Latency Minimization in SSD Clusters for Free

Dimitris Skourtis, Noah Watkins, Dimitris Achlioptas, Carlos Maltzahn, Scott Brandt

Department of Computer Science
University of California, Santa Cruz
{skourtis,jayhawk,optas,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, 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, under read/write workloads solid-state drives frequently block and exceed hard-drive latency.

In this paper, we propose an efficient approach for achieving performance predictability in distributed storage systems comprised of solid-state drives. By observing that virtually all storage systems incorporate significant redundancy for the purpose of reliability, we propose exploiting this latent resource to achieve the separation of reads from writes across nodes, allowing each drive to periodically serve either read or write workloads. Our proposed approach provides high performance and low latency for reads under read/write workloads while adding no extra cost to the system.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Organization and Design—*Distributed systems*; D.4.8 [**Operating Systems**]: Performance

Keywords

Storage, Distributed Systems, Virtualization

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.

Towards improving performance and predictability, solid-state drives are becoming increasingly common in enterprise storage systems, where they are used as permanent storage or large caches. One of their main advantages over hard-drives is fast random access, which can lead to higher and more predictable performance. Unfortunately, although solid-state drives perform as desired for read-only workloads, under demanding read/write workloads, such as those appearing in consolidated environments, SSDs suffer from high latency and performance unpredictability [2, 3, 5, 7, 9]. To mitigate this, current flash solutions for the enterprise are often aggressively over-provisioned, costing many times more than commodity SSDs or offer low write throughput.

In this paper, we propose a new approach for efficiently providing minimal latency, high throughput and predictable read performance in SSD clusters under read/write workloads. In particular, we observe that many - if not most - distributed 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 periodically 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, i.e., we provide latency minimization for free.

2. PRELIMINARIES

2.1 SSD Performance Predictability

In SSD drives, writes have large latency variance due to the fact that a single write can cause a great amount of "reorganizational" work inside the drive, e.g., garbage collection. In particular, as soon as a drive begins to run low on free space the garbage collector cannot keep up and frequently blocks all queued requests for tens of milliseconds. At a more aggregate level, blocking events can consume half of

the device's time. In contrast, reads on their own have virtually no variance. Unfortunately, in the presence of read/write interleaving, the variance in the latency of writes, "pollutes" the variance of reads, because a read can be blocked on a write that triggered reorganizational work. In other words, the drive performance becomes unpredictable.

2.2 Read/Write Separation

To solve the problem of high latency under read/write workloads one may physically separate reads from writes using two SSDs and a cache in a single node, as was proposed in Ianus [9]. More specifically, 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 as a whole is receiving both reads and writes. This allows the system to achieve optimal read performance. In this paper, we generalize the above and discuss how to achieve the same effect in distributed systems without adding extra drives on each node or over-provisioning flash devices. Instead, we exploit a system's *existing* overprovisioning for reliability to minimize latency for free.

3. PROPOSED METHOD OVERVIEW

We want to build a fault-tolerant storage system by using M identical solid-state drives. We will model redundancy as follows. Each object O stored in our system 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 that the $q|O|$ bits used to represent O are distributed (evenly) among the M drives in such a way that O can be reconstituted from any set of M/q drives. In practice, M is large (in the order of hundreds or thousands), requiring that we distribute objects among redundancy groups of $N \ll M$ drives and use a data placement technique to provide load-balancing and fault-tolerance. Assuming such a data placement method, a natural way to achieve load-balancing within each group 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 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 2.1, 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 $N/q \le s \le N$. 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 $|O| \times q/N$. 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 it has handled 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. Thus, each drive is a read-only drive for $P/N \times s \ge P/q$ 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 performance (both in terms of throughput and in terms of latency). On the other hand, the smaller P is, the smaller the amount of memory needed for each drive.

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 $R/r + Wq/w \le N$ can be supported, where R and W are the read and write loads, respectively.

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 capacity $N-s$ of the system provides write throughput, which thus ranges between 0 and $W_{\text{sep}} = w \times (N - N/q)/q = w \times N(q-1)/q^2 \le wN/4$. As long as the write load does not exceed W_{sep} the system perfoms perfect read/write separation and offers the read latency of a read-only system. We expect that in most 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.

