UNIVERSITY OF CALIFORNIA
SANTA CRUZ

PROVIDING PREDICTABLE PERFORMANCE IN FLASH AND BLACK-BOX STORAGE

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Dimitris Skourtis

September 2014

The Dissertation of Dimitris Skourtis
is approved:

Professor Scott Brandt, Co-Chair

Professor Carlos Maltzahn, Co-Chair

Professor Dimitris Achlioptas

Professor Ike Nassi

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Providing Predictable Performance in Flash and Black-box Storage

by

Dimitris Skourtis

Many storage systems are shared by multiple clients with different types of workloads and performance targets. To achieve performance targets without over-provisioning, a storage 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.

At the same time, virtualization and many other applications such as online analytics and transaction processing often require access to predictable, low-latency storage. Hard-drives have low and unpredictable performance under random workloads, while keeping everything in DRAM, in many cases, is still prohibitively expensive or unnecessary. Solid-state drives offer a balance between performance and cost, and are becoming increasingly popular in storage systems, playing the role of large caches and permanent storage. Although their read performance is high and predictable, SSDs frequently block in the presence of writes, due to internal operations such as garbage collection, exceeding hard-drive latency and leading to unpredictable performance. Many systems with read/write workloads have low latency requirements or require predictable performance

and guarantees. In such cases the performance variance of SSDs becomes a problem for both predictability and raw performance.

First, we present QBox, a new utilization-based approach for 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 our method provides good isolation and achieves the target utilizations of its clients.

Second, we present Rails, a flash storage system based on redundancy, which provides predictable performance and low latency for reads under read/write workloads by physically separating reads from writes. More specifically, reads achieve read-only performance while writes perform at least as well as before. We evaluate our design using micro-benchmarks and real traces, illustrating the performance benefits of Rails and read/write separation in flash.

Compared to hard-drives, flash is a more expensive option in terms of raw storage space. We present eRails, a scalable flash storage system on top of Rails that achieves read/write separation using erasure coding without the storage cost of replication. To support an arbitrary number of drives efficiently we describe a design allowing us to scale eRails by constructing overlapping erasure coding groups that preserve read/write separation. Through benchmarks we demonstrate that eRails achieves read/write separation and consistent read performance under read/write workloads.

Finally, we demonstrate that the guaranteeable performance in SSDs is low without consistent performance. Using Rails and time-based scheduling similar to QBox we

demonstrate that we can achieve performance guarantees that are close to optimal.

Acknowledgments

I would like to thank my collaborators Dimitris Achlioptas, Scott Brandt, Shinpei Kato, Carlos Maltzahn, and Noah Watkins.

Chapter 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 in disk-based storage 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.

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 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 [ADAD01], however, we require fewer algorithmic assumptions about the underlying system. In the first part of this dissertation 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 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 [PKB+08]. Our utilization-based approach can work with Service Level Agreements (SLA); requirements can be converted to utilization as demonstrated in [PSB10] 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 [LMA03, KKZ05, MUP+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. Without very specific knowledge about the workloads, the system must make worst-case assumptions, leading to extremely low reservable throughput. Moreover, those approaches do not provide workload isolation. A number of schedulers [GAW09, GMV07, GMV10, HPC04, WGLBs06] provide relative rather than absolute throughput guarantees by proportionally allocating the total system throughput, which is workload dependent. On the other hand, providing latency guarantees [LMA03, KKZ05, CAP+03]

by throttling requests can lead to low throughput. Other approaches such as Maestro [MUP+11] provide both throughput and latency guarantees without isolation, while Argon [WAEMTG07] does provide insulation but assumes low-level access. Moreover, throughput-based solutions create other challenges such as admission control. On the other hand, existing solutions based on disk utilization [KWG+08, PKB+08, PSB10] only support single drives and if used in a clustered storage system they require their scheduler to be present on every node.

Our novel method, QBox, manages 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 [LL73, SBA96]. We use EDF because it is an optimal algorithm for preemptive tasks in the sense that if there is a schedule satisfying the deadline requirements of a task-set, EDF will produce such a schedule. In particular, given the execution costs e_i and deadlines d_i, EDF will produce a schedule that satisfies all deadlines as long as the total required utilization is bounded by 100%, i.e., $\sum_i e_i/d_i \leq 1$. 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.

Virtualization and many other applications such as online analytics and transaction processing often require access to predictable, low-latency storage. Cost-effectively satisfying such performance requirements is hard due to the low performance of hard-drives for random workloads, while storing all data in DRAM, in many cases, is still prohibitively expensive and often unnecessary. In addition, offering high performance storage in a virtualized cloud environment is more challenging due to the loss of predictability, throughput, and latency incurred by mixed workloads in a shared storage system. This is because most, if not all, storage systems come without the ability to provide isolation and absolute guarantees. Given the popularity of cloud systems and virtualization, and the storage performance demands of modern applications, there is a clear need for scalable storage systems that provide high and predictable performance efficiently, under any mixture of workloads.

Solid-state drives and more generally flash memory have become an important component of many enterprise storage systems towards the goal of improving performance and predictability. They are commonly used as large caches and as permanent storage, often on top of hard-drives operating as long-term storage. A main advantage

over hard-drives is their fast random access. One would like SSDs to be the answer to predictability, throughput, latency, and performance isolation for consolidated storage in cloud environments. Unfortunately, though, SSD performance is heavily workload dependent. Depending on the drive and the workload latencies as high as 100ms can occur frequently (for both writes and reads) due to internal operations such as garbage collection, making SSDs multiple times slower than hard-drives in such cases. In particular, we could only find a single SSD with predictable performance which, however, is multiple times more expensive than commodity drives, possibly due to additional hardware it employs (e.g., extra DRAM).

Such read-write interference results in unpredictable performance and creates significant challenges, especially in consolidated environments, where different types of workloads are mixed and clients require high throughput and low latency consistently, often in the form of reservations. Similar behavior has been observed in previous work [CKZ09, CLZ11, MKC+12, PS12] for various device models and is well-known in the industry. Even so, most SSDs continue to exhibit unpredictability.

Although there is a continuing spread of solid-state drives in storage systems, research on providing efficient and predictable performance for SSDs is limited. In particular, most related work focuses on performance characteristics [CKZ09, CLZ11, BJB09], while other work, including [APW+08, BD10, Des12] is related to topics on the design of drives, such as wear-leveling, parallelism and the Flash Translation Layer (FTL). Instead, we use such performance observations to provide consistent performance. With regards to scheduling, FIOS [PS12] provides fair-sharing while trying to improve the

drive efficiency. To mitigate performance variance, current flash-based solutions for the enterprise are often aggressively over provisioned, costing many times more than commodity solid-state drives, or offer lower write throughput. Given the fast spread of SSDs, we believe that providing predictable performance and low latency efficiently is important for many systems.

In the second part of this dissertation we target the following problem: *given a set of clients and a set of flash storage devices, our goal is to provide consistent performance efficiently for the read operations of each client while performing at least as well for writes as before, independently of the client workloads and without having internal control of the flash storage devices, or requiring device modifications.* To that end, we contribute and evaluate a method for flash storage that achieves consistent performance and low latency under arbitrary read/write workloads by exploiting redundancy. Specifically, using our method read requests experience read-only throughput and latency, while write requests experience performance that is at least as well as before. To achieve this we physically separate reads from writes by placing the drives on a ring and using redundancy, e.g., replication. On this ring consider a sliding window, whose size depends on the desired data redundancy and read-to-write throughput ratio. The window moves along the ring one location at a time at a constant speed, transitioning between successive locations "instantaneously". Drives inside the sliding window do not perform any writes, hence bringing read-latency to read-only levels. All write requests received while inside the window are stored in memory (local cache/DRAM) and optionally to a log, and are actually written to the drive while outside the window.

Ultimately, we want a scalable flash storage system that provides read/write separation without the storage cost of replication. To that end, we consider the applicability of erasure coding in Rails, in a new system called eRails. As in Rails, we maintain a subset of read drives, however, to perform a read in eRails we read a chunk from each read drive and perform a decoding operation. We consider the effects of computation due to encoding/decoding on the raw performance, as well as its effect on performance consistency. We demonstrate that up to a certain number of drives the performance remains unaffected while the computation cost remains modest. After that point, the computational cost grows quickly due to coding itself making further scaling inefficient. To support an arbitrary number of drives we present a design allowing us to scale eRails by constructing overlapping erasure coding groups that preserve read/write separation. Through benchmarks we demonstrate that eRails achieves read/write separation and consistent read performance under read/write workloads. Finally, we discuss how eRails can be applied to distributed storage systems. In particular, we observe that many - if not all - distributed storage systems already employ a form of redundancy across nodes, such as replication or erasure coding. We show how this *existing* redundancy can be exploited to separate reads from writes across the cluster nodes, thus presenting each drive with either read-only or write-only load for the vast majority of time. This allows the system to take full advantage of the SSDs' capabilities of fast and predictable read operations without requiring additional drives and independently of the workload nature.

Lastly, we return to the problem of providing performance guarantees. We demonstrate

that performance guarantees for solid-state drives are low due to the variance of reads and writes (due to writes). Using Rails to separate reads from writes, and time-based scheduling we illustrate through experiments that providing high performance guarantees to flash storage is possible.

1.1 Contributions

The primary contributions of this dissertation are as follows:

1. QBox, a scheduler that manages 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. A version of QBox appeared in HPDC'12 [SKB12]. (Chapter 3)

2. Flash on Rails, a method for enabling consistent performance in flash storage by physically separating reads from writes through redundancy. Rails provides predictable performance and low latency for reads under read/write workloads. More specifically, reads achieve read-only performance while writes perform at least as well as before. We evaluate our design using micro-benchmarks and real traces, illustrating the performance benefits of Rails and read/write separation in solid-state drives. A version of Rails appeared in USENIX ATC'14 [SAW+14]. (Chapter 4)

3. eRails, a scalable flash storage system that provides read/write separation through erasure coding to provide reliability without the storage cost of replication. We consider the effects of computation due to encoding/decoding on the raw performance, as well as its effect on performance consistency. To support an arbitrary number of drives we present a design allowing us to scale eRails by constructing overlapping erasure coding groups that preserve read/write separation. Through benchmarks we demonstrate that eRails achieves read/write separation and consistent read performance under read/write workloads. (Chapter 5)

4. Finally, we demonstrate that performance guarantees on a single SSD are low due to the variance of writes. By applying time-based scheduling, similar to that used in QBox, on top of Rails we show that providing high performance guarantees becomes possible. (Chapter 6)

Chapter 2

Background and Related Work

This chapter presents related work on performance management for disk I/O and flash storage. First, Section 2.1 covers work on performance guarantees for disk or disk-based storage systems. We present different approaches for providing guarantees, in particular, proportional throughput allocation, latency management, and finally, absolute guarantees and performance isolation. In this work we are interested in the last approach. In Section 2.2 we describe work on flash storage. In particular, we first present work on performance characteristics of flash and approaches for scheduling. Finally, we look into work on erasure coding for storage.

2.1 Performance Guarantees

A large body of literature exists related to providing guarantees over storage devices. Typically they either aim to satisfy throughput or latency requirements, or attempt to proportionally distribute throughput. Most solutions do not distinguish between sequen-

tial 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, our method uses disk service time rather than IOPS or Bytes/s to solve that problem. Finally, most work does not address isolation of workloads. Exceptions are time-based approaches: [KWG+08], Fahrrad [PKB+08] and Horizon [PSB10]. Moreover, Argon [WAEMTG07] aims to provide insulation by sending sequential requests in batches. In the rest of this section we briefly describe some of the work that follows each of the main approaches for storage performance management.

2.1.1 Proportional Throughput Allocation

A number of schedulers [GAW09, GMV07, GMV10, HPC04, WGLBs06] allocate throughput proportionally so that relative guarantees are achieved. However, providing relative guarantees leads to the allocation of a proportion of the total throughput to each client, which is highly variable and workload dependent. Therefore, neither absolute throughput guarantees nor workload isolation is generally provided by such approaches.

Zygaria [WGLBs06] allows applications to specify a reserve and limit for their average I/O target. Each application is guaranteed to execute I/Os at a rate up to their reserve and any unused resources are shared fairly across all applications with each application being offered a throughput up to its limit. Moreover, Zygaria groups sequential requests up to a fixed size of 32KB and uses worst-case execution times to perform its scheduling, which can lead to low reservations. On the other hand, our approach does not schedule

requests according to their worst-case execution times and does not limit the total size of sequential requests sent to the storage at a time because we charge clients by time rather than by counting I/Os.

PARDA [GAW09] provides efficient proportional-share fairness among hosts (containing multiple Virtual Machines) accessing a storage array. It is based on latency measurements on a per host level, based on which it adjusts the queue depth of each host. Although the storage is treated as a black-box, PARDA follows an almost decentralized approach and assumes access to each of the hosts. In particular, each host schedules its requests locally in a fair manner, however, there can still be communication between the hosts to compute the aggregate latency periodically. Finally, as in other approaches such as [GMV07, WAEMTG07], it groups IOs that exhibit spatial locality to improve overall IO performance. Although, PARDA cannot be directly compared to our approach since we are interested in the centralized scenario with no control over the hosts, firstly, each host has the same queue size for all its VMs and secondly it is still based on throughput scheduling and can thus lead to low reservations.

pClock [GMV07] uses a service curve for each stream, describing the maximum and average I/Os per second as well as the maximum latency of requests. Based on the service curves, it computes deadlines and dispatches requests in an Earliest Deadline First manner. pClock manages the latencies as long as the workloads adhere to their I/O rates and burst size. However, due to workload variations, the pre-defined service curves may lead to a low reservable throughput. The unreserved capacity is split to the streams or offered to a background stream (e.g. back-up.) Moreover, pClock does not

take full advantage of the differences between sequential and random streams and may only send sequential requests in batches for background jobs if there is spare capacity.

Stonehenge [HPC04] clusters storage systems in terms of bandwidth, capacity and latency. It allows applications to reserve a rate of the total bandwidth and admits workloads depending on the current system bandwidth. Depending on the bandwidth reservations, deadlines are assigned to requests, which are then scheduled in order to meet the deadlines. Although scheduling based on the total system bandwidth can provide a higher reserved throughput than worst-case based approaches, the total bandwidth depends on workload variations and thus reservations may still fail.

2.1.2 Latency Management

A basic observation is that the latency of a request on average is proportional to the queue depth of the storage system. That observation is used by a number of schedulers [LMA03, KKZ05, CAP+03] to throttle requests and was studied more carefully in [GSA+11]. The main problem with throttling requests is that it can lead to a lower throughput than what is possible. In particular, as long as a subset of the requests has low latency requirements, the queue size of the system decreases leading to a lower throughput for all other requests. Facade [LMA03], described below, is an example of such a scheduler.

Facade [LMA03] aims to provide performance guarantees described by an Service Level Agreement (SLA) containing the desired request rate and latency of 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. Unfortunately, a single set of low latency requests may decrease the queue size of the system, which may reduce the throughput significantly. On the other hand, a time-based approach would reduce the queued requests depending on their total cost leading to a higher throughput. Moreover, as is the case with throughput-based approaches it is hard to determine whether a new workload may be admitted. In Facade, it is assumed that a capacity planner is provided to handle the task of admission and make sure that the aggregate demands of the workloads can be satisfied. In terms of scheduling, although Facade uses EDF, it does not include a stream rate in the deadline computation, which is not helpful in managing the performance of each workload separately. Instead, our approach allows each host to receive a percentage of the storage device making it clear whether there are available resources for additional streams.

Triage [KKZ05] takes a decentralized approach and throttles requests from different clients by having an online feedback loop. In particular, it uses SLA bands, which lets administrators specify the proportion of the throughput each client is given if the total throughput is within certain throughput values. For example, for the throughput band between 100 to 200 IOPS, one client may be assigned 30% and the other 70%, whereas if there is additional throughput available (200 to 300 IOPS), the percentages may differ for that band. Unlike our approach, throttling based on throughput bands can lead to substantial losses in terms of throughput reservation and under-utilization of the storage. Moreover, it is not an easy task to assign throughput targets for each

band and client.

SLEDS [CAP+03] attempts to provide absolute guarantees by throttling the requests of each workload. In particular, if a workloads is given a throughput under its SLA, then SLEDS lowers the throughput of other workloads that exceed their limit or are above their SLA target. As noted in the corresponding paper [CAP+03], the above is only a heuristic. Note that in our work we distinguish sequential from random requests and charge the appropriate streams based on time. In Triage, if one of the workloads contain both sequential and random streams the above heuristic is unlikely to provide a fair throughput leading to a low throughput reservations.

2.1.3 Absolute Targets and Isolation

Besides relative throughput and latency targets, there exist schedulers trying to guarantee absolute targets. In addition, some of that work also attempts to provide workload isolation.

Virtualizing Disk Performance [KWG+08] presents a time-based solution for guaranteeing certain throughput indirectly by providing a specific amount of disk time to each stream. Since there are different types of workloads and not all requests are equally expensive, with best-case requests costing orders of magnitude more than worst-case requests, charging streams by time leads to high throughput reservations. Moreover, due to time-based scheduling, admission control in [KWG+08] becomes easy since a workload may be admitted to the system as long as the total utilization remains bounded by 100%. [KWG+08] provides workload isolation by charging the appropriate streams for any

seeks they induce and by minimizing inter-stream seeks through appropriate scheduling. Given the above, instead of keeping an estimate of the request costs, [KWG+08] fixes those costs. That is not realistic, since the cost of requests varies depending on the workload (e.g., enqueueing multiple random requests improves throughput.) Secondly, their method works for a single disk and we are interested in black-box storage systems. Fahrrad [PKB+08] provides hard guarantees efficiently using time as a metric. Based on each stream utilization target rate and execution time estimations, each request is assigned a deadline. The execution times are estimated by keeping statistics of the time between consecutive request responses. Requests are scheduled according to a modified version of the Earliest Deadline First algorithm, which tries grouping sequential requests. In terms of satisfying its hard deadlines, Fahrrad ensures that dispatching a request is not going to make another request miss its deadline. It is possible to adjust the level of confidence, which may lead to a higher or lower throughput accordingly. For example, if all requests are assumed to have a worst-case execution time the throughput may drop significantly. Fahrrad manages a single disk. We are interested in managing black-box storage systems. Unfortunately, keeping statistics regarding the inter-arrival times is hard in a black-box setting. In particular, two consecutive responses may be coming back from two different disks. In that case, the time difference between those two arrivals do not allow us to estimate the execution time of the second response. Moreover, in our work we do not assume the type of workload for each stream is provided and instead try to classify each request as it enters our controller. Finally, schedulers such as Fahrrad require modifications to existing systems something that can be impractical, while our

approach works on top of them transparently without requiring any modifications.

Horizon [PSB10] being similar to Fahrrad manages disk I/O utilization in terms of time. It provides soft throughput and latency guarantees indirectly by converting client targets to utilization rates. Given that its goal is not to provide hard guarantees it does more efficient reordering than Fahrrad by using expected execution times. Moreover, it provides better support for low-latency bursty workloads. Unlike Fahrrad, Horizon was tested on a distributed environment, where each node manages its disk using the Horizon scheduler.

Maestro [MUP+11] attempts to satisfy absolute throughput or latency requirements for multiple clients on large disk arrays. It maintains a moving concurrency bound for the number of outstanding requests for each application so that their absolute requirements are satisfied if the array has enough capacity. In particular, it uses a linear model and estimates its parameters through small changes in the concurrency bound so that the bound stays close to the right value. On the other hand, if the disk array does not have enough capacity to handle the workloads (overload), then Maestro reduces the performance of each application based on its priority. Although it can be useful to allow the storage to be overloaded in certain cases, Maestro does not provide an alternative, i.e., an admission control mechanism for the cases where the client performance should not drop below its targets. Finally, the main issue with Maestro is that it does not isolate streams, leading to a low reservable throughput.

YouChoose [ZXJ11] tries to provide the performance of user-selected reference storage systems instead of an SLA by measuring their performance (service times) off-line and

mimicking it online by scheduling client requests according to the predicted service times. It is based on an off-line machine learning process, similar to [WAA+04], which can be hard to prepare due to the challenging task of selecting a representative set of training data for the workloads to be executed. Moreover, the admission of new virtual storage devices without affecting the performance of the other users can be challenging. Finally, YouChoose attempts to isolate the performance of each virtual device by sending requests in batches.

Argon [WAEMTG07] consists of three components: scheduling, amortization and caching. Most importantly, it does not satisfy our black box requirements since it assumes that the service time is measured by the scheduler and is available for scheduling purposes. Scheduling happens in quanta. For each workload a quantum size is computed during which it has exclusive disk access. Client quanta are scheduled in a round-robin manner. However, that can lead to lower throughput as mixing random workloads can have a positive effect in performance and as long as they are dispatched according to their rates insulation can also be achieved. Amortization: By using a simple model for the service time, which takes into account the disk seek, rotation and transfer costs it computes the I/O size the sequential requests should have in order to achieve a certain degree of stream insulation. In particular, the sequential requests are merged into one large request so that overhead costs are reduced to a cost comparable to one random request. On the other hand, sequential streams do not necessarily have perfectly sequential streams, which could reduce the chances for amortization or could lead to reading unnecessary data. Instead, in our work we assign the appropriate deadlines to requests

and keep a large (e.g. 200) number of requests queued up in the storage system. By doing so, sequential requests are grouped by the operating system. Of course, if needed that effect could be replicated in our controller. Caching: Argon avoids hitting the filesystem cache by using O_DIRECT and manages its own cache. Doing so allows Argon to force each workload to use only part of of the cache leading to an improvement in their insulation. Note that in a black box environment, although it is possible to manage a local cache it would be a violation to assume that the filesystem cache is not used.

In our work we are interested in absolute latency and throughput guarantees while providing workload isolation. We take a time-based approach and treat sequential and random requests differently as in [PSB10, PKB+08, KWG+08]. Our work differentiates from those solutions by not assuming low-level control over each hard-drive. Instead, we treat the storage device as a black box and do not assume our own scheduler is in front of every hard-drive. Finally, [KCGK04] attempts to predict response times through statistical models, while we are interested in service times, which do not include queueing delays.

2.2 Flash Storage

2.2.1 Performance and Device Characteristics

Flash has high read performance, however its performance under workloads containing random writes can suffer. We now provide an overview of why that is the case. NAND

flash organizes data in blocks, each of which contains multiple pages. Although the number of pages varies, it is common for a block to have 128 or 256 pages, each storing 4KB, for a total of 512KB or 1MB, respectively. Reads and writes happen at a page level, while erasing data (or overwriting in-place) can only happen safely at a block level, due to the physical nature of flash. A block erase operation takes about 1.5ms, whereas reading a page takes only 80μs and programming it 200μs [XYH10, SA13]. Because of that performance difference, it is common for flash devices to append writes instead of performing them in-place. In other words, writes are applied on clean pages in an analogous way to the write process in log-structured file systems. As new data is appended, outdated pages are flagged accordingly.

After a certain threshold of internally used storage space and depending on the flash device policy, garbage collection is triggered so that flagged pages are cleaned and appending remains possible. However, if the flash device receives enough random writes, the garbage collector cannot keep up because erasing a block is more expensive compared to programming a page. In that case, the flash device runs out of space and cannot immediately append data. The device then performs garbage collection on-line. Performing a write may require that a whole block is moved to memory, where the outdated data is replaced by the new data, and the whole block is then written back to flash. This leads to write amplification [LSZ13], high latencies (up to 100ms) and unpredictable behavior for writes. Finally, due to that variance, queued read requests on the same device are affected similarly, since the device is blocked frequently.

A large part of research on flash and SSDs focuses on the properties of the devices

themselves and improvements that can be performed internally. For example, multiple papers study the performance characteristics of SSDs. uFlip [BJB09] presents a benchmark, illustrates flash performance patterns, and presents design hints for flash-based systems. They find that random writes should be limited to areas of 4-16MB to maintain sequential write performance while large block sizes (32KB) are beneficial for writes. Moreover, they conclude that blocks should be aligned to flash pages, otherwise the penalty is quite severe.

The authors in [CLZ11] study the effect of internal parallelism in SSDs on performance. They note that exploiting parallelism can significantly improve I/O performance and decrease the performance limitations of SSDs. Second, particular attention must be paid in the interference of reads and writes in parallelism. The authors conclude by mentioning that optimizations tailored to hard-drives can be ineffective or even harmful to SSDs.

The work in [CKZ09] presents a set of experiments on the performance of SSDs. It provides experiments on the effect of reads/writes and access patterns on performance. In particular, they demonstrate the effect of random writes on the performance of SSDs and mention that the in the worst case, the average latency could increase by a factor of 89 and bandwidth could drop to only 0.025MB/s. On the other hand, the improvements of SSDs on read-only performance are noted.

Rajimwale et al. present system-level assumptions that need to be revisited in the context of SSDs [RPD09] and discusses the implications on designing storage systems. In particular, they propose that the block management is removed from the file system

and delegated to the SSD to avoid accumulation of storage-specific assumptions. Finally, they find that object-based storage is an appropriate way to achieve this.

