Erasure Coding & Read/Write Separation in Flash Storage

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

Abstract

We want to create a scalable flash storage system that provides read/write separation and uses erasure coding to provide reliability without the storage cost of replication. Flash on Rails [19] is a system for enabling consistent performance in flash storage by physically separating reads from writes through redundancy. In principle, Rails supports erasure codes. However, it has only been evaluated using replication in small arrays, so it is currently uncertain how it would scale with erasure coding.

In this work we consider the applicability of erasure coding in Rails, in a new system called eRails. 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. Finally, through benchmarks we demonstrate that eRails achieves read/write separation and consistent read performance under read/write workloads.

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 previous work [4, 5, 12, 13, 19], 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, Flash on Rails [19] was proposed as 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 paper 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, represented 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.

2 Overview

The contribution of this paper 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 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, 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 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 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.

2.1 System details

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 [14] open-source library with SIMD support enabled [15]. Moreover, we decided to use the Reed-Solomon Vandermonde method because its decoding performance with SIMD support appears higher than

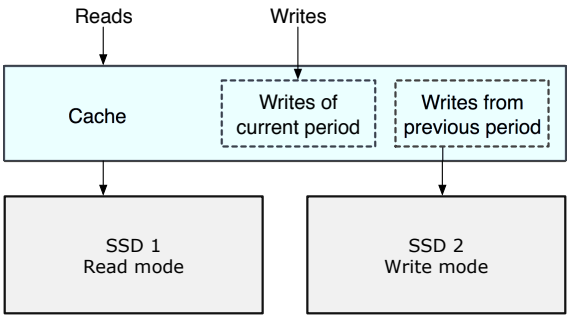


Figure 1: The basic design for read/write separation using replication and two solid-state drives.

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 [19]. In what follows, we provide the background required for the rest of the paper on read/write separation and erasure coding.

3 Background

3.1 Read/Write Separation

To solve the problem of high latency under read/write workloads one may physically separate reads from writes as described in Rails [19, 18]. In this paper, 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 briefly describe how Rails separates reads from writes when using two drives, and afterwards its more space-efficient generalization for multiple drives, which we use for the rest of the paper. Details of how Rails operates and other concerns are covered in the original paper [19] and are not discussed here. Finally, Figures 1 and 2 are include from Flash on Rails [19] for the convenience of the reader.

3.1.1 Basic 2-drive design

The basic approach performs read/write separation using two SSDs and a cache in a single node. By dedicating one SSD to reads, one to writes, and periodically switching their roles, each drive is effectively presented with a read-only workload even if the system receives both reads and writes. This allows the system to achieve optimal read performance. More specifically, at any point in time each drive is either reading or writing. Every T seconds, where T is a large enough time period, e.g., 10 seconds, the drives switch roles. The new write drive performs the writes of the previous period and incoming

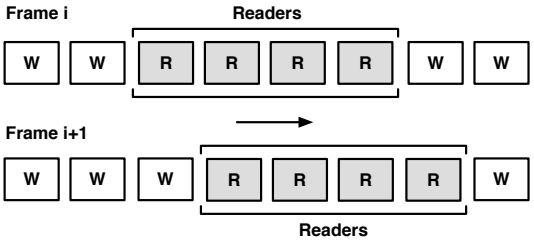


Figure 2: The reading window moves along the drives.

writes, while the new reader drive serves reads (Figure 1). At any point in time the union of the cache with the first drive is equal to the union of the cache with the second drive.

3.1.2 Generalization

The above design can be generalized to N drives, either through replication or erasure coding. In both cases, the drives are placed on a ring. On this ring consider a sliding window, whose size depends on the desired data redundancy. For example, using replication, the window size is allowed to be anywhere between 1 and N drives. Using erasure codes, the window size is bounded from below (Section 4), but can grow up to N. The window moves along the ring one location at a time at a constant speed, transitioning between successive locations "instantaneously" (Figure 2). 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, and while outside the window it empties all information in memory to drive, i.e., it performs the actual writes.

3.2 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 paper, 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.

