QBox: Guaranteeing I/O Performance on Black Box Storage Systems

Dimitris Skourtis
skourtis@cs.ucsc.edu

Shinpei Kato
shinpei@cs.ucsc.edu

Scott Brandt
scott@cs.ucsc.edu

Department of Computer Science
University of California, Santa Cruz

ABSTRACT

Many storage systems are shared by multiple clients with different types of workloads and performance targets. To achieve performance targets without over-provisioning, a system must provide isolation between clients. Throughput-based reservations are challenging due to the mix of workloads and the stateful nature of disk drives, leading to low reservable throughput, while existing utilization-based solutions require specialized I/O scheduling for each device in the storage system.

Qbox is a new utilization-based approach for generic black box storage systems that enforces utilization (and, indirectly, throughput) requirements and provides isolation between clients, without specialized low-level I/O scheduling. Our experimental results show that Qbox provides good isolation and achieves the target utilizations of its clients.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.8 [**Operating Systems**]: Performance

Keywords

Storage virtualization, quality of service, resource allocation, performance

1. INTRODUCTION

During the past decade there has been a significant growth of data with no signs of slowing. Due to that growth there is a real need for storage devices to be shared efficiently by different applications and avoid the extra costs of having more and more under-utilized devices dedicated to specific applications. In environments such as cloud systems, where multiple "clients", i.e., streams of requests, compete for the same storage device, it is especially important to manage the performance of each client. Failure to do so leads to low performance for some or all clients depending on complex factors such as the I/O schedulers used, the mix of client workloads, as well as storage-specific characteristics. Unfortunately, due to the nature of storage devices, managing the performance of each client and isolating them from each other is a non-trivial task. In a shared system, each client may have a different workload and each workload may affect the performance of the rest in undesirable and possibly unpredictable ways. A typical example would be a stream of random requests reducing the performance of a sequential or semi-sequential stream, mostly due to the storage device performing unnecessary seeks. The above is the result of storage devices trying to be equally fair to all requests by providing similar throughput to every stream. Of course, not all requests are equally costly, with sequential requests taking only a small fraction of a millisecond and random requests taking several milliseconds, 2-3 orders of magnitude longer. Note that a sequential stream does not have to be perfectly sequential–none ever truly are–and that real workloads often exhibit such behavior.

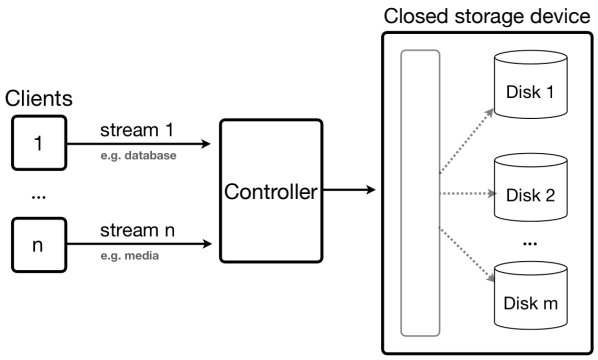

Figure 1: Given that we have no access to the storage device, we place a controller between the clients and the device to provide performance management to the clients, i.e., the request streams.

Providing a solution to the above problem may require using specific I/O schedulers for every disk-drive or node in a clustered storage system. Moreover, it could require changes to current infrastructure such as the replacement of the I/O scheduler of every client. Such changes may create compatibility issues preventing upgrades or other modifications to be applied to the storage system. Instead of making modifications to the infrastructure of an existing system it is often easier and in practice cheaper to deploy a solution between the clients and the storage. We call that the black box approach since it imposes minimal requirements on the

HPDC'12, June 18–22, 2012, Delft, The Netherlands.

clients and storage, and because it is agnostic to the specifications of either side. Our approach partly fits the grey-box framework for systems presented in [1], however, QBox requires fewer algorithmic assumptions about the underlying system. In this paper we take an almost agnostic approach and target the following problem: given a set of clients and a storage device, our goal is to manage the performance of each client's request stream in terms of disk-time utilization and provide each client with a pre-specified proportion of the device's time, while having no internal control of either the clients or the storage device, or requiring any modifications to the infrastructure of either side.

Clients want throughput reservations. However, except for highly regular workloads, throughput varies by orders of magnitude depending upon workload (Figure 2) and only a fixed fraction of the (highly variable) total may be guaranteed. By isolating each stream from the rest, utilization reservations allow a system to indirectly guarantee a specific throughput (not just a share of the total) based on direct or inferred knowledge about the workload of an individual stream, independent of any other workloads on the system and can allow much greater total throughput than throughput-based reservations [17]. Our utilization-based approach can work with Service Level Agreements (SLA); requirements can be converted to utilization as demonstrated in [18] and as long as we can guarantee utilization, we can guarantee throughput provided by an SLA.

To our knowledge there is no prior work on utilization-based performance guarantees for black box storage devices. Most work that is close to our scenario such as [15, 11] is based on throughput and latency requirements, which are hard to reserve directly without under-utilizing the storage for a number of reasons such the orders-of-magnitude cost differences between best- and worst-case requests. Moreover, throughput-based solutions create other challenges such as admission control. Without very specific knowledge about the workloads, the system must make worst-case assumptions, leading to extremely low reservable throughput. On the other hand, existing solutions based on disk utilization [18, 17, 10] only support single drives and if used in a clustered storage system they require their scheduler to be present on every node.

In this paper, we present a novel method for managing the performance of multiple clients on a storage device in terms of disk-time utilization. Unlike the management of a single drive, in the black box scenario it is hard to measure the service time of each request. Instead, our solution is based on the periodic estimation of the average cost of sequential and random requests as well as the observation that their costs have an orders-of-magnitude difference. We observe the throughput of each request type in consecutive time windows and maintain separate moving estimates for the cost of sequential and random requests. By taking into account the desired utilization of each client we schedule their requests by assigning them deadlines and dispatching them to the storage device according to the Earliest Deadline First (EDF) algorithm [13, 21]. Our results show that the desired utilization rates are achieved closely enough, achieving both good performance guarantees and isolation. Those results stand over any combination of random, sequential, and semi-sequential workloads. Moreover, due to our utilization-based approach, it is easy to decide whether a new client may be admitted to the storage system, possibly by modifying the rates of other clients. Finally, all clients may access any file on the storage device and we make no assumptions about the location of the data on a per client basis.

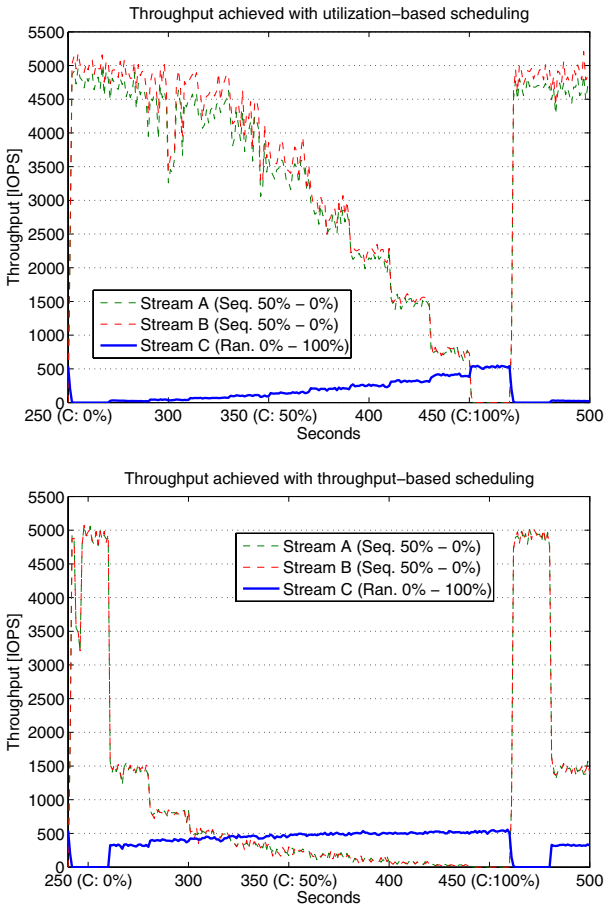

Figure 2: QBox (top) provides isolation, while the introduction of a random stream makes the throughput of sequential streams drop dramatically with throughput-based scheduling (bottom.)