Other work focuses on design improvements, and touches on a number of aspects of performance such as parallelism and write ordering [APW+08]. Agrawal et al. [APW+08], mention that SSD performance and lifetime is highly workload dependent and that complex systems problems that normally appear higher in the stack, now are relevant to the device firmware. The authors in [Des12] propose a solution for write amplification, while [BD10] focuses on write endurance and its implications on disk scheduling. Moreover, Grupp et al. focus on the future of flash and the relation between its density and performance [GDS12].

Jung et al. note that GC (garbage collection) and resource contention on SSD channels are critical bottlenecks while schedulers at the host interface logic (HIL) are unaware of such bottlenecks. This can result in violation of quality of service (QoS) requirements. HIOS [JCS+14] is a host interface scheduler for SSDs that is GC- and QoS-aware. It redistributes garbage collection across non-critical I/O requests and reduces channel resource contention. The main goal of HIOS is to ensure that the overheads due to GC or contention are bounded by within a request deadline, e.g., by stealing slack-times from non-critical I/O operations. Through simulation, the authors mention that HIOS reduces the standard deviation of latency by 52.5%, and the worst-case latency by 86.6%, compared to existing schedulers.

Ouyang et al. note that commodity drives typically only provide 50-70% of their space to the user to accommodate non-sequential writes, which is not effective enough for

their data-center needs at Baidu. They present SDF [OLJ+14] is a software-defined flash hardware/software storage system, which exposes individual flash channels to the host software and eliminates space over provisioning. Their device interface allows small reads but requires writes to have a size that is a multiple of the flash erase block size and requires write addresses to be block-aligned. That way write amplification is eliminated because no flash block can contain both valid and invalid data pages when garbage collection takes place. Moreover, each channel is presented to the applications as a separate device. Finally, the authors demonstrate that their system increases I/O bandwidth by 300% and reduces the per-GB hard-ware cost by 50% compared to commodity drives used at Baidu.

2.2.2 Scheduling

There is little work taking advantage of performance results in the context of scheduling and QoS. A fair scheduler optimized for flash is FIOS [PS12] (and its successor FlashFQ [SP13]). FIOS tries to improve flash efficiency and increase parallelism and is an improvement over OS schedulers that are designed with hard-drive characteristics in mind. Part of FIOS gives priority to reads over writes, which provides improvements for certain drive models. However, FIOS is designed as an efficient flash scheduler rather than a method for guaranteeing low latency. For example, a drive that blocks for garbage collection will block the incoming requests even if reads are given priority. Instead, our work on flash uses a drive-agnostic method to achieve read-only performance.

In another direction, SFS [MKC+12] presents a log-structured filesystem designed for

SSDs, with the goal of exploiting their maximum write bandwidth. To achieve that, SFS transforms all random writes to sequential ones at a filesystem level SFS. Moreover, SFS puts data blocks with similar update likelihood into the same segment to minimize the segment cleaning that log-structured filesystems have. According to the prototype implementation, the authors concluded that SFS has $2.5X$ the throughput of LFS, and reduces the block erase count inside the SSD by up to $7.5X$ compared to filesystems such as ext4 and btrfs.

SSDs are often used as a high performance tier (a large cache) on top of hard-drives [AM+13, MW08], so work using flash in storage systems often treats flash as a cache [AM+13, LSD+14, QBG14]. Nitro [LSD+14] is a flash cache architecture optimizes for lower capacity through adjustable deduplication, compression, and large replacement units. The authors experimentally present the trade-offs between data reduction, RAM requirements, SSD writes and storage performance. Their evaluation shows an increase of IOPS up to 120% and reduction of response time up to 55% compared to an SSD cache without Nitro. Advantages of Nitro include the reduction of random writes to the SSD by up to 53%.

Janus [AM+13] provisions flash caches on top of hard-drives in cloud-scale distributed file systems. The authors found that most IO in their systems (at Google) was on newly created files. Moving new files from flash to disk in FIFO or LRU resulted in 28% of the reads being served from flash by placing in flash 1% of all data. Moreover, all read ages were well represented: there were jobs accessing very young to very old data (1min to 1 year). The read rate is maximized based on workload priority through an optimization

program, while the flash write rate is bounded to avoid flash wear and reduce impact of flash writes (which are slow) on read latencies. To adjust to workload changes the optimization program is run periodically. Due to the scale of the system the traces provided as the input to the optimization program is based on trace sampling. The authors found that the resulting allocation improves the flash hit rate by 47% to %76 compared to a unified tier shared by all workloads. The concluded that flash-storage is a cost-effective complement to disks in data centers.

Koller et al. [KMR+13] present write policies for enabling host-side write-back flash caching while providing certain degrees of data consistency. As the authors mention, flash-based caching on the host side is a promising direction for optimizing networked storage. Currently, host-side flash caches are primarily used as read caches and employ a write-through policy which provides the strictest consistency and durability guarantees. In that case, the flash cache remains underutilized if the workloads mostly consist of writes. Instead, the authors present two consistent write-back policies, ordered and journaled, that outperform write-through and provide a trade-off across performance, data consistency, and data staleness.

As in the previous paper [KMR+13], Qin et al. [QBG14] focus on the problem of increasing the utilization of client-side flash-caches by treating it as a write-back cache. They present two write-back caching policies: write-back flush and write-back persist, that provide strong reliability guarantees. These policies rely on the file systems (or databases) issuing write barriers to persist data reliably, and implement barrier semantics. Their evaluation shows that their write-back persist policy is close to write-back

for both read-heavy and write-heavy workloads.

File systems often use ordering points to achieve consistency in case of a system crash. Requiring reordering lowers the performance and increases complexity. Moreover, it requires lower layers to enforce the ordering of writes, (which also introduces challenges in write-back caches). No-Order File System (NoFS) [CSADAD12] is a lightweight file system that employs a novel techniques called backpointer-based consistency to (provably) provide crash consistency without ordering writes. The authors present a performance evaluation showing that NoFS performs better than ext3 on workloads that frequently flush data to disk explicitly. Finally, it is mentioned that NoFS may be of special significance in cloud computing, where writes are multiplexed due to virtual machines.

Disk contention in cloud storage can lead to reduced throughput by an order of magnitude. Gecko [SBMW13] is a log-structured design for eliminating workload interference in hard-drive storage by chaining together a small number of drives into a single log. By spreading the log across multiple hard-drives, it can decrease the effect of garbage collection on the incoming workload. In particular, it separates the tail of the log (where writes are appended) from its body. Therefore, writes are performed without interference from reads (garbage collection of first-class reads), which are performed at the body of the log according to a specific caching policy. The authors present an evaluation showing that the achieved random write bandwidth is between 60 and 120 MB/s, despite concurrent garbage collection, application reads, and an adversarial workload.

2.3 Erasure Coding in Storage

Replication has become the standard method for storing information in data-centers [ZDM+10a]. In particular, the default storage policy for cloud file systems such as the Windows Azure Storage (WAS) [CWO+11] and the Google File System [GGL03] is triplication (triple replication). Triplication is easy to implement, has good reliability, and provides good read and recovery performance [KBP+12], however, its storage overhead is a concern.

Erasure coding [DGW+07] is a method for adding storage space redundancy to increase fault-tolerance without the storage overhead of replication. Given N drives, we store an object O of size $|O|$ by encoding it, producing $q|O|$ amount of information, where $q > 1$ is the amount of redundancy. The encoded object is then split into N equal chunks, which can be stored in distinct drives. To read an object O we retrieve *any* N/q chunks of the encoded object and reconstruct the original object O through decoding. If the code used is systematic, then the first N/q chunks correspond to the actual object, and the remaining ones to the coding information. If the node is non-systematic, then all chunks contain coded information and decoding is required for every read operation. For example, let $N = 15$ and $q = 1.5$. Given an object O of size $|O| = 100$, the encoded object is of size $q|O| = 150$ and can be split into $N = 15$ chunks, each of size $q|O|/N = 10$. To read the original object, any $N/q = 10$ chunks (or more) can be used to reconstruct (decode) the original object.

An advantage of erasure coding over replication is the increase of reliability without

additional storage space [WK02]. Erasure coding is used today in large-scale storage systems such as Windows Azure [HSX+12], and for big data in Hadoop HDFS [SAP+13]. Work on erasure coding performance includes improving degraded reads [KBP+12] (with an emphasis on few failures). Using Intel SIMD instructions has been shown to improve coding performance [PGM13], and is something we take advantage of in this work by using the Jerasure library [PG14]. Employing GPUs appears as another direction for improving computation in erasure codes [SRC12, BRV12, Cur10, GAKGR10], which would improve the performance of degraded reads. However, research on erasure coding and flash in particular, appears limited. In what follows we describe the related work in more detail.

Replication has become the standard in data centers. Zhang et al. [ZDM+10b] present new trade offs of erasure coding in data centers related to power consumption and complexity and compare with replication. They find that depending on the data center setup the storage cost advantage of erasure coding over replication ranges from 8.4% to 37%. Moreover, they find that the performance advantage is highly workload-dependent and varies from $< 5\%$ to $> 50\%$ while the energy advantage is highly correlated. Finally, they mention that the main penalty paid by erasure coding is the decreased performance during recovery.

Plank et al. [PLS+09] present a performance evaluation of different open-source erasure coding libraries. They found that RAID-6 codes outperform their general-purpose counterparts while Cauchy Reed-Solomon coding outperforms class Reed-Solomon coding significantly. They note that parameter selection can have a huge impact on how

well an implementation performs and that the interactions of the code with the memory and caches must be considered. Finally, the mention that a challenge for erasure coding open-source libraries will be to exploit multicore architectures.

A variety of erasure codes, e.g., Reed-solomon and LRC codes, use Galois Field arithmetic for encoding and decoding operations. The authors in [PGM13] mention that most implementations of Galois Field arithmetic use multiplication tables or discrete logarithms. To improve the performance of encoding/decoding in erasure codes, they propose the use of Intel SIMD instructions for achieving fast Galois Field arithmetic. Through experiments they demonstrate that their implementation is 2.7 to 12 times after than other implementations. Because of these improvements, the authors expect that future code design will reply more on Galois Field arithmetic than XOR's. Note that in our work we take advantage of Intel SIMD instructions by using the Jerasure [PG14] library released as open-source by the same authors.

The authors in [KBP+12] consider erasure codes as a replacement of replication in cloud file systems. They note that in such systems there are two performance critical operations with respect to erasure coding: Degraded reads due to temporarily unavailable data and recovery from single failures. In their work, they generalize results on reducing data requirements of recovering in RAID-6. They present an algorithm for finding the optimal number of symbols needed for recovering from an arbitrary number of drive failures. Moreover, they examine the amount of data needed to perform degraded reads, showing that it can use fewer symbols than recovery. Finally, they develop a new class of codes, called Rotated Reed-Solomon codes that improve the degraded read performance.

Azure is a cloud storage system used in production by Microsoft. Huang et al. [HSX+12] apply erasure coding in Azure through a new set of codes called Local Reconstruction Codes (LRC). LRC reduces the coded chunks required to reconstruct unavailable data while keeping the storage overhead low, which reduces the bandwidth and I/Os required for degraded reads. The main idea of LRC is the use of local parity; the drives are divided into groups and each group has a local parity, in addition to the global parity shared by all groups.

HDFS-Xorbas [SAP+13] is a system based on Hadoop HDFS that uses a new family of erasure codes called Locally Repairable Codes (LRC). Compared to the HDFS module, which uses Reed-Solomon codes, HDFS-Xorbas achieves approximately 2x reduction on the repair disk I/O and repair network traffic, but requires 14% more storage. The basic idea of LRCs is to add additional local parities to make repair efficient.

As mentioned in GPUStore [SRC12], storage systems have features such as encryption, checksums and error correcting codes, which can be computationally expensive. Many of those computational tasks are inherently parallel making them a good fit for GPUs since consumer GPUs have hundreds of cores. However, the current software frameworks for GPUs were designed primarily for long-running, data-intensive workloads and are not a good fit to storage systems. GPUStore [SRC12] is a framework for integrating GPU computing in storage systems. Through a prototype, supporting encryption and RAID-6 data recovery, the authors demonstrate that their framework achieves performance improvements of up to an order-of-magnitude compared to a CPU-based implementation.

Shredder [BRV12] is a framework for content-based chunking using GPU acceleration. The authors addressed both compute-and data-intensive environments and provided optimizations for reducing the cost of transferring data between the CPU and GPU and reducing the GPU memory access latency. They improved the chunking bandwidth by over $5X$ compared to the same host without GPU, and present an application of Shredder to HDFS and a cloud backup system.

Chapter 3

Guaranteeing I/O Performance on Black Box Storage Systems

3.1 Introduction

In environments such as cloud-based systems, where multiple clients compete for the same storage device, it is especially important to manage the performance of each client. 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.

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

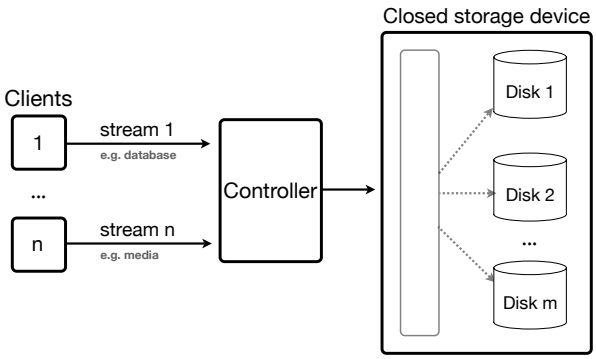


Figure 3.1: Given that we have no access to the storage device, we place a controller between the clients and the device to provide performance management to the clients, i.e., the request streams.

the clients or the storage device, or requiring any modifications to the infrastructure of either side. Instead of modifying the internals of the clients or storage, which can be impractical and expensive, we place a controller between the clients and the storage device (Figure 3.1).

Clients want throughput reservations. However, except for highly regular workloads, throughput varies by orders of magnitude depending upon workload (Figure 3.2) 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 [PKB+08]. Our utilization-based approach can work with Service Level Agreements (SLA); requirements can be converted to utilization as demonstrated in [PSB10] and as long as we can guarantee utilization, we can guarantee throughput provided by an SLA.

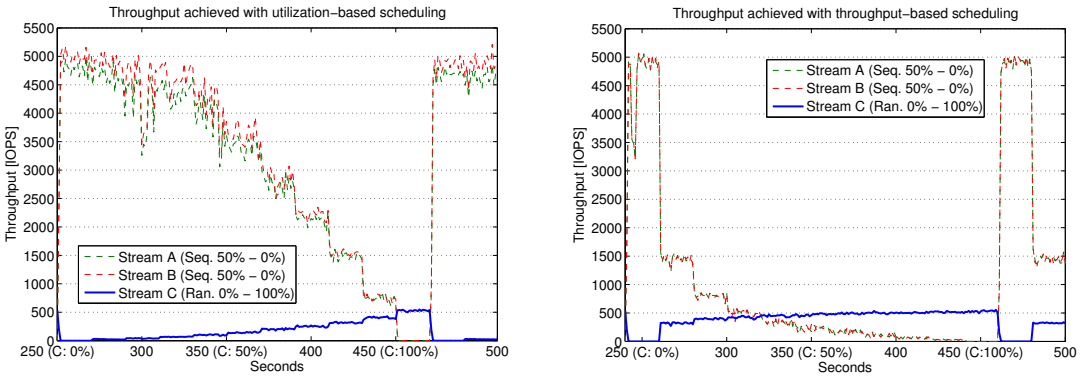


Figure 3.2: Our controller provides isolation (left). With throughput-based scheduling (right) the introduction of a random stream makes the throughput of sequential streams drop dramatically.

Our main contribution is QBox, 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 [LL73, SBA96]. 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.

3.2 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 [PKB+08, PSB10] or [KWG+08]. 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 [Pov10].

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 and receiving responses. For example, we cannot modify or replace the device's scheduler as is the case in [PSB10] 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

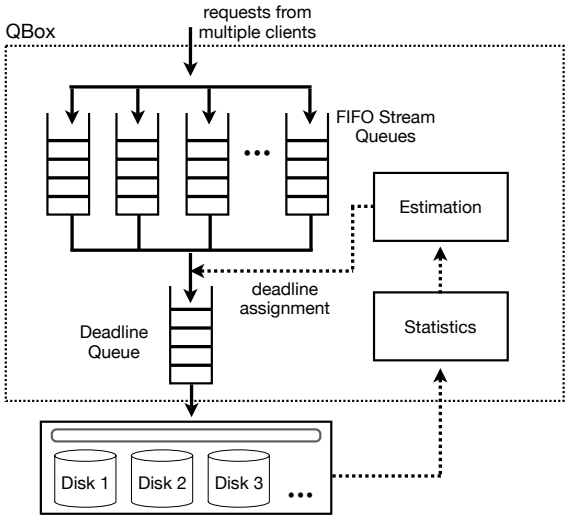


Figure 3.3: The controller architecture.

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 (Figure 3.3). 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. The order at which requests are dispatched to the device depends

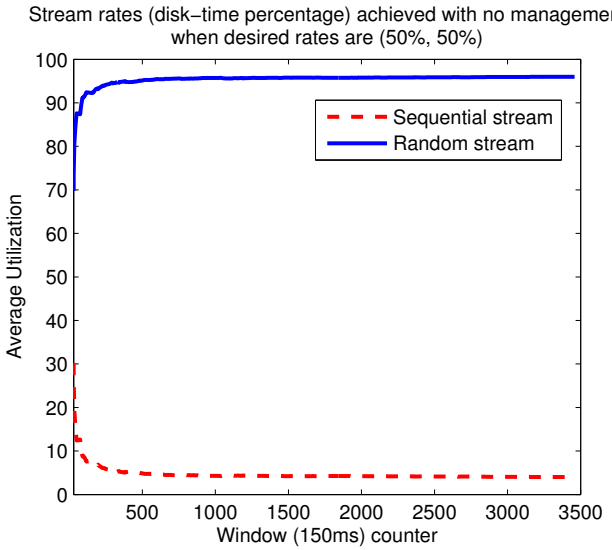


Figure 3.4: When our controller sends the requests in the order they are received from the clients, the system fails to provide the desired rates.

on their assigned deadlines and happens according to the Earliest Deadline First (EDF) algorithm [LL73]. 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., [PSB10], then the absolute times would be important for replacing the expected cost with the actual cost after the request was completed. In our work so far we have not focused 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 [PSB10] 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.

3.3 Performance Management

In our controller 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.

3.3.1 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

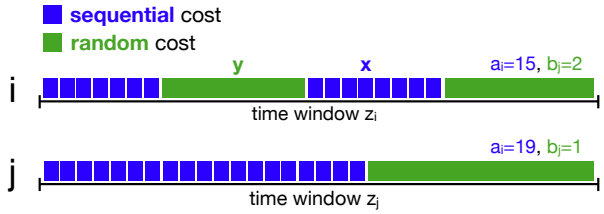


Figure 3.5: Counting sequential and random completions per window lets us estimate their average cost.

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 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 (Figure 3.5).

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:

$$\begin{cases} \alpha_i x + \beta_i y = z_i \\ \\ \alpha_j x + \beta_j y = z_j, \end{cases} \tag{3.1}$$

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:

$$x = \frac{z_j\beta_i - \beta_j z_i}{\alpha_j\beta_i - \alpha_i\beta_j}, \quad y = \frac{z_i}{\beta_i} - \frac{\alpha_i}{\beta_i}x. \tag{3.2}$$

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 3.6), i.e., satisfying:

$$\begin{cases} z_i/\alpha_i < z_j/\alpha_j \\ \\ z_i/\beta_i > z_j/\beta_j \\ \\ \alpha_i/\beta_i > \alpha_j/\beta_j \end{cases} \text{ or } \begin{cases} z_i/\alpha_i > z_j/\alpha_j \\ \\ z_i/\beta_i < z_j/\beta_j. \\ \\ \alpha_i/\beta_i < \alpha_j/\beta_j \end{cases} \tag{3.3}$$

Intuitively, setting z_i equal to z_j in (3.3) 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

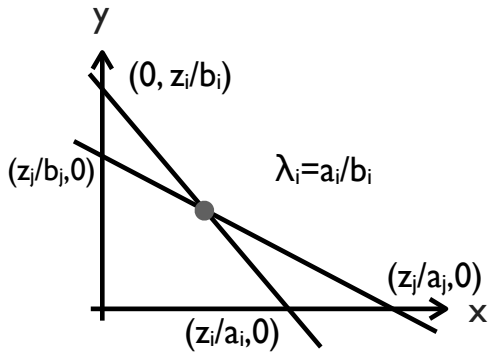


Figure 3.6: The intersection of the two lines from (3.3) gives us the average cost x of a sequential and y of a random request in the time windows z_i and z_j.

intuitively understand. In that case the solution becomes

$$x = \frac{z}{\alpha_i + \beta_i \lambda}, \quad y = \lambda x, \tag{3.4}$$

where

$$\lambda = \frac{\alpha_i - \alpha_j}{\beta_j - \beta_i}. \tag{3.5}$$

From (3.5) 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 3.6 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

$$x = \frac{\alpha_j z_i}{\alpha_i(\mu\beta_j + \alpha_j)}, \quad y = \mu x, \tag{3.6}$$

where

$$\mu = \frac{1}{\beta_i}\left(\frac{z_i}{x} - \alpha_i\right). \tag{3.7}$$

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.

3.3.2 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 (3.2) 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

$$\left|x - x'\right| = \frac{\beta_i}{|\alpha_j \beta_i - \alpha_i \beta_j|} \left|(\alpha_j x + \beta_j y) - z_j\right| \tag{3.8}$$

and

$$\left|y - y'\right| = \frac{\alpha_i}{\beta_i} \left|x' - x\right|. \tag{3.9}$$

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 (3.9) 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 (3.2) 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 3.6. Specifically, fixing every variable in (3.2), 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.

3.3.3 Write Support and Estimating in Practice

In our work so far, 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 (Section 3.5.1).

In our implementation we took the approach of having small windows, e.g. 100ms, to

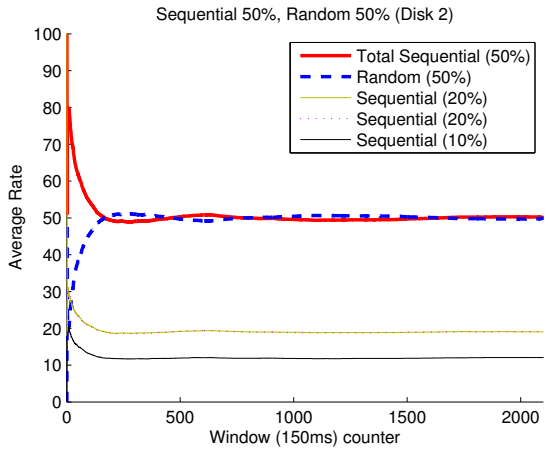


Figure 3.7: Using one disk and a mixture of sequential and random streams the rates are achieved and convergence happens quickly.

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 (3.5) 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.

3.4 Experimental Evaluation

In this section, we evaluate our controller, 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.

3.4.1 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 3.4.4 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 3.4.4 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 [ELMS03] NFS trace with request sizes typically 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.

3.4.2 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 3.8(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 3.8(b) and (c) the sequential streams are given higher utilizations. In all three cases, the achieved utilizations are close to the desired

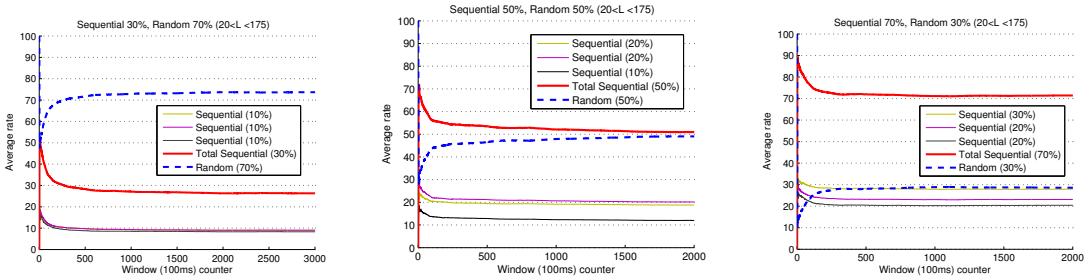


Figure 3.8: Using two disks and our scheduling and estimation method we achieve the desired rates most of the time relatively well. In the above we have three sequential streams and a random one.

