavior, we again use `Program 1` and `Program 2`, the microbenchmarks described in Section 5.1.2. `Program 1` is configured to generate requests with an inter-request seek distance of 1M, but `Program 2` is configured to generate requests with an inter-request seek distance of 1M, 16M, 32M, and 64M sectors for different experiments.

Figure 5 shows the execution times of both applications running alone (bottom two lines) and the execution times of each when executed concurrently (top two lines). The figure shows the impact of the different seek distances of requests from `Program 2` on the execution times of both applications. As can be seen, as the seek distance between requests of `Program 2` increases, only its execution time increases – there is virtually no impact on the execution time of the other application. Thus, CFQ-CRR provides perfect performance isolation in disk-time allocation, and is insensitive to the seek characteristics of competing applications.

Together, these experiments answer our question, i.e., they show that the disk-time sharing notion in CFQ-CRR and, therefore, VIOS and the disk-time resource measure result in fairness and performance isolation. It is rather straight forward to see that the VIOS can provide differential quality of service with differing values of quanta.

Applications that require scheduling algorithms other than FCFS to satisfy their particular application data delivery requirements may use a suitable algorithm. With the knowledge of the number of busy queues in the system and their respective weights, such an algorithm can calculate an upper bound on the amount of time it has to wait between quantum allocations and schedule the requests accordingly. Due to the predictable disk-time allocation across queues, scheduling optimizations based on application data delivery requirements and those that seek to improve aggregate disk utilization can be performed simultaneously. This particular feature of our framework is suitable for commodity operating systems, such as Linux, as they are required to service multiple appli-

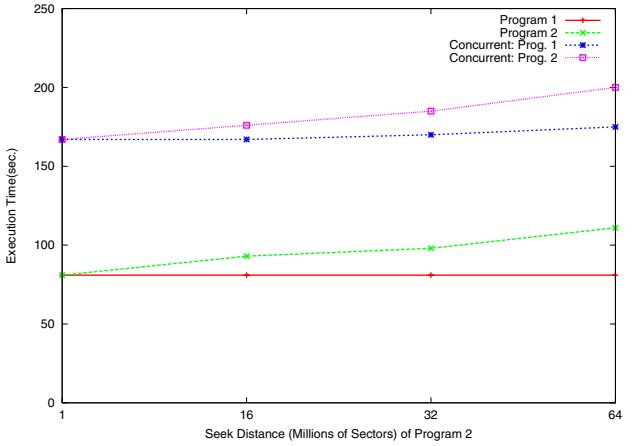

Figure 5. VIOS Application Execution Time: Varying Seek Characteristics of `Program 2`

cations with potentially orthogonal data delivery requirements. We are working on pushing this into the Linux kernel.

6. Extensions to VIOS: Handling Complex Devices

With Small Computer System Interface 1 (SCSI-1) or IDE/ATA, only one request can be pending at any time. This fits well with the VIOS, which allows the application-dependent schedulers to dispatch only one request at a time. However, this does not permit application-dependent schedulers to exploit advances in device technologies, in particular, device queueing and intelligent controller systems (such as RAID) that support out-of-order delivery.

Tagged Command Queuing (TCQ) and Native Command Queuing (NCQ) mechanisms [11, 15, 32, 33] and intelligent controller systems allow host systems to submit multiple I/O commands to disk systems. We refer to TCQ and NCQ simply as *device queuing*. In device queuing, a command is tagged at the drive with an identifier and then reordered to minimize the mechanical movement of the disk head (both rotational and seek latencies). This results in improved I/O request latencies and improved disk utilization. For

random I/O workloads executed in server and network storage systems, this reordering can result in significant performance improvements. In contrast, for sequential I/O workloads there is not much room for performance improvement [30, 31]. Under random workloads, these mechanisms contribute from 0 to 30% improvement in disk utilization. Applications may take advantage of this feature by using asynchronous I/O [4, 11] because the asynchronous I/O is non-blocking and can increase the number of requests in the device queue. Historically, a TCQ disk supports up to 128 pending requests, whereas an NCQ disk supports up to 32. However, the performance improvement can be realized only when the device queue contains "several" pending requests.

