Ursa: Hybrid Block Storage for Cloud-Scale Virtual Disks

Huiba Li*
Alibaba
Beijing, China

Yiming Zhang*†
NiceX Lab, PDL, NUDT
Changsha, Hunan, China
zhangyiming@nudt.edu.cn

Dongsheng Li
NUDT
Changsha, Hunan, China

Zhiming Zhang
mos.meituan.com
Beijing, China

Shengyun Liu
NUDT
Changsha, Hunan, China

Peng Huang
Johns Hopkins University
Baltimore, Maryland, USA

Zheng Qin
NUDT
Changsha, Hunan, China

Kai Chen
HKUST
Hong Kong, China

Yongqiang Xiong
Microsoft
Beijing, China

Abstract

This paper presents Ursa, a hybrid block store that provides virtual disks for various applications to run efficiently on cloud VMs. Trace analysis shows that the I/O patterns served by block storage have limited locality to exploit. Therefore, instead of using SSDs as a cache layer, Ursa proposes an SSD-HDD-hybrid storage structure that directly stores primary replicas on SSDs and replicates backup replicas on HDDs, using journals to bridge the performance gap between SSDs and HDDs. Ursa integrates the hybrid structure with designs for high reliability, scalability, and availability. Experiments show that Ursa in its hybrid mode achieves almost the same performance as in its SSD-only mode (storing all replicas on SSDs), and outperforms other block stores (Ceph and Sheepdog) even in their SSD-only mode while achieving much higher CPU efficiency (performance per core). We also discuss some practical issues in our deployment.

CCS Concepts • **Computer systems organization → Cloud computing**; **Reliability**; **Availability**; **Secondary storage organization**.

Keywords SSD-HDD-hybrid, cloud storage, virtual disks

ACM Reference Format:
Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. 2019. Ursa: Hybrid Block Storage for Cloud-Scale Virtual Disks. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3302424.3303967

1 Introduction

In recent years, we see a trend of traditional desktop [19] and server [74] applications migrating to the cloud. These applications run inside VMs seamlessly with virtual disks that are backed by remote block storage, such as Amazon EBS (Elastic Block Service) [3], Microsoft Azure Storage [5], OpenStack Cinder [18], Alibaba Storage [13], and Tencent QCloud [20]. However, the I/O patterns and consistency requirements of these traditional applications are different from cloud-native applications that adopt specific programming models and paradigms such as MapReduce [47] and RDD [100]. The discrepancies make it difficult for the migrated applications to enjoy the high performance of cloud storage. Although recent advances have largely improved the throughput of virtual disks, e.g., Blizzard exploits disk parallelism [74], it is still challenging to satisfy the requirements of cloud virtual disks in high performance at low cost.

The traces from both existing work [77] and our production block store demonstrate two characteristics (detailed in §2) of the workloads from migrated applications. First, small I/O is dominant in these applications, with occasional large sequential I/O. Second, both reads and writes have limited locality perceived by the block storage.

Since small I/O is dominant in the migrated applications, the performance of block storage is largely determined by the performance of small reads/writes, making SSDs (solid state drives) preferable to HDDs (hard disk drives) in building block storage. However, the much higher price and energy

*Co-primary authors.
†The work was done when Yiming Zhang was a visiting researcher at MSRA.

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303967

cost of SSDs [83] make it too expensive for a large fraction of customers to store all block replicas on SSDs. The limited locality suggests that solutions using SSDs as a cache layer [66] are ineffective in accelerating I/O of the migrated applications because of both the high cache miss ratio and the huge SSD-HDD performance gap. As discussed in [101], even a 1% cache miss ratio might degrade the average I/O performance by a factor of 10, considering that high-end SSDs are three orders of magnitude faster than HDDs in both latency and IOPS. Further, using SSDs as a cache layer helps little in improving tail latency, which is particularly important for cloud storage for guaranteeing SLA [50]. An additional cache layer also makes block storage highly prone to consistency problems [82], such as the severe outage in Facebook's service due to cache misconfiguration [23].