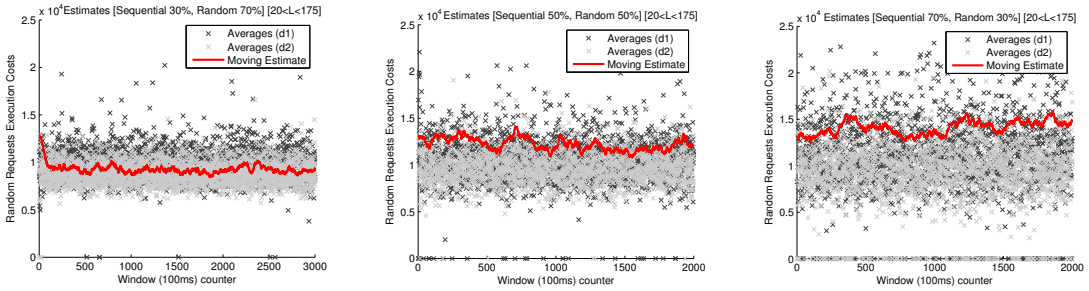


Figure 3.9: Using two disks (d1, d2) and our estimation method we maintain a moving estimate of the average random execution cost on the storage device.

ones. Again in Figures 3.8(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 3.9 we notice that estimates get above the average cost when there is many sequential requests even if the utilization targets are achieved (Figure 3.8). 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 3.8.

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 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 3.8) is 100ms. Other values such as 150ms provide similar estimation quality and later we look at smaller windows of 75ms. Note that in Figure 3.9 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 3.10 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 3.10 we see that the rates are still achieved, while there is slightly more noise in the estimates compared to Figure 3.9. We noted that if the idle times are larger than the window

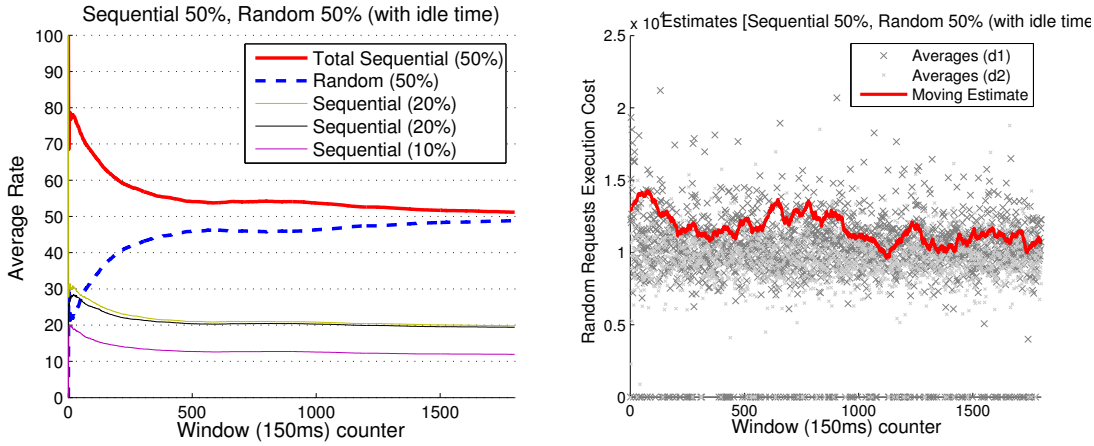


Figure 3.10: Using two disks (d1, d2), the desired rates are achieved well enough (reach 45% quickly) even when there is idle time in the workload.

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.

3.4.3 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

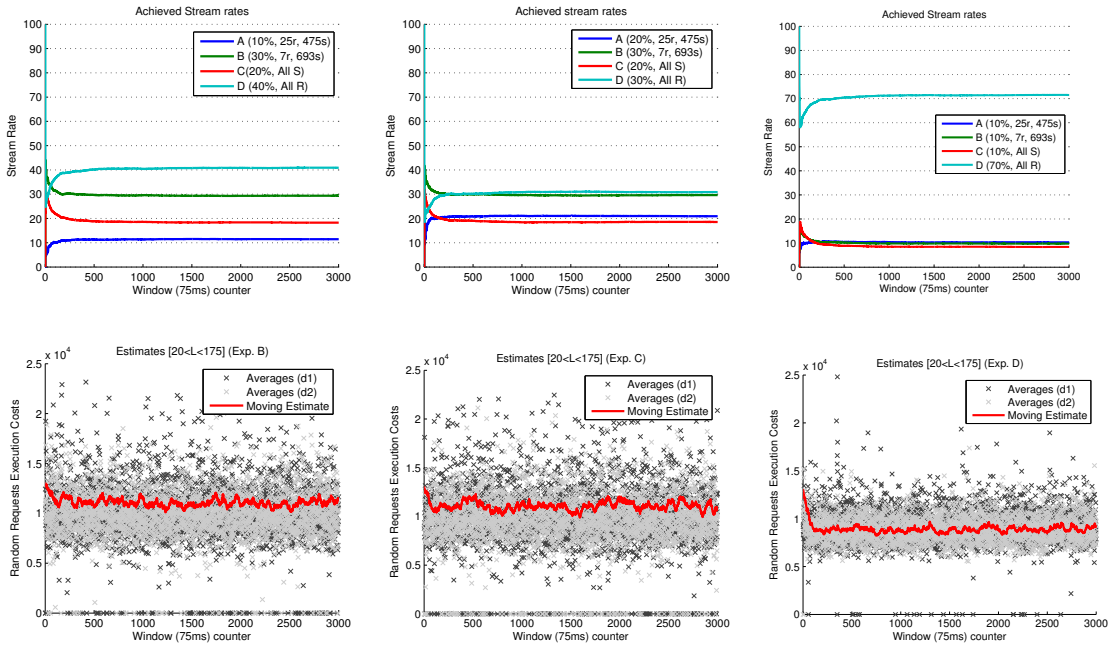


Figure 3.11: Using two disks and our scheduling and estimation method we achieve the desired rates most of the time relatively well (top row). Stream A requires 20% of the disk time and sends 25 random requests every 475 sequential requests. Similarly for the rest of the streams. The moving estimate of the average random execution cost on the storage device (bottom row).

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 [KWG+08], good utilization management can still be provided when random requests are assumed to cost the same. Moreover, from [ELMS03] 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

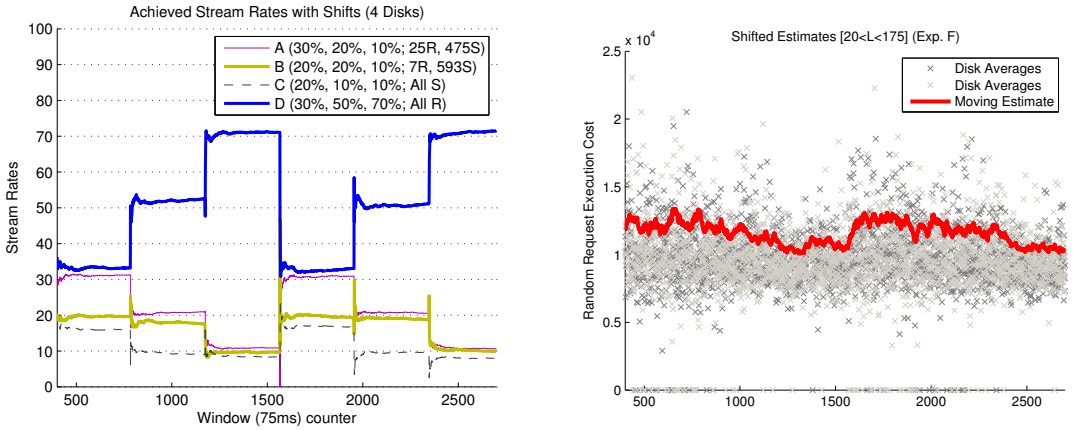


Figure 3.12: Using four disks and desired rates that shift over time, the rates are still achieved quickly under semi-sequential and random workloads.

future work. From Figure 3.11 we see that the targets are achieved in the presence of semi-sequential streams. In particular, in 3.11(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 3.11 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 3.12(a) illustrates that the utilization rates are satisfied and Figure 3.12(b) shows how the random estimate

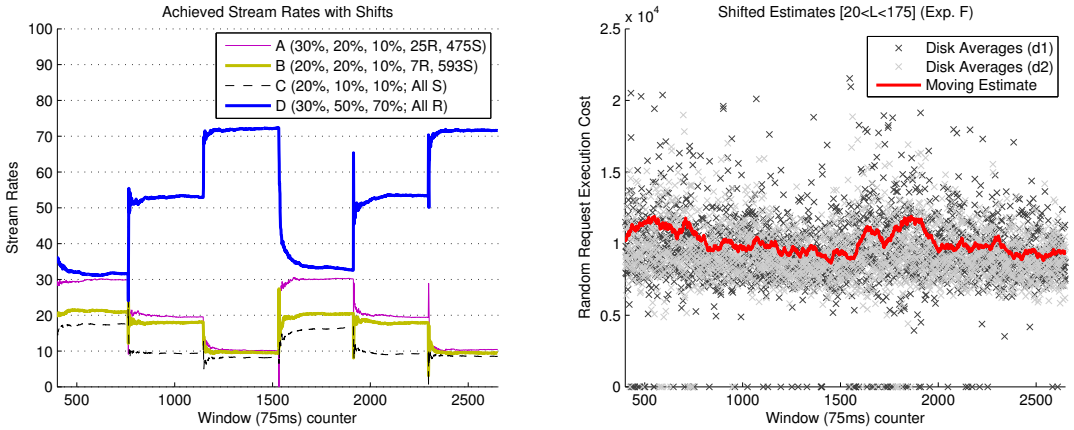


Figure 3.13: Utilization targets are met in the presence of semi-sequential streams and varying target rates. In this experiment we used two disks.

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 3.13).

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 3.14, 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.

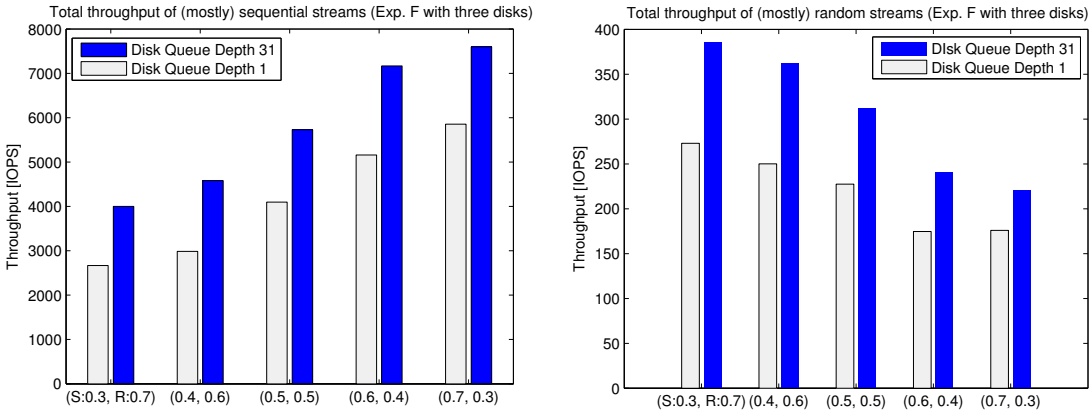


Figure 3.14: The throughput with an NCQ of 1 and 31 is maintained while the desired rates vary.

3.4.4 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 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 3.15 stream A starts with a target rate of 0.5 and goes down

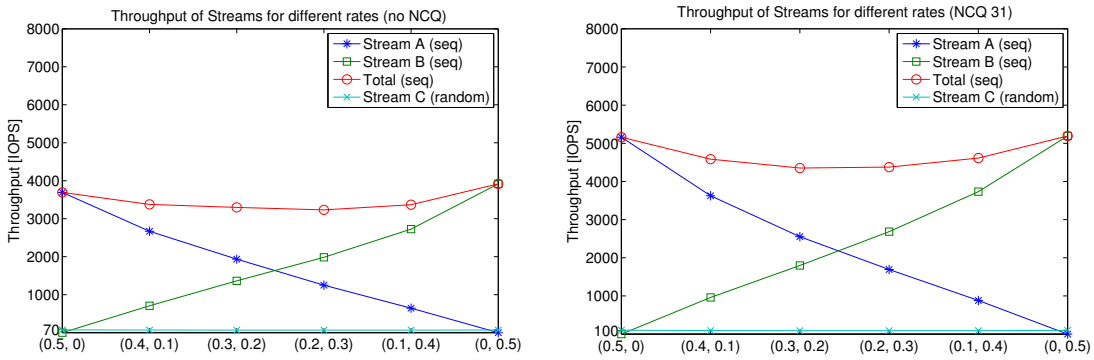


Figure 3.15: Using RAID 0 the throughput achieved by each sequential stream is in agreement with their target rates. Stream A has a varied target rate from 50% to 0 and the opposite for B. Random stream C requires a fixed rate of 50% of the storage time. Similarly for a large disk queue depth (NCQ.)

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 3.16 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 3.15, 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 3.16 there is no drop in the total throughput, which is expected since the cost of seeks between random requests are similar to the typical cost

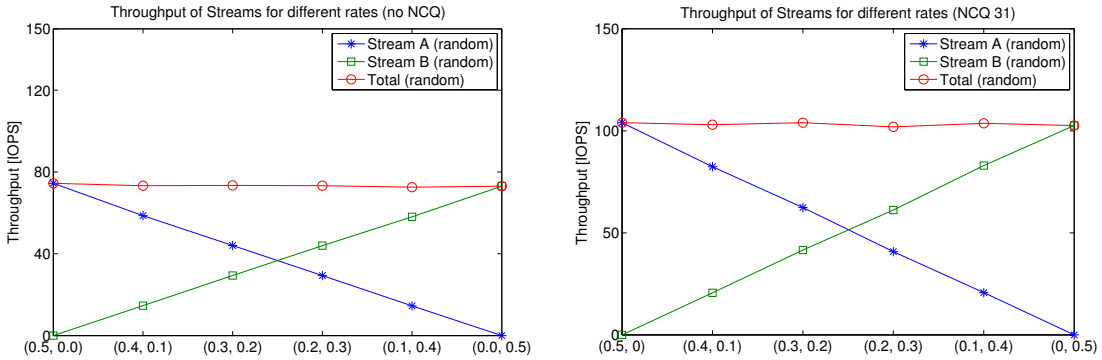


Figure 3.16: Using RAID 0 the throughput of each random stream is in agreement with its target. Stream A has a varied target rate from 50% to 0 and B from 0 to 50%. The sequential stream (not plotted) has a utilization of 50% leading to an average of 3600 and 5060 IOPS with no NCQ and a depth of 31, respectively.

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 3.15. Finally, note that whether we use no NCQ or a depth of 31 the throughput behavior is similar in both Figures (3.15 and 3.16), which is desired since a large depth can provide a higher throughput in certain cases [YSEY10], along with other benefits such as reducing power consumption [Wan06].

3.4.5 Evaluation Using Traces

To strengthen our evaluation, besides synthetic workloads we run QBox using two different days of the Deasna2 [ELMS03] 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

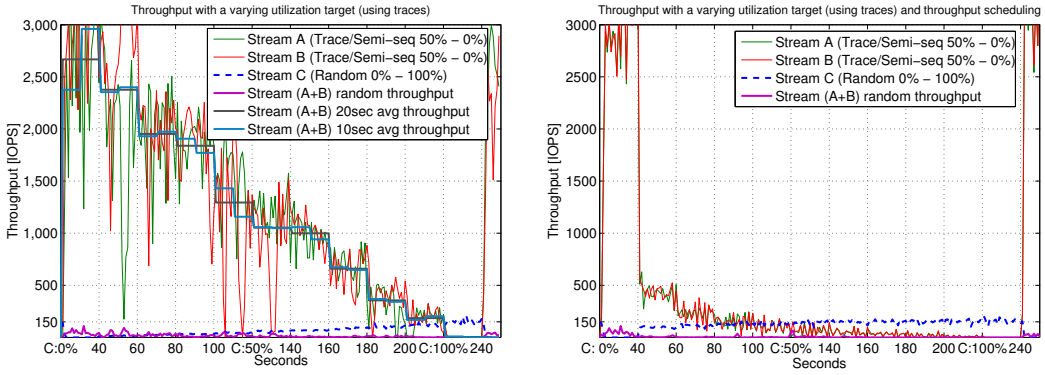


Figure 3.17: QBox provides streams of real traces the throughput corresponding to their rate close enough even in the presence of a random stream. Random stream C affects throughput-scheduling leading to a low throughput for A and B.

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 3.17(a) and the results of throughput-based scheduling in Figure 3.17(b) 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 3.17(a), 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 3.17(b) demonstrates the destructive interference inherent in throughput-based reservation schemes with semi-sequential streams receiving a very low throughput.

3.4.6 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 blocking 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 [GSA+11]–between the queue size and the average latency may also be useful to improve accuracy as well as grey-box methods [ADAD01]. 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 these experiments we treat hits as regular completions for simplicity.

Without modifying QBox we enable the file system cache and see from Figure 3.18(a) 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 3.18(b)) gives similar results to Figure 3.17(b), supporting our position on throughput-based scheduling. Finally, using synthetic workloads we get an output of the same form as Figure 3.15 with a maximum sequential throughput of 1700 IOPS (figure omitted.)

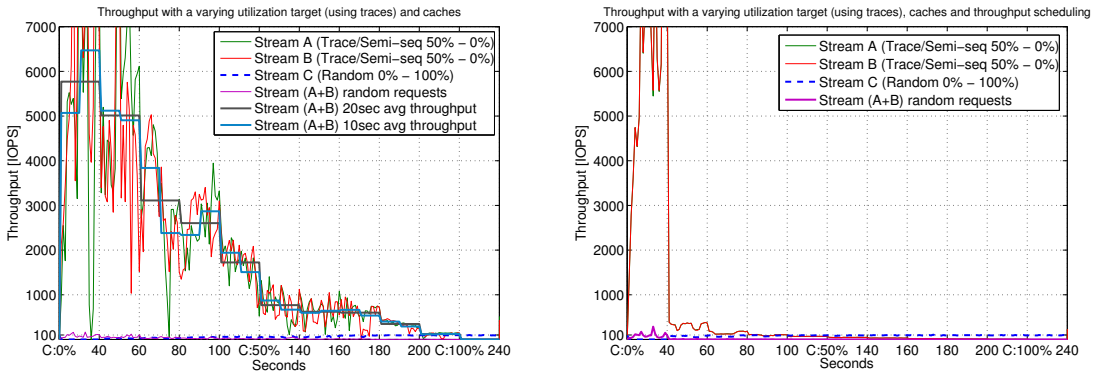


Figure 3.18: The throughput achieved by QBox in the presence of caches follows the target rates. Throughput-based scheduling fails to isolate stream performance leading to a low throughput for semi-sequential streams.

3.4.7 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.

3.5 Discussion: Writes and Hybrid Storage

3.5.1 Managing Writes

Typically, on a write request the sender receives a response as soon as the data is written to some cache (filesystem cache, SSD, etc.), which happens before the data is written, (if at all) to the disk. When the size of the data in the cache reaches a certain threshold, a number of write requests are dispatched to the disk at once by the operating system. Due to that behavior, the total write throughput observed by the clients over small time intervals (e.g. 200ms) may not be reflective of the actual number of disk writes that take place in the storage device. Note that for read requests that is not the case, i.e., a read response implies that the data was either in a cache or it was physically read from the disk, and it is easy to differentiate between those two cases as a physical read is orders-of-magnitude more expensive than a cache read. Under a heavy-enough write workload, throughput averages over time windows of a few seconds (e.g. 5 seconds,) reflect the disk write throughput more accurately. That is expected since most schedulers do not let requests starve and set an expiration time to each of them (e.g. 5 seconds after a request is enqueued.) Therefore, if we look over large enough windows the average throughput can be close to the disk throughput.

In order to properly manage workloads containing write requests we have to keep an estimate of their average sequential and random cost. Adding a third variable to our linear system (3.1) and extending it to three windows is technically easy. However, as already mentioned, read and write requests have a different behavior in terms of throughput.

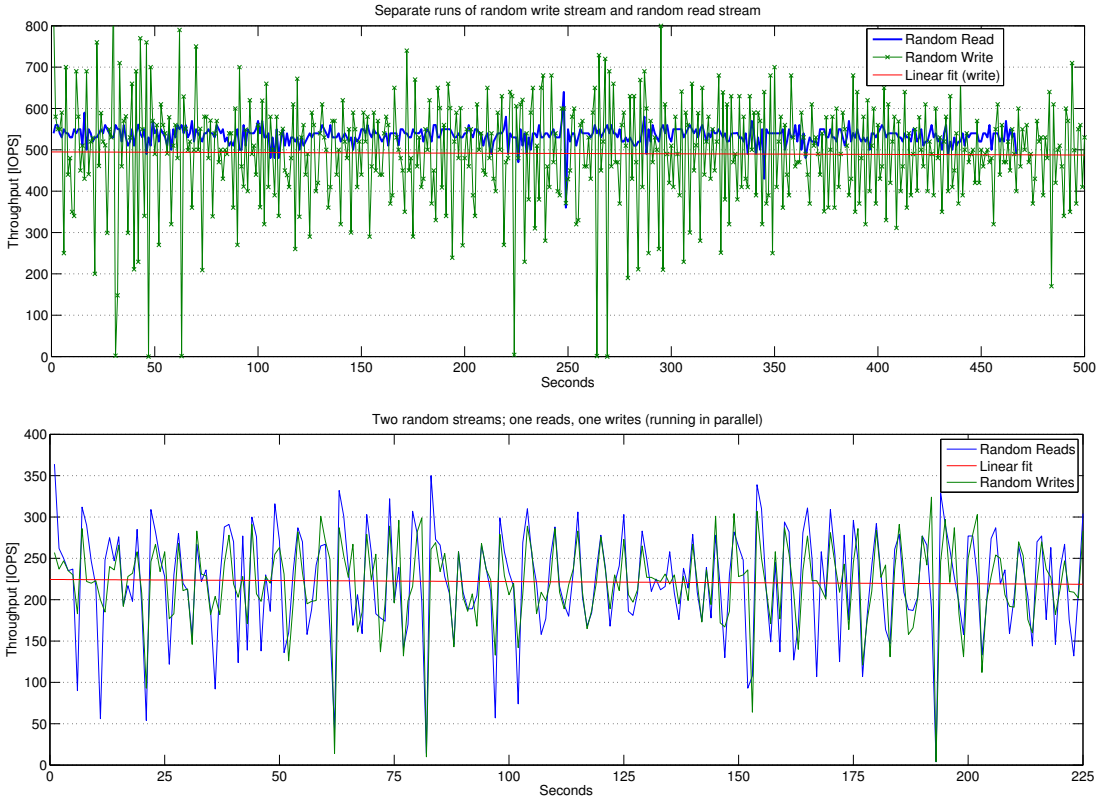


Figure 3.19: Random read requests in isolation (a) achieve a proportionally higher throughput than when the storage is shared (b) with write requests ($2 \cdot 225 = 450 < 500$.)

Although considering the average write throughput is a reasonable approach it is hard to decide the interval over which to compute that average. That is because the numerical solution would depend on that interval size. Indeed, from early experiments we have observed that mixing reads and writes, and extending our estimation method to three variables (sequential reads, random reads and random writes) gives good but not good enough estimates. In particular, we observed that the relation between random reads and writes was accurate enough but the sequential read cost was overestimated. On the other hand, estimating write requests by using average throughput values separately from reads gave us good estimates, which was expected since sequential and random writes can be compared as they have a similar behavior.

