Flash on Rails: Consistent Flash Performance through Redundancy

Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, Scott Brandt
University of California, Santa Cruz
{*skourtis, optas, jayhawk, carlosm, scott*}*@cs.ucsc.edu*

Abstract

Modern applications and virtualization require fast and predictable storage. Hard-drives have low and unpredictable performance, while keeping everything in DRAM is still prohibitively expensive or unnecessary in many cases. 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, exceeding hard-drive latency and leading to unpredictable performance.

Many systems with mixed 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. In this paper, we propose Rails, a design 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 solid-state drives.

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 [5, 6, 14, 17] 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 [5, 6, 4], while other work, including

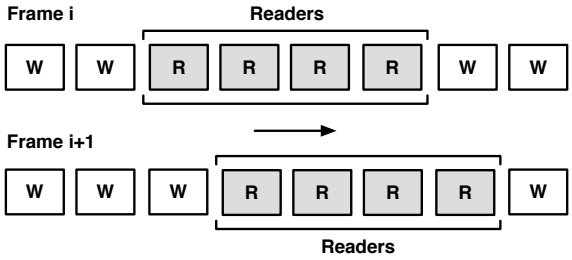


Figure 1: The sliding window moves along the drives. Drives inside the sliding window only perform reads and temporarily store writes in memory.

[1, 3, 8] 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 [17] 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 paper 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 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.

2 Overview

The contribution of this paper 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 paper. Instead, we focus on achieving predictable and efficient read performance under read/write workloads. We present our results in three parts. In Section 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 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.

2.1 System Notes

As described in Section 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 [6], 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:

	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

We chose the above drive models to develop a method, which unlike heuristics (Section 3.2), works under different types of drives, and especially commodity drives. A small number of recent data-center oriented models and

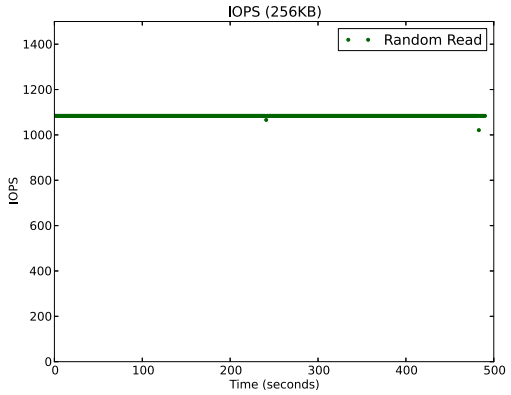
in particular model *C* have placed a greater emphasis on performance stability. For the drives we are aware of, the stability either comes at a cost that is multiple times that of commodity drives (as with model *C*), or is achieved by lowering the random write throughput significantly compared to other 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.

3 Performance and Stability

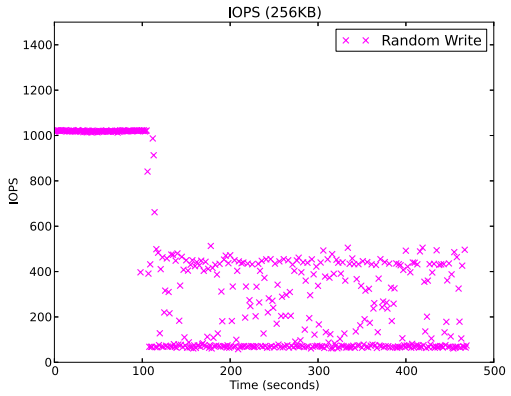
Solid-state drives have orders-of-magnitude faster random access than hard-drives. On the other hand, SSDs are stateful and their performance depends heavily on the past history of the workload. The influence of writes on reads has been noted in prior work [5, 6, 14, 17] for various drive models, and 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 2(a) 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, 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 2(b)), due to write-induced drive operations.