2. SYSTEM MODEL

Our basic scenario consists of a set of clients each associated with a stream of requests and a single storage device containing multiple disks. Clients send requests to the storage device and each stream uses a proportion of the device's execution time. We call that proportion the utilization rate of a stream and it is either provided by the client or in practice, by a broker, which is part of our controller and translates SLAs into throughput and latency requirements as in [17, 18] or [10]. Briefly, to translate an SLA to utilization, we measure the aggregate throughput of the system for sequential and random requests separately over small amounts of requests (e.g., 20) and set a confidence level (e.g., 95%) to avoid treating all requests as outliers. Details about arrival patterns and issues such as head/track switches and bad layout are presented in [19].

The main characteristic of our scenario is that we treat the storage device as a black box. In other words, we only interact with the storage device by passing it client requests

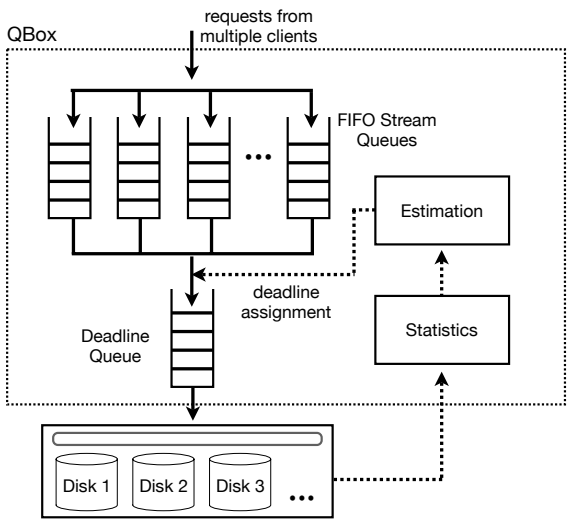

Figure 3: The controller architecture.

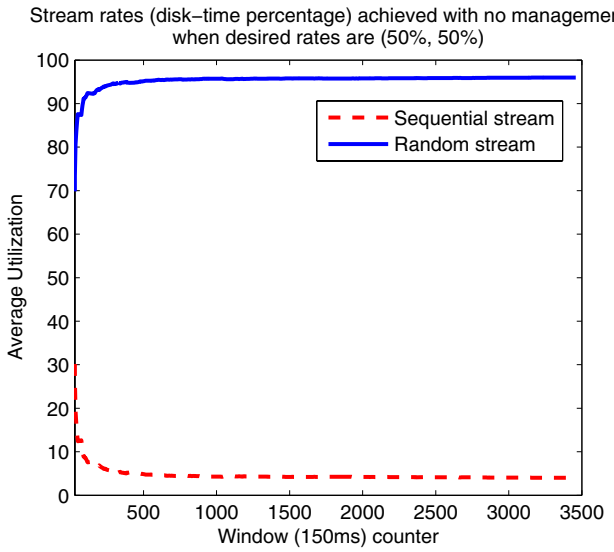

Figure 4: When our controller sends the requests in the order they are received from the clients, the system fails to provide the desired rates.

and receiving responses. For example, we cannot modify or replace the device's scheduler as is the case in [18] and do not assume it uses a particular scheduler. Moreover, we cannot control which disk(s) are going to execute each request and do not restrict clients to specific parts of the storage device. Due to those requirements the natural choice is to place a controller between the clients and the storage device. Hence, all client requests go through the controller, where they are scheduled and eventually dispatched to the device. As we will see, this setup allows us to gather little information regarding the disk execution times, which turns scheduling and therefore black box management into a challenge.

We manage the performance of the streams in a time-based manner. After a request reaches our controller, we assign it a deadline by keeping an estimate of the expected execution time e for each type of request (sequential or random) and by using the stream's rate r provided by the broker. Using e and r we compute the request's deadline by $d = e/r$. The absolute deadline of a request coming from stream s is set to $D_s = T_s + d$, where T_s is the sum of all the relative deadlines assigned so far to the requests of stream s. Although, we are using "deadlines" for scheduling, our goal is not to strictly satisfy deadlines. Instead, it is the relative values that matter with regards to the dispatching order. On the other hand, if we used a stricter dispatching approach e.g., [18], then the absolute times would be important for replacing the expected cost with the actual cost after the request was completed. In this paper we do not focus on urgent requests, however, it is possible to place such requests ahead of others in the corresponding stream queue by simply assigning them earlier deadlines.

Although we do not assume the storage device is using a specific disk scheduler, it is better to have a scheduler which tries to avoid starvation and orders the requests in a reasonable manner (as most do). A stricter dispatching policy such as [18] can be used on the controller side to avoid starvation by placing more emphasis on satisfying the assigned deadlines instead of overall performance. The next section presents our method for estimating execution times and managing the performance of each stream in terms of time. We also discuss practical issues we faced while applying our method and discuss how we addressed them.

3. PERFORMANCE MANAGEMENT

In QBox we maintain a FIFO queue for each stream and a deadline queue, which may contain requests from any stream. The deadline queue is ordered according to the Earliest Deadline First (EDF) scheduler and the deadlines are computed as described in the previous section. Whenever we are ready to dispatch a request to the storage device the request with the smallest absolute deadline out of all the stream queues is moved to the deadline queue. To find the earliest-deadline request it suffices to look at the oldest request from each stream queue, since any other request before that has either arrived at a later time or is less urgent. Next, the request with the earliest deadline is removed from the deadline queue and dispatched to the device.

3.1 Estimating execution times

As mentioned earlier, we aim to provide performance management through a controller placed between the clients and the storage device. We wish to achieve this goal without knowledge of how the storage device schedules and distributes the requests among its disks and without access to the storage system internals. Most importantly, we are unaware of the time each request takes on a single disk, which we could otherwise measure by looking at the time difference between two consecutive responses, i.e., the inter-arrival time. In our case, the time between two consecutive responses does not necessarily reflect the time spent by the device executing the second request, because those two requests may have been satisfied by different disks.

On the other hand, we know the number of requests executed from each stream on the storage device. If all requests had the same cost, then we could take the average over a time window T, i.e., $e = T/n$, where n is the number of requests completed in T. Clearly, that would not solve

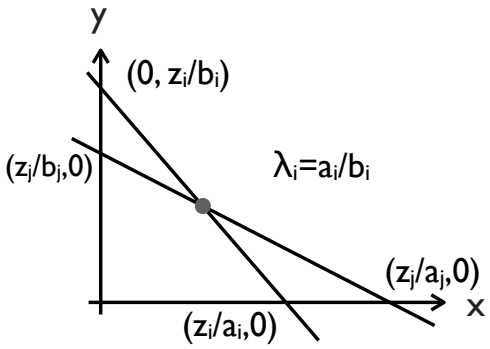

Figure 5: The intersection of the two lines from (3) gives us the average cost x of a sequential and y of a random request in the time windows z_i and z_j.