In addition, from our experiments we have concluded that writes interfere with reads by slightly lowering the total read throughput and making it as volatile as the write throughput (Figure 3.19.) That is independent of our controller. The drop of throughput is around 10%, which is close to the difference between the cost induced by a read seek and that of a write seek according to disk manufacturers. Therefore, the read to write difference for random requests is expected to be relatively stable. On the other hand, from Figure 3.20 we see that increasing the time window between read and write intervals lowers their interference. To compute accurate estimates we need to separate reads from writes as much as possible with a minimal effect on the throughput and management quality. Keeping writes in our queue for a long time would affect the client performance and increase the latency. Instead, we propose to respond back to the client quickly by having a large buffer (e.g. an array of SSDs) in our controller, dedicated

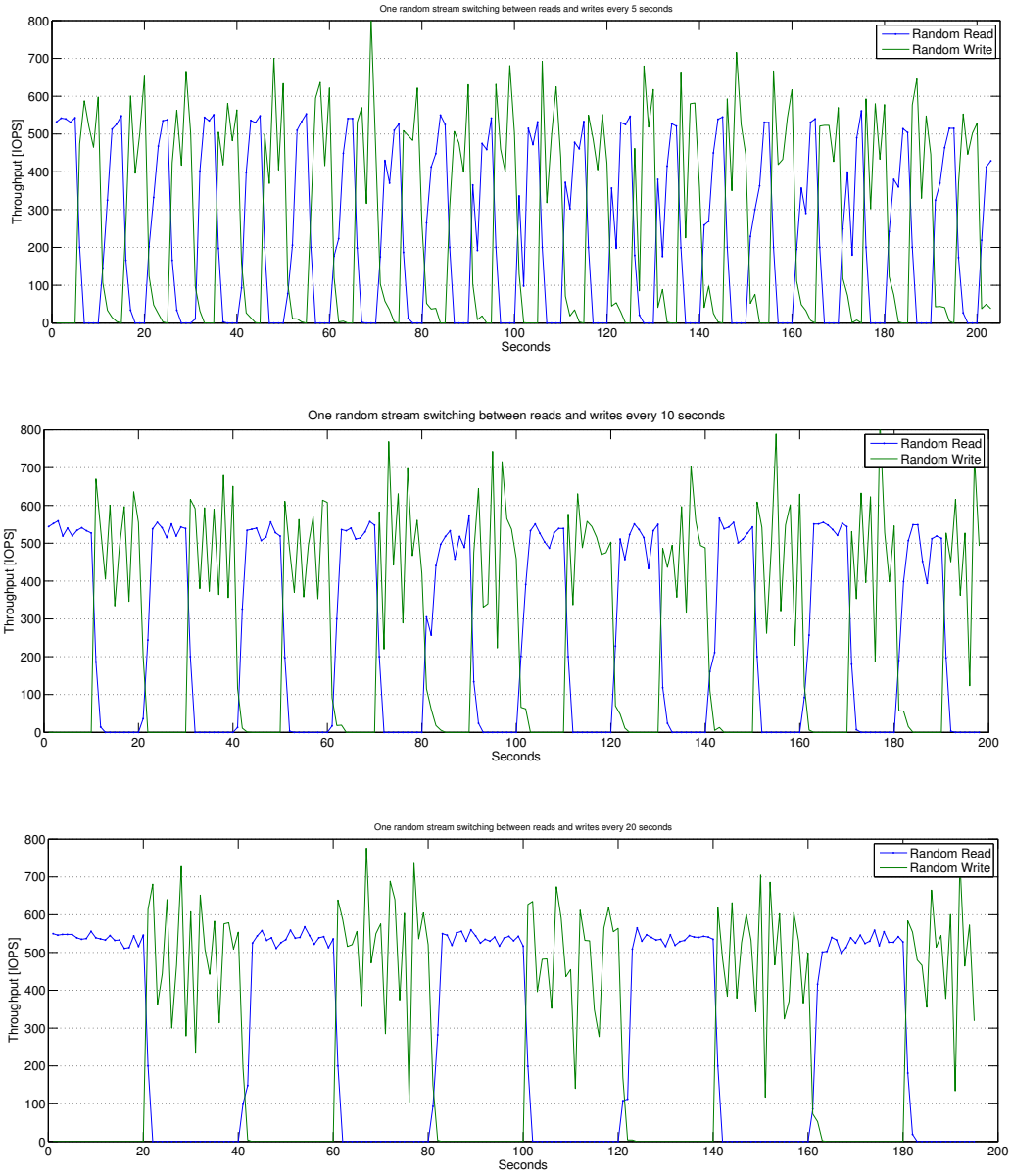


Figure 3.20: Switching between reads and writes every some interval (5, 10 and 20 seconds) decreases the interference of writes on reads.

to write requests. Periodically and transparently from the client we send those write requests to the storage device to be stored permanently. By doing so, we minimize the interference of writes on reads, increase the write throughput for the clients and potentially offer almost perfect local management for the writes in our controller. On the other hand, we cannot delay reads. That is, although we can respond immediately to a write request even if the actual write is sent to the storage device after a few seconds, reads have to be sent to the device as soon as possible. To address that we propose to keep sending reads while also sending writes but still have some intervals, where only read requests are dispatched to the storage. By doing so, firstly, we increase the read throughput, secondly, we are able to estimate the read costs accurately (as we have been doing so far) and thirdly, given that we have a good estimate for the reads we have the chance to accurately estimate the write cost to a large degree.

3.5.2 Solid-State Drives and Hybrid Storage

SSDs are becoming more and more popular partially due to their fast random access leading to higher throughput and lower latency when compared to hard disks. They are often part of hybrid storage systems playing the role of a large cache or acting as complementary storage. In addition, there are storage systems with no hard drives, which are based solely on solid-state drives. Although sequential requests are still less expensive than random ones in SSDs, the difference is much smaller than in disks depending on the SSD device model. On the other hand, reads and writes in SSDs can have a higher cost difference than in disks, with writes being more expensive. We

expect our method to work on storage systems having only SSDs without significant modifications and plan to explore that direction.

Hybrid storage systems have both traditional disks and SSDs. In order to always charge and schedule streams properly we need a method to predict whether a request or collection of requests are going to be serviced by SSDs rather than hard-drives. Knowing that during the scheduling phase with certainty and no extra knowledge about the client or setup seems impossible. Moreover, making such a prediction for writes would be harder than reads, since a write could first hit an SSD and eventually be written to the disk. In that case, the scheduling should happen as if the request is going to utilize some SSD, otherwise, the client would experience a much higher latency due to our scheduling. On the other hand, if a write request is scheduled as an SSD request, there could be many write requests ready to be dispatched to the disk, which could harm the performance guarantees for read requests. To reduce that problem, each write may happen in an SSD array in our controller and then scheduled to be dispatched to the storage system as a disk write. If the write delay needed is short enough the SSD may remain unused. Read responses may be classified as SSD-hits or disk-hits. If we assume that a stream we believe has been hitting the SSDs will keep doing so, then we can schedule future requests accordingly. If based on the response classification we have mispredicted a request in the scheduling phase, we may charge the stream for that cost difference by penalizing future requests. A possible approach for classifying responses as SSD- or disk-hits is by looking at the latency of the requests. As long as both SSD and disk queues in the storage are always busy, the disk responses are expected to have a greater

latency than the SSD responses. Given the challenges of hybrid storage systems in general and in particular write requests under such systems, our first step in the direction of supporting SSDs is the management of SSD-only storage systems.

3.6 Summary

There has been a number of attempts to provide performance guarantees for storage systems. Throughput or latency based approaches fail to provide either absolute guarantees or isolation, or both. On the other hand, there has been no work on time-based approaches for storage systems treated as black boxes. In our work we target the problem of providing isolation and absolute performance guarantees through time-based scheduling 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.

Managing the I/O performance of black box storage systems is challenging and there are multiple directions for future work. Extensions include support for SSDs, writes and Network Attached Storage. Another direction is extending QBox to cloud systems

with multiple storage devices, where more than one controller is expected to be needed.

Chapter 4

Providing Consistent Performance in Flash through Redundancy

4.1 Introduction

Virtualization and many other applications such as online analytics and transaction processing often require access to predictable, low-latency storage. Cost-effectively satisfying such performance requirements is hard due to the low and unpredictable performance of hard-drives, while storing all data in DRAM, in many cases, is still prohibitively expensive and often unnecessary. In addition, offering high performance storage in a virtualized cloud environment is more challenging due to the loss of predictability, throughput, and latency incurred by mixed workloads in a shared storage system. Given the popularity of cloud systems and virtualization, and the storage performance demands of modern applications, there is a clear need for scalable storage

systems that provide high and predictable performance efficiently, under any mixture of workloads.

Solid-state drives and more generally flash memory have become an important component of many enterprise storage systems towards the goal of improving performance and predictability. They are commonly used as large caches and as permanent storage, often on top of hard-drives operating as long-term storage. A main advantage over hard-drives is their fast random access. One would like SSDs to be the answer to predictability, throughput, latency, and performance isolation for consolidated storage in cloud environments. Unfortunately, though, SSD performance is heavily workload dependent. Depending on the drive and the workload latencies as high as 100ms can occur frequently (for both writes and reads), making SSDs multiple times slower than hard-drives in such cases. In particular, we could only find a single SSD with predictable performance which, however, is multiple times more expensive than commodity drives, possibly due to additional hardware it employs (e.g., extra DRAM).

Such read-write interference results in unpredictable performance and creates significant challenges, especially in consolidated environments, where different types of workloads are mixed and clients require high throughput and low latency consistently, often in the form of reservations. Similar behavior has been observed in previous work [CKZ09, CLZ11, MKC+12, PS12] for various device models and is well-known in the industry. Even so, most SSDs continue to exhibit unpredictability.

Although there is a continuing spread of solid-state drives in storage systems, research on providing efficient and predictable performance for SSDs is limited. In particu-

lar, most related work focuses on performance characteristics [CKZ09, CLZ11, BJB09], while other work, including [APW+08, BD10, Des12] is related to topics on the design of drives, such as wear-leveling, parallelism and the Flash Translation Layer (FTL). Instead, we use such performance observations to provide consistent performance. With regards to scheduling, FIOS [PS12] provides fair-sharing while trying to improve the drive efficiency. To mitigate performance variance, current flash-based solutions for the enterprise are often aggressively over provisioned, costing many times more than commodity solid-state drives, or offer lower write throughput. Given the fast spread of SSDs, we believe that providing predictable performance and low latency efficiently is important for many systems.

In this work we propose and evaluate a method for achieving consistent performance and low latency under arbitrary read/write workloads by exploiting redundancy. Specifically, using our method read requests experience read-only throughput and latency, while write requests experience performance at least as well as before. To achieve this we physically separate reads from writes by placing the drives on a ring and using redundancy, e.g., replication. On this ring consider a sliding window (Figure 4.1), whose size depends on the desired data redundancy and read-to-write throughput ratio. The window moves along the ring one location at a time at a constant speed, transitioning between successive locations "instantaneously". Drives inside the sliding window do not perform any writes, hence bringing read-latency to read-only levels. All write requests received while inside the window are stored in memory (local cache/DRAM) and optionally to a log, and are actually written to the drive while outside the window.

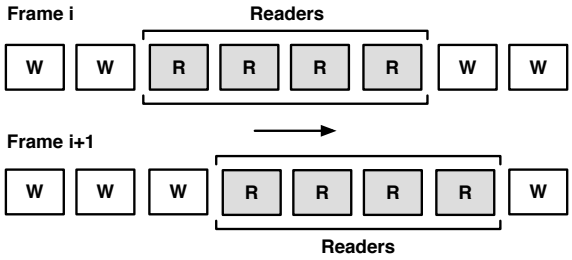


Figure 4.1: The sliding window moves along the drives. Drives inside the sliding window only perform reads and temporarily store writes in memory.

4.2 Overview

The contribution of this chapter is a design based on redundancy that provides read-only performance for reads under arbitrary read/write workloads. In other words, we provide consistent performance and minimal latency for reads while performing at least as well for writes as before. In addition, as we will see, there is opportunity to improve the write throughput through batch writing, however, this is out of the scope of this work. Instead, we focus on achieving predictable and efficient read performance under read/write workloads. We present our results in three parts. In Section 4.3, we study the performance of multiple SSD models. We observe that in most cases their performance can become significantly unpredictable and that instantaneous performance depends heavily on past history of the workload. As we coarsen the measurement granularity we see, as expected, that at some point the worst-case throughput increases and stabilizes. This point, though, is quite significant, in the order of multiple seconds. Note that we illustrate specific drive performance characteristics to motivate our design for Rails rather than present a thorough study of the performance of each drive. Based on the

above, in Section 4.4 we present Rails. In particular, we first show how to provide read/write separation using two drives and replication. Then we generalize our design to SSD arrays performing replication or erasure coding. Finally, we evaluate our method using replication under micro-benchmarks and real workload traces.

4.2.1 System Notes

As described in Section 4.4.3, the design of Rails supports both replication and erasure coding. To this date we have implemented a prototype of the described design under replication rather than erasure coding. We believe that a thorough study of Rails under erasure coding requires a more extensive evaluation and is left as future work.

For our experiments we perform direct I/O to bypass the OS cache and use Kernel AIO to asynchronously dispatch requests to the raw device. To make our results easier to interpret, we do not use a filesystem. Limited experiments on top of ext3 and ext4 suggest our method would work in those cases. Moreover, our experiments were performed with both our queue and NCQ (Native Command Queueing) depth set to 31. Other queue depths had similar effects to what is presented in [CLZ11], that is throughput increased with the queue size. Finally, the SATA connector used was of 3.0Gb/s. For all our experiments we used the following SSD models:

We chose the above drive models to develop a method, which unlike heuristics (Section 4.3.2), works under different types of drives, and especially commodity drives. A small number of recent data-center oriented models and in particular model C have placed a greater emphasis on performance stability. For the drives we are aware of, the stability

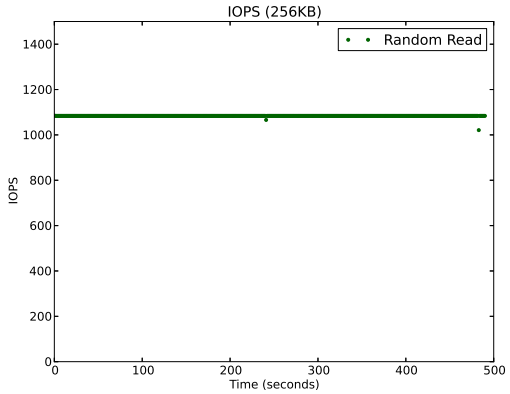
	Model	Capacity	Cache	Year
A	Intel X-25E	65GB	16MB	2008
B	Intel 510	250GB	128MB	2011
C	Intel DC3700	400GB	512MB	2012
D	Samsung 840EVO	120GB	256MB	2013

Table 4.1: Solid-state drive models used

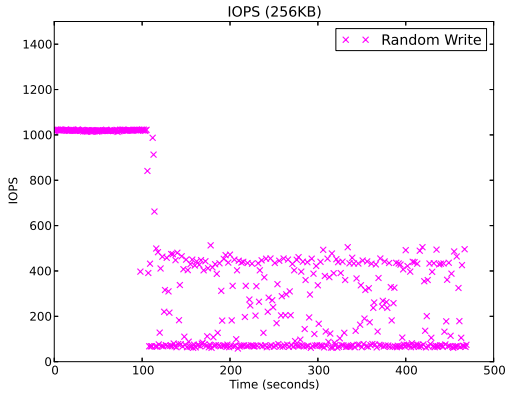
either comes at a cost that is multiple times that of commodity drives (as with model C), or is achieved by lowering the random write throughput significantly compared to other models. In particular, commodity SSDs typically have a price between \$0.5 and \$1/GB while model C has a cost close to \$2.5/GB. Most importantly, we are interested in a versatile, open solution supporting replication and erasure coding, which can be applied on existing systems by taking advantage of commodity hardware, rather than expensive black-box solutions.

4.3 Performance and Stability

Solid-state drives have orders-of-magnitude faster random access than hard-drives. On the other hand, SSDs are stateful and their performance depends heavily on the past history of the workload. The influence of writes on reads has been noted in prior work [CKZ09, CLZ11, MKC+12, PS12] for various drive models, and is widely known in the industry. We first verify the behavior of reads and writes on drive B. By running a simple read workload with requests of 256KB over the first 200GB of drive B, we see from Figure 4.2a that the throughput is high and virtually variance-free. We noticed a similar behavior for smaller request sizes and for drive A. On the other hand,



(a) The read-only workload performance has virtually no variance.

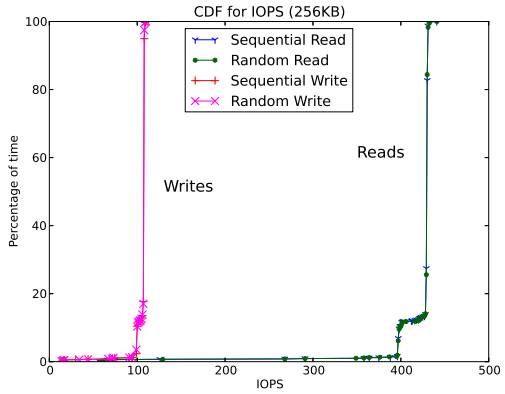


(b) When the drive has limited free space, random writes trigger the garbage collector resulting in unpredictable performance.

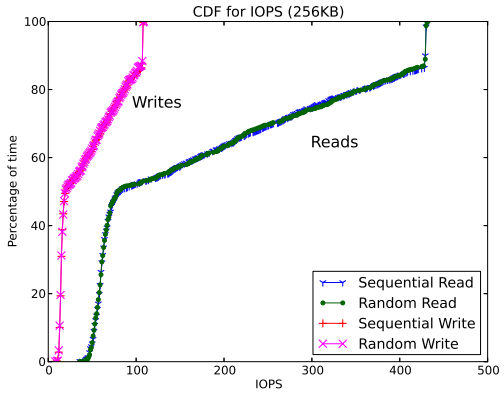
Figure 4.2: Under random writes the performance eventually drops and becomes unpredictable. (Drive B; 256KB)

performing the same experiment but with random writes gives stable throughput up to a certain moment, after which the performance degrades and becomes unpredictable (Figure 4.2b), due to write-induced drive operations.

To illustrate the interference of writes on reads, we run the following two experiments on drive B. Consider two streams performing reads (one sequential and one random), and two streams performing writes (one sequential and one random). Each read stream has a dispatch rate of 0.4 while each write stream has a dispatch rate of 0.1, i.e., for each write we send four reads. In the first experiment, all streams perform operations on the same logical range of 100MB. Figure 4.3a shows the CDF of the throughput in IOPS (input/output per second) achieved by each stream over time. We note that the performance behavior is predictable. In the second experiment, we consider the same set of streams and dispatch rates, with the difference that the requests span the first 200GB,



(a) Reads and writes over a range of 100MB lead to predictable write performance and the effect of writes on reads is small.



(b) Performing writes over 200GB leads to unpredictable read throughput due to write-induced blocking events.

Figure 4.3: The performance varies according to the writing range. (Drive B; 256KB)

instead of only 100MB. Figure 4.3b illustrates the drive performance unpredictability under such conditions. We attribute this to the garbage collector not being able to keep up, which turns background operations into blocking ones.

Different SSD models can exhibit different throughput variance for the same workload. Performing random writes over the first 50GB of drive A, which has a capacity of 65GB, initially gives a throughput variance close to that of reads (figure skipped). Still, the average performance eventually degrades to that of B, with the total blocking time corresponding to more than 60% of the device time (Figure 4.4). Finally, although there is ongoing work, even newly released commodity SSDs (e.g., drive D) can have write latency that is 10 times that of reads, in addition to the significant write variance (figure skipped).

Throughout our experiments we found that model C was the only drive with high and consistent performance under mixed read/write workloads. More specifically, after

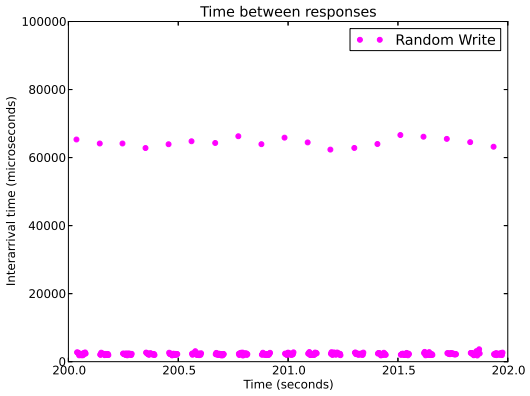


Figure 4.4: The drive blocks for over $600ms$/sec, leading to high latencies for all queued requests. (Drive A; 256KB)

filling the drive multiple times by performing random writes, we presented it with a workload having a decreasing rate of (random) writes, from 90% down to 10%. To stress the device, we performed those writes over the whole drive's logical space. From Figure 4.5, we see that the read performance remains relatively predictable throughout the whole experiment. As mentioned earlier, a disadvantage of this device is its cost, part of which could be attributed to extra components it employs to achieve this level of performance. As mentioned earlier, we are interested in an open, extensible and cheaper solution using existing commodity hardware. In addition, Rails, unlike model C, requires an amount of DRAM that is not proportional to the drive capacity. Finally, although we only tested drive C with a 3Gb/s SATA controller, we expect it to provide comparably predictable performance under 6Gb/s.

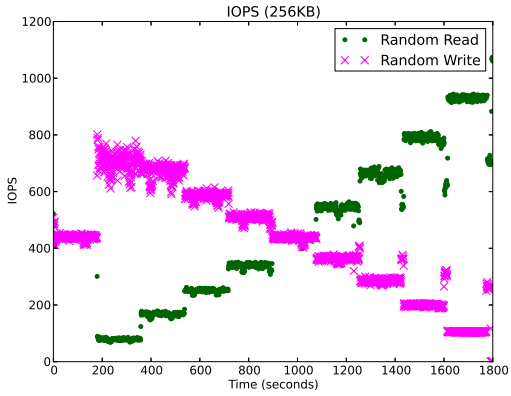


Figure 4.5: Random reads/writes at a decreasing write rate. Writes have little effect on reads. (Drive C; 256KB)

4.3.1 Performance over long periods

As mentioned in Section 4.1, writes targeting drives in read mode are accumulated and performed only when the drives start writing, which may be after multiple seconds. To that end, we are interested in the write throughput and predictability of drives over various time periods. For certain workloads and depending on the drive, the worst-case throughput can reach zero. A simple example of such workload on drive A consists of 4KB sequential writes. We noted that when the sequential writes enter a randomly written area, the throughput oscillates between 0 and 40,000 writes/sec. To illustrate how the throughput variance depends on the length of observation period, we computed the achieved throughput over different averaging window sizes, and for each window size we computed the corresponding CDF of throughputs. In Figure 4.6, we plot the 5%-ile of these throughput measurements as we increase the window size. We see that increasing the window size to about 5 seconds improves the 5%-ile, and that the increase is fast

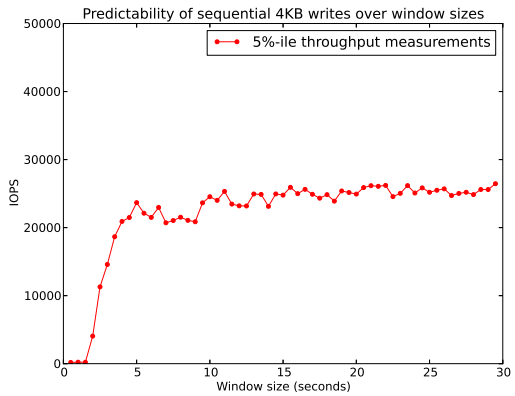


Figure 4.6: The bottom 5% throughput against the averaging window size. (Drive A; 4KB)

but then flattens out. That is, the throughput of SSDs over a window size of a few seconds becomes as predictable as the throughput over large window sizes. Drive *B* exhibits similar behavior. Finally, we found that the SSD write cache contributes to the throughput variance and although in our experiments we keep it enabled by default, disabling it improves stability but often at the cost of lower throughput, especially for small writes.

4.3.2 Heuristic Improvements

By studying drive models *A* and *B*, we found the behavior of *A*, which has a small cache, to be more easily affected by the workload type. First, writing sequentially over blocks that were previously written with a random pattern has low and unstable behavior, while writing sequentially over sequentially written blocks has high and stable performance. Although such patterns may appear under certain workloads and could be a filesystem

optimization for certain drives, we cannot assume that in general. Moreover, switching from random to sequential writes on drive A, adds significant variance.

To reduce that variance we tried to disaggregate sequential from random writes (e.g., in 10-second batches). Doing so doubled the throughput and reduced the variance significantly (to 10% of the average). On the other hand, we should emphasize that the above heuristic does not improve the read variance of drive B unless the random writes happen over a small range. This strengthens the position of not relying on heuristics due to differences between SSDs. In contrast to the above, we next present a generic method for achieving efficient and predictable read performance under mixed workloads that is virtually independent of drive model and workload history.

4.4 Efficient and Predictable Performance

In the previous section, we observed that high latency events become common under read/write workloads leading to unpredictable performance, which is prohibitive for many applications. We now present a generic design based on redundancy that when applied on SSDs provides predictable performance and low latency for reads, by physically isolating them from writes. We expect this design to be significantly less prone to differences between drives than heuristics, and demonstrate its benefits under models A and B. In what follows, we first present a minimal version of our design, where we have two drives and perform replication. In Section 4.4.3, we generalize that to support more than two drives, erasure codes, and describe its achievable throughput.

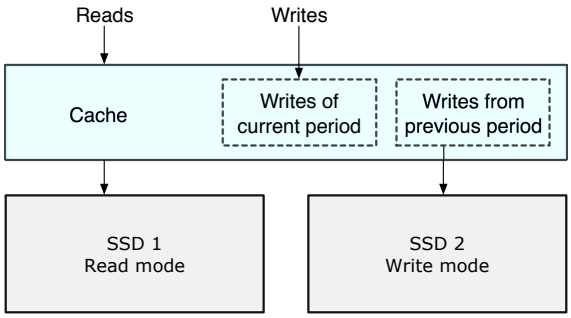


Figure 4.7: At any given time each of the two drives is either performing reads or writes. While one drive is reading the other drive is performing the writes of the previous period.

4.4.1 Basic Design

Solid-state drives have fast random access and can exhibit high performance. However, as shown in Section 4.3, depending on the current and past workloads, performance can degrade quickly. For example, performing random writes over a wide logical range of a drive can lead to high latencies for all queued requests due to write-induced blocking events (Figure 4.2b). Such events can last up to 100ms and account for a significant proportion of the device's time, e.g., 60% (Figure 4.4). Therefore, when mixing read and write workloads, reads also block considerably, which can be prohibitive.