4. DISTRIBUTED STORAGE

In this section we present a distributed storage model and two instantiations, one where the system is used as persistent storage and the other as a large cache. In the next section, we will use this model to describe, in a generic context, our approach for achieving read/write separation in flash-based distributed storage and the challenges we have to face.

4.1 Distributed Storage Model

Distributed systems are complex and proposing a modification can take different forms when applying it on a specific implementation. In what follows, we present an abstract distributed storage model to help us describe system modifications without assuming a specific implementation. To achieve that we treat different system components as logical rather than physical and push most logic to a single layer. We can then map abstract solutions to existing systems, an example of which is shown in Section 6 for Ceph [11].

4.1.1 Model Layers

Our model consists of three logical layers or components illustrated in Figure 1: clients, coordinators, and storage. Clients are end-users 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. Finally, we assume each layer can communicate with any other layer at a cost. We now describe each logical layer:

Clients: From a client's perspective the whole storage system is seen as a single storage device with multiple entry points. In particular, we assume that the system provides to the clients a list of entry points and optionally a suggested rule based on which to pick entries. For example, entries may be chosen randomly to perform load-balancing.

Coordinators: Coordinators accept arbitrary client requests and route them to storage nodes. They have a data placement function $D(O)$, which takes as input an object name and returns a vector with the storage locations of the object's pieces. Moreover, a selection function $L(D(O), \vec{P})$ takes as input the data locations of O and a vector $\vec{P}$ of system or request parameters and returns a subset of drives from which O can be read. Coordinators can communicate with each other to reach a consensus regarding data placement or forwarding parameters. As the number of coordinators increases their logic must be as scalable as the rest of the system. Finally, coordinators contain additional logic to orchestrate operations such as 2-phase commit and journaling.

Storage: Storage is spread across nodes, consisting of a memory with a minimum size and a drive.

The above separation between layers is purely logical. A physical implementation may merge layers to decrease network effects, improve scalability and generally optimize performance. For example, the coordinator logic (e.g., data placement) can be split between clients and storage nodes, removing the coordinator overhead. This model provides an abstraction for creating mappings from abstract modifications onto specific system implementations (Section 6).

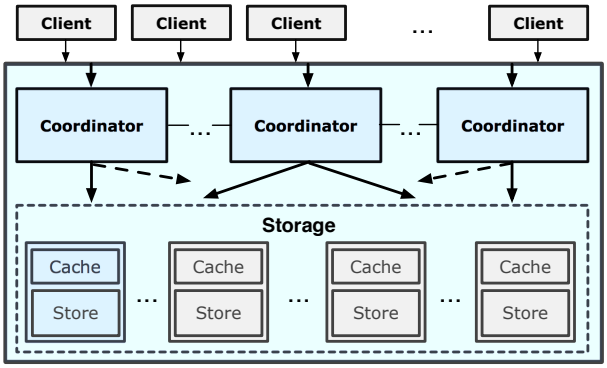

Figure 1: A logical distributed storage system model

4.2 Model Instances

In what follows we briefly describe how our model can be configured to handle permanent, journaled storage and the case where the drives are treated as a large cache.

4.2.1 Journal Support and Data Consistency

In practice, distributed storage systems often perform journaling for fault-tolerance and improved write performance. To describe journaling through our model, we place the required logic in the coordinator layer. In particular, when a write request arrives at a coordinator, the placement function L returns a list containing both data and journal destinations in the storage layer. A write is then performed depending on the consistency requirements of the request or system.

The strong consistency logic is as follows: Besides a single journal, which we call the primary the coordinator sends a write request to all journal destinations, which we call secondary. As soon as each secondary receives the write data, it acknowledges to the coordinator and starts performing the write request. When the coordinator receives acknowledgment from all secondaries, the primary is sent the write. When all journals commit, the coordinator responds to the client, and forwards the write request to the corresponding data drives, where a similar operation takes place.

When a coordinator receives a read, it forwards it to the corresponding primary journal. If there is no cache hit, the request is forwarded by the coordinator to the corresponding primary data node. Under eventual consistency, any journal or data drive can receive reads directly. In the case of a node failure, the coordinator retrieves the write operations from the journal and replays them on the data storage.