Based on these observations, we propose an SSD-HDD-hybrid storage structure (§3.2) that stores primary replicas on SSDs and replicates backup replicas on HDDs, so as to achieve I/O performance similar to SSDs but at cost comparable to HDDs. To address the challenge of the huge performance gap between primary SSDs and backup HDDs for random small writes, we design journals that bridge the SSD/HDD performance gap by transforming the dominant random small backup writes into sequential journal appends, which are then asynchronously replayed and merged to backup HDDs. For efficiency, occasional large sequential writes are directly performed on backup HDDs (bypassing journals). We adopt this hybrid structure to design Ursa, a high-performance and low-cost block store that provides efficient virtual disks for cloud VMs. This paper makes the following contributions.

First, it presents the design of an SSD-HDD-hybrid block storage system. To address the complexity of locating backup data due to the combination of journal appending (for small backup writes) and replica copying (for large backup writes), we design an efficient LSMT (log-structured merge-tree) [84] based index structure for journals (§3.3); this structure supports fast journal queries for (i) quick invalidation of stale journal appends and (ii) fast reads of journal data during failure recovery.

Second, we systematically exploit multi-level parallelisms in Ursa (§3.4), mainly including (i) *on-disk* parallel I/O, (ii) *inter-disk* striping, out-of-order execution, and out-of-order completion, and (iii) *in-network* pipelining, so as to improve Ursa's IOPS and throughput performance.

Third, we design Ursa's Replication Protocol (§4.1) to satisfy the strong consistency (linearizability) [58] requirement for migrated desktop and server applications [74]. We also design rich-featured client (§5.1) and efficient mechanisms for online component upgrade (§5.2).

Experiments show that Ursa in its hybrid mode provides almost the same performance as in its SSD-only mode (storing all replicas on SSDs), and outperforms state-of-the-art open-source block stores (Ceph [98] and Sheepdog [17]) even in their SSD-only mode while achieving much higher CPU efficiency (IOPS and throughput per core). Real-world deployment shows that Ursa has similar or better performance compared to commercial SSD-only block storage services (Amazon AWS [3] and Tencent QCloud [20]) while using much fewer SSDs.

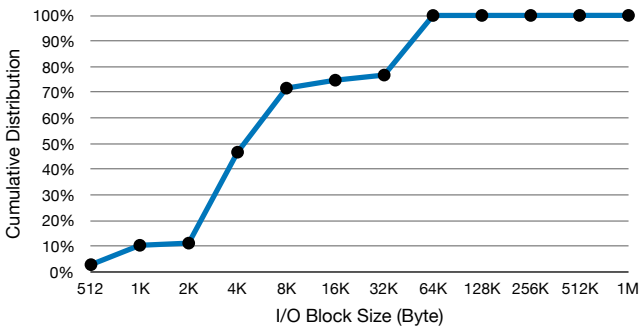


Figure 1. CDF of I/O block sizes. Note that HDDs support 512-byte sectors in physical or emulated modes [7].

2 Motivation

To understand the I/O patterns of traditional desktop and server applications served by block storage, we analyze the traces [77] of Microsoft's block storage service gathered below the filesystem cache and capturing all the block-level reads and writes performed over one week by different applications on different volumes. We first calculate the cumulative distribution of all I/O block sizes. The result shown in Fig. 1 indicates that more than 70% of I/O sizes are at most 8KB, and almost all I/O sizes do not exceed 64KB.[1] This implies that small I/O sizes are dominant for block storage when serving desktop/server applications.

Because HDDs are extremely slow for random small I/O, SSDs are gaining increasing popularity for building high-performance storage systems [27, 28, 48, 49, 69]. Traditional SATA SSDs outperform HDDs by more than two orders of magnitude in IOPS and I/O latency, as well as by several times in throughput for random small I/O workloads, while even faster PCI-Express (PCIe) SSDs have already been widely used for high-end storage. Besides, SSDs also have much lower failure rate than HDDs (owing to the lack of mechanical movement) [24] and have similar median lifespans (of about 5 years) as HDDs [22, 78].

The main drawback of SSDs is the much higher price and energy usage per bit compared to HDDs [83], making it too expensive for many customers to adopt SSD-only block storage with n-way replication[2] (storing all the n replicas on

[1] There do exist some block reads/writes with sizes larger than 64KB, but the ratio is very small.