Accordingly, in order to make VIOS applicable to contemporary devices, it must allow the application-dependent schedulers to send multiple requests at once. For this, we propose a new scheduling algorithm that still provides fairness and performance isolation, while exploiting the command queuing features of the disk systems.

6.1 Introduction to Application-Dependent P Scheduler

To take advantage of device queuing, in each round of round-robin service, the VIOS must allow each application-dependent scheduler to dispatch multiple requests from each busy queue, while ensuring that the requests take approximately the allocated quantum of disk time. To determine the number of requests that satisfy this constraint, we need an approximation of the number of requests that can be scheduled in a given quantum. The approximation must be such that the number of requests scheduled takes *no less* than the given quantum of disk time and also takes only a bounded amount of time over the allocated quantum. These two objectives can be achieved by using the maximum bandwidth obtainable from the disk system as a reference point. Let Q be the quantum of disk time allowed in milliseconds and BW be the maximum obtainable bandwidth of the disk system in MB/s. Then the number of bytes that can be scheduled from the queue is $(Q \cdot BW)/1000$ MB $\approx 2 \cdot Q \cdot BW$ sectors (assuming that each sector is 512 bytes). From this, a scheduler can easily schedule requests from its queue such that the sum of the sectors scheduled (in bytes) is equal to or slightly greater than the above number. Notice that the scheduled requests take Q or longer disk time but never take smaller than Q time because the estimation is based on the maximum bandwidth. (This may not be true for disk systems with local caches, especially when the requests are serviced by the cache.) We call the new scheduler P because it exploits parallelism using device queuing. The complete algorithm is described in the next subsection.

6.2 P: Scheduling Algorithm for Exploiting Device Queuing

Let BW be the maximum obtainable bandwidth and K be the number of requests that can be outstanding at the device queue. In Linux, the device queue depth can be measured from the **sys** file system; for other operating systems and virtual systems, this can be gleaned from other locations (see, e.g., [34]). For each busy queue q_i, the corresponding scheduler:

1. Compute the total number of sectors to be serviced, SQ_i: $SQ_i = 2 \cdot Q_i \cdot BW$.
2. Dispatch requests from queue q_i as long as the sum of the sectors of the dispatched requests is less than SQ_i and the total number of pending requests in the driver queue is less than K
3. If there are no more requests in queue q_i and the number of pending requests is less than K, go to step 1 and repeat the process for queue q_{i+1}; else continue to step 4.
4. Once the number of pending requests reaches K, only begin dispatching requests again when there are two pending requests. The reason for this heuristic is explained later.
5. When all the schedulers that are allowed to reorder requests have completed their round, the VIOS takes over the scheduling responsibility.
6. The VIOS calculates the compensating quantum CQ_i, allocates the quanta Q_i for the next round, and applies round-robin scheduling for all busy queues with positive quanta.

There are a few issues in the above algorithm that need clarification. First, once the device queue is full or we finish scheduling requests in a round of round-robin service, why do we wait for the device queue to dispatch all but two requests? This is done to avoid starvation of requests in the device queue. Recall that command queuing reorders the outstanding requests to minimize disk head seek movement using a Shortest Seek First (SSF) [15] like scheduler. Thus, if we continuously fill the device queue and the incoming requests are for sectors closer to the current head position, then requests that require larger seeks or rotations will be delayed indefinitely. Thus, when the disk system is loaded heavily, the disk head localizes itself to a particular location and may result in request starvation. For this reason, the VIOS does not start the next round until almost all of the requests in the device queue are dispatched to the disk. However, if we wait until all the requests are dispatched, we will be idling the disk for a short duration of time. Although, this duration may not be too long, the VIOS takes a proactive step and allows P to start dispatching requests as soon it sees that there are two requests left in the device queue. Even though the two-request rule is a heuristic, our experimentation shows no signs of request starvation. On the other hand, experiments with three requests show that there is clear request starvation in some cases, hence, we recommend using either one or two requests as the triggering point.

Second, the VIOS maintains strict fairness by scheduling only one request at a time, whereas the VIOS with P results in a less stringent fairness measure because it schedules several requests based on the quantum and the measured peak bandwidth. Therefore, we expect to see an increase in scheduling unfairness using VIOS with P, however, such unfairness is tightly bounded because of the compensation mechanism. Experimental evaluation of P is discussed in the next section, which shows that VIOS still provides fairness and performance isolation while exploiting the command queuing features of the disk.