To illustrate the interference of writes on reads, we run the following two experiments on drive *B*. First, we consider three streams performing reads (two sequential and one random), and a single one performing random writes, all covering the same logical range of 100MB. The drive is 99.96% empty. Figure 3(a) 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 relatively stable, which is expected, since more than 80% of the device blocks are clean, reducing write amplification. Second, we consider two streams performing reads (one sequential and one random), and two performing writes (one sequential and one random). The requests now span the first 200GB, instead of only 100MB, and we ensure the drive is first filled completely. Figure 3(b) 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



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

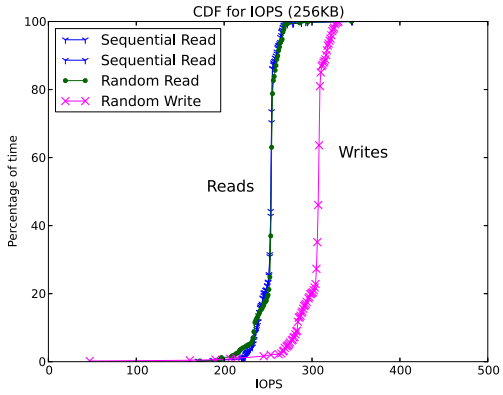


(b) When the drive has limited free space, random writes trigger the garbage collector resulting in unpredictable performance.

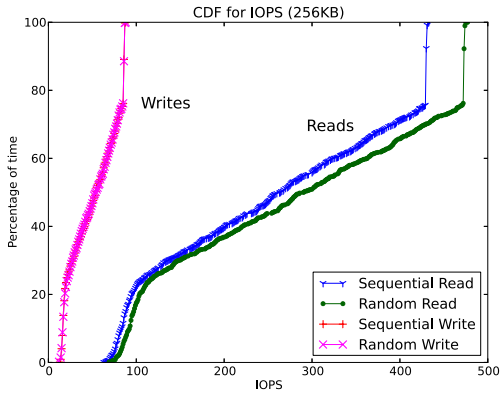
Figure 2: Under random writes the performance eventually drops and becomes unpredictable. (Drive B; 256KB)

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). 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 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 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. As



(a) Writes over a range of 100MB in an otherwise empty SSD lead to stable write performance and the effect of writes on reads is small.



(b) When the drive is filled, performing writes over 200GB leads to unpredictable read throughput due to write-induced blocking events.

Figure 3: The performance varies according to the writing range and the drive's free space. (Drive B; 256KB)

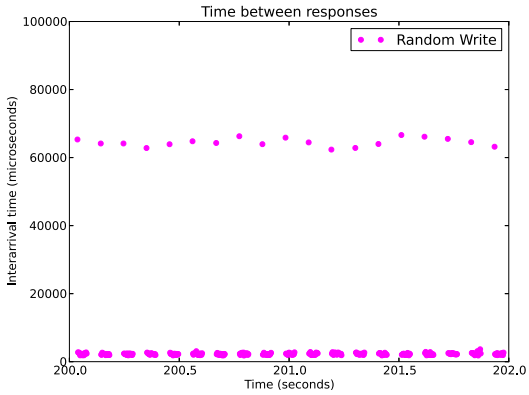


Figure 4: The drive blocks for over 600*ms*/sec, leading to high latencies for all queued requests. (Drive A; 256KB)

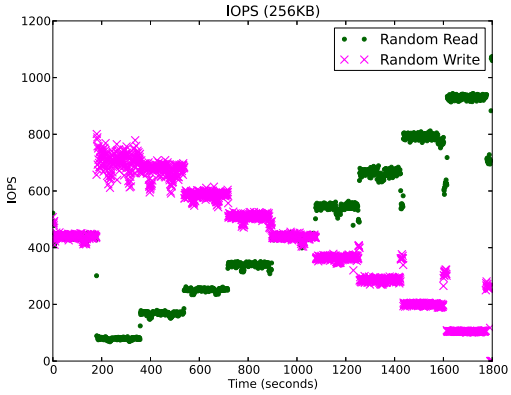


Figure 5: Random reads/writes at a decreasing write rate. Writes have little effect on reads. (Drive *C*; 256KB)

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.