the problem since random requests are orders-of-magnitude more expensive than sequential requests, i.e., the disk has to spend significantly more time to complete a random request. Based on that observation, for each stream we classify its requests into sequential and random while keeping track of the number of requests completed by type per window. Assuming the clients saturate the device and the cost x of the average sequential request and the cost y of the average random request remain the same across two time windows z_i and z_j we are lead to the following system of linear equations:

$$\begin{cases} \alpha_i x + \beta_i y = z_i \\ \alpha_j x + \beta_j y = z_j, \end{cases} \tag{1}$$

where α_i is the number of sequential requests completed in window i, and similarly for the number of random requests denoted by β_i. Often, j will be equal to $i+1$. Solving the above system gives us the sequential and random average request costs for windows z_i and z_j:

$$x = \frac{z_j\beta_i - \beta_j z_i}{\alpha_j\beta_i - \alpha_i\beta_j}, \quad y = \frac{z_i}{\beta_i} - \frac{\alpha_i}{\beta_i}x. \tag{2}$$

The above equations may give us negative solutions due to system noise and other factors. Since execution costs may only be positive we restrict the solutions to positive (x,y) pairs (Figure 5), i.e., satisfying:

$$\begin{cases} z_i/\alpha_i < z_j/\alpha_j \\ z_i/\beta_i > z_j/\beta_j \\ \alpha_i/\beta_i > \alpha_j/\beta_j \end{cases} \text{ or } \begin{cases} z_i/\alpha_i > z_j/\alpha_j \\ z_i/\beta_i < z_j/\beta_j. \\ \alpha_i/\beta_i < \alpha_j/\beta_j \end{cases} \tag{3}$$

Intuitively, setting z_i equal to z_j in (3) would require that if the number of completed sequential requests goes down in window j, then the number of random requests has to go up (and vice-versa.) Otherwise, the intersection would contain a negative component. By focusing on the case where every time window has the same length we reduce the chances of getting highly volatile solutions and make the analytical solution simpler to intuitively understand. In that case the solution becomes

$$x = \frac{z}{\alpha_i + \beta_i\lambda}, \quad y = \lambda x, \tag{4}$$

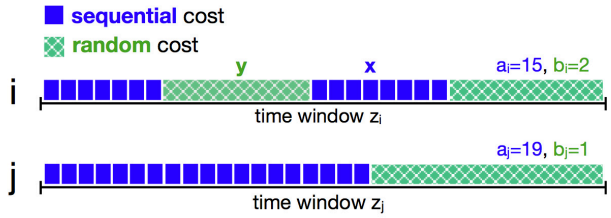

Figure 6: Counting sequential and random completions per window lets us estimate their average cost.

where

$$\lambda = \frac{\alpha_i - \alpha_j}{\beta_j - \beta_i}. \tag{5}$$

From (5) we see that the intersection solutions are expected to be volatile if the window size is small. On the other hand, if the window size is large and the throughput does not change, the intersection will often be negative, i.e., it will happen on a negative quadrant, since the two lines from Figure 5 will often have a similar slope. It would be easy to ignore negative solutions by skipping windows. However, depending on the window size and workload it is possible to get negative solutions more often than positive ones. That leads to fewer updates and therefore a slower convergence to a stable estimate. To face that issue we looked into two directions. One direction is to observe that if the window size is small enough, it is not important whether we take the intersection of the current window with the previous one or some other window not too far in the past. Based on that observation we consider the positive intersections of the current window with a number of the previous ones and take the average. That method increases the chance of getting a valid solution. In addition, updating more frequently allows the moving estimate to converge more quickly without giving a large weight on any of the individual estimates.

The other way we propose to face negative solutions is to compute the projection of the previous estimate on the current window assuming the x/y ratio remains the same along those two windows. In particular, we may assume that α_i/β_i is close to α_j/β_j. In that case, we can project the previous intersection point or estimate on the line describing the second window. The projection is given by

$$x = \frac{\alpha_j z_i}{\alpha_i(\mu\beta_j + \alpha_j)}, \quad y = \mu x, \tag{6}$$

where

$$\mu = \frac{1}{\beta_i}\left(\frac{z_i}{x} - \alpha_i\right). \tag{7}$$

The idea is that if both the number of completed sequential and random requests in a window drops (or increases) proportionally the cost must have shifted accordingly. Although we observed that the projection method works especially well, its correctness depends on the previous estimate. It could still be used when some intersection is invalid to keep updating the estimate but leave it as future work to determine whether it can enhance our estimates.

3.2 Estimation error and seek times

A key assumption is that the request costs are the same among windows. Assuming that at some point we have the

true (x, y) cost and that the cost in the next window is not exactly the same due to system noise we expect to have error. To compute that error we replace z_j in the solution for x in (2) by its definition i.e., $\alpha_j x + \beta_j y$ and denote that expression by x'. Taking the difference between x and x' gives

$$\left|x - x'\right| = \frac{\beta_i}{\left|\alpha_j \beta_i - \alpha_i \beta_j\right|} \left|(\alpha_j x + \beta_j y) - z_j\right| \quad (8)$$

and

$$\left|y - y'\right| = \frac{\alpha_i}{\beta_i} \left|x' - x\right|. \quad (9)$$

So far we have not considered seek times between streams and how they might affect our estimates. In the typical case where m random requests are executed by a disk followed by n sequential requests, the first request out of the sequential ones will incur a seek. That seek is not fully charged to either type of request in our model, simply because it is either hard or impossible in our scenario. Intuitively, the total seek cost of a window is distributed across both request types. Firstly, because fewer requests of both types will end up being executed in that window and secondly due to the error formula (9) for y. In particular, assuming the delayed requests in some window i would also follow the α_i/β_i ratio we now show that seeks do not affect our scheduling.

Let $\alpha'_i = \alpha_i - \delta_i^{(\alpha)}$ and $\beta'_i = \beta_i - \delta_i^{(\beta)}$, where δ_i^{τ} is the number of requests of type τ that are not executed in window i due to seek events. From the above assumption, $\delta_i^{(\beta)} = \beta_i/\alpha_i \delta_i^{(\alpha)}$. Then $\alpha'_i/\beta'_i = (\alpha_i - \delta_i^{(\alpha)})/(\alpha_i - \delta_i^{(\beta)})$, which gives α_i/β_i and similarly, for window j. Using the original solution (2) for the sequential and random costs, consider the ratio of y/x as well as y'/x', which uses α' instead of α and similarly for β. By substituting, we get that $y/x = y'/x' = -\alpha'_i/\beta'_i$, which is independent of the number of seeks δ and by the above is equal to $-\alpha_i/\beta_i$.

From the above, we conclude that seeks do not affect the relative estimation costs and consequently our schedule. The reason the ratios are negative can be seen from Figure 5. Specifically, fixing every variable in (2), while increasing the x-cost, reduces the y-cost and vice versa. Therefore, the slope y/x is negative whether we have seeks or not.

3.3 Write support and estimating in practice

In this work, we only deal with read requests. Since writes typically respond immediately, it is harder to approximate the disk throughput over small time intervals. On the other hand, if a system is busy enough, the write throughput over large intervals (e.g., 5 seconds) is expected to have a smaller variance and be closer to the true throughput. Preliminary results suggest the above holds. There are still some challenges, such as the effect of writes on reads when there is significant write activity, which may be addressed by dispatching writes in groups. Adding support for writes is a priority for future work and is expected to lead into a more general solution supporting SSDs and hybrid systems.

In our implementation we took the approach of having small windows, e.g. 100ms, to increase the frequency of estimates and to give a small weight to each of them. As we compute intersections we keep a moving average and weight each estimate depending on its distance from the previous one. Due to the frequent updates, if there is a shift in the cost, the moving estimate will reach that value quickly. Moreover, to improve estimates, for each window we find its intersection with a number of the previous windows (e.g., 10.) Finally, if the λ cost ratio as defined in (5) is too small or too large we ignore that pair of costs. We set the bounds to what we consider safe values in that they will only take out clearly wrong intersections.