7. Exploiting Device Queueing by VIOS with P – VIOS(P): Case Study with Multiple Threads

The test system is an IBM 16-way, 1.4 GHz Power4 SMP configurable up to 32 GB of main memory. We use two processors and 4 GB memory for this experimentation. The experimental platform contains two SCSI disks that support TCQ, in addition to similar SCSI disks that host the operating system. Using `tiobench`, we create a set of 32 threads, each of which reads 1,000 random 4KB blocks from 32 files on the test TCQ disk. At the end of the execution, we capture four performance measurements for six different schedulers; they are shown in Figure 6. The six schedulers are VIOS(P), VIOS, and FCFS, and the four in the Linux operating system, i.e., Deadline, AS, CFQ, and Noop [17].

The top part of the figure shows the thread average and maximum request latencies associated with the different scheduling algorithms and the percentage of requests that exceed a one-second latency. The bottom part of the figure shows the thread execution times; each bar is comprised of two differently colored segments: the bottom, lighter colored segment represents the time at which the first thread, of the group of 32, finished its execution, and the top, darker colored segment represents the time at which the last thread finished its execution. The difference between the two execution times represents the unfairness – a strict fair scheduling algorithm

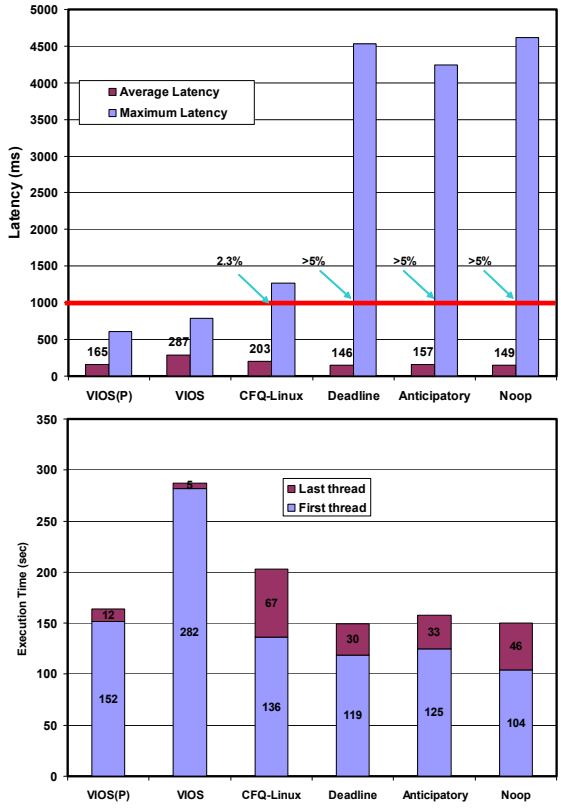

Figure 6. Case Study with Multiple Threads: Thread Average and Maximum Latency (top) and Execution Time (bottom).

must result in all threads finishing at approximately the same time. Note that the threads may not finish at exactly the same time due to OS interference.

7.1 Analysis of Scheduler Performance

From Figure 6, in terms of fairness, the VIOS results in the best fairness, followed closely by VIOS(P); and both VIOS and VIOS(P) result in the best maximum latency, in that order. Notice that, compared to VIOS, using VIOS(P), unfairness increases slightly. In contrast, Deadline, Anticipatory, and Noop result in higher unfairness; in addition, they result in 5% of requests exceeding a latency of one second. In contrast, VIOS(P) results in no requests with a latency of one second. We further analyze these findings below.

7.2 Analysis of VIOS and VIOS(P)

As shown in Figure 6, the VIOS results in the best fairness. However, it also results in the worst execution time. Although, following the theoretical results, one might expect this unfairness to be much lower, it is exacerbated by the non-deterministic nature of OS process scheduling in Linux. Due to the strict sequential scheduling of requests with interleaved disk idle times, and failing to exploit the benefits of device queuing, the VIOS results in the worst execution time.