We want a solution that provides read-only performance for reads under mixed workloads. SSD models differ from each other and a heuristic solution working on one model may not work well on another (Section 4.3.2). We are interested in an approach that works across various models. We propose a new design based on redundancy that achieves those goals by physically isolating reads from writes. By doing so, we nearly eliminate the latency that reads have to pay due to writes, which is crucial for many

low-latency applications, such as online analytics. Moreover, we have the opportunity to further optimize reads and writes separately. Note that using a single drive and dispatching reads and writes in small time-based batches and prioritizing reads as in [PS12], may improve the performance under certain workloads and SSDs. However, it cannot eliminate the frequent blocking due to garbage collection under a generic workload.

The basic design, illustrated in Figure 4.7, works as follows: given two drives D_1 and D_2, we separate reads from writes by sending reads to D_1 and writes to D_2. After a variable amount of time $T \geq T_{min}$, the drives switch roles with D_1 performing writes and D_2 reads. When the switch takes place, D_1 performs all the writes D_2 completed that D_1 has not, so that the drives are in sync. We call those two consecutive time windows a *period*. If D_1 completes syncing and the window is not yet over ($t < T_{min}$), D_1 continues with new writes until $t \geq T_{min}$. In order to achieve the above, we place a cache on top of the drives. While the writing drive D_w performs the old writes all new writes are written to the cache. In terms of the write performance, by default, the user perceives write performance as perfectly stable and half that of a drive dedicated to writes. As will be discussed in Section 4.4.2.3, the above may be modified to allow new writes to be performed directly on the write drive, in addition to the cache, leading to a smaller memory footprint. In what follows we present certain properties of the above design and in Section 4.4.3 a generalization supporting an arbitrary number of drives, allowing us to trade read and write throughput, as well as erasure codes.

4.4.2 Properties and Challenges

4.4.2.1 Data consistency & fault-tolerance

All data is always accessible. In particular, by the above design, reads always have access to the latest data, possibly through the cache, independently of which drive is in read mode. This is because the union of the cache with any of the two drives always contains exactly the same (and latest) data. By the same argument, if any of the two drives fail at any point in time, there is no data loss and we continue having access to the latest data. While the system operates with one drive, the performance will be degraded until the replacement drive syncs up.

4.4.2.2 Cache size

Assuming we switch drive modes every T seconds and the write throughput of each drive is w MB/s, the cache size has to be at least $2T \times w$. This is because a total of $T \times w$ new writes are accepted while performing the previous $T \times w$ writes to each drive. We may lower that value to an average of $3/2 \times T \times w$ by removing from memory a write that is performed to both drives. As an example, if we switch every 10 seconds and the write throughput per drive is 200MB/s, then we need a cache of $T \times 2w = 4000$MB.

The above requires that the drives have the same throughput on average (over T seconds), which is reasonable to assume if T is not small (Figure 4.6). In general though, the write throughput of an SSD can vary in time depending on past workloads. This implies that even drives of the same model may not always have identical performance,

It follows that a drive in our system may not manage to flush all its data while in write mode.

In a typical storage system an array of replica drives has the performance of its slowest drive. We want to retain that property while providing read/write separation to prevent the accumulated data of each drive from growing unbounded. To that end we ensure that the system accepts writes at the rate of its slowest drive through throttling. In particular, in the above 2-drive design, we ensure that the rate at which writes are accepted is $w/2$, i.e., half the write throughput of a drive. That condition is necessary to hold over large periods since replication implies that the write throughput, as perceived by the client, has to be half the drive throughput. Of course, extra cache may be added to handle bursts. Finally, the cache factor can become $w \times T$ by sacrificing up to half the read throughput if the syncing drive retrieves the required data from D_r instead of the cache. However, that would also sacrifice fault-tolerance and given the low cost of memory it may be an inferior choice.

4.4.2.3 Power failure

In an event of a power failure our design as described so far will result in a data loss of $T \times w$ MBs, which is less than 2GB in the above example. Shorter switching periods have a smaller possible data loss. Given the advantages of the original design, limited data loss may be tolerable by certain applications, such as applications streaming data that is not sensitive to small, bounded losses. However, other applications may not tolerate a potential data loss. To prevent data loss, non-volatile memory can be used

to keep the design and implementation simple while retaining the option of predictable write throughput. As NVRAM becomes more popular and easily available, we expect that future implementations of Rails will assume its availability in the system.

In the absence of NVRAM, an approach to solve the power-failure problem is to turn incoming writes into synchronous ones. Assuming we split the bandwidth fairly between cached and incoming writes, the incoming $T \times w/2$ MBs of data is then written to D_w in addition to the cache. In that case, the amount of cache required reduces to an average of $T \times w/2$. In the above approach we first perform the writes of the previous period, and then any incoming writes. In practice, to avoid write starvation for the incoming writes, we can coalesce writes while sharing the bandwidth between the writes of the previous period and the incoming ones. As in write-back caches, and especially large flash caches, a challenge with the above is preserving write-ordering. Preserving the write order is required by a proportion of writes in certain applications (e.g., most file systems). To achieve this efficiently, a method such as ordered write-back [KMR$^+$13] may be used, which preserves the original order during eviction by using dependency graphs. On the other hand, not all applications require write order preservation, including NoFS [CSADAD12]. Note that other approaches such as performing writes to a permanent journal may be possible, especially in distributed storage systems where a separate journal drive often exists on each node. Finally, after the power is restored, we need to know which drive has the latest data, which can be achieved by storing metadata on D_w. The implementation details of the above are out of the scope of this work.

4.4.2.4 Capacity and cost

Doubling the capacity required to store the same amount of data appears as doubling the storage cost. However, there are reasons why this is not entirely true. First, cheaper SSDs may be used in our design because we are taking away responsibility from the SSD controller by not mixing reads and writes. In other words, any reasonable SSD has high and stable read-only performance, and stable average write performance over large time intervals (Figure 4.6). Second, in practice, significant over-provisioning is already present to handle the unpredictable performance of mixed workloads. Third, providing a drive with a write-only load for multiple seconds instead of interleaving reads and writes is expected to improve its lifetime. Finally, and most importantly, Rails can take advantage of the redundancy that local and distributed systems often employ for fault-tolerance. In such cases, the hardware is already available. In particular, we next generalize the above design to more than two drives and reduce the storage space penalty through erasure codes while providing read/write separation.

4.4.3 Design Generalization

We now describe the generalization of the previous design to support an arbitrary number of identical SSDs and reader-to-writer drive ratios through erasure coding.

Imagine that we want to build a fault-tolerant storage system by using N identical solid-state drives connected over the network to a single controller. We will model redundancy as follows. Each object O stored will occupy $q|O|$ space, for some $q > 1$. Having fixed q, the best we can hope in terms of fault-tolerance and load-balancing is

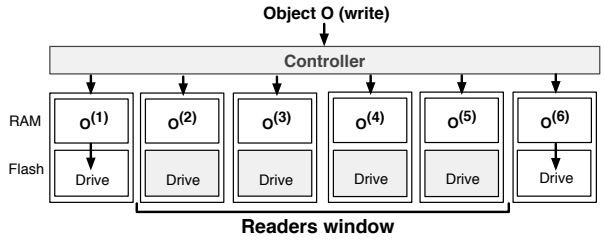


Figure 4.8: Each object is obfuscated and its chunks are spread across all drives. Reading drives store their chunk in memory until they become writers.

that the $q|O|$ bits used to represent O are distributed (evenly) among the N drives in such a way that O can be reconstituted from any set of N/q drives. A natural way to achieve load-balancing is the following. To handle a write request for an object O, each of the N drives receives a write request of size $|O| \times q/N$. To handle a read request for an object O, each of N/q randomly selected drives receives a read request of size $|O| \times q/N$.

In the simple system above, writes are load-balanced deterministically since each write request places exactly the same load on each drive. Reads, on the other hand, are load-balanced via randomization. Each drive receives a stream of read and write requests whose interleaving mirrors the interleaving of read/write requests coming from the external world (more precisely, each external-world write request generates a write on each drive, while each external-world read request generates a read with probability $1/q$ on each drive.)

As discussed in Section 4.3, in the presence of read/write interleaving the write latency "pollutes" the variance of reads. We would like to avoid this latency contamination and bring read latency down to the levels that would be experienced if each drive was

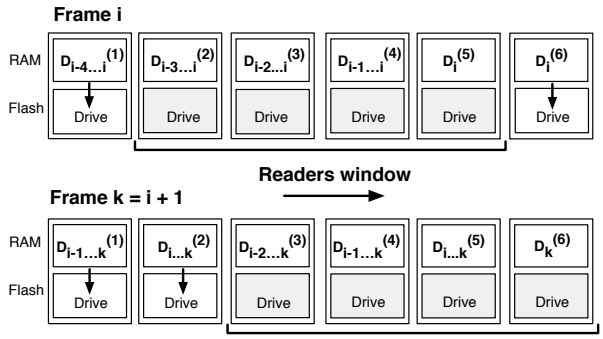


Figure 4.9: Each node m accumulates the incoming writes across frames $f \dots f'$ in memory, $D^{(m)}_{f \dots f'}$. While outside the reading window nodes flush their data.

read-only. To this effect, we propose making the load-balancing of reads partially deterministic, as follows. Place the N drives on a ring. On this ring consider a sliding window of size s, such that $N/q \leq s \leq N$. The window moves along the ring one location at a time at a constant speed, transitioning between successive locations "instantaneously". The time it takes the window to complete a rotation is called the period P. The amount of time, P/N, that the window stays in each location is called a frame.

To handle a write request for an object O, each of the N drives receives one write request of size $|O| \times q/N$ (Figure 4.8, with $N = 6$ and $q = 3/2$). To handle a read request for an object O, out of the s drives in the window N/q drives are selected at random and each receives one read request of size $|O| \times q/N$. In other words, the only difference between the two systems is that reads are not handled by a random subset of nodes per read request, but by random nodes from a coordinated subset which changes only after handling a large number of read requests.

In the new system, drives inside the sliding window do not perform any writes, hence

bringing read-latency to read-only levels. Instead, while inside the window, each drive stores all write requests received in memory (local cache/DRAM) and optionally to a log. While outside the window, each drive empties all information in memory, i.e., it performs the actual writes (Figure 4.9). Thus, each drive is a read-only drive for $P/N \times s \geq P/q$ successive time units and a write-only drive for at most $P(1 - 1/q)$ successive time units.

Clearly, there is a tradeoff regarding P. The bigger P is, the longer the stretches for which each drive will only serve requests of one type and, therefore, the better the read performance predictability and latency. On the other hand, the smaller P is, the smaller the amount of memory needed for each drive.

4.4.3.1 Throughput Performance

Let us now look at the throughput difference between the two systems. The first system can accommodate any ratio of read and write loads, as long as the total demand placed on the system does not exceed capacity. Specifically, if r is the read-rate of each drive and w is the write-rate of each drive, then any load such that $R/r + Wq/w \leq N$ can be supported, where R and W are the read and write loads, respectively. In this system read and write workloads are mixed.

In the second system, s can be readily adjusted on the fly to any value in $[N/q, N]$, thus allowing the system to handle any read load up to the maximum possible rN. For each such choice of s, the other $N - s$ drives provide write throughput, which thus ranges between 0 and $W_{\text{sep}} = w \times (N - N/q)/q = w \times N(q-1)/q^2 \leq wN/4$. (Note

that taking $s = 0$ creates a write-only system with optimal write-throughput wN/q.)

We see that as long as the write load $W \leq W_{\text{sep}}$, by adjusting s the system performs perfect read/write separation and offers the read latency and predictability of a read-only system. We expect that in many shared storage systems, the reads-to-writes ratio and the redundancy are such that the above restriction is satisfied in the typical mode of operation. For example, for all $q \in [3/2, 3]$, having $R > 4W$ suffices.

When $W > W_{\text{sep}}$ some of the dedicated read nodes must become read/write nodes to handle the write load. As a result, read/write separation is only partial. Nevertheless, by construction, in every such case the second system performs at least as well as the first system in terms of read-latency. On the other hand, when performing replication ($q = N$) we have complete flexibility with respect to trading between read and write drives (or throughput) so we never have to mix reads and writes at a steady state.

Depending on the workload, ensuring that we adjust fast enough may require that drives switch modes quickly. In practice, to maintain low latency drives should not be changing mode quickly, otherwise reads could be blocked by write-induced background operations on the drive. Those operations could be triggered by previous writes. For example, we found that model B can switch between reads and writes every 5 seconds almost perfectly (Figure 4.10) while model A exhibits few blocking events that affect the system predictability (Figure 4.13a) when switching every 10 seconds.

We conclude that part of the performance predictability provided by Rails may have to be traded for the full utilization of every drive under fast-changing workloads. The strategy for managing that trade-off is out of the scope of this work. Having said that,

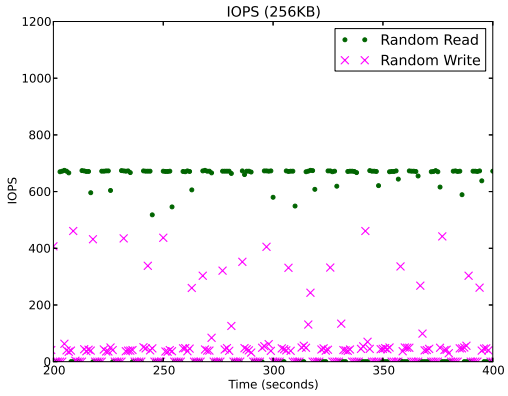


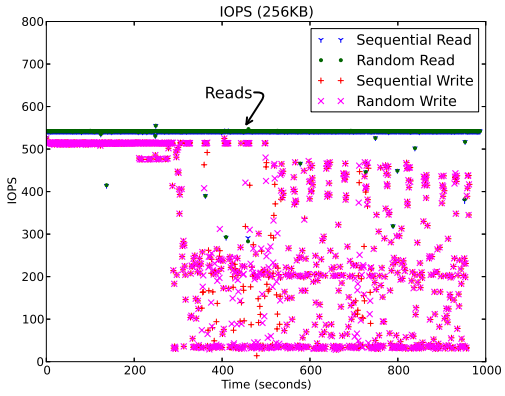
Figure 4.10: Switching modes as frequent as every 5 seconds creates little variance on reads. (Drive B; 256KB)

we expect that if the number of workloads seen by a shared storage is large enough, then the aggregate behavior will be stable enough and Rails would only see minor disruptions.

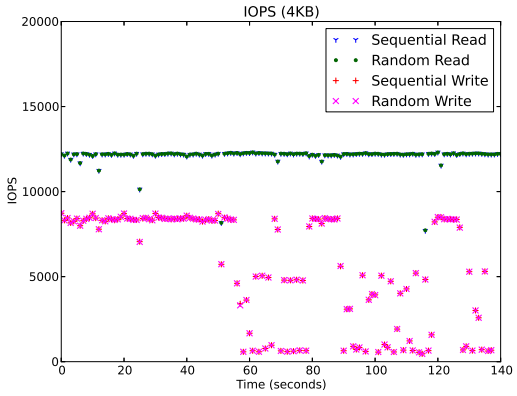
4.4.3.2 Feasibility and Efficiency

Consider the following four dimensions: storage space, reliability, computation and read performance variance. Systems performing replication have high space requirements. On the other hand, they offer reliability, and no computation costs for reconstruction. Moreover, applying our method improves their variance without affecting the other quantities. In other words, the system becomes strictly better.

Systems performing erasure coding have smaller storage space requirements and offer reliability, but add computation cost due to the reconstruction when there is a failure. Adding our method to such systems improves the performance variance. The price is that reading entails reconstruction, i.e., computation.



(a) The read streams throughput remains constant at the maximum possible while writes perform as before. (Drive B; 256KB)



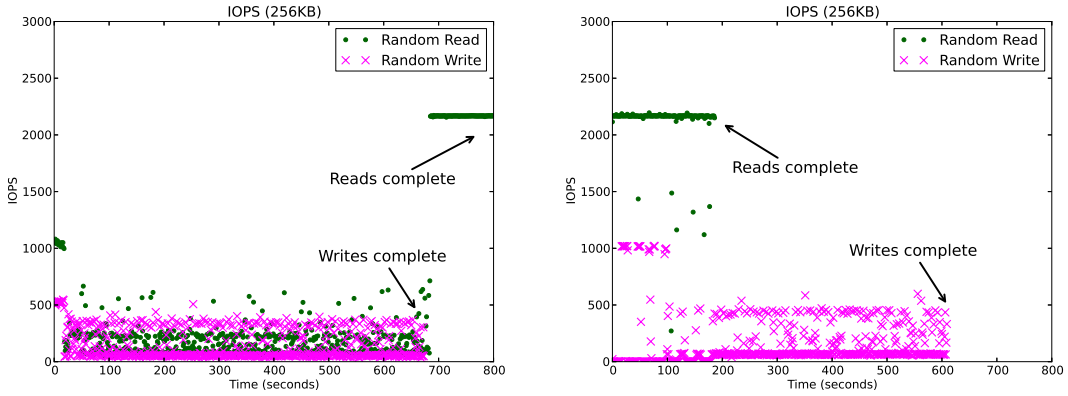
(b) The read throughput remains stable at its maximum performance. (Drive B; 4KB)

Figure 4.11: Using Rails to physically separate reads from writes leads to a stable and high read performance.

Nevertheless, reconstruction speed has improved [PGM13] while optimizing degraded read performance is a possibility, as has been done for a single failure [KBP+12]. In practice, there are SSD systems performing reconstruction frequently to separate reads from writes, illustrating that reconstruction costs are tolerable for small N (e.g., 6). We are interested in the behavior of such systems under various codes as N grows, and identifying the value of N after which read/write separation becomes inefficient due to excessive computation or other bottlenecks.

4.4.4 Experimental Evaluation

We built a prototype of Rails as presented in Sections 4.4.1 and 4.4.3 using replication and verified that it provides predictable and efficient performance, and low latency for reads under read/write workloads using two and three drives. For simplicity, we ignored



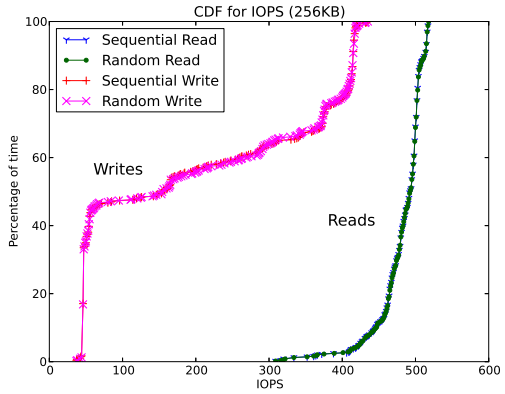
(a) Mixing reads and writes on all 3 drives leads to slow performance for reads until writes complete.

(b) Using Rails to separate reads from writes across drives nearly eliminates the interference of writes on reads.

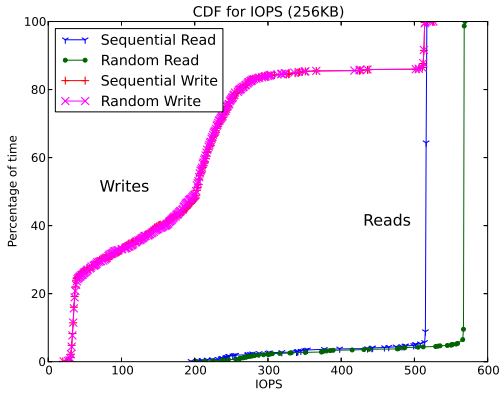
Figure 4.12: Client throughput under 3-replication, (a) without Rails, (b) with Rails. (Drive B; 256KB)

the possibility of cache hits or overwriting data still in the cache and focused on the worst-case performance. In what follows, we consider two drives that switch roles every $T_{min} = 10$ seconds. For the first experiment we used two instances of drive B. The workload consists of four streams, each sending requests of 256KB as fast as possible. From Figure 4.11a, we see that reads happen at a total constant rate of 1100 reads/sec. and are not affected by writes. Writes however have a variable behavior as in earlier experiments, e.g., Figure 4.2b. Without Rails reads have unstable performance due to the writes (Figure 4.3b).

Increasing the number of drives to three, and using the sliding window technique (Section 4.4.3), provides similar results. In particular, we set the number of read drives (window size) to two and therefore had a single write drive at a time. Figure 4.12a shows the performance without Rails is inconsistent when mixing reads and writes. In particular,



(a) Using Rails, the read throughput of drive A is mostly predictable, with a small variance due to writes after each drive switch.

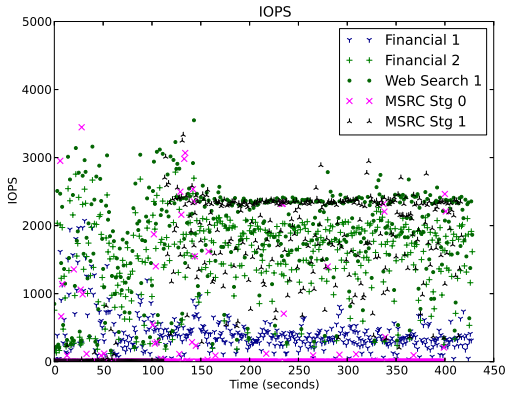


(b) Using Rails, on drive B we can provide predictable performance for reads more than 95% of the time without any heuristics.

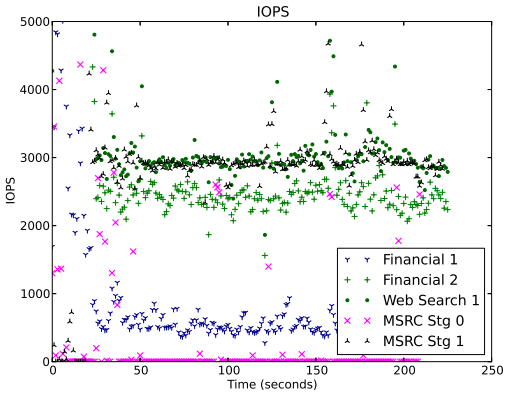
Figure 4.13: IOPS CDF using Rails on (a) drive A, (b) drive B. (256KB)

in the first figure, the reads are being blocked by writes until writes complete. On the other hand, when using Rails the read performance is unaffected by the writes, and both the read and write workload finish earlier (Figure 4.12b). Note that when the reads complete, all three drives start performing writes.

Although we physically separate reads from writes, in the worst-case there can still be interference due to remaining background work right after a window shift. In the previous experiment we noticed little interference, which was partly due to the drive itself. Specifically, from Figure 4.13b we see that reads have predictable performance around 95% of the time, which is significantly more predictable than without Rails (Figure 4.3b). Moreover, in Figure 4.13a we see that drive A has predictable read performance when using Rails, though reads do not appear as a perfect line, possibly due to its small cache. Since that may also happen with other drives we propose letting the write drive idle before each shift in order to reduce any remaining background work.



(a) Without Rails, reads are blocked by writes (not shown) making read performance unpredictable.

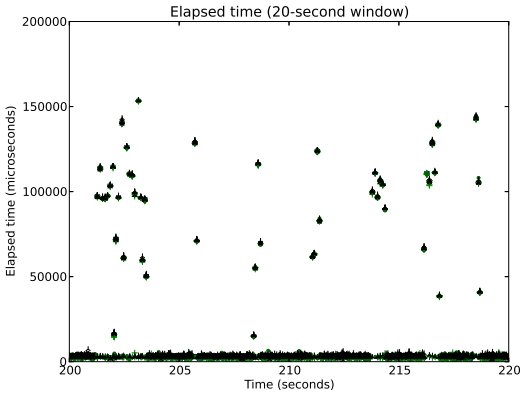


(b) With Rails, reads are not affected by writes (not shown).

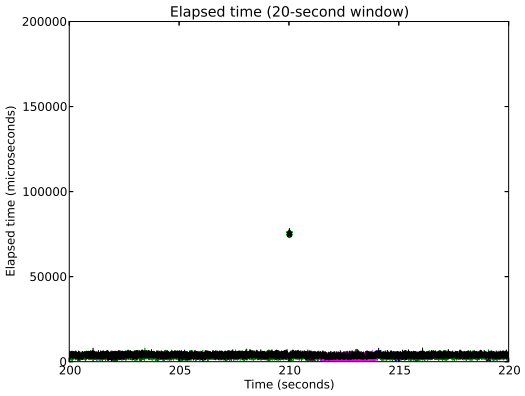
Figure 4.14: Read throughput (a) without Rails, (b) with Rails, under a mixture of real workloads. (Drive B)

That way, we found that the interference becomes minimal and 99% of the time the throughput is stable. If providing QoS, that idle time can be charged to the write streams, since they are responsible for the blocking. Small amounts of interference may be acceptable, however, certain users may prefer to sacrifice part of the write throughput to further reduce the chance of high read latency.

The random write throughput achieved by the commodity drives we used (models A, B, and D) drops significantly after some number of operations (Figure 4.2b). Instead, model C, which is more expensive as discussed earlier, retains its write performance. We believe there is an opportunity to increase the write throughput in Rails for commodity drives through batch writing, or through a version of SFS [14] adapted to redundancy. That is because many writes in Rails happen in the background.