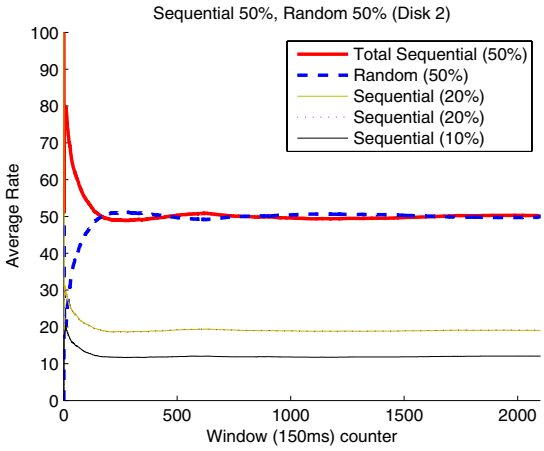

Figure 7: Using one disk and a mixture of sequential and random streams the rates are achieved and convergence happens quickly.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate QBox in terms of utilization and throughput management. We first verify that the sequential and random request cost estimates are accurate enough and that the desired stream rates are satisfied in different scenarios. Next, we show that the throughput achieved is to a large degree in agreement with the target rates of each stream.

4.1 Prototype

In all our experiments we use up to four disks (different models) or a software RAID 0 over two disks. We forward stream requests to the disks asynchronously using Kernel AIO. We avoided using threads in order to keep a large number of requests queued up (e.g., 200) and to avoid race conditions leading to inaccurate inter-arrival time measurements. Up to subsection 4.4 we are interested in evaluating QBox in a time-based manner. For that purpose we avoid hitting the filesystem cache by enabling O_DIRECT and do not use Native Command Queuing (NCQ) in any of the disks. Moreover, we send requests in a RAID 0 fashion rather than using a true RAID. The above allows us to know the disk each request targets, which consequently lets us compute the service times by measuring the inter-arrival times and compare those with our estimates. The extra information is not used by our method since it is normally unavailable. It is used only for evaluation purposes. Starting from subsection 4.4 we gradually remove all the above restrictions and evaluate QBox implicitly in a throughput-based manner.

We evaluate QBox both with synthetic and real workloads depending on the goal of the experiment. All synthetic requests are reads of size 4KB unless we are using a RAID over two disks in which case they are 8KB. For the synthetic workload, each disk contains a hundred 1GB files. We use a subset of the Deasna2 [3] NFS trace with request sizes typ-

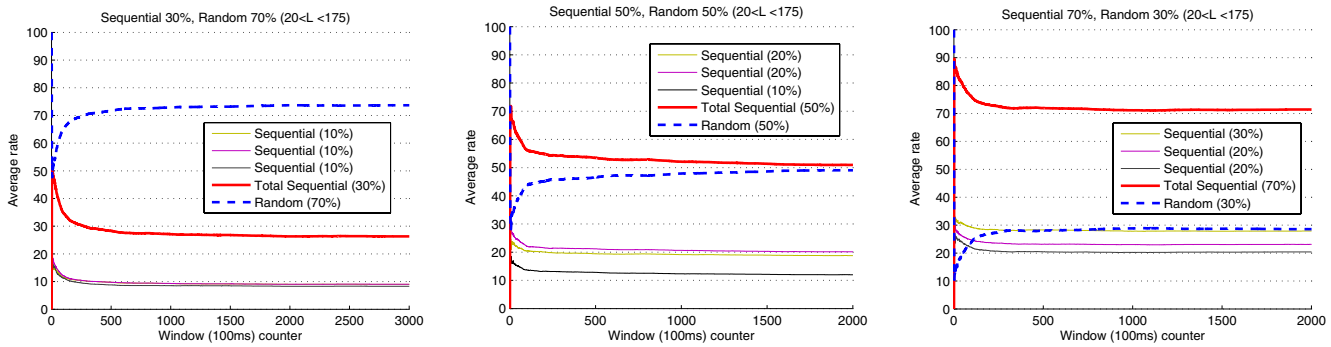

Figure 8: Using two disks and our scheduling and estimation method we achieve the desired rates most of the time relatively well. In the above we have three sequential streams and a random one.

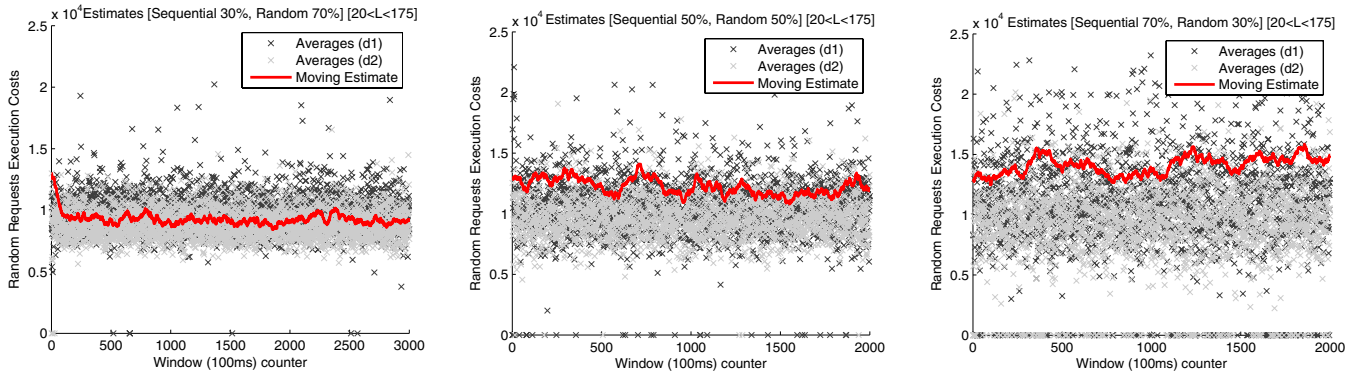

Figure 9: Using two disks (d1, d2) and our estimation method we maintain a moving estimate of the average random execution cost on the storage device.

ically being 32KB or 64KB. Finally, except for a workload containing idle time, we assume there are always requests queued up, since that is the most interesting scenario, and we bound the number of pending requests on the storage by a constant, e.g., 200. Finally,we do not assume a specific I/O scheduler is used by the storage device. In our experiments, the "Deadline Scheduler" was used, however, we have tried other schedulers and observed similar results.

4.2 Sequential and random streams

Our approach is based on the differentiation between sequential and random requests and so the first step in evaluating QBox is to consider a workload of fully sequential and random streams with the goal of providing isolation between them. Note that to provide isolation a prerequisite is that our cost estimates for the average sequential and random request are close enough to the true values, which are not known, and it is not possible to explicitly measure them in our black box scenario. In this set of experiments, the workload consists of three sequential streams and one random. Each sequential stream starts at a different file to ensure there are inter-stream seeks. Each request of the random stream targets a file and offset uniformly at random. For each stream we measure the average utilization provided by the storage device. We look into three sets of desired utilizations. Figure 8(a) shows a random stream with a utilization target of 70%, while each sequential stream has a target of 10% for a total of 30%. As the experiment runs, the cost estimates take values within a small range and the average achieved utilization converges. In Figures 8(b) and (c) the sequential streams are given higher utilizations. In all three cases, the achieved utilizations are close to the desired ones. Again in Figures 8(b) and (c) the initial estimate was relatively close to the actual cost, so the moving rates approach the converging rates more quickly.

From Figure 9 we notice that estimates get above the average cost when there is many sequential requests even if the utilization targets are achieved (Figure 8). The main reason for that is that we keep track and store (in memory) large amounts of otherwise unnecessary statistics per request. Therefore, if in a window of e.g., 100ms there is a very large number of request completions, i.e., when the rate of sequential streams is high, 10ms ($10\mu s \cdot 1000$ requests) may be given to that processing and therefore the estimates are scaled up. Of course, those operations can be optimized or eliminated without affecting QBox. As expected, a similar effect happens with the estimated cost of sequential requests (not shown), therefore the ratio of the costs stays valid leading to proper scheduling as shown in Figure 8.