[2] In n-way replication, the block storage system stores n replicas for each data chunk. Chunks are fixed-size (normally 64MB) data blocks for organizing the data of virtual disks.

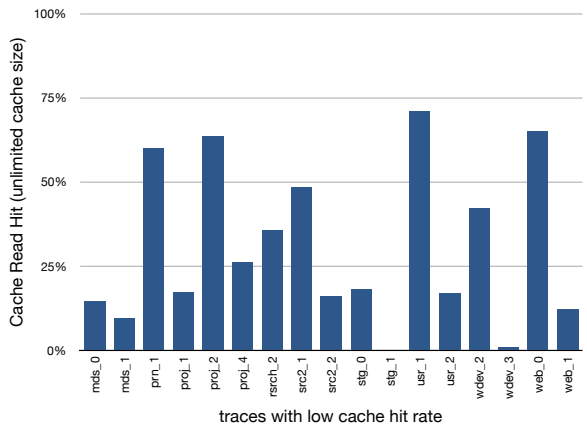


Figure 2. Cache hit ratio for reads in block storage.

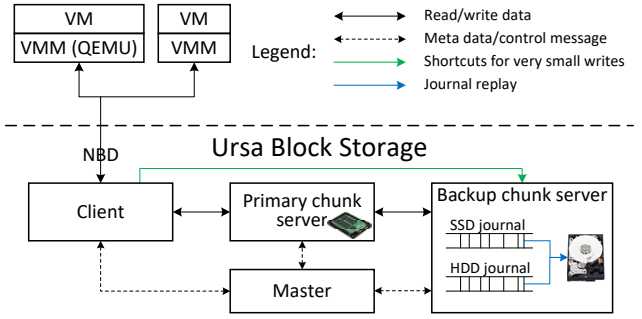


Figure 3. Ursa architecture. Clients are a portal providing the block interface to VMMs (virtual machine monitors).

SSDs). A possible solution is to use RAM-/SSD-based cache between clients and the primary storage [66] to mitigate the performance inefficiency of HDDs. However, cache has proven to have little effect in absorbing writes [30, 77]. Our analysis of the effect of cache of block storage in absorbing reads reaches similar conclusions. We replay the per volume traces [77] against a simulated cache in the write-back mode. We simplify the simulation by assuming an unlimited cache size (to avoid overflows) as well as an infinite write-back speed (to keep cached blocks always clean). The cache hit ratio in practice is lower than in our simulation, because HDDs have limited cache size and relatively-low write-back speed.

Fig. 2 depicts the 17 traces that have low cache hit ratios (less than 75%) of reads. Nearly half of the total 36 traces [77] have low cache hit ratios because large amounts of data blocks are read only once. The cache hit ratios would be even lower in realistic scenario with a limited cache size because eviction policies could potentially evict some hot data.

Considering the 2~3 orders of magnitude performance gap between SSDs and HDDs, even a small miss ratio would dramatically degrade the I/O performance. Further, an additional cache layer would significantly complicate consistency issues [23, 82]. In addition, the uncertainty of cache hit ratio prevents cloud vendors from providing clear SLAs (service-level agreements) to their customers, which are critical for business.

Therefore, in this paper we design an SSD-HDD-hybrid block storage structure, which directly stores primary replicas on SSDs and copies backup replicas on HDDs. All client reads are (normally) served by the primary replicas on SSDs, so there are no cache misses at all. A direct challenge for the hybrid structure is the huge performance gap in random small writes between primary SSDs and backup HDDs: if every write had to be copied to the corresponding backup HDDs before returning, then the SSD-HDD-hybrid structure would have little effect, because the write performance would always be bounded by the backup HDDs.

To address this problem, we propose to use journals to transform random backup writes to journal appends, which are asynchronously replayed and merged to the backup HDDs. The journals amortize the backup load and make it sustainable for backup HDDs. Although the long-term average performance of random writes is still bounded by the speed of journal replay (to the backup HDDs), in practice the *client-perceived* performance of random small writes could be as high as the journal's sequential write performance.

3 Design

3.1 Architecture Overview

As shown in Fig. 3, an Ursa block store consists of (i) chunk servers storing primary/backup data chunks, (ii) clients serving as portals providing block interface for VMs to read/write data, and (iii) a master managing the cluster.