(a) Without Rails, the garbage collector blocks reads for tens of milliseconds, or for 25% of the device time.



(b) With Rails, reads are virtually unaffected by writes - they are blocked for less than %1 of the time.

Figure 4.15: High-latency events (a) without Rails, (b) with Rails, using traces of real workloads. (Drive B)

4.4.5 Evaluation with Traces

We next evaluate Rails with two drives (of model B) and replication using the Stg dataset from the MSR Cambridge Traces [NDR08], OLTP traces from a financial institution and traces from a popular search engine [uma]. Other combinations of MSRC traces gave us similar results with respect to read/write isolation and skip them. For the following experiments we performed large writes to fill the drive cache before running the traces. Evaluating results using traces can be more challenging to interpret due to request size differences leading to a variable throughput even under a storage system capable of delivering perfectly predictable performance.

In terms of read/write isolation, Figure 4.14a shows the high variance of the read throughput when mixing reads and writes under a single device. The write plots for both cases are skipped as they are as unstable as Figure 4.14a. Under the same work-

load our method provides predictable performance (Figure 4.14b) in the sense that high-latency events become rare. To clearly see that, Figure 4.15a focuses on twenty arbitrary seconds of the same experiment and illustrates that without Rails there are multiple response times in the range of 100ms. Looking more closely, we see that about 25% of the time reads are blocked due to the write operations. On the other hand, from Figure 4.15b we see that Rails nearly eliminates the high latencies, therefore providing read-only response time that is low and predictable, almost always.

4.5 Summary

The performance of SSDs degrades and becomes significantly unpredictable under demanding read/write workloads. In this chapter, we introduced Rails, an approach based on redundancy that physically separates reads from writes to achieve read-only performance in the presence of writes. Through experiments, we demonstrated that under replication, Rails enables efficient and predictable performance for reads under read/write workloads. A direction for future work is studying the implementation details of Rails regarding write order preservation in the lack of NVRAM. Finally, we plan to study the scalability of Rails using erasure codes, as well as its application on peta-scale distributed flash-only storage systems, as proposed in [SWA+13].

Chapter 5

Erasure Coding and Read/Write Separation in Flash

5.1 Introduction

Flash memory and solid-state drives in particular, have become a standard component in enterprise storage, where they are commonly used as a large cache, and in some cases as primary storage. Solid-state drives provide a good balance between cost and performance, and in that respect may be placed between DRAM and hard-drives. However, as demonstrated in Chapter 4 and in previous work [CKZ09, CLZ11, MKC+12, PS12, SAW+14], the performance of SSDs is workload dependent and can be inconsistent. In particular, their performance can degrade due to writes leading to high read latencies in read/write workloads. Furthermore, this effect is amplified in SSD arrays, where the latency of a read request takes the maximum over the latencies of multiple drives. It

follows that clients may experience high read latencies due to writes at an increasing rate as the array size grows.

To eliminate the performance variance of reads due to writes, in Chapter 4 we proposed Flash on Rails [SAW+14], a system for enabling consistent performance in flash storage by physically separating reads from writes through redundancy. Rails supports replication and erasure coding, and has already been evaluated under replication. Although replication can be used as a redundancy method to eliminate read variance, in general it is not a cost-effective or performant approach for scaling an array of drives. That is mainly due to the storage space overhead and the write throughput being equal to at most a single drive independently of the array size. Instead, erasure coding is more space-efficient, and provides higher write throughput since objects are not replicated across all drives.

In this chapter we focus on the applicability of erasure coding on Rails, through a new system called eRails. In particular, we explore the computational cost of erasure coding, its effect on the throughput observed by clients, and the scalability of eRails. Following the Rails design, we maintain a set of k dedicated readers and m dedicated writers. To perform a read, we read k data chunks out of which we reconstruct the original data through decoding. Note that decoding entails computational cost, which has to be low enough to prevent computation from becoming the bottleneck. We find that using commodity hardware, the computational cost grows rapidly in the array size but has no observable effect by the client for medium-sized arrays. Finally, to construct large-scale arrays efficiently, we create overlapping logical redundancy groups, repre-

sented as hypergraphs. Those hypergraphs allow us to generate relatively small groups that maintain a low coding cost due to their bounded size while enabling read/write separation.

In the above we do not specify the scale at which we could scale Rails. Theoretically, our grouping technique allows us to scale Rails theoretically indefinitely. In practice, to have a peta-scale storage system, it is reasonable, if not necessary, to employ multiple storage nodes. To that end, we discuss the different scenarios where Rails can be applied to, from a local node to a distributed storage system. and discuss certain challenges, such as time synchronization, that appear in a large-scale distributed storage system. Finally, as an example, we highlight how Rails could be integrated into Ceph.

5.2 Overview

The contribution of this chapter is a design allowing us to efficiently scale Rails to an arbitrary number of drives when using erasure codes. To that end, we study the applicability of erasure coding in read/write separation with respect to its computational cost. We present our results in three parts. In Section 5.4, we present an analysis of the achievable throughput using Rails and erasure coding. We find that the maximum write throughput is attained when the number of readers equals the number of writers ($k = m$). Increasing the number of writers further decreases the write throughput due to the higher amount of redundancy required to maintain read/write separation. In Section 5.5, we look into the computational overhead of erasure codes (without using SSDs) in

the context of read/write separation. In particular, we focus on the case where half the drives are unavailable ($k = m$), because the "unavailable" drives in Rails are those performing writes. We find that when $k = m$ the erasure coding overhead decreases quickly in k, however, the decoding throughput is still high enough in comparison to that of an SSD for multiple values of k.

In Section 5.6 we first look into the read throughput achieved with and without decoding. We find that the read throughput achieved is the same as without decoding up to (and including) six reading drives. Following that result, we present a design that allows us to increase the number of drives proportionally to the computational cost by overlapping multiple logical drive arrays while maintaining read/write separation. Finally, in Section 5.7 we evaluate eRails with respect to read/write separation and the total read throughput it achieves. We find that employing erasure codes has no negative effect on the performance consistency as soon as the size of a single array, i.e., without grouping, does not become too large, e.g., more than ten drives.

5.2.1 System Notes

For our experiments we used a single node with an Asus P6T motherboard, an Intel Core i7 CPU at 2.67MHz (with 4 cores), and 12GB of DRAM. Because the SATA throughput of our motherboard could only reach about 800MB/s in total, we used three PCIe to SATA cards to allow every drive to operate at a bandwidth of about 250MB/s. Reads and writes are performed using direct I/O to bypass the OS cache and we use Kernel AIO to asynchronously dispatch requests to the raw devices.

We added erasure coding support to Rails by integrating the Jerasure [PG14] open-source library with SIMD support enabled [PGM13]. Moreover, we decided to use the Reed-Solomon Vandermonde method because its decoding performance with SIMD support appears higher than that of other codes in the same library. Finally, for all our experiments that required an SSD, we used the Intel 510 model. Similar results are expected for different models (Chapter 4). In what follows, we provide the background required for the rest of the chapter on read/write separation and erasure coding.

5.3 Background

To solve the problem of high latency under read/write workloads one may physically separate reads from writes as described in Rails (Chapter 4). In this work, we apply erasure coding to Rails and demonstrate its performance as well as the incurred computational cost while maintaining read/write separation. In what follows, we describe how erasure coding is applied on Rails to reduce the storage overhead of replication.

5.3.1 Erasure Coding and Separation

Erasure codes allow us to trade performance for fault-tolerance without the space overhead of replication. Given N drives, we denote by m the number of failures supported by the system. Each object O stored occupies $q|O|$ space, for some $q > 1$. The $q|O|$ bits used to represent O are distributed (evenly) among the N drives in such a way that O can be reconstituted from any set of N/q drives.

As mentioned earlier, in the context of Rails k denotes the number of readers and

$m = N - k$ the number of writers. In eRails there are as many writers as the maximum number of tolerable failures. More precisely, the number of writers is bounded by m, since at least k drives are required to read an object (by reconstructing it). For example, when $N = 6$, $k = 3$ and $m = 3$, an object O of 3MB would be obfuscated to 6MB ($q = 2$) and each drive would store 1MB. In the same example, we can tolerate up to three "failures" since reading O requires any $N/q = 3$ drives. Note that if we limit the read drives to k, each read entails computation due to decoding. In the context of Rails, using erasure coding allows us to reduce the storage penalty of replication and enable a higher write throughput in arrays of more than two drives. In what follows, we provide an analysis of the achievable write throughput of eRails and note that setting $k = m$ maximizes the achievable write throughput. Finally, for the purposes of this work, we assume there are no true drive failures. In the case of a true failure, we may start mixing reads and writes on a subset of the devices or plan ahead by having a set of spare drives.

5.4 Achievable Throughput

We now look into the achievable throughput under read/write separation and erasure codes. Let m and k be the number of drives writing and reading, respectively. The sustainable system write throughput grows in the number of write drives, until $k = m$. We now explain that relation. Let $q > 1$ be the obfuscation factor defined as $q = (k + m)/k$. Since a drive is in write mode for m time units, each writer is in read mode

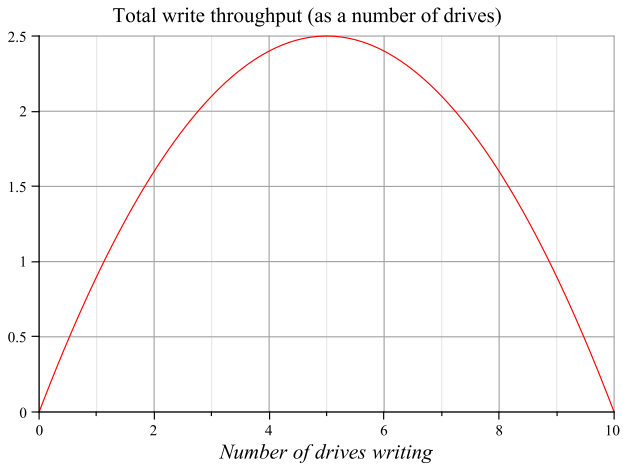


Figure 5.1: The achievable write throughput of eRails peaks when the number of readers equals the number of writers, i.e., when $k = m$.

for k time units. Let W be the amount of writes to be supported by the system. Due to the obfuscation effect, internally the system needs to support qW amount of writes. Hence, each writer is given $qW/(k+m)$ amount of writes per time unit for a total of qW. Since each drive has m time units to perform qW amount of writes, we require that $qW \leq mw$, or $W \leq kmw/(k+m)$. Setting $N = k+m$, we get $W \leq (N-m)kw/N$, concluding that the maximum write throughput is attained when $k = m$, i.e., when we have as many read as write drives.

Figure 5.1 shows the achievable write throughput when $N = 10$, against m. From the same figure we see that the maximum throughput is attained when $k = m = 5$ and is equal to $1/4$ of the maximum possible (if there were no reads). If a higher write throughput is required, mixing reads and writes on some drives may be an option. Other practical approaches for throughput improvement may be possible but not studied here. In what follows we study the encoding/decoding performance when $k = m$,

irrespectively of the storage device.

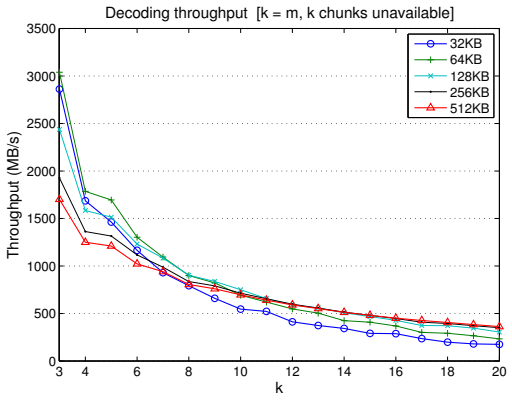
5.5 Erasure Coding Overhead

The sliding window (Section **??**) contains k drives all of which serve reads. To read an object O out of those k drives it is required to perform decoding, i.e., computation. It follows that eRails must perform a decoding operation for every read request, so the computational cost has to be small enough to not become a bottleneck. In other words, the decoding (and encoding) processes must be able to keep up with the drives performance or eRails will reduce the total storage throughput. Note that in this work, we assume that reconstruction is always required to avoid occasional periods with higher throughput when systematic codes are used.

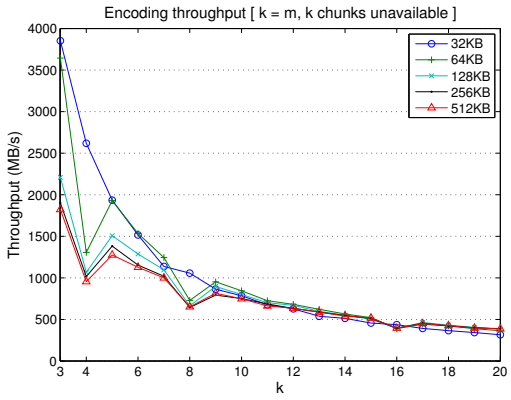
5.5.1 Decoding Throughput

In this section we study the computational cost of decoding with a focus on a large number of "failures". Based on the throughput analysis from Section 5.4 we know that the maximum achievable throughput is reached when $k = m$. In that case, half the drives are considered unavailable for reading (instead they are writing) while the array size becomes $2k$. In what follows we consider the case where $k = m$ and experimentally demonstrate the computational cost for various values of k.

First, we consider a single thread performing decoding operations over multiple values of k. From Figure 5.2a we observe that the decoding throughput quickly decreases in k until about $k = 4$, after which the decrease slows down. Note that for small values



(a) Decoding throughput. (256KB)

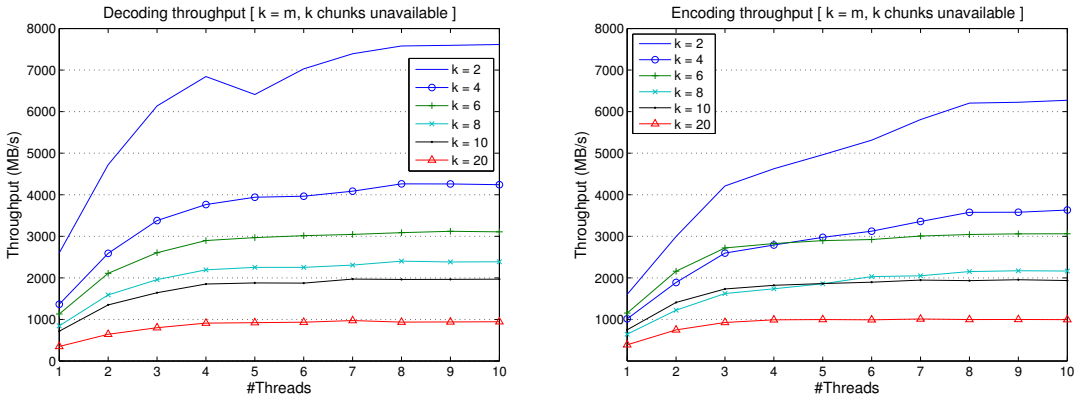


(b) Encoding throughput. (256KB)

Figure 5.2: Encoding/decoding throughput against k, for various request sizes using a single thread. The throughput drops quickly in k. (256KB)

of k, e.g., $k = 3$, the throughput is at least 2GB/s for most request sizes. Given that $k = m$ we have $q = 2$, so to saturate 2GB/s worth of decoding we require a total read throughput of 4GB/s from the drives. Assuming each drive achieves a maximum read throughput of 512MB/s, we would need to have $k = 8$ readers to saturate a single processor (if the decoding cost remained the same). Therefore, when $k = 3$ we consume 3/8 of that computational power. Given that we used a single core, the cost for small k values appears modest compared to the computational power of modern commodity systems. Figure 5.2b shows the encoding performance, which has a similar trend to decoding.

To improve the coding throughput we may use multiple threads for decoding (and encoding). Figure 5.3a shows that the decoding throughput scales up to the number of threads, depending on the value of k. In particular, the larger the k, the sooner the throughput flattens. Since our machine has four cores (and hyper threading), the



(a) Decoding throughput.

(b) Encoding throughput.

Figure 5.3: Encoding/decoding throughput for various values of k in the number of threads. (256KB)

increase in throughput after four threads is small, and the throughput flattens out soon. The above suggests that the decoding throughputs scales well, but not particularly well (not linearly) in the number of cores, potentially leaving space for improvement. The encoding throughput as shown in Figure 5.3b is similar to decoding, with the difference that it takes lower values for $k \leq 4$.

Although adding more threads improves the total decoding throughput, the cost is disproportional to k for non-small k values. For example, the decoding throughput when $k = 4$ is around 1600MB/s, whereas the throughput when $k = 12$ is around 600MB/s. If we had three arrays of $k = 4$, and therefore required three times the computation, we would achieve up to $3 \times 1600 = 4800$MB/s, instead of $3 \times 600 = 1800$MB/s. Ignoring any potential benefits of a large array, this example demonstrates that building many smaller arrays appears more efficient in terms of the computational cost.

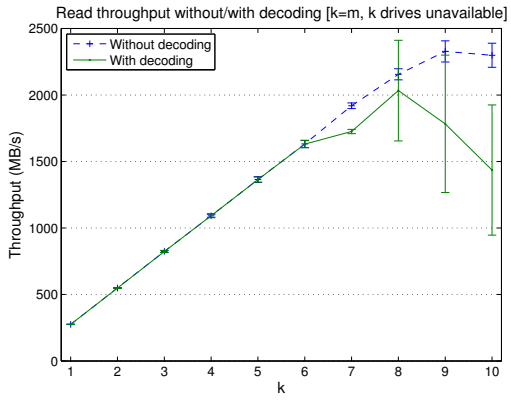


Figure 5.4: Read throughput using a single solid-state drive and varying k. ($k = m$; 128 KB)

5.6 Scaling and Computational Cost

5.6.1 Throughput after Decoding

In the previous section we saw that the computational cost of coding increases in k, i.e., as we add read and write drives. Here we look into the effect of that computational overhead to the actual read throughput. (The write overhead is similar.) Figure 5.4 shows the read throughput of 128KB requests (using actual drives) as we increase k, with and without decoding to illustrate the computational overhead. As expected, there is a point where adding more devices, i.e., increasing k, leads to lower rather than equal or higher read throughput. In our experiments that point is at $k = 9$. In addition, for $k > 6$, the throughput increase slows down and becomes lower than that without decoding. Note that even without decoding, the read throughput flattens out soon after $k = 9$ showing that decoding is not adding significant overhead unless k becomes large. Of course, the smaller the requests the higher the CPU utilization

becomes, irrespectively of the coding cost. From the above we conclude that growing an array by simply increasing k requires a disproportional increase in computational power, otherwise the array throughput drops considerably. For example, from Figure 5.4 we see that when $k = 3$ the read throughput is 820MB/s. For $k = 9$, we achieve 1,780MB/s, instead of $3 \times 820 = 2460$MB/s. The above is a consequence of the decoding throughput decreasing quickly in k (until around $k = 10$) as shown in Figure 5.2a, and the fact that CPU resources are limited.

5.6.2 Scaling through Grouping

In the previous sections, we saw that scaling the read throughput in k requires a disproportional increase of computational power. Instead, we would want for the decoding (and encoding) throughput to grow proportionally to the number of drives, so that the read throughput grows in the same manner. A naive approach to achieve scalability is to use multiple disjoint arrays with small k. However, that could potentially lead to suboptimal load-balancing depending on the data placement, because certain drives or arrays could contain highly-accessible objects and others rarely accessible objects. Moreover, spreading each object over all arrays as in RAID-0 could result in suboptimal performance if the objects are not large enough, and can limit scalability.

In what follows we propose an approach for scaling the above naive system. In particular, in the new system the computational cost increases proportionally to the total achievable throughput while maintaining read/write separation. Our approach is based on the ideas of CRUSH [WBMM06], which maps objects to storage devices according to a pseudo-

random function. In particular, we create logical arrays, or redundancy groups, and spread objects according to a pseudo-random function. The difference from CRUSH is that each redundancy group is constructed in a way that enables read/write separation. Still, the group construction remains highly flexible, in the sense that the number of valid redundancy groups that can be constructed remains high. In what follows we describe how we can construct such redundancy groups.

5.6.2.1 Data Placement

To perform read/write separation and maintain availability we need to create redundancy groups and arrange data appropriately. In particular, every object should be accessible for reading at any point in time. Moreover, even if a drive is part of more than a single redundancy group, its role (reader or writer) has to be the same in every group. To keep our design simple we make two observations. First, in practice, reads are served by a fixed number of nodes inside each group - just one under replication and k with erasure codes. (Clients requiring higher performance typically stripe their objects across groups.) Therefore, we concentrate on groups with a fixed number of readers. Second, a data center typically uses a limited number of redundancy configurations (e.g., 2- or 3-replication and limited erasure coding configurations). Hence, we require that there is no overlap between redundancy groups of different size or functionality. We note that neither of these two restrictions are necessary for our scheme. We focus on them because they are largely realistic and greatly simplify exposition.

Consider an n-uniform hypergraph H, where vertices represent physical drives (or stor-

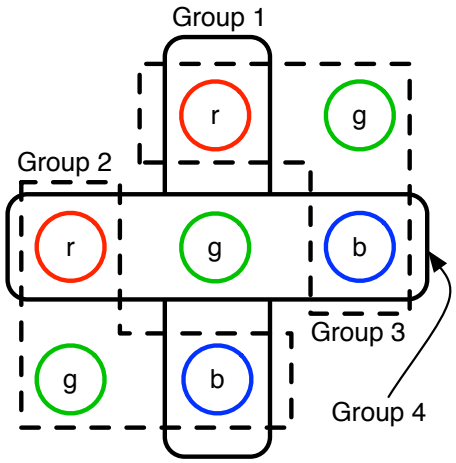


Figure 5.5: A hypergraph with four (partially) overlapping hyperedges (redundancy groups), each containing three vertices (drives).

age nodes with a single drive) and hyperedges represent redundancy groups of size n. We want to be able to generate n-uniform hypergraphs that allow us to perform read/write separation and which provide good load-balancing. As an example, consider the 3-uniform hypergraph with four hyperedges (redundancy groups) shown in Figure 5.5. From the same figure each vertex is given a color (r:red, g:green and b:blue). Assume that $k = 2$ and $m = 1$, and that the red and green vertices start as reader drives while the blue ones as writer drives. It can be observed that as soon as the vertices switch roles every T seconds and they start this process at the same time, this particular hypergraph enables read/write separation. Note that in practice we do not require perfect time synchronization since drives only change roles after multiple seconds.

More generally, a class of n-uniform hypergraphs satisfying read/write separation is illustrated in Figure 5.6 and can be described as follows: Consider all N nodes in the system. Partition the nodes into k sets P_i of equal size, with each set corresponding to a

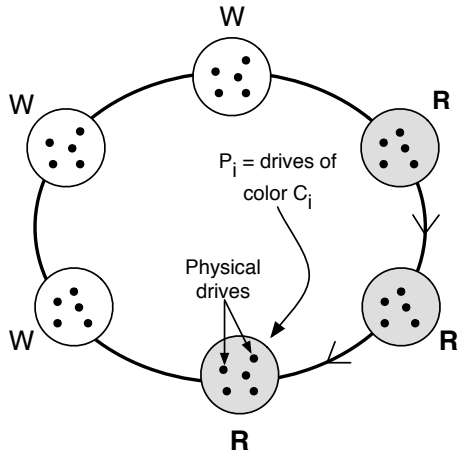
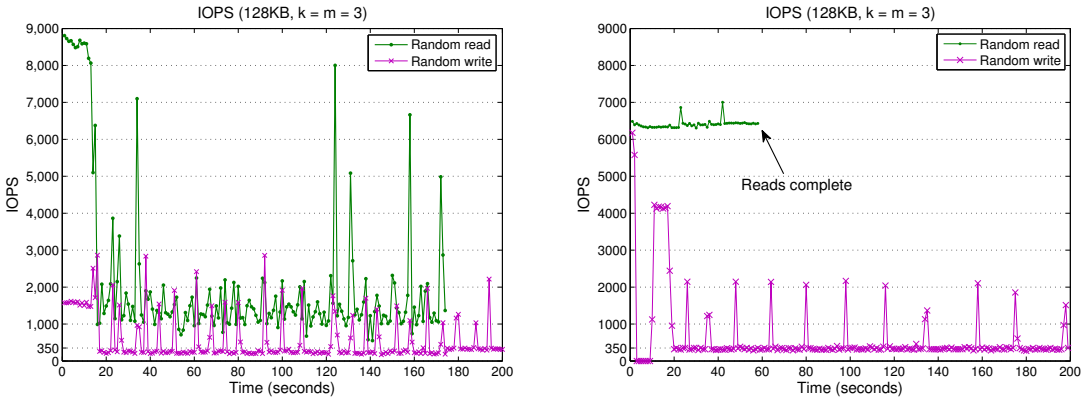


Figure 5.6: To construct valid redundancy groups of size six, drives are partitioned into six sets. Each group is then constructed by selecting a single drive per set.

color C_i. To form a group we select exactly one node from each P_i. Generating multiple groups consists of repeating this process, with the difference that to provide good load-balancing by the end of the process each node will be selected an equal number of times. To provide read/write separation, we initially pick a random subset C_R of colors, with size equal to the number of readers. The drives having color in C_R correspond to the initial readers. The sliding window now shifts over the colors and initially contains exactly C_R. Any node having color that is inside the sliding window performs reads, otherwise writes. We expect the above class of hyper graphs to be large enough for practical usage and leave open the possibility of creating more classes. For example, the above may be extended by noticing that the hyperedges do not have to be of the same size as long as the ratio of the readers to writers remains the same.