Besides the initial estimates, the convergence rate also depends on the window size, since a smaller size implies more frequent updates and faster convergence. However, if the window size becomes too small the number of completed requests become too few and the quality of the estimate may not be accurate enough due to the significant noise. Note that whether the window size is considered too small

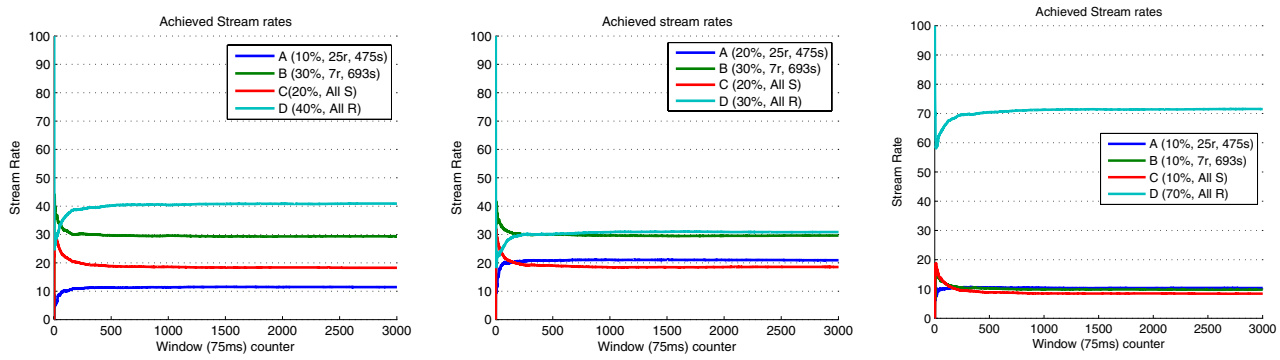

Figure 10: Using two disks and our scheduling and estimation method we achieve the desired rates most of the time relatively well. Stream A requires 20% of the disk time and sends 25 random requests every 475 sequential requests. Similarly for the rest of the streams.

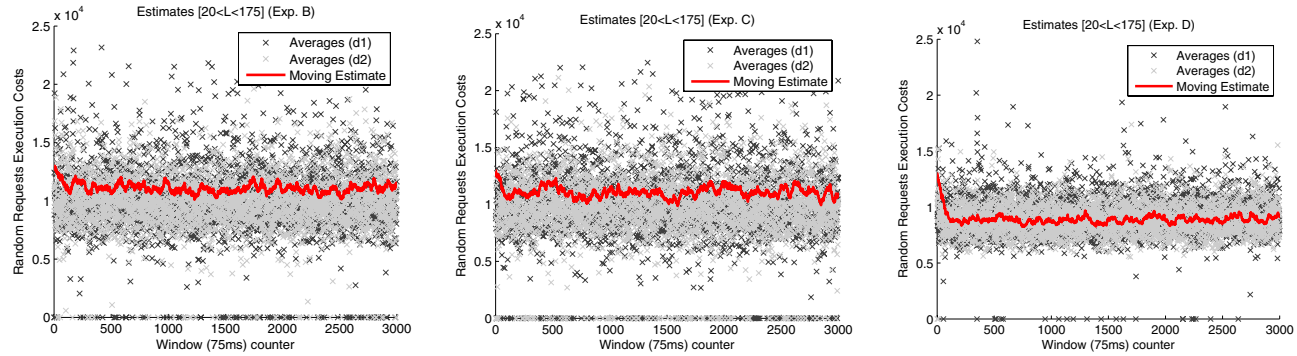

Figure 11: Using two disks (d1, d2) and our estimation method we maintain a moving estimate of the average random execution cost on the storage device.

depends on the number of disks in the storage system as having more disks implies that a greater number of requests complete per window. The window size we picked in the above experiment (Figure 8) is 100ms. Other values such as 150ms provide similar estimation quality and later we look at smaller windows of 75ms. Note that in Figure 9 there is a number of recorded averages that are 0 because random streams with low target rates are more likely to have no arrivals in a window. Not having any completed random requests in a window implies that we can estimate a new sequential estimate more easily.

Finally, in the above, we assume there is always enough queued requests from all streams. Without any modification to our method we see from Figure 12 that under idle time it is still possible to manage the rates. In particular, every 5000 requests (on average) dispatched to the storage device we delay dispatching the next request(s) for a (uniformly at) random amount of time between 0.5 to 1 second. From Figure 12 we see that the rates are still achieved, while there is slightly more noise in the estimates compared to Figure 9. We noted that if the idle times are larger than the window size, then our method is less affected. That was expected, since idling over a number of consecutive windows implies that new requests will be scheduled according to the previous estimates as the estimates will not be updated. Finally, although the start and end of the idle time window may affect the estimate, the effect is not significant since the estimate moves only by a small amount on each update and most updates are not affected.

4.3 Mixed-workload streams

In practice, most streams are not perfectly sequential. For example, a stream of requests may consist of m random requests for every n sequential requests, where m is often significantly smaller than n. To face that issue, instead of characterizing each stream as either sequential or random we classify each request. Note that the first request of a sequential group of requests after m random ones is considered random if m is large enough. Although not all random requests cost exactly the same, we do not differentiate between them since we work on top of the filesystem and do not assume we have access to the logical block number of each file. Therefore, we do not have a real measure of sequentiality for any two I/O requests. However, as long as the cost of random requests does not vary significantly between streams we expect to achieve the desired utilization for each stream. Indeed, as it has been observed in [10], good utilization management can still be provided when random requests are assumed to cost the same. Moreover, from [3] we see that requests from common workloads are usually either almost sequential or fully random. Differentiating between cost estimates on a per stream basis is expected to improve the management quality and leave it as future work.

From Figure 10 we see that the targets are achieved in the

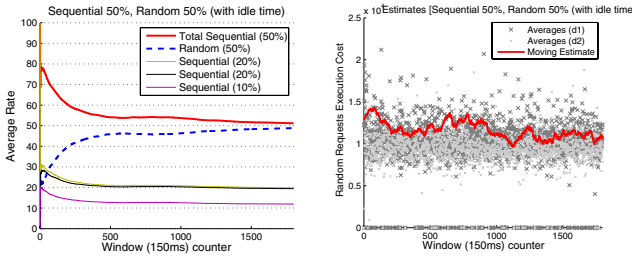

Figure 12: Using two disks (d1, d2), the desired rates are achieved well enough (reach 45% quickly) even when there is idle time in the workload.

presence of semi-sequential streams. In particular, in 10(a), stream A sends 25 random requests for every 475 sequential ones. Stream B sends 7 random requests for every 693 sequential ones, while streams C and D are purely sequential and random, respectively. Other target sets in Figure 10 are satisfied equally well. Note that each group of requests does not have to be completed before the next one is sent. Instead, requests are continuously dequeued and scheduled.

So far we have seen scenarios with fixed target rates. Our method supports changing the target rates online as long as the rate sum is up to 100%. Depending on the new target rates, the cost estimation updates can be crucial in achieving those rates. For example, increasing the rate of a random stream decreases the average cost of a random request and our estimates are adjusted automatically to reflect that. Figure 13(a) illustrates that the utilization rates are satisfied and Figure 13(b) shows how the random estimate changes as the clients adjust their desired utilization rates every thirty seconds. For this experiment we set the number of disks to four to illustrate our method works with a higher number of disks and to support our claim that it can work with any number of disks. The same experiment was run with two disks giving nearly identical results (figure omitted.)