Chunk server. A data chunk has one primary replica and multiple backup replicas for durability, which are stored on different machines. Each machine installs multiple SSDs and HDDs, and runs both primary (chunk) servers storing primary replicas on SSDs and backup (chunk) servers storing backup replicas on HDDs.

Client. VMMs access the block storage using clients as a portal via the NBD (network block device) protocol [15]. Clients interact with the master for metadata queries and management requests (such as virtual disk creation, opening, and deletion). The interactions between clients and chunk servers are designed to be *stateless*, because otherwise it would be complex and error-prone to maintain consensus about their states in large-scale storage systems.

Master. Ursa adopts a global master in the same vein as GFS [54] to simplify management. We minimize the master's involvement in normal I/O paths to prevent it from becoming a bottleneck. The master provides coordination services, including virtual disk creation/opening/deletion, metadata query, status monitoring, failure recovery, etc.

3.2 SSD-HDD-Hybrid Storage Structure

Ursa follows the primary-backup paradigm [83], with clients normally sending read/write requests to the primary servers (Fig. 3). We achieve SSD-level I/O performance at HDD-like cost by collaboratively storing the primary replicas on SSDs and the backup replicas on HDDs. For a read request, the primary server directly reads the data from SSD. For a write request, the primary server writes the data on SSD, replicates it onto the backup servers (which return after writing to their on-disk journals), and finally returns to the client.

In this hybrid structure, primary SSDs and backup HDDs have a huge performance gap in handling the random small writes that dominate the workloads of block storage. Therefore, Ursa uses a journal for each backup HDD, which transforms small backup writes to sequential appends and asynchronously replays the appends to the backup HDD.

Although journal appends are well-supported by both SSDs and HDDs, Ursa prefers to store the journals on SSDs, so that the journals can be *continuously* replayed without affecting other simultaneous reads/writes on the same SSDs, by leveraging SSDs' highly efficient support for parallel I/O. Ursa uses an in-memory index (§3.3) for each data chunk to map from the chunk offsets in backup writes onto the journal offsets.

SSD journals enable the *short-term*, client-perceived backup performance to be as high as the sequential write performance. In addition, for *long-term* backup performance, it is also more efficient to use journals as a bridge between primary SSDs and backup HDDs than to directly perform backup writes on backup HDDs, although both of them are bounded by the random small write performance of backup HDDs. This is because (i) a large proportion of writes is *overwrites* in the workloads of block storage [40, 77], and overwrites between two successive replays can be merged to reduce the actual number of writes; and (ii) we could perform combination and scheduling [75] in journal replay to reduce the movement of magnetic heads of backup HDDs.

The journals are co-located with their corresponding backup HDDs, because local journal replay is much simpler and more efficient than inter-machine journal replay. Empirically, Ursa bounds the maximum size of an SSD that can be used for journals to 1/10 of the SSD's capacity: we find that this is enough for most burst of writes to the corresponding backup HDDs.

On-demand journal expansion. To handle the uncommon scenario where an SSD exhausts its quota for storing journals, Ursa supports dynamically expanding the journals to the least-loaded SSDs co-located on the same machine. The expansion could be simply done by mapping overflowed backup writes to the new SSDs in the in-memory index (§3.3). Further, in the rare case that all SSDs in a machine exhaust their quotas, Ursa also supports expanding the journals to HDDs in the same machine. Theoretically, the size of HDD journals could be as large as needed. But unlike log-structured file systems [85], journal replay is necessary no matter how large the journals could be, not only for space efficiency but also for fast recovery.

The design of HDD journals is similar to that of SSD journals, except that HDD journals are replayed *only* when HDDs are idle since, because of disk seeks, HDDs inherently have no parallelism for random writes. Consequently, although large HDD journals solve the overflow problem of SSD journals, they provide lower long-term average backup performance than SSD journals. In practice, we observe that HDD journals were seldom used for replication and never used for recovery in Ursa's over two-year deployment [14]. This is because striping distributes the backup writes of a client to many machines (§3.4), and it is unlikely for many clients to simultaneously perform writes very aggressively to a specific machine for a long period of time. Further, clients that are too aggressive are rate-limited by the master before SSDs on one machine exhaust their journal quotas.