4.2.2 Flash as a Cache

SSDs are often used as a high performance tier (a large cache) on top of hard-drives [1, 6]. Treating SSDs as a cache removes the need for journaling. Instead, the system can respond back to each write request as soon as the data is in the DRAM of every storage node in $D(x)$. This allows us to

provide predictable, high performance writes. In particular, the coordinator may throttle requests to match the drive's average write throughput. Throttling may be achieved by assigning a time cost to each request based on its characteristics. For the rest of this paper we concentrate on the first configuration, since it is easy to reduce it to the second one.

5. PROVIDING MINIMAL LATENCY

Given a distributed system and a number of clients with arbitrary read/write 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 2.2. We assume the provided system was designed properly, in that it has enough network capacity to take advantage of SSDs. In what follows we describe some of the challenges to be addressed in achieving read/write separation in distributed systems and proposed techniques.

5.1 Distributed Read/Write Separation

A simple approach for achieving read/write separation in distributed systems is to place $N \ll M$ drives on each node as a group and apply the Ianus technique (Section 3) locally. For example, when $N = q = 2$ each node has two drives and performs replication locally (Section 2.2). 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 paper is as the above with the difference that the drives of each group are spread among nodes in order to use existing resources.

5.2 Challenges and Techniques

There is a number of challenges in achieving read/write separation across nodes that are specific to distributed systems. We now describe some of those challenges and for each of them discuss possible approaches or solutions.

5.2.1 Data Placement

In a system with M drives, where M is large (hundreds or thousands), splitting an object O into M pieces is impractical. Instead, a fault-tolerant distributed system would split O over a set of $N \ll M$ drives. To retain load-balancing, each drive is part of multiple sets 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, to allow read/write separation we need to construct drive sets carefully.

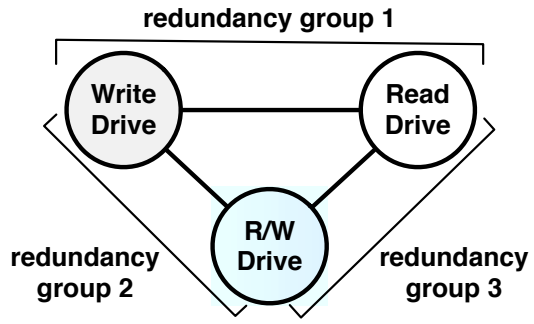

Figure 2: Odd-length cycles makes read/write separation impossible, since a node performs both reads and writes.

To formally discuss data placement, we use the term redundancy group to refer to the set of drives over which the pieces of a given object are spread. Also, we consider data placement graphs, where nodes represent drives and edges indicate that nodes have to be in different mode (read/write). Consider a set of three drives and three redundancy groups, as shown in Figure 2, with each group receiving a subset of the workload. In the same graph, it follows that read/write separation is impossible because one of the drives must be in mixed mode. Instead, by adding a fourth drive we can construct a cycle of length four which enables separation. We infer that when half the drives in each group perform reads (and similarly writes), read/write separation is possible, exactly when the graph is 2-colorable, i.e., bipartite.

More generally, we have the following: If all redundancy groups are even-sized and the ratio of read to write drives is the same across groups in a graph component, then read/write separation is possible exactly when the graph is 2-colorable. An extension to both even and odd-sized groups is as follows: If the number of drives in a group is odd, we treat the odd drive as always write-only and consequently require the graph to be 3-colorable, where the third color corresponds to write-only drives. These results allow for a number of configurations including the option to trade writers for readers and vice versa at the system (or graph component) level, which is useful for systems with unstable read/write load.

The above can be extended to increase the system's flexibility by supporting different reader/writer ratios across overlapping groups. In that case, we adjust the writing period of each group based on its neighbors and ensure each drive can perform the sent writes while in write mode. Due to limited space in this paper we will describe that extension elsewhere.

5.2.2 Synchronous Mode Switching

Coordinators distribute requests among the storage nodes. Therefore, we need to ensure that they agree on which drives should be receiving reads and consequently, which ones are in write mode during each frame. The above implies that coordinators need to reach a consensus. We refer to the switch-

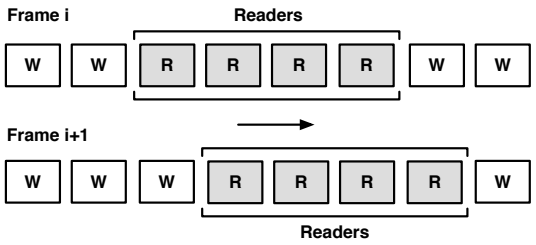

Figure 3: The reading window moves along the drives.

ing policy of a system as the rule based on which drives switch roles between reading and 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. Given that we switch drive modes every multiple seconds, e.g., 10 seconds, NTP is expected to provide good enough synchronization. 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 log storage or memory cache (depending on the system), and only eventually written to a data drive. Hence, the storage can give priority to reads when briefly presented with a mixed workload. In other words, brief synchronization delays have no effect on read isolation. Finally, it is important that we quickly identify nodes whose clock is continuously drifting and resolve the issue.