As explained earlier, the disk queue depth is set to one for evaluation purposes. However, since a large queue depth can improve the disk throughput we implicitly evaluate QBox by comparing the throughput achieved when the depth is 1 and 31, while the target rates change. In particular, we look at semi-sequential and random streams. As expected and illustrated in Figure 15, having a depth of 31 achieves a higher throughput over a range of rates. Although, this does not verify our method works perfectly due to lack of information, it provides evidence that it works and, as we will see in the next subsection, that is indeed the case.

4.4 RAID utilization management

In our experiments so far, we have been sending requests to disks manually in a striping fashion instead of using an actual RAID device. That was done for evaluation purposes. Here, we use a (software) RAID 0 device and instead evaluate QBox indirectly. The RAID configuration consists of two disks with a chunk size of 4KB to match our previous experiments, while requests have a size of 8KB.

In the first experiment we focus on the throughput achieved by two (semi-)sequential streams as we vary their desired rates. Moreover, we add a random stream to make it more realistic and challenging. We fix the target rate of the random stream since otherwise it would have a variable effect on

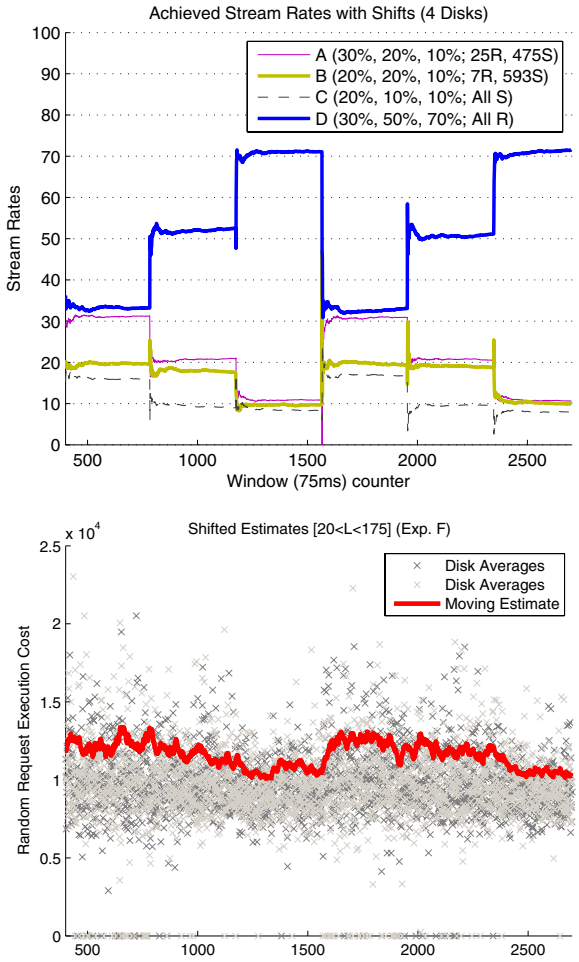

Figure 13: Using four disks and desired rates that shift over time, the rates are still achieved quickly under semi-sequential and random workloads.

the sequential streams throughput and make the evaluation uncertain. As long as the throughput achieved by each of the sequential streams varies in a linear fashion we are able to conclude that our method works. Indeed, from Figure 14 stream A starts with a target rate of 0.5 and goes down to 0, while stream B moves in the opposite direction. As the throughput of stream A goes down, the difference is provided to stream B. Moreover, in Figure 16 we see that having two random streams and a sequential one fixed at 50% (not plotted) has a similar behavior. The difference between those two cases is the drop in the total throughput of the first case with streams A and B having a lower throughput when their rates get closer to each other. That is due to the more balanced number of requests being executed from each sequential stream leading to a greater number of seeks between them. Since seeks are relatively expensive compared to the typical sequential request the overall throughput drops slightly. If that effect was not observed in Figure 14, then the random stream (C) would be getting a smaller amount of the storage time, which would go against its performance targets. Instead, the random stream throughput remains unchanged. On the other hand, in Figure 16 there

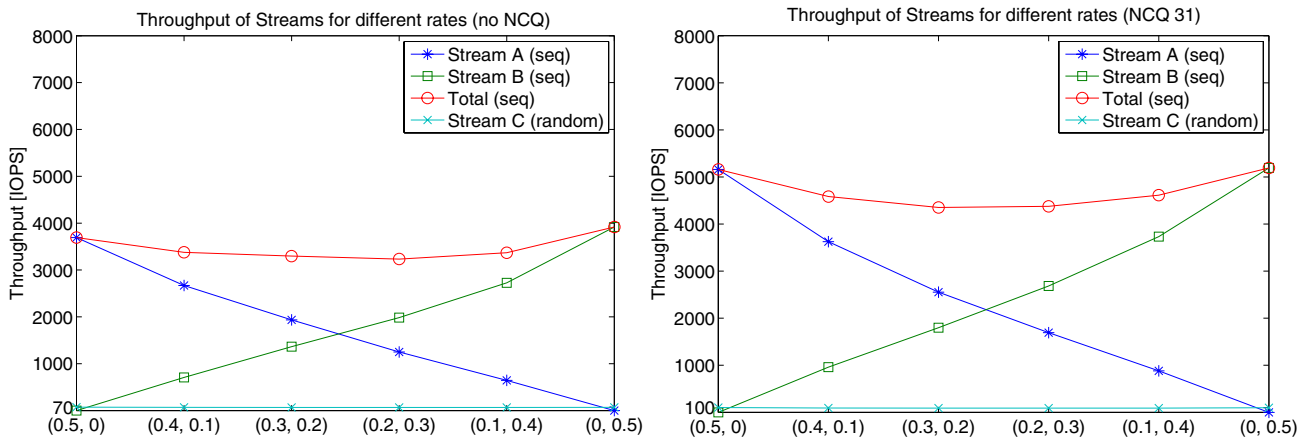

Figure 14: Using RAID 0 the throughput achieved by each sequential stream is in agreement with their target rates. Stream A has a varied target rate from 50% to 0 and the opposite for B. Random stream C requires a fixed rate of 50% of the storage time. Similarly for a large disk queue depth (NCQ.)

is no drop in the total throughput, which is expected since the cost of seeks between random requests are similar to the typical cost of a random request. Therefore, the total throughput remains constant. Moreover, the sequential stream (not plotted) reaches an average throughput of 3600 and 5060 IOPS with a depth of 1 and 31, respectively as in Figure 14. Finally, note that whether we use no NCQ or a depth of 31 the throughput behavior is similar in both Figures (14 and 16), which is desired since a large depth can provide a higher throughput in certain cases [26], along with other benefits such as reducing power consumption [24].

4.5 Evaluation using traces

To strengthen our evaluation, besides synthetic workloads we run QBox using two different days of the Deasna2 [3] trace as two of the three read streams, while the third stream sends random requests. Deasna2 contains semi-sequential traces of email and workloads from Harvard's division of engineering and applied sciences. As the requests wait to be dispatched, we classify them as either sequential or random depending on the other requests in their queue.

Unlike time, evaluating a method by comparing throughput values is hard because the achieved throughput depends on the stream workloads. However, by looking at the throughput achieved using QBox in Figure 17 and the results of throughput-based scheduling in Figure 18 it is easy to conclude that QBox provides a significantly higher degree of isolation and that the target rates of the streams are respected well enough. Moreover, looking more closely at Figure 17, we see that wherever the throughput is not in perfect accordance with the targets of streams A and B, there is an increase of random requests coming from the same streams. That effect is valid and due to the trace itself. On the other hand, Figure 18 demonstrates the destructive interference inherent in throughput-based reservation schemes with semi-sequential streams receiving a very low throughput.