Journal bypassing. In addition to small writes, the migrated desktop/server applications also have workloads of large sequential writes, for which backup HDDs could provide comparable I/O performance with primary SSDs [40]. Ursa uses a threshold T_j to decide whether to use journals for backup writes, so as to improve the space efficiency of journals when handling large sequential writes. Only writes no larger than T_j are handled by journals, and writes exceeding the threshold bypass journals and are directly replicated to the backup HDDs. Obsolete overlapped journal appends of previous small writes (which have not yet been replayed) are invalidated by updating the index (§3.3). Larger thresholds will lead to heavier use of journals but higher overall backup performance, and vice versa. As shown in Fig. 1, most reads/writes are no larger than 64 KB, so we set the journal bypassing threshold T_j to be 64 KB.

Client-directed replication. The migrated applications also have workloads of *tiny* writes. To further reduce the latency of tiny writes without affecting the throughput, if a write is no larger than a tiny write threshold T_c, Ursa has the client simultaneously initiate the write both to the primary server and to the backup servers, instead of relying on the primary server to initiate the replication. Currently, most of our machines have two 10GbE NICs, and we wish to limit the maximum bandwidth a client consumes to half the available bandwidth of a machine. Given our specification for peak performance of 40K IOPS and a replication factor of three, the maximum T_c is $20\text{Gb}/2/40\text{K}/3 \approx 10.4\text{KB}$; so we set the tiny write threshold T_c to 8 KB.

3.3 Journal Index

The combination of journal appending and replica copying in the SSD-HDD-hybrid structure complicates the processing of data replication (for bypassing journals in large writes)

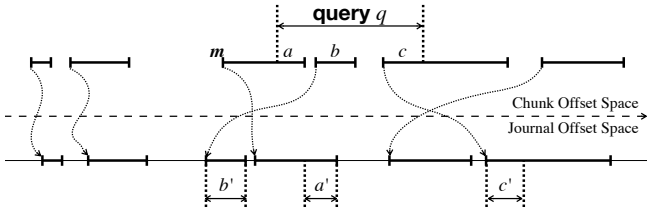


Figure 4. Index range query

and recovery (for locating the newest data). To address this problem, URSA designs a per-chunk in-memory index structure, which maps the chunk offset space to the journal offset space in order to support (i) quick invalidation of stale journal appends and (ii) fast recovery of chunk failures.

Journal index is usually implemented with log-structured merge-tree (LSMT) [31, 84, 87] in previous studies. LSMT is a general approach for maintaining mappings from keys (chunk offsets) to values (journal offsets). However, the original LSMT-based index (as used in PebblesDB [87] and TRIAD [31]) cannot efficiently support URSA's stringent performance requirements on invalidation and recovery. There are two reasons for this. First, the key space of the index is a continuous integer interval (from 0 to CHUNK_SIZE−1) and most queries and updates are for a range of keys. Second, index queries and updates are on the critical path for journal reads/writes, so URSA's I/O performance is largely determined by the index efficiency.

In order to adapt LSMT to URSA's characteristics, we make the following optimizations by which URSA outperforms state-of-the-art LSMT-based designs (like PebblesDB [87]) significantly both in range queries and in range insertions (§6.2).

Composite keys. We combine continuous keys into a single composite key `{offset,length}` if their mapped journal offsets are also continuous. A composite key indicates a range mapping from a half-closed, half-open offset interval [`offset`, `offset+length`) in the chunk offset space to the journal offset space, recorded by a KV (`{offset,length}` → `j_offset`). We define the *LESS* relationship between two composite keys x and y as follows: x is *LESS* than y iff x's `offset+length` is not greater than y's `offset`. There are no intersecting composite keys, and thus *LESS* is a total order relationship among the composite keys.

Index operations. The total order relationship allows fast range queries and updates over the composite keys. Suppose that there is a range query q (which could be viewed as a composite key) over the chunk offset space, as shown in Fig. 4. We can find the minimal composite key m that is not *LESS* than q in $O(\log n)$ time, where n is the number of keys in the per-chunk index. Then we start from m to check whether a series of composite keys intersect with q, and get the intersected regions a, b and c. For a read request, the mapped regions of the intersections, a', b' and c', are the regions to be read from the journal; and for a write request, the intersections are to be erased from the index prior to inserting the new composite key q, so as to keep the total order relationship among the composite keys.