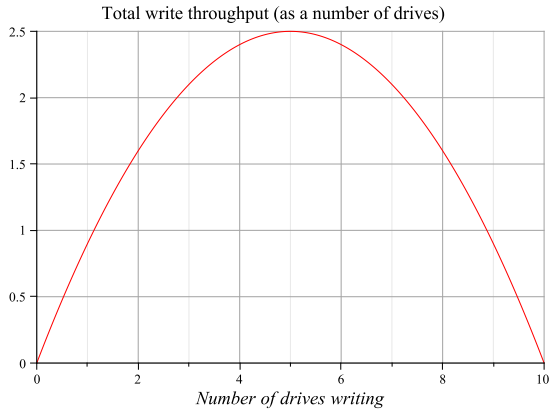


Figure 3: The achievable write throughput of eRails peaks when the number of readers equals the number of writers, i.e., when $k = m$.

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 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 3 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.

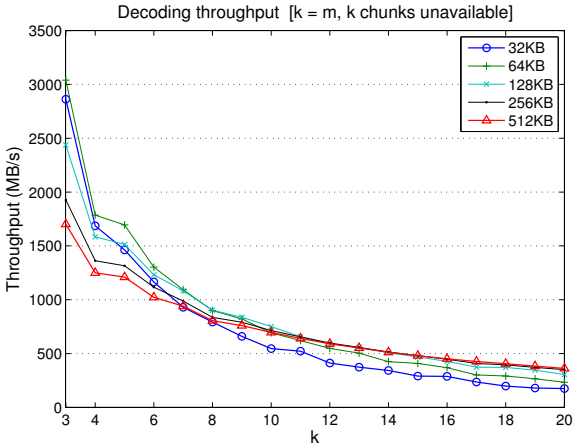
5 Erasure coding overhead

The sliding window (Section 3.1.2) 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.

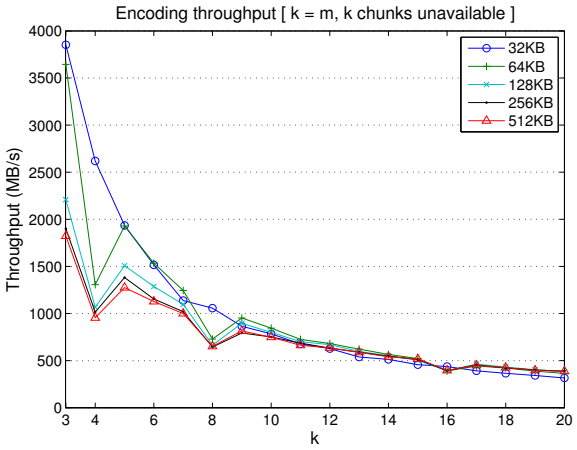
5.1 Throughput and overhead

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 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 4a 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 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



(a) Decoding throughput. (256KB)