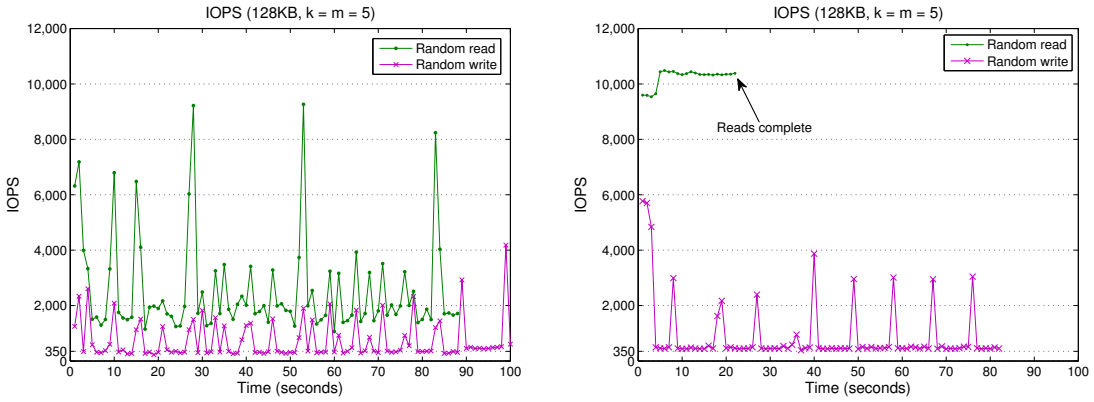
(a) Without eRails. (b) With eRails.

Figure 5.7: Without eRails the variance of writes "pollutes" that of reads. Using eRails eliminates that variance. ($k = m = 3$; 128KB)

5.7 Evaluation: Read/Write Separation

We now provide an evaluation of eRails. The main point of the evaluation is to show that under erasure codes we can provide read/write separation as well as Rails does under replication. We perform two sets of experiments. In the first set we use six drives and let $k = m = 3$, whereas in the second one we use ten drives and set $k = m = 5$, so in both cases we use an equal number of read and write drives. The workload consists two independent streams, a read stream and a write stream, both sending 128KB requests as fast as possible. We set the system dispatch rate of the read stream to 0.75 and the rate of write stream to 0.25. In other words, for every write operation dispatched, three reads are dispatched.

Figure 5.7 demonstrates the case where $k = 3$. In particular, Figure 5.7a shows that, as expected, the performance without Rails drops quickly for both reads and writes (due to writes). On the other hand, with Rails (Figure 5.7b) the read performance remains high



(a) Without eRails.

(b) With eRails.

Figure 5.8: Using eRails to physically separate reads from writes leads to a stable and high read performance. ($k = m = 5$; 128KB)

and almost variance free regardless of the write workload. The two (upward) spikes in the read performance happen when the sliding window moves. In that case, if there are remaining reads for a drive that just became a writer, then those are executed before writing starts. Therefore, the number of read drives and consequently the read throughput are temporarily higher. The number of the remaining reads is small and bounded, and in our experiments it was set to 100. Note that with eRails we achieve about 6,400 reads/second (800MB/s), which is close to the read throughput of 820MB/s achieved when only performing reads (Figure 5.4). Finally, in the above set of experiments, the total number of requests was set to 500,000 and it took 200 seconds for the workload to complete.

Increasing k from $k = 3$ to $k = 5$ gives similar results (Figure 5.8). First, Figure 5.8a shows that without Rails the read performance is low and inconsistent. With Rails, Figure 5.8b illustrates that reads have high and consistent performance. (The

lower performance at the start is due to read/write mixing before separation started). In particular, Rails achieves 10,400 reads/second (1300MB/s). As was the case with $k = 3$, 1300MB/s is also close to the read throughput of 1360MB/s achieved when only performing reads (Figure 5.4). The difference in throughput may be attributed to the additional CPU pressure due to the write operations. The workload in the above two experiments consists of 300,000 requests and it took between 80 and 100 seconds to complete, with the variance being due to the write performance inconsistency. In both experiments, the write performance is clearly low, however, that is due to the drives themselves. Improving the write throughput is a possibility in eRails especially due to batch writing, for example, through its integration with a log-structured block store. However, that is left as future work.

5.8 Read/Write Separation in Distributed Storage

In the previous sections we described how to scale eRails efficiently. Since a single node has limitations in terms of its processing power, to build very large-scale storage systems using eRails it appears necessary that we have a distributed version of it, i.e., without relying on a shared local memory. In this section we discuss some of the challenges in making eRails distributed and show how the grouping presented in Section 5.6 can be directly applied.

Given a distributed system and a set of clients with arbitrary workloads, our goal is to provide read-only performance for the reads, while performing at least as good for writes

as before. To achieve this, we follow the general approach of physically separating reads from writes, as presented in Section **??**. We assume the existing cluster has enough network capacity to take advantage of SSDs, which is a reasonable assumption given that 10Gb or faster networks are becoming common.

5.8.1 Distributed Storage Model

Distributed systems are complex and a modification can take different forms when applied on a specific implementation. In what follows, we present a model for a distributed storage system that will be used a basis for describing our solution in subsequent sections. In Section 5.8.5, we outline our solution specifically for Ceph [WBM+06].

5.8.1.1 Model Layers

The employed model consists of three logical layers as shown in Figure 5.9: clients, coordinators, and storage. Clients are end-users such as applications that view the system as a storage device. Coordinators accept client requests and forward them to storage nodes, where they are stored in persistent media. We now describe each layer:

Clients: From a client's perspective the whole storage system is seen as a single storage device with multiple entries. We assume that the system provides to the clients a list of entry points and a rule for picking entry points.

Coordinators: Coordinators accept client requests and route them to storage nodes. They have a data placement function, which takes as input an object name and returns a list with the storage locations of the object's pieces. Moreover, a selection function

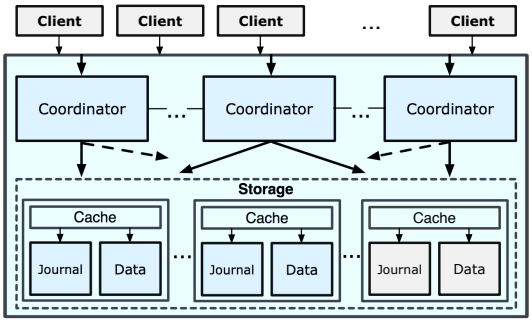


Figure 5.9: A logical distributed storage system model

takes as input the data locations of the object and a vector of system parameters and returns a subset of drives from which O should be read. Finally, coordinators may orchestrate operations such as 2-phase commit and erasure coding operations.

Storage: Storage is spread across nodes. Each storage node consists of a memory with a minimum size, a single data drive and a journal drive. When a write arrives at a storage node, it is first written to its cache and journal, and then to its data drive asynchronously. Reads are served by the cache or data drive.

Note that the above separation between layers is logical. A physical implementation may merge layers to decrease network effects, improve scalability and generally optimize performance. In what follows, we show how to provide read/write separation in systems of increasing complexity, starting from having local redundancy (the case so far), then a pair of nodes and leading to the generic case of multiple nodes and complex data placement.

5.8.2 Local Redundancy

A simple non-distributed approach for achieving read/write separation in a distributed system with M drives is to place $N << M$ drives on each node and apply read/write separation, as described in Section 4.4, locally. For example, when $N = 2$ each node has two drives and can replicate locally (Section 4.4.1). This design creates a logical device per node, which behaves as a drive with optimal read performance under read/write workloads. The disadvantage of this approach is that it adds local redundancy, which increases the system cost without especially adding to its fault-tolerance and availability. The advantage of it is simplicity. Even if the nodes are part of a distributed system, the read/write separation happens seamlessly at a local level, without modifying the system itself. Also, the above design is cheaper than typical over-provisioned flash cards or drives, which cost many times more than commodity SSDs. The approach we consider in this work is as the above with the difference that the drives are spread among nodes in order to use existing resources.

5.8.3 Basic Setting

Consider the system model described in Section 5.8.1.1 with a single coordinator and two storage nodes, namely n_1 and n_2. We refer to n_1 and n_2 as a redundancy group or simply as a group, with n_1 acting as the "primary" and n_2 as the "secondary". Without read/write separation the system operates as follows: The coordinator sends reads and writes to node n_1, which acts as the entry point of the group. Reads are served exclusively by n_1. Writes are stored on n_1 and forwarded to n_2, where they are

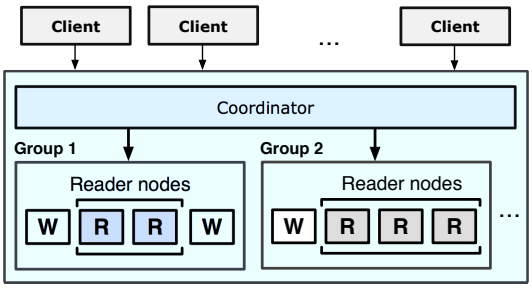


Figure 5.10: A special case of read/write separation under a single coordinator and disjoint groups of nodes.

also stored, i.e., they are written to the memory and journal. After a write is stored on both nodes, the coordinator receives a response from n_1. Depending on the system policy, nodes eventually flush the writes to their data drive. In this system writes can cause reads to block.

To separate reads from writes, we modify the above as follows: The coordinator decides whether a node should be reading or writing. As before, n_1 is the entry point of the group, however, if node n_1 receives a read with n_2 as its final target, it forwards that read to node n_2, which responds directly to the client with the data. Writes are performed as before with the difference that only nodes in write mode flush any accumulated or incoming writes. The read node accumulates the writes and only flushes them to its data drive once the nodes switch roles. To provide strong consistency, we ensure that only data committed on both nodes are returned. Finally, the above could be modified to provide weaker versions of consistency for potential performance gains.

5.8.4 Generalization and Challenges

In the above, we considered a single group of two nodes. We can increase the size of the group and the coordinator can assign read/write roles to the nodes according to the sliding window technique (Section 4.4.3). Moreover, we can create additional redundancy groups under the same coordinator assuming those groups are disjoint, i.e., they have no nodes/drives in common (Figure 5.10). In addition, multiple coordinators may be used if we assume no two coordinators point to the same group.

In practice, the above assumptions may not hold because redundancy groups often overlap to achieve better load-balancing [WBMM06]. This may also lead to data placements that make read/write separation impossible. Hence, in general the coordinators have to be in agreement on which nodes should be receiving reads, and consequently which ones should be flushing writes to their data drive. For example, the redundancy groups in Figure 5.11 make read/write separation impossible. Moreover, we have to restrict the data placement to ensure that for all data there is a drive in read mode containing that data. In what follows, we describe those challenges in more detail and propose techniques to address them.

5.8.4.1 Data Placement

In a system with M drives, where M is large (hundreds or thousands), data is typically replicated to a much smaller number of drives, $N << M$. Similarly, when using erasure coding, splitting an object O into M pieces is impractical. Instead, a fault-tolerant distributed system would split O over a set of $N << M$ drives. In both cases, to retain

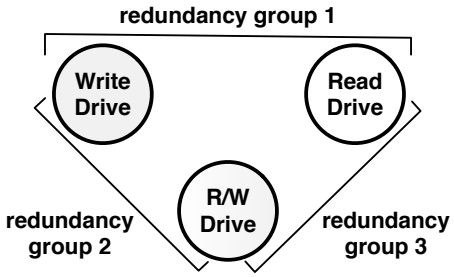


Figure 5.11: Odd-length cycles makes read/write separation impossible, since a node performs both reads and writes.

load-balancing, each drive is part of multiple groups according to a data placement method. However, if all drives corresponding to some data are in write mode, a read over that data leads to read/write mixing. Therefore, we need to construct redundancy groups allowing read/write separation rather than place objects arbitrarily. To achieve that our grouping technique from Section 5.6.2 can be readily applied. Applying that technique allows for a number of configurations including the option to trade writers for readers and vice versa at the system level, which is useful for systems with unstable read/write load.

5.8.4.2 Aggreeing on the Drive Mode

Coordinators distribute requests among storage nodes. When multiple coordinators manage a set of drives, they must all agree on which drives should be receiving reads and consequently, which ones are in write mode. This implies that coordinators need to reach a consensus. We refer to the switching policy as the rule based on which drives switch roles (reading or writing). Depending on the switching policy, we propose the following methods for keeping the coordinators synchronized.

Time-based policy: A time-based policy divides time into frames and sets the mode of each device according to the current frame. Assuming a time-based switching policy, we propose the use of a protocol such as NTP (Network Time Protocol) to keep the time of each coordinator in sync. We expect NTP to be sufficient, since the granularity at which it is expected to achieve synchronization is significantly smaller than the granularity at which we switch modes. In principle, synchronization inaccuracies result in read/write mixing on a single drive. We expect those delays to be small enough (in the order of a few milliseconds). In addition, writes are first committed to a journal drive, and only eventually written to a data drive. Hence, the storage can give priority to reads when briefly presented with a mixed workload so that delays have no effect on read isolation.

Generic switching policies: The switching policy proposed in this work is purely time-based. To support generic switch policies, e.g, based on the amount of data written, purely time-based policies are not sufficient. Instead, the coordinators may have to reach a consensus regarding the time of the next switch. In general, reaching a consensus over a growing set of nodes limits scalability. In our case, we only need to do so every multiple seconds and can perform all related operations in memory, since we do not need the fault-tolerance guarantees of storage-backed consensus algorithms. Based on the above, we argue that in practice scalability is possible in our case.

5.8.4.3 Other Challenges and Techniques

Node failures: When a node goes down, we have to find a replacement quickly. The difference from a typical distributed system is we have to ensure the replacement node is

in the same mode as the failed one, i.e., it is of the same color. Although this reduces the flexibility in picking a replacement it does not appear as a problem. Finally, additional flexibility may be added by having a small amount of extra nodes for such cases until the original node is available again.

SSD performance heterogeneity: The write throughput of an SSD can vary in time depending on past workloads. This implies that even drives of the same model may not always have identical performance. It follows that a node in our system may not manage to flush all its data while in write mode. Instead, in a typical storage system a redundancy group has the performance of its slowest node. We want to retain that property while providing read/write separation to prevent the accumulated data of each node from growing unbounded. To that end we propose that each secondary node informs the primary of the remaining amount of writes it has to flush when it starts accepting reads. Based on those values, the primary reduces the rate at which it sends write operations to the secondaries.

5.8.5 Example Integration with Ceph

5.8.5.1 Overview

Ceph is a scalable, distributed storage system supporting object, block and POSIX file system interfaces [WBM+06, WLBM07]. At a high level it works as follows. Clients map object names using a hash function called CRUSH [WBMM06] onto redundancy groups called placement groups (PGs). A PG is a set of Object Storage Devices (OSDs), which together form the redundancy group for a subset of objects. For our purposes an OSD

may be thought of as a storage node with a data drive and an associated journal drive. The first node in a PG acts as the primary - it receives all requests targeting that group. Reads are served locally by the primary while write requests are performed on every node in the group. When a node receives a write request it stores it in its journal and eventually flushes it to its data drive. After the primary receives acknowledgement from all PG nodes that the data have been written to their journal, the primary performs the write locally and then responds to the client. Observe that the above mechanism provides strong consistency.

5.8.5.2 Model Mapping

From the above, we note that the coordinator logic is now split between the clients and the storage nodes. In particular, the data placement function is provided to the clients and is composed of two operations, the mapping from object names to PG identifiers and the mapping provided by CRUSH to return the OSDs of the placement group. On the other hand, the load-balancing function is part of the primary nodes logic and by default points to the primary.

To provide read/write separation, unlike the default behavior of Ceph, we need to send reads to group nodes other than the primary. In fact, the load-balancing function could return any of the drives in read mode. To follow Ceph's default behavior, we require that all reads go through the primary, but not that they are served by the primary. For instance, the primary forwards the read request to a replica, and the replica responds directly to the client. To apply our data placement techniques to Ceph, we need to

restrict its placement group generation. Fortunately, the restrictions in Section 5.6.2 still allow for a significant number of graphs and we expect the load-balancing to be comparable after our modification.

5.9 Summary

Rails (Section 4) is a system for enabling consistent performance in flash storage by physically separating reads from writes through redundancy. In this chapter, we presented eRails, a system built on top of Rails that employs erasure coding as its redundancy method. To provide read/write separation, eRails reads only from a specific subset of drives at a time and performs a decoding operation for each read request. Through experiments of up to ten drives, we demonstrated that eRails enables predictable performance for reads under read/write workloads without reducing the raw throughput. Moreover, we presented a design enabling eRails to support an arbitrary number of drives by creating small overlapping erasure coding groups of drives to limit the computational cost of encoding and decoding. Finally, we considered some challenges of applying our scaling solution to large-scale flash-based distributed storage systems. An interesting direction is to implement and evaluate eRails at a larger scale in a distributed environment by taking advantage of the hypergraph construction presented in this work. We expect that to lead us to the peta-scale distributed flash-only storage system that provides consistent performance, as we proposed in [SWA+13].

Chapter 6

Guaranteeing I/O Performance in Solid-State Drives

6.1 Performance Guarantees

In this chapter we present a time-based scheduling method for providing throughput guarantees to SSDs. First, we rely on the SSD capabilities and focus on providing guarantees without attempting to improve the device performance or variance. In particular, we construct a generic time-based QoS method for guarantees, rather than design heuristics for performance improvements. Second, we apply our scheduling method to Rails and demonstrate through experiments that providing high performance guarantees efficiently becomes possible.

6.1.1 Overview

Storage users want Service Level Agreements (SLAs) in the form of throughput or latency requirements. Our solution supports throughput guarantees at different granularities for shared SSDs with each client receiving the promised performance irrespectively of other workloads. What follows is an overview of the steps we take to achieve that. First, we profile the drive by running simple synthetic workloads. Using the profiling results we construct throughput distributions allowing us to estimate the cost of each type of request at any granularity and confidence level. Based on those distributions, we convert the client requirements into utilization, which is defined as the fraction of device time provided to a client/workload. Afterwards, each request is assigned a deadline based on its cost estimate and the utilization assigned to its workload. Finally, each request is dispatched based on its deadline and according to the Earliest Deadline First algorithm. In the rest of this section, we describe the above in more detail and present our evaluation results.

6.1.2 Scheduling for Guarantees

In shared environments, where multiple streams utilize the same storage system, each stream expects its performance objectives to be satisfied independently of other workloads. Hence, any interference between different types of requests due to request cost inequalities has to be minimized and the utilization provided to each stream adjusted accordingly. To achieve that we follow a time-based approach [PKB+08] and provide each stream with a proportion of the drive time based on its requests and performance

needs. By reserving time we can guarantee an absolute throughput for each stream rather than a proportion of the total throughput, which can vary significantly and lead to low reservations.

In order to reserve a specific percentage of the device time for each stream, we need to know the cost of the queued requests. The amount of time it takes for a request to complete, irrespectively of any queueing overhead varies, and estimating it online is a separate challenge (Chapter 3). From our experiments and from previous work [CLZ11] we know that in SSDs the request size and the queue depth are significant factors in a request's cost. Moreover, it is known from [CKZ09], and we verify it here, that whether a request is a read or write, and whether it is sequential or random can significantly affect its cost. We conclude that by ignoring the state of an SSD, it is possible to estimate the cost of a request to a large degree based on its size, type and the queue depth. On the other hand, SSDs are stateful and their behavior is shaped by current and previous workloads. Depending on the state of the drive, e.g., whether it has enough free space, the cost of a request may differ. Therefore, to provide close to worst-case guarantees we should consider a worst-case behavior.

To estimate request costs we profile the drive under requests of different access type, size and queue sizes. Since the throughput during profiling may fluctuate significantly, instead of taking the average cost, each stream has a confidence level assigned to its reservations, with higher levels implying a higher request cost. On a similar note, if the profiling range is small enough (e.g., 1GB,) the write performance may be more consistent as well as higher and the guarantees granularity can be higher. Having said

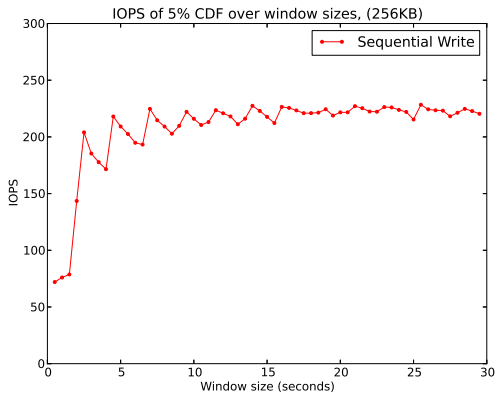


Figure 6.1: The throughput of the bottom 5% of sequential writes can be relatively low, making high granularity guarantees expensive.

that, the write range is not part of our model. Instead, if the storage user expects to consistently have a lighter workload it is up to them to adjust the profiling accordingly. If the workload is not known, then the profiling should cover the worst case as described in the previous section.

Although scheduling by time allows us to minimize stream interference and provide tighter guarantees, clients expect throughput reservations expressed by SLAs. To that end, we use our profiling results to convert throughput requirements to the corresponding stream utilization u_s, which is defined as the proportion of the device time provided to stream s. Note that throughput requirements need to have a granularity associated with them and u_s depends on that value. In particular, a higher granularity is expected to imply a lower throughput (Figures 4.6 and 6.1) and a higher utilization. The notion of granularity is useful because there are applications such as video streaming, which would trade granularity for higher throughput guarantees, whereas online processes

such as audio streams would accept lower throughput guarantees for higher granularity. By converting throughput to utilization we can also perform admission control, since the total utilization cannot exceed 100%. On the other hand, for systems that can be over-utilized, where guarantees are less important, this model offers stream isolation by providing each stream with its relative proportion of the device time.

In order to schedule the stream requests, given the expected cost of a request we assign it a deadline that depends on the time utilization rate u_s of the corresponding stream s. In particular, if the expected cost of a request is e, then its (relative) deadline is defined as $d = e/u$. Also, the absolute deadline of the request is set to $D_s = T_s + d$, where T_s is the sum of all relative deadlines of stream s up to that point. After a request is assigned a deadline, it is inserted to a priority queue and dispatched according to the Earliest Deadline First algorithm. To summarize, at a high level our method consists of the following sequence of operations: drive profiling, throughput requirements to utilization conversion, (i.e., resource allocation) and time-based request dispatching.

6.1.3 Providing Guarantees

In this section, we apply our scheduling method to provide guarantees to multiple streams sharing the same drive and illustrate the importance of associating granularity to throughput guarantees. A natural question is how to decide the granularity at which we can provide certain guarantees. Since drives often behave differently, we take advantage of the profiling we already perform. For example, in Figure 6.1, we saw that increasing the granularity, decreases the guarantees for large sequential writes (similarly

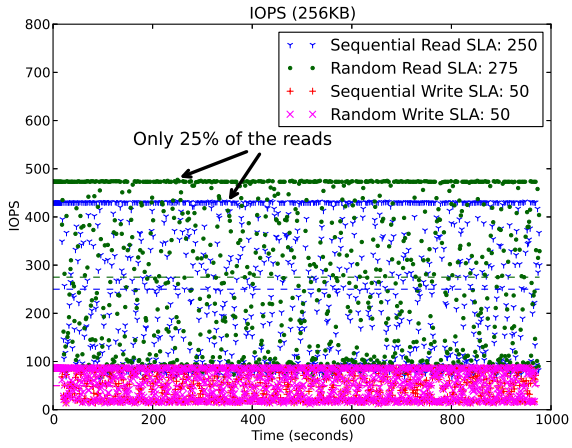


Figure 6.2: Due to write-induced blocking events, read throughput is unpredictable at a 1-second granularity and the targets are only met on average.

for random ones). Based on that information we can convert throughput requirements of certain granularity to the required utilization u, and decide whether those requirements can be guaranteed, i.e., if $u + \sum_i u_i \leq 1$.