Index storage. We pack the KVs (`{offset, length}` → `j_offset`) into 8-byte `structs` by carefully setting the bit lengths of the fields. URSA adopts a two-level structure to store the KVs. The first level is a red-black tree, which is highly efficient in insertion but inefficient in storage, because every tree node has an extra overhead of three pointers and one bit flag. The second level is a sorted array, whose insertion speed is slower than the red-black tree, but which is more efficient in both storage and query.

When adding a new KV, URSA first quickly inserts it into the red-black tree, which will be asynchronously merged into the array by a low-priority background worker thread. Since KVs in the array might be obsolete because of previous insertions to the red-black tree, queries are solved first in the red-black tree and then in the array for missed ranges, as in the sub range between a and b and the sub range between b and c in Fig. 4 (assuming the KVs in Fig. 4 are stored in the red-black tree). The first-level red-black tree acts as a small-size write cache of the second-level array. For the array, 8 GB memory can keep one billion records, corresponding to at least 16 TB of journal data assuming an average write size of 16 KB, which is far more than enough for our needs. Besides, a large write will immediately free a large number of composite keys within the range of the write, further reducing the memory footprint of the index.

3.4 Multi-Level Parallelism

URSA extends Blizzard's exploitation of disk parallelism [74] to systematically exploit three levels of parallelism, namely, (i) on-disk concurrent I/O, (ii) inter-disk concurrent chunk reads/writes, and (iii) in-network pipelining.

Disk parallelism. URSA differentiates between SSDs and HDDs in exploiting disk-level parallelism. To fully exploit SSD's parallelism, an SSD runs multiple chunk server processes, each of which has one thread issuing parallel I/O requests via `libaio` [10]. Internally, URSA converts events of `libaio` into coroutines [72] and hides `libaio` behind a synchronous interface similar to `read()/write()`. In contrast, a backup HDD runs only one single-threaded process without using `libaio`, because disk seeks inherently prevent parallelism on HDDs.

Inter-disk parallelism. At the inter-disk level, URSA exploits three kinds of parallelism, including striping [74], out-of-order execution, and out-of-order completion. First, URSA splits a virtual disk into fixed-size chunks and organizes two or more chunks into a striping group, so that large reads and writes can be parallelized to multiple chunks. The service manager is responsible for the data placement policy,

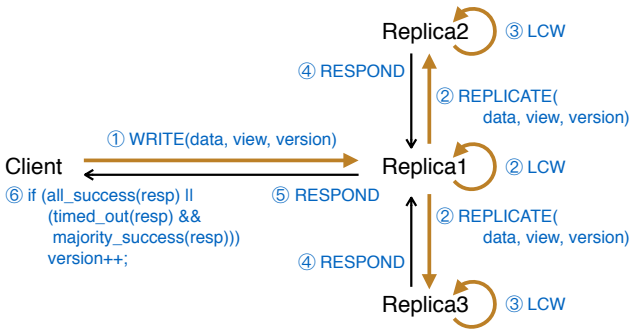


Figure 5. Writing with independently maintained chunk version numbers. LCW: local chunk write.

ensuring that all the chunks in a striping group do not reside on the same disk or machine. Second, Ursa handles I/O requests in parallel and allows out-of-order execution of requests as long as they access different data chunks. Random accesses tend to significantly benefit from such execution. Third, Ursa supports sending responses back to the client in an out-of-order manner. For instance, it is possible for a chunk server to receive I/O requests in the order of r_1, r_2, and return responses in the reverse order.

Network parallelism. Network congestion and OS scheduling can cause end-to-end network delay [99]. Ursa processes I/O requests in a pipeline for each connection, so that network delay is masked by the requests flying over the wire and waiting in the pipeline and thus has little impact on the overall IOPS and throughput.

Note that cloud virtual disks, like all other block devices including HDDs and SSDs, process I/O requests in parallel [44, 91]. It is the responsibility of guest file systems and/or applications to correctly recover from crashes while enjoying block devices' parallelism. For example, Linux Ext4 [38] and XFS [94] file systems use journaling to handle the "out-of-order" issue, and OptFS [43] introduces two primitives, `osync()` and `dsync()`, to respectively ensure (i) ordering between writes but only eventual durability, and (ii) immediate durability as well as ordering.