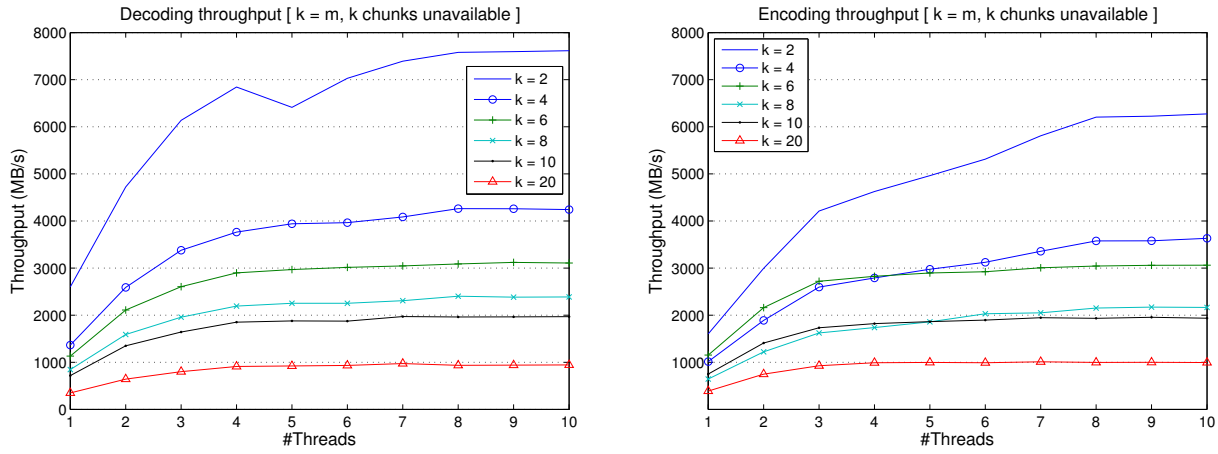


(b) Encoding throughput. (256KB)

Figure 4: Encoding/decoding throughput against k, for various request sizes using a single thread. The throughput drops quickly in k. (256KB)

power of modern commodity systems. Figure 4b 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 5a 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 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 5b is similar to decoding, with the difference that it takes lower values for $k \leq 4$.



(a) Decoding throughput.

(b) Encoding throughput.

Figure 5: Encoding/decoding throughput for various values of k in the number of threads. (256KB)

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.

6 Scaling and computational cost

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 6 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

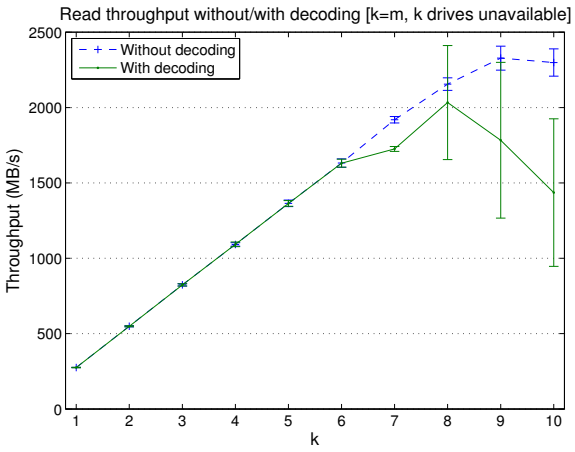


Figure 6: Read throughput using $k+m$ drives with k varying. ($k = m$; 128 KB)

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 6 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 4a, and the fact that CPU resources are limited.

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

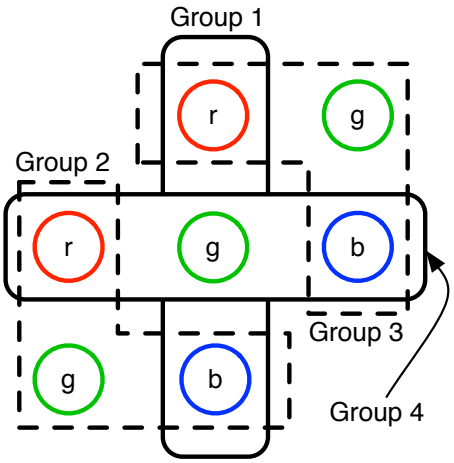


Figure 7: A hypergraph with four (partially) overlapping hyperedges (redundancy groups), each containing three vertices (drives).

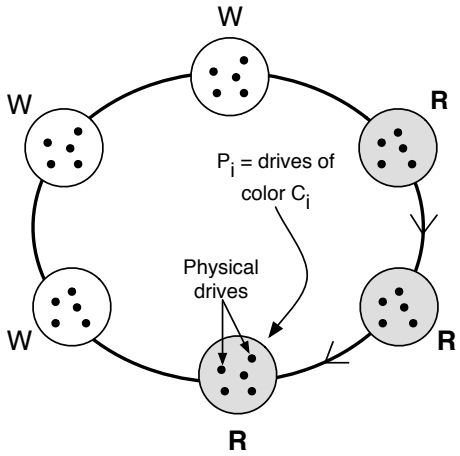


Figure 8: To construct valid redundancy groups of size six, drives are partitioned into six sets. Each group is then constructed by selecting a single drive per set.

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 [22], 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.

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.