On the other hand, as compared to the VIOS, VIOS(P), which dispatches multiple requests at a time, results in improved execution. In terms of fairness, since both apply similar scheduling logic, one might expect that both result in the same degree of fairness. However, this is not the case – VIOS(P) increases unfairness because it dispatches multiple requests at a time. This unfairness also is exacerbated by the non-deterministic nature of OS process scheduling in Linux. One could achieve different degrees of performance isolation by using VIOS and VIOS(P) in virtualized storage systems.

7.3 Analysis of the Performance of Deadline, Anticipatory, Noop, and CFQ-Linux Schedulers

The scheduling behavior with and without device queuing is quite different in the Deadline, Anticipatory, Noop, and CFQ-Linux schedulers because with device queueing the device scheduler determines the final scheduling of the requests to disk. These schedulers initially fill the device queue and keep it full as long as there are outstanding requests in the queue, thus, they become pseudo-SSF; and it is well known that any SSF scheduling algorithm results in starvation of requests (see, e.g., [23, 10, 22]). This can be seen in Figure 6, where the maximum latency is over four seconds and over 5% of requests exceed the one-second latency bound.

On the other hand, CFQ-Linux keeps at most one request in the device queue to eliminate the pseudo-SSF behavior of the queue and results in better maximum latency. However, it results in a larger degree of unfairness: the last thread takes 50% more time than the first thread to finish. In addition, CFQ-Linux allows limited back-seeking and, thus, schedules requests to disk in pseudo-SSF order; however, because it enforces a 125 ms maximum latency, it results in a lower maximum latency.

8. Concluding Remarks

In this paper we presented an I/O scheduling framework that virtualizes the shared storage for meeting diverse data delivery requirements of applications running on a system with a single disk storage system. It allows the co-existence of multiple applications and their corresponding schedulers.

Our application-independent Virtual I/O Scheduler – VIOS – controls coarse-grain allocation of disk time to applications and, instead of requiring a detailed performance model, it allows applications to take slightly larger quanta than those allocated and compensates for the excess time in succeeding rounds of round-robin service. We analytically and experimentally demonstrated that our framework provides fairness in apportioned disk time. In addition, it provides performance isolation among applications such that the I/O characteristics of any application cannot impact the performance of other applications. We described a new application-dependent scheduler – P – that exploits device command queuing, which takes multiple requests at a time, and demonstrated its performance isolation properties and compared its performance with several other schedulers.

Currently, our scheduling framework does not enforce latency requirements explicitly and also does not handle storage systems such as RAIDs, however, we are extending it to provide this capability. We are leveraging this framework in our ongoing work to build a complete I/O virtualization solution.

9. Acknowledgments

This work is supported by the Department of Energy under Grant No. DE-FG02-04ER25622, an IBM SUR grant, and the University of Texas at El Paso. The authors gratefully acknowledge Jens Axboe for implementing CFQ in Linux and for his continued commitment to improving it. We would like to thank him for sharing his thoughts and for answering our questions. We would like to thank Jan Beck, Yipkei Kwok, Sarala Arunagiri, and Maria del Carmen Ruiz Varela for providing valuable feedback and for careful reading of the paper.

A. Proofs

Lemma 2: For all time intervals $[t_1, t_2]$ during which queue q_i, where $0 \leq i < n$, is backlogged and there are m rounds of scheduling, the disk time $S_i^T(t_1, t_2)$ allocated to q_i in the interval is given by

$$m \cdot Q_i - T_{max} < S_i^T(t_1, t_2) < m \cdot Q_i + T_{max}.$$

Proof: Suppose that there are m rounds of service to q_i in the interval $[t_1, t_2]$. Also assume that t_1 is at the beginning of the first round and at the end of the hypothetical zeroth round. Recall that quantum Q_i for the first round is adjusted by the compensating quantum from the end of round zero.

With the above observations in mind, let CQ_i^k be the compensating quantum for queue q_i after the end of round $k, (0 \leq k \leq m)$, and let T_i^k be the aggregate time allocated to q_i in the same round. Let $S_i^T(t_1, t_2)$ be the cumulative disk time allocated to q_i for m rounds, $S_i^T(t_1, t_2) = \sum_{j=1}^{m} T_i^j$.