3.1 Performance over long periods

As mentioned in Section 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 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 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.

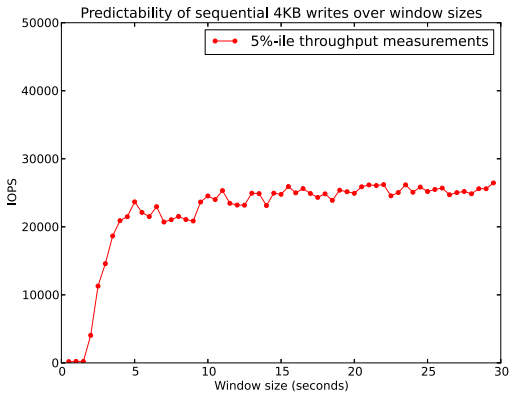


Figure 6: The bottom 5% throughput against the averaging window size. (Drive A; 4KB)

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 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 4.3, we generalize that to support more than two drives, erasure codes, and describe its achievable throughput.

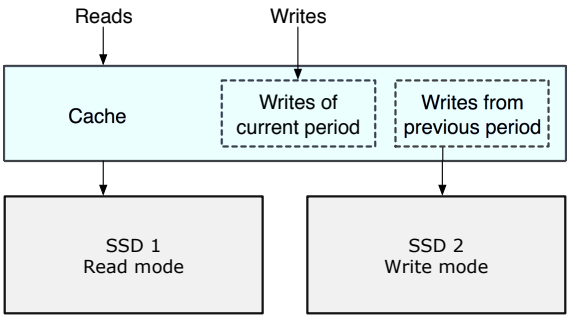


Figure 7: At any given time each of the two drives is either performing reads or writes. While one drive is reading the other drive is performing the writes of the previous period.

4.1 Basic Design

Solid-state drives have fast random access and can exhibit high performance. However, as shown in Section 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 2(b)). Such events can last up to 100ms and account for a significant proportion of the device's time, e.g., 60% (Figure 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 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 [17], 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 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 4.2.3, 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 4.3 a generalization supporting an arbitrary number of drives, allowing us to trade read and write throughput, as well as erasure codes.

4.2 Properties and Challenges

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.

4.2.2 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 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.

4.2.3 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 star-

vation 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 [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 [7]. 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 paper.

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 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.

4.3 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

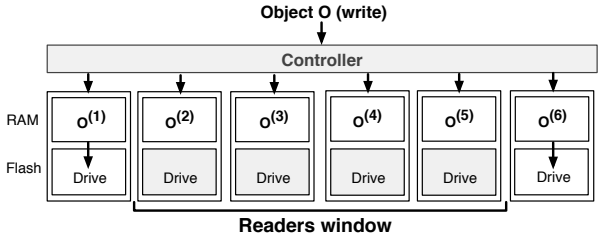


Figure 8: Each object is obfuscated and its chunks are spread across all drives. Reading drives store their chunk in memory until they become writers.

is 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 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 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 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

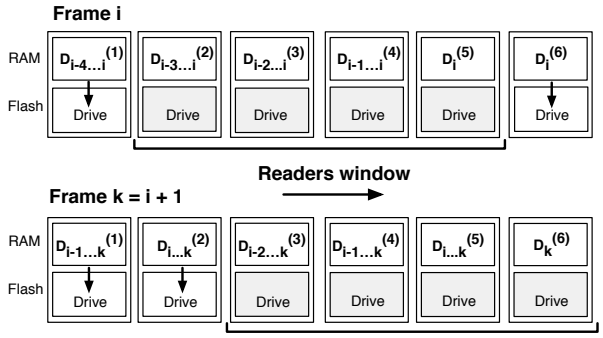


Figure 9: Each node m accumulates the incoming writes across frames $f \ldots f'$ in memory, $D^{(m)}_{f \ldots f'}$. While outside the reading window nodes flush their data.