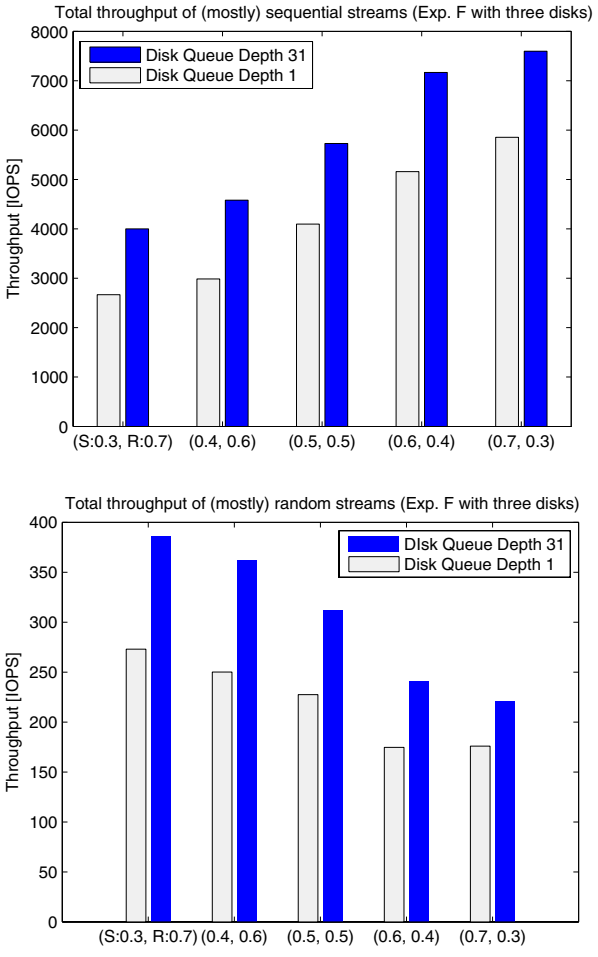

Figure 15: The throughput with an NCQ of 1 and 31 is maintained while the desired rates vary.

4.6 Caches

So far our experiments have skipped the file system cache to more easily evaluate our method and to send requests asynchronously, since without O_DIRECT they become block-

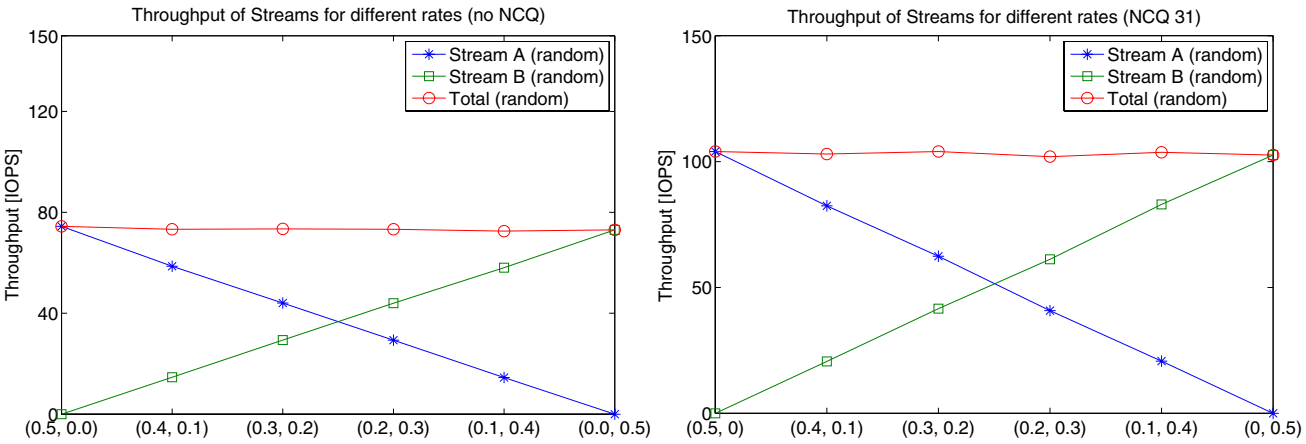

Figure 16: Using RAID 0 the throughput of each random stream is in agreement with its target. Stream A has a varied target rate from 50% to 0 and B from 0 to 50%. The sequential stream (not plotted) has a utilization of 50% leading to an average of 3600 and 5060 IOPS with no NCQ and a depth of 31, respectively.

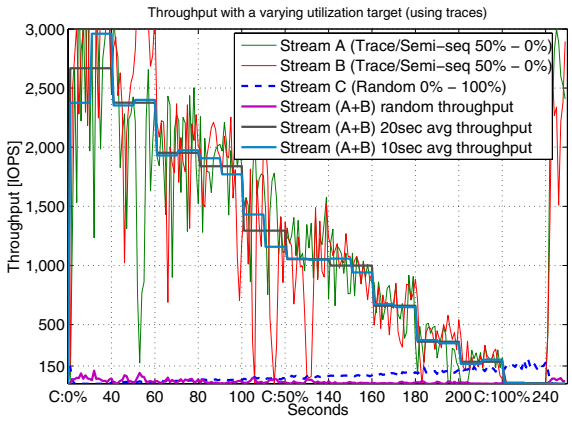

Figure 17: QBox provides streams of real traces the throughput corresponding to their rate close enough even in the presence of a random stream.

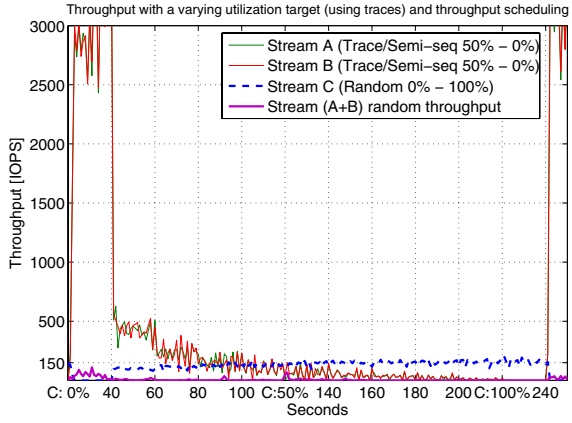

Figure 18: Random stream C affects throughput-scheduling leading to a low throughput for A and B.

ing requests. Although applications such as databases may avoid file system caches, we are interested in QBox being applicable in a general setting. For our purposes, request completions resulting from cache hits could be ignored or accounted differently. From our experiments, detection of random cache hits seems reliable and the well-known relation–as explained in [7]–between the queue size and the average latency may also be useful to improve accuracy as well as grey-box methods [1]. However, we cannot say the same for sequential requests due to prefetching. Moreover, since random workloads may cover a large segment of the storage, hits are not as likely. Hence, in this paper we treat hits as regular completions for simplicity.

Without modifying QBox we enable the file system cache and see from Figure 19 that although the throughput is noisier than in the previous experiments due to the nature of cache hits, we still manage to achieve throughput rates that are in accordance with the target rates. Scheduling based on throughput (Figure 20) gives similar results to Figure 18, supporting our position on throughput-based scheduling. Finally, using synthetic workloads we get an output of the same form as Figure 14 with a maximum sequential throughput of 1700 IOPS (figure omitted.)

4.7 Overhead

The computational overhead is trivial. We know the most urgent request in each stream queue and thus picking the next request to dispatch requires as many operations as the number of streams. Since the number of streams is expected to be low, that cost is trivial. In addition, on a request completion we increase a fixed number of counters and at the end of each window we compute a fixed, small number of intersections. The time it takes to compute each intersection is insignificant. Finally, updating the moving estimate only requires computing the new estimate weight. In total the procedure at the end of each window takes less than 10μs.