Recall that, because of the compensating quantum from round $k-1$, the disk time allocated to q_i for round k is given by $T_i^k = Q_i - CQ_i^{k-1} + CQ_i^k$. Now, let us compute the sum of T_i^k for m rounds, i.e.,

$$S_i^T(t_1, t_2) = Q_i - CQ_i^0 + CQ_i^1 + \ldots + Q_i - CQ_i^{m-1} + CQ_i^m.$$

Because of the telescoping series, this can be written as:

$$S_i^T(t_1, t_2) = m \cdot Q_i - CQ_i^0 + CQ_i^m. \quad (2)$$

Using Lemma 1, for CQ_i^0 and CQ_i^m in Equation 2, it is trivial to see that

$$m \cdot Q_i - T_{max} < S_i^T(t_1, t_2) < m \cdot Q_i + T_{max}.$$

■

Theorem 1: For any time interval $[t_1, t_2]$ during which queue q_i and q_j, where $0 \leq i, j < n$, are backlogged, the difference in the normalized disk time allocated to the two queues q_i, q_j by CFQ-CRR can be expressed as follows:

$$\left| \frac{S_i^T(t_1, t_2)}{Q_i} - \frac{S_j^T(t_1, t_2)}{Q_j} \right| < 1 + T_{max} \cdot \left(\frac{1}{Q_i} + \frac{1}{Q_j} \right).$$

Proof: Let us assume that there are m rounds of service to q_i in the interval $[t_1, t_2]$ and m' rounds of service to q_j in the same interval. Recall that in any round-robin servicing, the property $|m - m'| \leq 1$ is true. In other words, between any two rounds of service to q_i, there must be one round of service to queue q_j and vice versa.

From Lemma 2, we can bound the disk time allocated in interval $[t_1, t_2]$ to q_i by:

$$m \cdot Q_i - T_{max} < S_i^T(t_1, t_2) < m \cdot Q_i + T_{max}$$

$$\Rightarrow m - \frac{T_{max}}{Q_i} < \frac{S_i^T(t_1, t_2)}{Q_i} < m + \frac{T_{max}}{Q_i}, \quad (3)$$

and the disk time allocated to q_j by:

$$m' \cdot Q_j - T_{max} < S_j^T(t_1, t_2) < m' \cdot Q_j + T_{max}$$

$$\Rightarrow m' - \frac{T_{max}}{Q_j} < \frac{S_j^T(t_1, t_2)}{Q_j} < m' + \frac{T_{max}}{Q_j}. \quad (4)$$

Because $|m - m'| \leq 1$, the difference between Equations 3 and 4 can be expressed as follows:

$$\left| \frac{S_i^T(t_1, t_2)}{Q_i} - \frac{S_j^T(t_1, t_2)}{Q_j} \right| < 1 + T_{max} \cdot \left(\frac{1}{Q_i} + \frac{1}{Q_j} \right).$$

■

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. R. Romer, R. B. Becker Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes, "Minerva: an Automated Resource Provisioning Tool for Large-scale Storage Systems," *ACM Transactions on Computer Systems*, November 2001, 19(4):483–518.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: Running Circles around Storage Administrators," *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002, pp. 175–188.

[3] J. Axboe, "Linux Block IO - Present and Future," *Proceedings of the Ottawa Linux Symposium 2004*, Ottawa, Ontario, Canada, July 21-24, 2004, pp. 51–61.

[4] S. Bhattacharya, S. Pratt, B. Pulavarty, J. Morgan, "Asynchronous I/O Support in Linux 2.5," *Proceedings of the Ottawa Linux Symposium 2003*, Ottawa, Ontario, Canada, July 23-26, pp. 371–386.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk Scheduling with Quality of Service Guarantees," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Vol. II, June 7-11, 1999, pp. 400–405.

[6] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, Austin, TX, September 1989, pp. 1–12.

[7] S. J. Golestani, "A Self-Clocked Fair Queueing Scheme for Broadband Applications," *Proceedings of IEEE INFOCOM'94*, Toronto, Canada, June 12-16, 1994, pp. 636–646.

[8] P. Goyal, H. M. Vin, and H. Chen, "Start-time Fair Queueing: a Scheduling Algorithm for Integrated Services Packet Switching Networks," *ACM SIGCOMM Computer Communication Review*, October 1996, 26(4):157–168.