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 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.

4.3.1 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 10) while model A exhibits few blocking events that affect the system predictability (Figure 13(a)) 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 paper. Having said that, 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.

4.3.2 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.

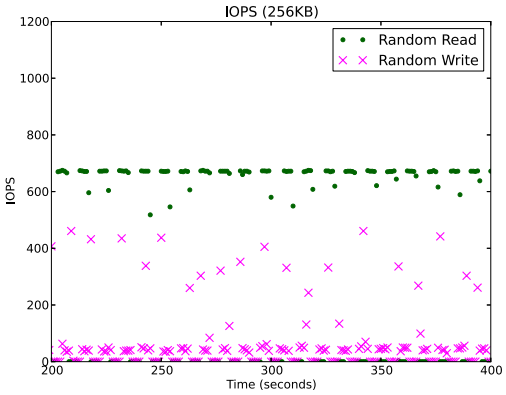


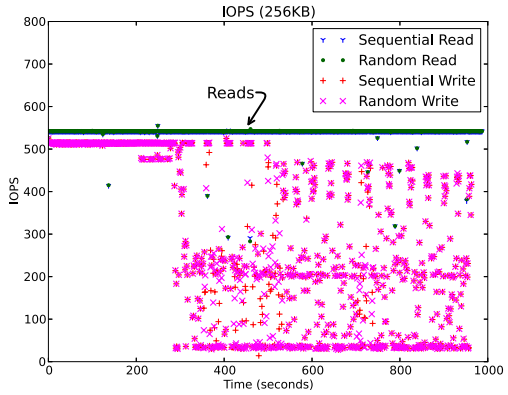
Figure 10: Switching modes as frequent as every 5 seconds creates little variance on reads. (Drive B; 256KB)

Nevertheless, reconstruction speed has improved [18] while optimizing degraded read performance is a possibility, as has been done for a single failure [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.

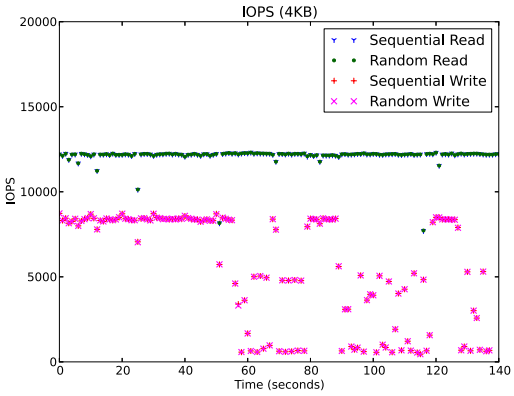
4.4 Experimental Evaluation

We built a prototype of Rails as presented in Sections 4.1 and 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 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 11(a), 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 2(b). Without Rails reads have unstable performance due to the writes (Figure 3(b)).

Increasing the number of drives to three, and using the sliding window technique (Section 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 12(a) shows the performance without Rails is inconsistent when mixing reads



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

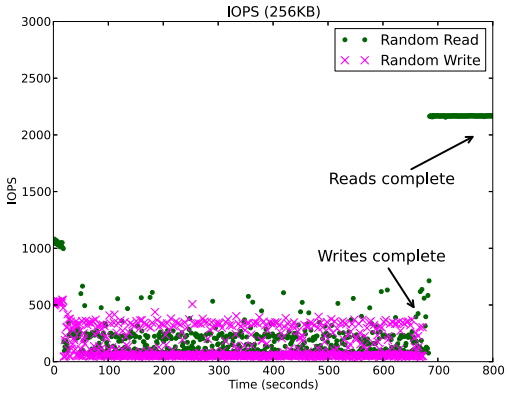


(b) The read throughput remains stable at its maximum performance. (Drive B; 4KB)