Generic switching policies: If the switching policy is not purely time-based, the coordinators have to reach a consensus regarding the time of the next switch based on parameters, such as the amount of data written per drive. In general, reaching a consensus over a growing set of nodes limits scalability. In our case, we only need to reach a consensus every multiple seconds and we 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 scalability within practical limits is possible in our case. Different methods are possible, such as reaching a consensus over a small number of coordinators or electing a leader. The decision could then either be broadcasted or propagated through a gossip protocol.

5.2.3 Node Failures

A challenge with respect to data placement is node failures. That is, if a node fails we have to find a replacement, which is in the same mode. To achieve that we would like to have a function, instead of a shared table, that would compute that node quickly to preserve performance.

5.2.4 Data Consistency

To provide strong consistency, each redundancy group has a primary node acting as the entry point of the group for reads and writes. Alternatively, under chain replication the tail of the group may be serving the reads. Since we are interested in alternating the mode of each drive, over time, we need to allow more than one node to serve the same data. If the system requires eventual consistency we can achieve that while allowing any drive in a redundancy group to serve reads directly. For writes, we may reply back to a write request as soon as the data is committed on every node.

There are at least two ways to provide strong consistency while performing read/write separation. One approach is to send every read request to the primary but ask it to forward the request to another group node according to L, unless there is a cache hit in the primary. Another approach is to generalize a method such as CRAQ [10] to handle read/write separation and help us avoid turning the primary into a bottleneck. CRAQ [10] improves the read throughput by allowing all drives in a group to perform reads while providing strong consistency. Next, we present an analysis for the achievable throughput under replication and erasure coding while providing read/write separation.

5.3 Replication Performance

Replication is a simple and common solution for fault-tolerance and availability. To support the physical isolation of reads/writes under various replication factors on the same system we can apply our methods for data placement as presented in Section 5.2.1. In terms of performance, the number of read and write drives within each redundancy group may be divided as required, allowing us to trade read for write performance and vice versa, at a local and global level. Let n be the number of drives reading and m the number of drives writing such that $(n+m)$ is even. If we allow half the drives in each group to perform writes at a time, then each group supports a throughput of $w/2$, where w is the write throughput of a single drive.

In general, each group supports up to $mw/(m+n)$ amount of writes. In other words, if we are given a fixed number of drives $N = m+n$, we can increase the write throughput linearly by turning more of the drives into writers. Of course, under replication the maximum write throughput is equal to that of a single drive and is attained when $N = m$. The read throughput is simply nr or $(N-m)r$. Note that we do not propose adding replicas to increase performance, there are other methods such as striping to achieve that. Instead, we note that under replication the system has complete flexibility in terms of its read-to-write throughput ratio.

5.4 Erasure Coding

Erasure codes provide fault-tolerance in storage systems at a much lower space cost than replication. We now look into the achievable throughput under read/write separation as described in Section 3.

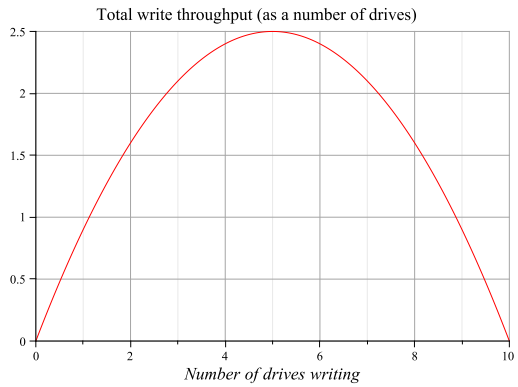

Figure 4: The achievable write throughput under erasure coding peaks when we have the same number of readers and writers.