We start with a set of four streams of different type, each having a throughput requirement. In particular, we set the target of the two read streams to 250 and 275 IOPS and set the write targets to 50 IOPS each. Although the write targets might appear low, according to the profiling results (for drive B) in order to achieve those targets at a low granularity their streams have to occupy a total of 50% of the drive time. Instead, at a granularity of a second the utilization required just for the sequential writes is over 60% (as implied by Figure 6.1) for a total of over 170%. This is due to the significantly higher worst-case cost of writes compared to that of reads when performed in isolation (as in profiling). As expected, from Figure 6.2 we see that the throughput

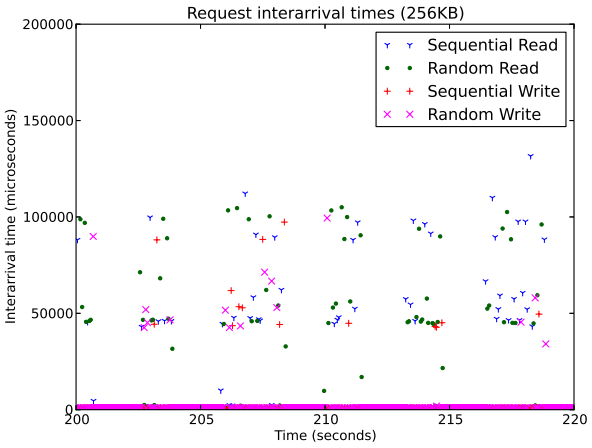
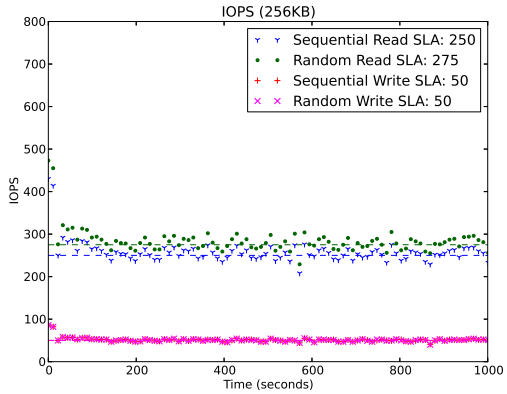


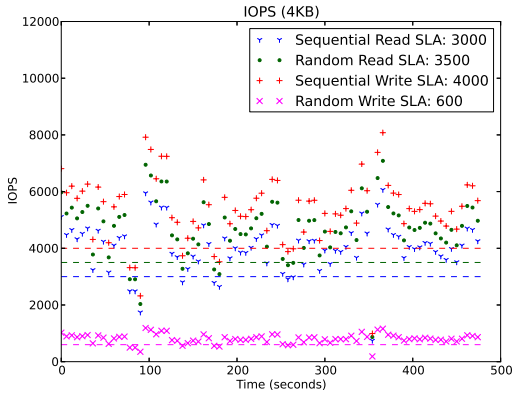
Figure 6.3: The blocking events do not have a short period, meaning that some, e.g., 1-second intervals have no or few such events while others have many.

targets are not achieved at a 1-second granularity since the total utilization required exceeds 100%. On the other hand, the average throughputs still satisfy the targets but there is an oscillating behavior for all streams. Another way to visualize why a 1-second granularity leads to a high utilization is by noting the blocking events over a typical 2-second subinterval as shown in Figure 6.3. We observe that the number of blocking events per second varies, while it remains similar over 10-second intervals leading to a lower worst-case request cost.

As expected, increasing the granularity to 10 seconds leads to a (worst-case) write cost drop while the required utilization becomes 99% to 100%. As illustrated in Figure 6.4a, the achieved throughput at a 10-second granularity remains close to the targets of each stream. Finally, since the total utilization is between 99% and 100%, we conclude that almost all of the drive's time is successfully reserved, while achieving the throughput



(a) At a granularity of 10-seconds, the throughput is close enough to the targets while the reserved utilization is almost 100%.



(b) At a granularity of 5-seconds, the throughput guarantees are almost always achieved while the reserved utilization is almost 100%.

Figure 6.4: Throughput achieved over 3-second and 5-second granularities.

guarantees to the granularity possible by drive B, without any heuristic optimizations. Moreover, this implies that the cost estimates extracted during profiling were accurate enough.

Large requests allow us to examine the drive behavior more easily by having a more consistent behavior than small (e.g., 4KB) requests. Although the behavior of small writes can be more variable, proper profiling and time scheduling still allow us to provide guarantees. As mentioned earlier, Figures 4.6 shows the relation between the throughput of sequential writes running in isolation on drive A and its granularity. We saw that reducing the window size to less than four seconds drops the throughput quickly to the point where it becomes zero. In what follows we set the window size to five seconds, which corresponds to about 20,000 writes/second, therefore, an average cost of $50\mu s$ per request. From Figure 6.4b, we see that the achieved throughput at a 5-second granularity meets the targets almost always, while the total reserved utilization reaches

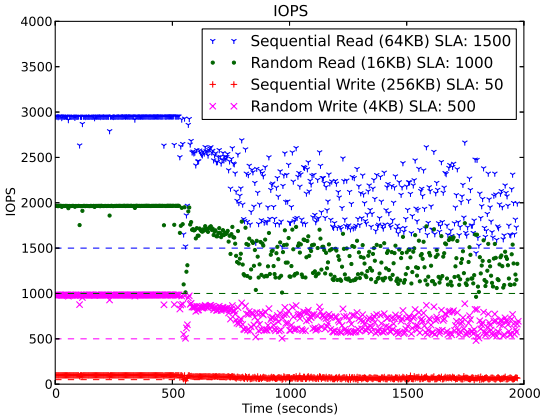
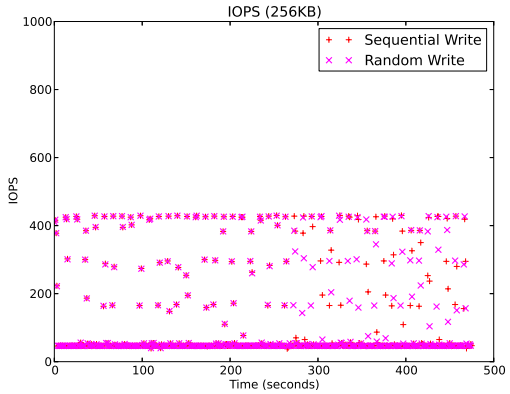
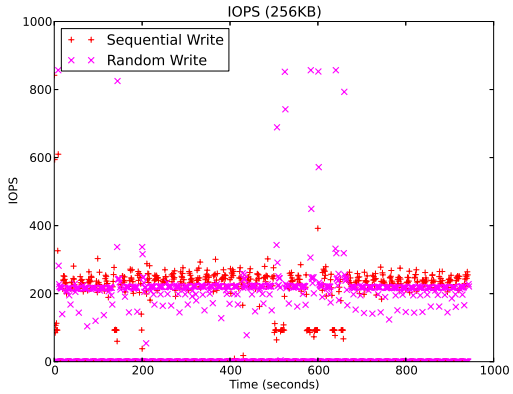


Figure 6.5: The guarantees (over 3-second intervals) for mixed request sizes are tightly satisfied, with a total device time reservation of 99%.

100%. In the next experiment, illustrated in Figure 6.5, we consider streams of different request sizes at a 3-second granularity, based on the granularity-throughput relation in Figure 6.7. Besides showing that the targets are met while reserving 99% of the device time, this experiment illustrates that a stream with larger requests can have a higher target than one with smaller requests without one making the other miss its targets. Moreover, since the scheduling happens based on the cost extracted from profiling, where each type of stream runs in isolation, we conclude that isolation is also provided to the granularity permitted by the device. Finally, our experiments so far were performed on a single SSD. We have noticed that on SSD arrays, dependencies between requests can lead to more frequent blocking. Therefore, to provide guarantees on arrays using the same method, the worst-case profiling has to be applied on top of the array.



(a) Mixing sequential with random writes on drive A, creates consistency problems.



(b) Disaggregating sequential from random writes improves the consistency of drive A.

Figure 6.6: Disaggregating sequential from random writes in drive A improves write consistency, while the average performance remains similar.

6.1.4 Heuristic Performance Improvements

By studying drive models A and B, we found the behavior of A, which has a small cache, to be more easily affected by the workload type. First, writing sequentially over blocks that were previously written with a random pattern has a low and unstable behavior, while writing sequentially over sequentially written blocks has a high and stable performance. Although such a pattern may appear under certain workloads and could be a filesystem optimization for certain drives, we cannot assume that in general. Moreover, switching from random to sequential writes on drive A adds significant variance and lowers its performance. To reduce that effect we tried to disaggregate sequential from random writes (e.g., in 10-second batches). From Figure 6.6 we see that doing so doubles the throughput and reduces the variance significantly (to about 10% of the average), which allows higher I/O guarantees. On the other hand, we should emphasize that the

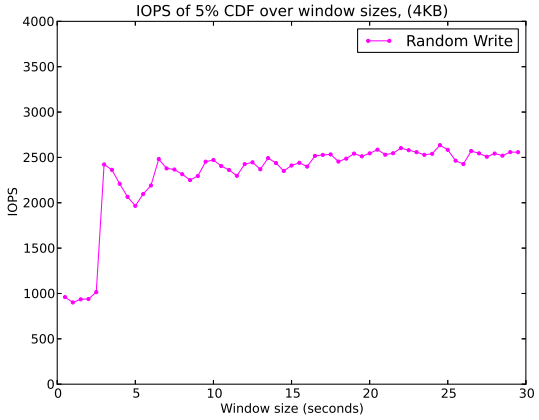
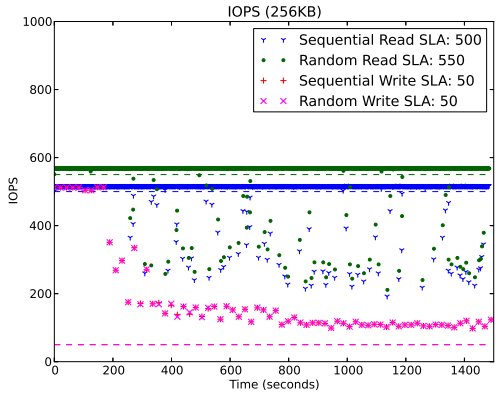


Figure 6.7: As long as we average over three seconds or more, the cost of a random write can be close to $400\mu s$. Otherwise, the cost doubles.

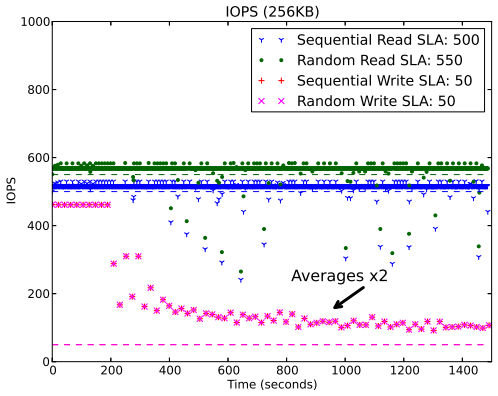
above heuristic does not improve the read variance of drive B unless the random writes happen over a small range, which strengthens the position of not relying on heuristics due to differences between SSD models. In contrast to the above, in the next section, we show how using our scheduling method on top of Rails (Chapter 4) provides high and stable read performance under mixed workloads for generic drives.

6.2 High Performance Guarantees

In the previous section, we observed that the granularity at which SSD devices achieve high performance under mixed (read/write) workloads is low, i.e., multiple seconds, instead of a single second or less. That leads to high read latencies, which are prohibitive for many applications. In this section, we use Rails (Chapter 4) as a way to eliminate the read variance, which then allows us to provide high performance guarantees. We show that Rails is significantly less prone to differences between SSDs than heuristics,



(a) Using our design and QoS method we can provide high guarantees for reads and achieve the targets more than 95% of the time without any heuristics.



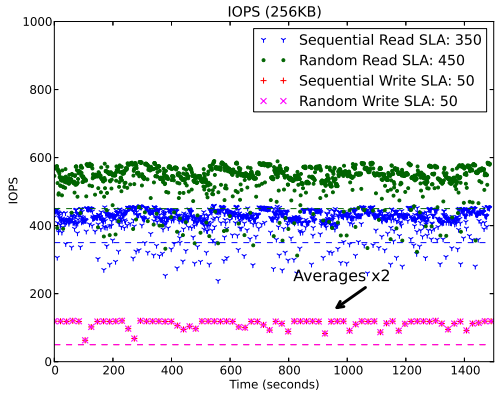
(b) Adding a second of idle time to writes before each switch reduces the read/write interference while satisfying the targets 99% of the time.

Figure 6.8: Guaranteeing I/O on Intel 510.

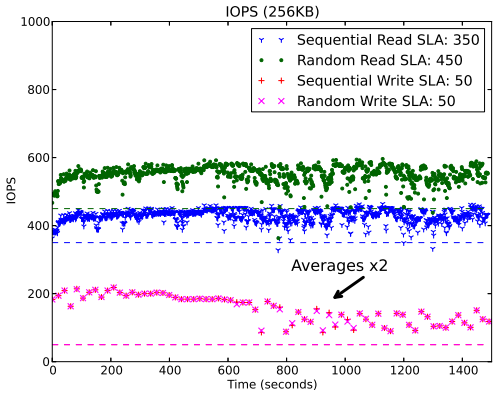
and demonstrate its benefits in providing guarantees under two different models.

6.2.1 Guaranteeing High Performance

Given that we now have stable read performance through Rails, it becomes easier to provide high performance guarantees by combining Rails with our scheduling method from the previous section. As before, we set four streams each with its own type and throughput requirements. In the following experiments, although the targets of the last two streams are 50 writes/sec, the average throughput plotted is 100 writes/sec or more. That is due to the same writes eventually being dispatched to both drives. Moreover, since the write performance has too much variance at a high granularity, we plot the average write throughput over each epoch (two switch periods) to show that the write guarantees at the drive level are also achieved at the granularity allowed by the device.



(a) On a drive switch there can be interference due to remaining background work. Without any heuristics, we still achieve 95% of the targets and attain a low latencies.



(b) To improve performance stability we add one second of idle time to the writes before a switch happens, increasing target satisfaction from 95% to 99%, while improving latencies.

Figure 6.9: Although drive A is more easily affected by background work we still achieve 95% of the targets with low latencies and can improve that to 99%.

From Figure 6.8a we see that under drive B the targets are satisfied 95% of the time. In addition, as shown in Figure 6.8b, by idling writes for a second before each switch the target satisfaction rate increases to more than 99%. In a similar fashion, using drive A we found the same success rates for both the idle and the non-idling case (Figure 6.9). However, due to the nature of drive A there is a small increase in the variation of the read throughput after each mode switch, which leads to slightly lower guarantees. Since drive B is a newer model one would expect that recent models of similar quality have a behavior closer to that of B. Independently of that, the read performance achieved with any the two drives is significantly more stable compared to that without redundancy. Finally, note that switching less often reduces the throughput drops but requires a larger cache and may leave more background work for the SSD after each switch.

6.3 Summary

The performance of solid-state drives degrades significantly under write-heavy workloads. In particular, the average throughput can drop many times while the latency often increases to the order of 100ms due to garbage collection. In this chapter, we showed how to provide QoS for solid-state drives by considering the worst case and found that providing tight guarantees is possible at the performance and granularity allowed by the drive it self. In addition, by placing our scheduling method on top of Rails we showed that guaranteeing consistent and high read performance becomes possible even under demanding read/write workloads.

Chapter 7

Conclusion

This dissertation has addressed two main research problems: How to provide I/O guarantees to black-box storage systems efficiently; and How to build a scalable flash storage system that provides consistent performance efficiently and enables I/O guarantees. Addressing the first problem is based on the following main ideas: place a proxy controller between the clients and the storage device, estimate the cost of requests through a linear model differentiating between sequential and random requests. Depending on the utilization reserved for each client and the predicted cost of each incoming request schedule the requests in a time-based manner, finally update the model parameters based on the storage device performance observed.

Providing consistent flash performance is important for virtualization and many modern applications. First we saw that solid-state drives have inconsistent performance under read/write workloads due to the writes, whereas reads alone have high and consistent performance. We addressed the inconsistent behavior of flash by physically separating

reads from writes through redundancy. In particular, we achieved read-only performance in the presence of writes while the write performance was at least as well as before.

To reduce the storage overhead and improve reliability we presented eRails, an extension of Rails that uses erasure coding. We considered the effects of computation due to encoding/decoding on the raw performance, and saw it does not effect the performance consistency of Rails. We found that up to a certain number of drives the performance remains unaffected while the computation cost remains modest. After that point, the computational cost grows quickly due to coding making further scaling inefficient.

To support an arbitrary number of drives we presented a design allowing us to scale eRails by constructing overlapping redundancy groups that preserve read/write separation. Through benchmarks we demonstrated that eRails achieves read/write separation and consistent read performance under read/write workloads. We noted that distributed storage systems already employ a form of redundancy for reliability purposes. We discussed the challenges of applying eRails to such systems by exploiting those existing resources to provide read/write separation.

Finally, we saw that applying time-based scheduling on top of a solid-state drive can only provide low I/O guarantees due to the inconsistent performance of SSDs. Through benchmarks we showed that guaranteeing I/O efficiently is possible by applying time-based scheduling on Rails, regardless of the write variance of solid-state drives.

Bibliography

[ADAD01] Andrea C. Arpaci-Dusseau and Remzi 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.

[AM+13] Christoph Albrecht, Arif Merchant, et al. Janus: Optimal flash provisioning for cloud storage workloads. In *ATC '13*, 2013.

[APW+08] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX ATC'08*, 2008.

[BD10] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: measurements and analysis. In *USENIX FAST'10*, 2010.

[BJB09] Luc Bouganim, Björn Jónsson, and Philippe Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR '09*. CIDR, 2009.

[BRV12] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. Shredder:

Gpu-accelerated incremental storage and computation. In *USENIX FAST'12*, FAST'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[CAP+03] David D. Chambliss, Guillermo A. Alvarez, Prashant Pandey, Divyesh Jadav, Jian Xu, Ram Menon, and Tzongyu P. Lee. Performance virtualization for large-scale storage systems. In *In Proceedings of the 22th International Symposium on Reliable Distributed Systems (SRDS03*, pages 109–118, 2003.

[CKZ09] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09*. ACM, 2009.

[CLZ11] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA '11*. IEEE, 2011.

[CSADAD12] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Consistency without ordering. In *USENIX FAST12*, 2012.

[Cur10] Matthew L. Curry. *A Highly Reliable Gpu-based Raid System*. PhD thesis, University of Alabama at Birmingham, Birmingham, AL, USA, 2010. AAI3434991.

[CWO+11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, and Arild et al. Skjolsvold. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[Des12] Peter Desnoyers. Analytic modeling of SSD write performance. In *SYSTOR '12*. ACM, 2012.

[DGW+07] Ros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin O. Wainwright, and Kannan Ramch. Network coding for distributed storage systems. In *In Proc. of IEEE INFOCOM*, 2007.

[ELMS03] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo 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.

[GAKGR10] Abdullah Gharaibeh, Samer Al-Kiswany, Sathish Gopalakrishnan, and Matei Ripeanu. A gpu accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 167–178, New York, NY, USA, 2010. ACM.

[GAW09] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Parda: propor-

tional 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.

[GDS12] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND flash memory. In *USENIX FAST'12*, 2012.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[GMV07] Ajay Gulati, Arif Merchant, and Peter J. Varman. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. In *SIGMETRICS '07*. ACM, 2007.

[GMV10] Ajay Gulati, Arif Merchant, and Peter 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.

[GSA+11] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Sym-*

posium on Cloud Computing, SOCC '11, pages 19:1–19:14, New York, NY, USA, 2011. ACM.

[HPC04] Lan Huang, Gang Peng, and Tzi-cker 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.

[HSX+12] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *USENIX ATC'12*, 2012.

[JCS+14] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. Hios: A host interface i/o scheduler for solid state disks. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 289–300, June 2014.

[KBP+12] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *USENIX FAST'12*, 2012.

[KCGK04] T. Kelly, I. Cohen, M. Goldszmidt, and K. Keeton. Inducing models of black-box storage arrays. *In Technical Report HPL-SSP-2004-108*, 2004.

[KKZ05] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage:

Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1:457–480, November 2005.

[KMR+13] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *USENIX FAST'13*, volume 13, 2013.

[KWG+08] T. Kaldewey, T.M. 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.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[LMA03] Christopher R. Lumb, Arif Merchant, and Guillermo 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.

[LSD+14] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX ATC'14*, pages 501–512, Philadelphia, PA, June 2014. USENIX Association.

[LSZ13] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San Jose, CA, 2013. USENIX.

[MKC+12] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random write considered harmful in solid state drives. In *USENIX FAST'12*, pages 12–12, 2012.

[MUP+11] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: quality-of-service in large disk arrays. In *ICAC '11*. ACM, 2011.

[MW08] Mark Moshayedi and Patrick Wilkison. Enterprise SSDs. *Queue*, 6(4), July 2008.

[NDR08] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: practical power management for enterprise storage. In *USENIX FAST'08*, 2008.

[OLJ+14] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on*

Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[PG14] J. S. Plank and K. M. Greenan. Jerasure: A library in C facilitating erasure coding for storage applications – version 2.0. Technical Report UT-EECS-14-721, University of Tennessee, January 2014.

[PGM13] James S. Plank, Kevin M. Greenan, and Ethan L. Miller. Screaming fast galois field arithmetic using Intel SIMD instructions. In *USENIX FAST'13*, 2013.

[PKB+08] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with Fahrrad. In *Eurosys '08*. ACM, 2008.

[PLS+09] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *USENIX FAST '09*, FAST '09, pages 253–265, Berkeley, CA, USA, 2009. USENIX Association.

[Pov10] Anna Sergeyevna Povzner. *Efficient guaranteed disk I/O performance management*. PhD thesis, University of California at Santa Cruz, Santa Cruz, CA, USA, 2010. AAI3429522.

[PS12] Stan Park and Kai Shen. FIOS: a fair, efficient flash I/O scheduler. In *USENIX FAST'12*, 2012.

[PSB10] Anna Povzner, Darren Sawyer, and Scott Brandt. Horizon: efficient deadline-driven disk I/O management for distributed storage systems. In *HPDC '10*. ACM, 2010.

[QBG14] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 451–462, Philadelphia, PA, June 2014. USENIX Association.

[RPD09] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *USENIX ATC'09*, 2009.

[SA13] Radu Stoica and Anastasia Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[SAP+13] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, et al. XORing elephants: novel erasure codes for big data. In *PVLDB'13*. VLDB Endowment, 2013.

[SAW+14] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on rails: Consistent flash performance

through redundancy. In *USENIX ATC'14*, Philadelphia, PA, June 2014. USENIX Association.

[SBA96] Marco Spuri, Giorgio Buttazzo, and Scuola Superiore S. Anna. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10:179–210, 1996.

[SBMW13] Ji Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *USENIX FAST'13*, 2013.

[SKB12] Dimitris Skourtis, Shinpei Kato, and Scott Brandt. QBox: guaranteeing I/O performance on black box storage systems. In *HPDC '12*. ACM, 2012.

[SP13] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX ATC'13*, 2013.

[SRC12] Weibin Sun, Robert Ricci, and Matthew L. Curry. Gpustore: Harnessing gpu computing for storage systems in the os kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 9:1–9:12, New York, NY, USA, 2012. ACM.

[SWA+13] Dimitris Skourtis, Noah Watkins, Dimitris Achlioptas, Carlos Maltzahn, and Scott Brandt. Latency minimization in SSD clusters for free. Technical Report UCSC-SOE-13-10, UC Santa Cruz, June 2013.

[uma] OLTP traces from the University of Massachusetts Amherst trace repository. http://traces.cs.umass.edu/index.php/Storage.

[WAA+04] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with cart models. *SIGMETRICS Perform. Eval. Rev.*, 32:412–413, June 2004.

[WAEMTG07] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: performance insulation for shared storage servers. In *USENIX FAST '07*. USENIX Association, 2007.

[Wan06] Yun Wang. Ncq for power efficiency. *White paper*, February 2006.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06*. USENIX Association, 2006.

[WBMM06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: controlled, scalable, decentralized placement of replicated data. In *SC '06*. ACM, 2006.

[WGLBs06] Theodore M. Wong, Richard A. Golding, Caixue Lin, and Ralph 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.

[WK02] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 328–338, London, UK, UK, 2002. Springer-Verlag.

[WLBM07] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *PDSW '07*. ACM, 2007.

[XYH10] Hu Xiao-Yu and Robert Haas. The fundamental limit of flash random write performance: understanding, analysis and performance modelling. In *In IBM Research Report, RZ 3771*, 2010.

[YSEY10] Young Jin Yu, Dong In Shin, Hyeonsang Eom, and Heon Young Yeom. NCQ vs. I/O scheduler: Preventing unexpected misbehaviors. *Trans. Storage*, 6:2:1–2:37, April 2010.

[ZDM+10a] Zhe Zhang, Amey Deshp, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center?, 2010.

[ZDM+10b] Zhe Zhang, Amey Deshp, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center?, 2010.

[ZXJ11] Xuechen Zhang, Yuehai Xu, and Song Jiang. YouChoose: Choosing your

storage device as a performance interface to consolidated I/O service. *Trans. Storage*, 7:9:1–9:18, October 2011.