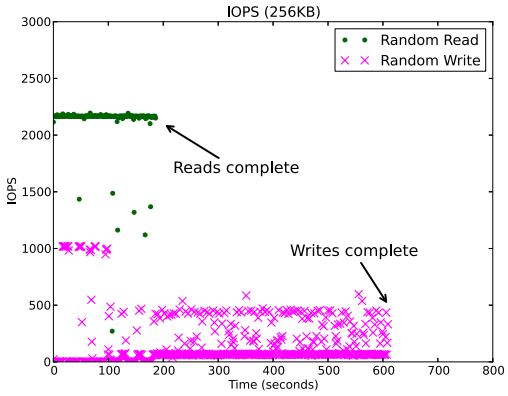
Figure 11: Using Rails to physically separate reads from writes leads to a stable and high read performance.

and writes. In particular, 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 12(b)). 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 13(b) we see that reads have predictable performance around 95% of the time, which is significantly more predictable than without Rails (Figure 3(b)). Moreover, in Figure 13(a) 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

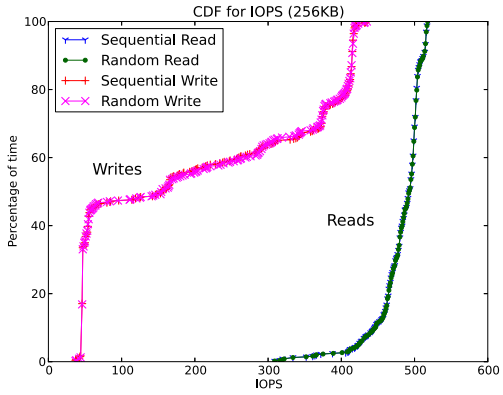


(a) Mixing reads and writes on all 3 drives leads to slow performance for reads until writes complete.

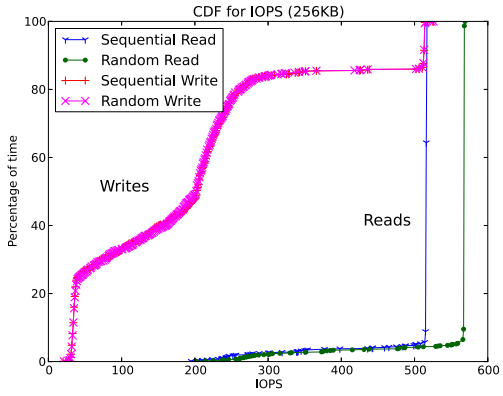


(b) Using Rails to separate reads from writes across drives nearly eliminates the interference of writes on reads.

Figure 12: Client throughput under 3-replication, (a) without Rails, (b) with Rails. (Drive B; 256KB)



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

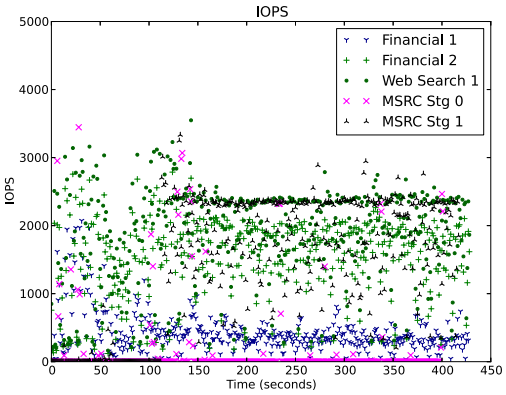
letting the write drive idle before each shift in order to reduce any remaining background work. That way, we found that the interference becomes minimal and 99% of the time the throughput is stable. If providing QoS, that idle time can be charged to the write streams, since they are responsible for 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 2(b)). 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.