5.4.1 *Performance Analysis*

Let n and m be the number of read and write drives, respectively. The sustainable system write throughput grows in the number of write drives as in Figure 4, until $m = n$.

To explain why the maximum write throughput occurs when $m = n$, note that if $q > 1$ is the obfuscation factor induced by erasure coding and the write load is W, then internally the system performs qW writes. Thus, $W \leq w \times m/q$. At the same time, there must be enough read drives to reconstruct from, i.e., $n \geq (n+m)/q$. Combining the two inequalities and setting $N = n+m$ yields $W \leq (N-m)nw/N$, which is maximized when $n = m$. Taking $m > N/2$ *reduces* the write throughput as it requires an even greater amount of obfuscation (since we must reconstruct from fewer readers).

The optimal values of n, m to the left of the peak are determined by q. Write throughput is maximized when $n = N/q$, i.e., the smallest possible allowing reconstruction. Of course, if the write load is below the maximum we can increase n further. By computing $(n+m)/(n+W)$, we see that a system with no redundancy whatsoever would never offer more than 33% additional write throughput and would only offer 6% when $m < N/4$, i.e., when erasure coding allows reconstruction from any subset of $0.8N$ drives. The price is that reading entails reconstruction, i.e., computation. Nevertheless, there are improvements regarding reconstruction speed [8] while optimizing degraded read performance is a possibility, as was done in [4] for a single failure.

6. MINIMAL LATENCY WITH CEPH

Earlier, we presented our approach for read/write separation using a model system. We now consider Ceph, and discuss how to map our proposed solution on it.

6.1 Overview

Ceph is a scalable, distributed storage system supporting object, block and POSIX file system interfaces. At a high level it works as follows. Clients map object names using a hash function called CRUSH [12] 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.

6.2 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 $D(x)$ 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 $L(D(x), \vec{P})$ is part of the primary nodes logic and by default points to the primary.

Unlike the default behavior of Ceph, to provide read/write separation we need to send reads to group nodes other than the primary. In fact, L 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. In particular, at a high level, reads pass through the primary's cache. If there is a cache hit the primary responds with the data, otherwise, the read request is forwarded to a read drive based on the L function. When the primary receives the read response it applies any changes it may have in its cache and responds to the client. The above ensures that clients always receive the latest committed version of their data. To apply our data placement techniques to Ceph, we need to restrict its placement group generation. Fortunately, the restrictions in Section 5.2.1, still allow for a significant number of graphs. Therefore, we expect the load-balancing performed in Ceph to still be as good as originally.

7. CONCLUSION

SSDs have high random access but exhibit high latency or low throughput under read/write workloads. We considered SSD clusters and presented an approach for providing read-only performance under mixed workloads. Our approach is based on the observation that many systems already perform redundancy. We exploit this by physically separating reads from writes without additional over-provisioning. We plan to further develop the presented ideas, and illustrate our performance goals can be achieved in production systems, at a fraction of the cost compared to current solutions.

8. REFERENCES

[1] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and C. E. Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC '13, Berkeley, CA, USA, 2013. USENIX Association.

[2] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[3] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 266–277, Washington, DC, USA, 2011. IEEE Computer Society.

[4] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[5] C. Min, K. Kim, H. Cho, S. Lee, and Y. Eom. Sfs: Random write considered harmful in solid state drives. In *FAST'12: Proceedings of the 10th Conference on File and Storage Technologies*, 2012.

[6] M. Moshayedi and P. Wilkison. Enterprise ssds. *Queue*, 6(4):32–39, July 2008.

[7] S. Park and K. Shen. Fios: a fair, efficient flash i/o scheduler. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 13–13, Berkeley, CA, USA, 2012. USENIX Association.

[8] J. Plank, K. Greenan, and E. L. Miller. Screaming fast galois field arithmetic using intel simd extensions. In *Proceedings of the 11th Conference on File and Storage Systems (FAST 2013)*, Feb. 2013.

[9] D. Skourtis, S. Brandt, and C. Maltzahn. Ianus: Guaranteeing high performance in solid-state drives. Technical Report UCSC-SOE-13-08, Department of Computer Science, University of California, Santa Cruz, May 2013.

[10] J. Terrace and M. J. Freedman. Object storage on craq: high-throughput chain replication for read-mostly workloads. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 11–11, Berkeley, CA, USA, 2009. USENIX Association.

[11] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[12] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.