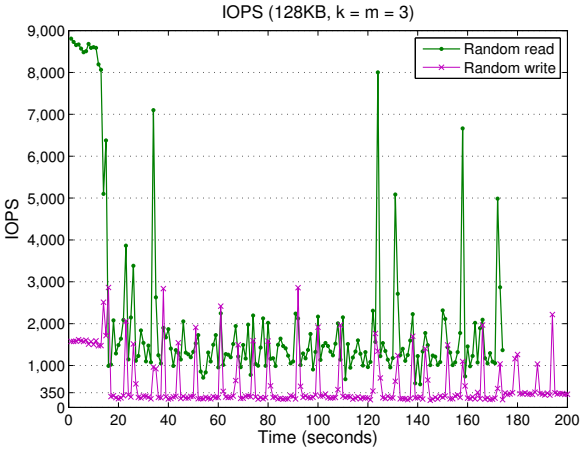
Consider an n-uniform hypergraph H, where vertices represent physical drives (or storage nodes with a single drive) and hyperedges represent redundancy groups of size n. We want to be able to generate n-uniform hypergraphs that allow us to perform read/write separation and which provide good load-balancing. As an example, consider the 3-uniform hypergraph with four hyperedges (redundancy groups) shown in Figure 7. 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 8 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 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.

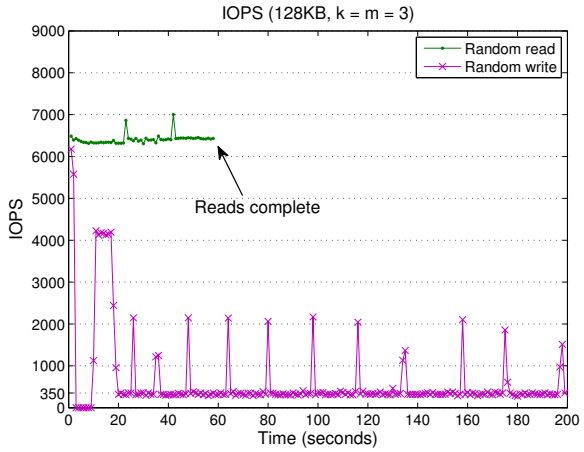
7 Evaluation: Read/write separation

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 9 demonstrates the case where $k = 3$. In particular, Figure 9a 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 9b) the read performance remains high 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



(a) Without eRails.

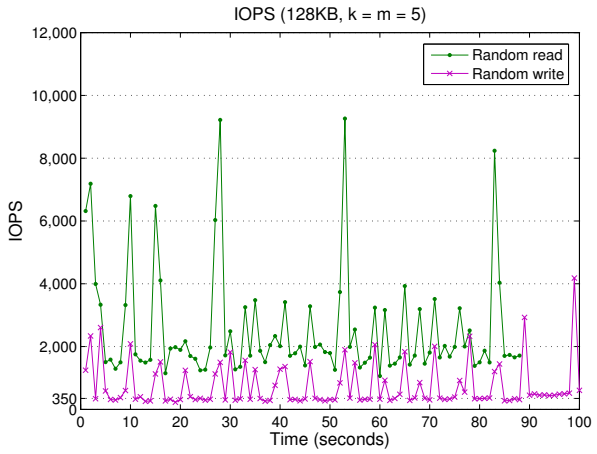


(b) With eRails.

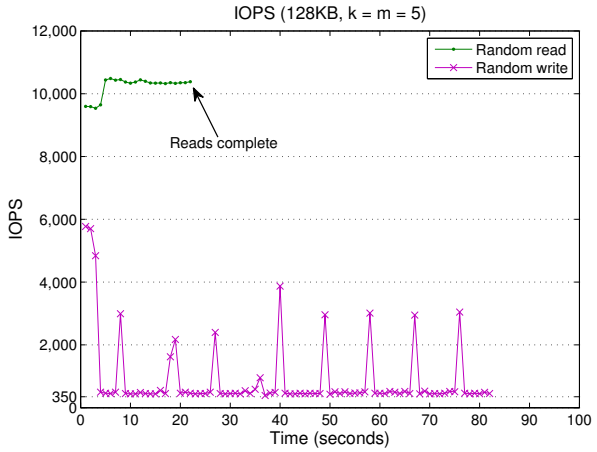
Figure 9: Without eRails the variance of writes "pollutes" that of reads. Using eRails eliminates that variance. ($k = m = 3$; 128KB)

reads (Figure 6). 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 10). First, Figure 10a shows that without Rails the read performance is low and inconsistent. With Rails, Figure 10b 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 6). 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 be-



(a) Without eRails.