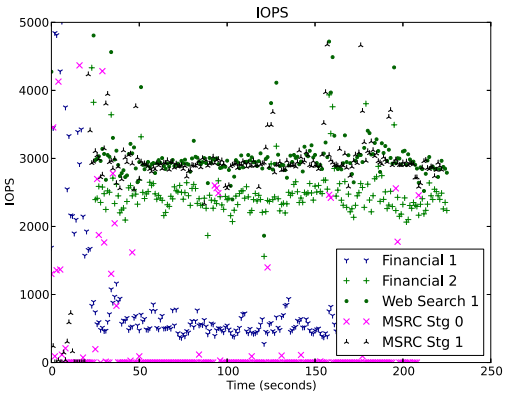
4.5 Evaluation with Traces

We next evaluate Rails with two drives (of model *B*) and replication using the Stg dataset from the MSR Cambridge Traces [16], OLTP traces from a financial institution and traces from a popular search engine [26]. 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 14(a) 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 14(a). Under the same workload our method pro-

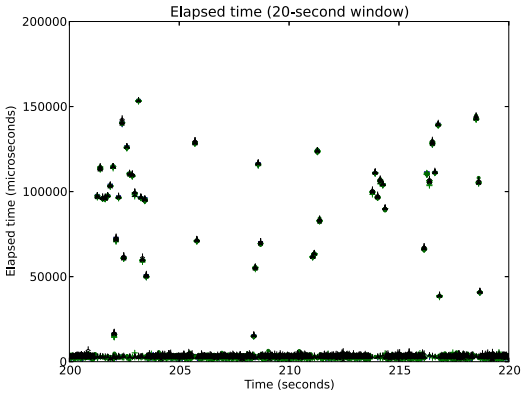


(a) Withour Rails, reads are blocked by writes (not shown) making read performance unpredictable.

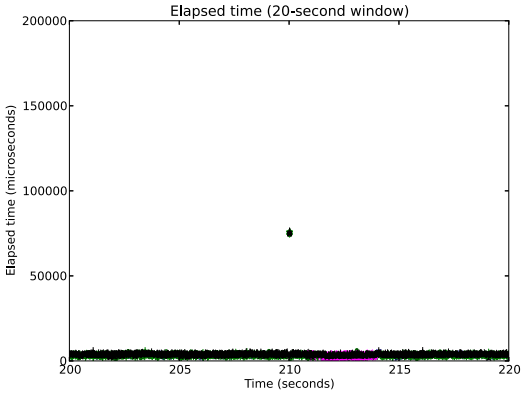


(b) With Rails, reads are not affected by writes (not shown).

Figure 14: Read throughput (a) without Rails, (b) with Rails, under a mixture of real workloads. (Drive B)



(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 15: High-latency events (a) without Rails, (b) with Rails. (Drive B)

vides predictable performance (Figure 14(b)) in the sense that high-latency events become rare. To clearly see that, Figure 15(a) 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 15(b) we see that Rails nearly eliminates the high latencies, therefore providing read-only response time that is low and predictable, almost always.

5 Related Work

Multiple papers study the performance characteristics of SSDs. uFlip [4] presents a benchmark and illustrates flash performance patterns, while the authors in [6] study the effect of parallelism on performance. The work in [5] includes a set of experiments on the effect of reads/writes and access patterns on performance. In addition, Rajimwale et al. present system-level assumptions that need to be revisited in the context of SSDs [20]. Other work focuses on design improvements, and touches on a number of aspects of performance such as parallelism and write ordering [1]. The authors in [8] propose a solution for write amplification, while [3] 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 [9].

In the context of hard-drive storage, DCD [10] proposes adding a log drive to cache small writes and destage them to the data drive. Fahrrad [19] treats sequential and random requests differently to provide predictable performance, while QBox [24] takes a similar approach for black-box storage. Gecko [23] is a log-structured design for reducing workload interference in hard-drive storage. In particular, it spreads the log across

multiple hard-drives, therefore decreasing the effect of garbage collection on the incoming workload.