[9] A. G. Greenberg and N. Madras, "How Fair is Fair Queuing," *Journal of the ACM*, July 1992, 39(3):568–598.

[10] M. Hofri, "Disk Scheduling: FCFS vs SSTF Revisited," *Communications of the ACM*, November 1980, 23(11):645–653.

[11] A. Huffman, J. Clark, "Serial ATA: Native Command Queuing – An Exciting New Performance Feature for Serial ATA," *A Joint White Paper By: Intel Corporation and Seagate Technology LLC*, July 2003.

[12] W. Jin, J. Chase, and J. Kaur, "Interposed Proportional Sharing for a Storage Service Utility," *Proceedings of the ACM Sigmetrics - Performance (SIGMETRICS'04)*, New York, June 2004.

[13] C. Lumb, A. Merchant, and G. Alvarez, "Facade: Virtual Storage Devices with Performance Guarantees," *Proceedings of the Conference on File and Storage Technology (FAST'03)*, San Francisco, CA, March 31-April 2, 2003, pp. 131–144.

[14] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, 2002, 6(1):4–15.

[15] N. Marushak and R. Jeppsen, "Deciding between SATA and SAS? Compare their Behind-the-Scenes Command Queuing Techniques," *Storage Networking Work Online*, September 2004.

[16] P. McKenney, "Stochastic Fairness Queueing," *Internetworking: Research and Experience*, January 1991, Vol. 2, pp. 113–131.

[17] S. Pratt, and D. Heger, "Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers," Proceedings of the Ottawa Linux Symposium 2004, Ottawa, Ontario, Canada, July 21-24, pp. 425–448.

[18] S. Seelam and P. Teller, "Fairness and Performance Isolation: an Analysis of Disk Scheduling Algorithms," *Proceedings of International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems (HiperIO'06)*, Barcelona, Spain, September 25-28, 2006.

[19] M. Shreedhar and G. Varghese, "Efficient Fair Queueing using Deficit Round-Robin," *IEEE/ACM Transactions on Networking (TON)*, June 1996, 4(3):375–385.

[20] P. Shenoy and H. Vin, "Cello: a Disk Scheduling Framework for Next Generation Operating Systems," *Real-Time Systems*, January 2002, 22(1-2):9–48.

[21] G. Tan and J. Guttag, "Time-based Fairness Improves Performance in Multi-rate WLANs," *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2004.

[22] A. S. Tanenbaum, *Modern Operating System*, Prentice Hall, 2001.

[23] J. Toby and T. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Communications of ACM*, 1972, 15(3):177–184.

[24] R. Wijayarante and A. L. N. Reddy, "Providing QOS Gurantees for Disk I/O," *Multimedia Systems*, January 2000, 8(1):57–68.

[25] J. Zhang, A. Riska, A. Sivasubramaniam, Q. Wang, E. Riedel, "Storage Performance Virtualization via Throughput and Latency Control," *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, September 2005.

[26] T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore, and M. Dagenais, "relayfs: an Efficient Unified Approach for Transmitting Data from Kernel to User Space," *Proceedings of the 2003 Linux Symposium*, Ottawa, Canada, July 23-26, 2003.

[27] tiobench: Thread I/O Bench for Linux, tiobench.sourceforge.net.

[28] J. Axboe, "Patch: Block Device IO Tracing," `http://lwn.net/Articles/155340/`.

[29] J. Axboe, "Time Sliced CFQ I/O Scheduler," `http://kerneltrap.org/node/4406`.

[30] From SCSI to SATA, `http://www.westerndigital.com/en/library/sata/2579-001097.pdf`

[31] TCQ, RAID, SCSI, and SATA, `http://www.storagereview.com/articles/200406/20040625TCQ_1.html`

[32] Tagged Command Queuing: `http://en.wikipedia.org/wiki/Tagged_Command_Queueing`

[33] Tagged Command Queuing (TCQ): `http://www.wdc.com/en/library/sata/2579-001076.pdf`

[34] Setting SCSI-Adapter and Disk-Device Queue Limits, `www.unet.univie.ac.at/aix/aixbman/prftungd/scsiqueue.htm`