(b) With eRails.

Figure 10: Using eRails to physically separate reads from writes leads to a stable and high read performance. ($k = m = 5$; 128KB)

tween 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 due to the batch writing, for example, through its integration with a log-structured block store. However, that is left as future work.

8 Related Work

A large part of research on flash and SSDs focuses on the properties of the devices themselves and improvements that can be performed internally [1, 3, 7]. Work using flash in storage systems often treats flash as a cache [2, 11, 16]. For example, Nitro [11] optimizes for lower capacity through deduplication and compression while Janus [2] provisions flash caches on top of hard-drives. Our work is applicable to flash in general, whether it is treated as a cache or primary storage. In another direction, SFS [12] presents a filesystem designed to improve write performance by turning random writes to sequential ones.

An advantage of erasure coding over replication is the increase of reliability without additional storage space [21]. Erasure coding is used today in large-scale storage systems such as Windows Azure [9], and for big data in Hadoop HDFS [17]. However, research on flash and erasure coding appears limited.

Work on erasure coding performance includes improving degraded reads [10] (with an emphasis on few failures). Using Intel SIMD instructions has been shown to improve coding performance [15], and is something we take advantage of in this work by using the Jerasure library [14]. Moreover, employing GPUs appears as another direction for improving the performance of degraded reads [6, 8]. Finally, with the performance improvements in erasure coding and computational power of commodity machines, we expect eRails to be applicable in real storage systems.that use erasure coding.

9 Conclusion

Flash on Rails [19] is a system for enabling consistent performance in flash storage by physically separating reads from writes through redundancy. In this paper, we presented eRails, a system built on top of Flash on 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. For future work, we plan to implement and evaluate eRails at a larger scale by taking advantage of the hypergraph construction presented in this paper. We expect that to lead us to the peta-scale distributed flash-only storage system, as proposed in [20].

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] 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.

[5] 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.

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

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

[8] GHARAIBEH, A., AL-KISWANY, S., GOPALAKRISHNAN, S., AND RIPEANU, M. A gpu accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (New York, NY, USA, 2010), HPDC '10, ACM, pp. 167–178.

[9] 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).

[10] 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).

[11] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX ATC'14* (Philadelphia, PA, June 2014), USENIX Association, pp. 501–512.

[12] 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.

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

[14] PLANK, J. S., AND GREENAN, K. M. Jerasure: A library in C facilitating erasure coding for storage applications – version 2.0. Tech. Rep. UT-EECS-14-721, University of Tennessee, January 2014.

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

[16] QIN, D., BROWN, A. D., AND GOEL, A. Reliable writeback for client-side flash caches. In *USENIX ATC'14* (Philadelphia, PA, June 2014), USENIX Association, pp. 451–462.

[17] 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.

[18] SKOURTIS, D., ACHLIOPTAS, D., MALTZAHN, C., AND BRANDT, S. High performance & low latency in solid-state drives through redundancy. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (2013), INFLOW '13, ACM.

[19] SKOURTIS, D., ACHLIOPTAS, D., WATKINS, N., MALTZAHN, C., AND BRANDT, S. Flash on rails: Consistent flash performance through redundancy. In *USENIX ATC'14* (Philadelphia, PA, June 2014), USENIX Association.

[20] 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.

[21] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, UK, 2002), IPTPS '01, Springer-Verlag, pp. 328–338.

[22] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. CRUSH: controlled, scalable, decentralized placement of replicated data. In *SC '06* (2006), ACM.