There is little work taking advantage of performance results in the context of scheduling and QoS. A fair scheduler optimized for flash is FIOS [17] (and its successor FlashFQ [22]). 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. Instead, we use a drive-agnostic method to achieve minimal latency. In another direction, SFS [14] presents a filesystem designed to improve write performance by turning random writes to sequential ones. SSDs are often used as a high performance tier (a large cache) on top of hard-drives [2, 15]. Our solution may be simplified and applied in such cases. Finally, there is recent work on the applications of erasure coding on large-scale storage [11, 21]. We expect our method to be applicable in practice on storage systems using erasure coding to separate reads from writes.

6 Conclusion

The performance of SSDs degrades and becomes significantly unpredictable under demanding read/write workloads. In this paper, 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 [25].

Acknowledgements: We would like to thank the anonymous USENIX ATC reviewers and our shepherd Kai Shen for comments that helped improve this paper.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *USENIX ATC'08* (2008).

[2] ALBRECHT, C., MERCHANT, A., ET AL. Janus: Optimal flash provisioning for cloud storage workloads. In *ATC '13* (2013).

[3] BOBOILA, S., AND DESNOYERS, P. Write endurance in flash drives: measurements and analysis. In *USENIX FAST'10* (2010).

[4] BOUGANIM, L., JÓNSSON, B., AND BONNET, P. uFLIP: Understanding flash IO patterns. In *CIDR '09* (2009), CIDR.

[5] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09* (2009), ACM.

[6] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA '11* (2011), IEEE.

[7] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *USENIX FAST12* (2012).

[8] DESNOYERS, P. Analytic modeling of SSD write performance. In *SYSTOR '12* (2012), ACM.

[9] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *USENIX FAST'12* (2012).

[10] HU, Y., AND YANG, Q. DCD-Disk Caching Disk: A new approach for boosting I/O performance. In *ISCA '96* (1996), ACM.

[11] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *USENIX ATC'12* (2012).

[12] KHAN, O., BURNS, R., PLANK, J., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *USENIX FAST'12* (2012).

[13] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *USENIX FAST'13* (2013), vol. 13.

[14] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. SFS: Random write considered harmful in solid state drives. In *USENIX FAST'12* (2012), pp. 12–12.

[15] MOSHAYEDI, M., AND WILKISON, P. Enterprise SSDs. *Queue 6*, 4 (July 2008).

[16] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write off-loading: practical power management for enterprise storage. In *USENIX FAST'08* (2008).

[17] PARK, S., AND SHEN, K. FIOS: a fair, efficient flash I/O scheduler. In *USENIX FAST'12* (2012).

[18] PLANK, J. S., GREENAN, K. M., AND MILLER, E. L. Screaming fast galois field arithmetic using Intel SIMD instructions. In *USENIX FAST'13* (2013).

[19] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T. M., AND MALTZAHN, C. Efficient guaranteed disk request scheduling with Fahrrad. In *Eurosys '08* (2008), ACM.

[20] RAJIMWALE, A., PRABHAKARAN, V., AND DAVIS, J. D. Block management in solid-state devices. In *USENIX ATC'09* (2009).

[21] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., ET AL. XORing elephants: novel erasure codes for big data. In *PVLDB'13* (2013), VLDB Endowment.

[22] SHEN, K., AND PARK, S. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX ATC'13* (2013).

[23] SHIN, J. Y., BALAKRISHNAN, M., MARIAN, T., AND WEATHERSPOON, H. Gecko: Contention-oblivious disk arrays for cloud storage. In *USENIX FAST'13* (2013).

[24] SKOURTIS, D., KATO, S., AND BRANDT, S. QBox: guaranteeing I/O performance on black box storage systems. In *HPDC '12* (2012), ACM.

[25] SKOURTIS, D., WATKINS, N., ACHLIOPTAS, D., MALTZAHN, C., AND BRANDT, S. Latency minimization in SSD clusters for free. Tech. Rep. UCSC-SOE-13-10, UC Santa Cruz, June 2013.

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