5. RELATED WORK

A large body of literature exists related to providing guarantees over storage devices. Typically they either aim to

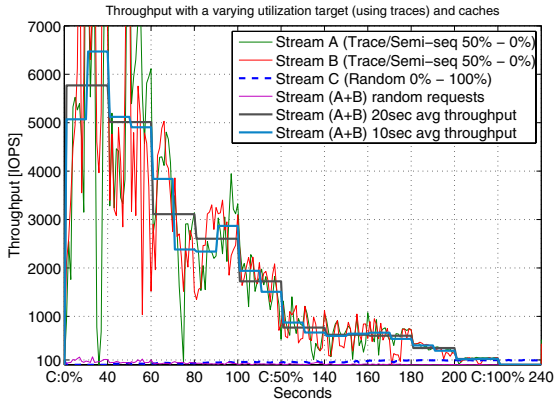

Figure 19: The throughput achieved by QBox in the presence of caches follows the target rates.

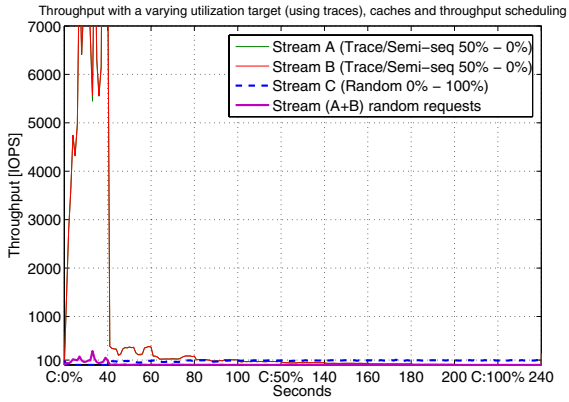

Figure 20: Throughput-based scheduling fails to isolate stream performance leading to a low throughput for semi-sequential streams.

satisfy throughput or latency requirements, or attempt to proportionally distribute throughput. Most solutions do not distinguish between sequential and random workloads, which leads to the storage being under-utilized. Avoiding that distinction leads to charging semi-sequential streams unfairly due to the significant cost difference between sequential and random requests. Instead, QBox uses disk service time rather than IOPS or Bytes/s to solve that problem.

Stonehenge [8] clusters storage systems in terms of bandwidth, capacity and latency, however, being based on bandwidth its reservations cover only a fraction of the disk performance. Other proposed solutions based on bandwidth, include [11, 15, 2] and take advantage of the relation - as was later explained in [7] - between the queue length and average latency to throttle requests. mClock [6] does not provide performance insulation, while both [6] and pClock [5] do not differentiate between sequential and random requests. Other solutions such as [9, 11, 15, 16] do not provide insulation either. On the other hand, Argon [22] provides insulation, however, workload changes may affect its provided soft bounds. In [14], distribution-based QoS is provided to a percentage of the workload to avoid over-provisioning. [12] attempts to predict response times rather than service times through statistical models. PARDA [4] provides guarantees by assuming a specific scheduler resides on each host, unlike QBox, which does not assume access to the hosts/clients.

Facade [15] aims to provide performance guarantees described by an SLA for each virtual storage device. It places a virtual store controller between a set of hosts and storage devices in a network and throttles client requests so that the devices do not saturate. In particular, it adjusts the queue size dynamically, which affects the latency of each workload. However, a single set of low latency requests may decrease the queue size of the system and it is hard to determine whether a new workload may be admitted.

YouChoose [27] tries to provide the performance of reference storage systems by measuring their performance off-line and mimicking it online. It is based on an off-line machine learning process, similar to [23], which can be hard to prepare due to the challenging task of selecting a representative set of training data. Moreover, the safe admission of new virtual storage devices can be challenging.

Solutions based on execution time estimates such as [18, 17, 10] assume we have low-level control over each hard-drive. Moreover, in Horizon [18] it was shown that such a solution can be used in distributed storage systems with single-disk nodes using the Horizon scheduler. Our work is also based on disk-time utilization and deadline assignment, however, we treat the storage device as a black box and therefore do not assume our own scheduler is in front of every hard-drive. Finally, [25, 20] reserve I/O rates using worst-case execution times, therefore, they can only reserve a fraction of the storage device time.

6. CONCLUSIONS

In this paper, we targeted the problem of providing isolation and performance guarantees in terms of storage device utilization to multiple clients with different types of workloads. We proposed a "plug-n-play" method for isolating the performance of clients accessing a single file-level storage device treated as a black box. Our solution is based on a novel method for estimating the expected execution times of sequential and random requests as well as on assigning deadlines and scheduling requests using the Earliest Deadline First (EDF) scheduling algorithm. Our experiments show that QBox provides isolation between streams having different characteristics with changing needs and on storage systems with a variable number of disks.

There are multiple directions for future work. Extensions include support for SSDs based on the cost difference of reads and writes as well as hybrid systems. Adding support for writes and RAID 4, 5 is another direction. Technical improvements include a better use of the history of requests in computing estimates and sophisticated methods to detect sudden and stable changes. Finally, we would like to verify QBox works on Network Attached Storage, test it at the hypervisor level in a virtualized environment and explore the case where there is multiple controllers and storage devices.

7. REFERENCES

[1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 43–56, New York, NY, USA, 2001. ACM.

[2] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *In Proceedings of the 22th International Symposium on Reliable Distributed Systems (SRDSÕ03*, pages 109–118, 2003.

[3] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive nfs tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 203–216, Berkeley, CA, USA, 2003. USENIX Association.

[4] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *Proccedings of the 7th conference on File and storage technologies*, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.

[5] A. Gulati, A. Merchant, and P. J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 13–24, New York, NY, USA, 2007. ACM.

[6] A. Gulati, A. Merchant, and P. J. Varman. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[7] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 19:1–19:14, New York, NY, USA, 2011. ACM.

[8] L. Huang, G. Peng, and T.-c. Chiueh. Multi-dimensional storage virtualization. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 14–24, New York, NY, USA, 2004. ACM.

[9] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Perform. Eval. Rev.*, 32:37–48, June 2004.

[10] T. Kaldewey, T. Wong, R. Golding, A. Povzner, S. Brand, and C. Maltzahn. Virtualizing disk performance. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 319 –330, April 2008.

[11] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1:457–480, November 2005.

[12] T. Kelly, I. Cohen, M. Goldszmidt, and K. Keeton. Inducing models of black-box storage arrays. *In Technical Report HPL-SSP-2004-108*, 2004.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[14] L. Lu, P. Varman, and K. Doshi. Graduated qos by decomposing bursts: Don't let the tail wag your server. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 12 –21, June 2009.

[15] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 131–144, Berkeley, CA, USA, 2003. USENIX Association.

[16] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: quality-of-service in large disk arrays. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 245–254, New York, NY, USA, 2011. ACM.

[17] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 13–25, New York, NY, USA, 2008. ACM.

[18] A. Povzner, D. Sawyer, and S. Brandt. Horizon: efficient deadline-driven disk i/o management for distributed storage systems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 1–12, New York, NY, USA, 2010. ACM.

[19] A. S. Povzner. *Efficient guaranteed disk i/o performance management*. PhD thesis, University of California at Santa Cruz, Santa Cruz, CA, USA, 2010. AAI3429522.

[20] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (das). In *In Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003). IEEE*, page 374. IEEE Computer Society, 2003.

[21] M. Spuri, G. Buttazzo, and S. S. S. Anna. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10:179–210, 1996.

[22] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.

[23] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. *SIGMETRICS Perform. Eval. Rev.*, 32:412–413, June 2004.

[24] Y. Wang. Ncq for power efficiency. *White paper*, February 2006.

[25] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-szendy. Zygaria: storage performance as a managed resource. In *In IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 06*, pages 125–134, 2006.

[26] Y. J. Yu, D. I. Shin, H. Eom, and H. Y. Yeom. Ncq vs. i/o scheduler: Preventing unexpected misbehaviors. *Trans. Storage*, 6:2:1–2:37, April 2010.

[27] X. Zhang, Y. Xu, and S. Jiang. Youchoose: Choosing your storage device as a performance interface to consolidated i/o service. *Trans. Storage*, 7:9:1–9:18, October 2011.