4 Consistency

This section introduces Ursa's replication protocol for strong consistency guarantees.

4.1 Overview

Ursa provides per-chunk linearizability [58] by totally ordering write requests across chunk servers (replicas). Specifically, per-chunk linearizability guarantees that if a write request to a chunk is committed at time t_1, then any following read request to that chunk issued at time $t_2 > t_1$ will see the committed (or newer) data.

In general, Ursa replication protocol is akin to a Replicated State Machine (RSM) such as Paxos [62] or Raft [81]. For each data chunk, a version number is maintained across chunk servers and is updated after a write request succeeds. Besides, chunk servers and the master also maintain a persistent view number [80], which is updated when a new chunk server is allocated during failure recovery.

Upon opening a virtual disk, the client gets the view number and the locations of all chunks of that virtual disk from the master, and then asks all replicas of each chunk for their version numbers. Once the version number is confirmed, the client chooses one of the replicas as the primary (preferably the one on SSD), and sends read and write requests to the primary, as shown in Fig. 5. Read requests in Ursa are served normally by primary replicas; write requests are propagated to and executed by all replicas. If a replica failure is detected by the client, the client notifies the master to allocate a new replica, update the chunk locations, and increase the view number.

Although in general Ursa's replication protocol follows the design principles of classical methods [54, 74], it differs from them mainly in two aspects. First, Ursa ensures that at most one client can access a virtual disk at any time; Second, Ursa adopts a hybrid fault model to treat replica and network faults separately. We briefly discuss these two aspects and their implications below.

Single-client scenario. Ursa leverages lease [55] and lock [35] protocols to ensure that at most one client can hold the lease of a virtual disk at any moment for reading/writing data chunks in the virtual disk. This single client scenario simplifies how Ursa guarantees strong consistency. It also does not impose undesirable constraints on virtual disk services, because a virtual disk is commonly mounted by a single VM instance [6, 37] where desktop/server applications use traditional filesystem APIs for data access.

The client periodically renews the lease, usually every tens of seconds. For safety, Ursa enforces reasonable timing constraints to ensure that leases of a virtual disk are not interleaved. The benefit is that the client is aware of the most recent state (i.e., the highest version number) and can choose which replicas will serve its requests. Especially, a read request can be served by any replica (preferably by the primary replica stored on SSD), as long as the replica's data has a matching version number. If the primary is temporarily unavailable, to preserve high availability the client can choose another replica. When the failed replica recovers, the client can switch back. In contrast, existing replication protocols that provide service to multiple clients either rely on a primary lease [36, 54], or communicate with a majority of replicas to confirm the most recent state [33, 95].

When multiple VMs mount the same virtual disk (which is uncommon in Ursa), we let a single client, which may reside on any machine, serve all the VMs and use a cluster file

system (like OCFS [16]) deployed on the VMs to coordinate concurrent I/O requests.

Hybrid fault model. Unlike both synchronous replication systems [54] (which replicate operations to $f + 1$ replicas to tolerate f crash faults), and asynchronous replication systems [95] (which replicate operations to $2f + 1$ replicas to tolerate f crash faults), Ursa considers replica and network faults separately, in light of recent advances in fault models such as VFT [86] and XFT [70]. In Ursa, the client tries to replicate a write request to all replicas, but also waits for a timeout and commits the request if a majority of replicas replies (step 6 in Fig. 5). Meanwhile, the client also notifies the master to fix the problem by, e.g., allocating a new replica to replace the crashed one. We will introduce failure recovery in §4.2.2.

Compared to synchronous replication systems that have to wait for failure recovery before proceeding, Ursa keeps the service available upon some replica crashes. Compared to asynchronous replication systems where $f + 1$ permanent faults may lead to data loss even if the other f replicas are functioning correctly, Ursa still guarantees data durability if at least one of the f replicas has executed the most recent write request. More precisely, a data chunk is durable in Ursa as long as either (i) fewer than a majority of replicas fail (as in asynchronous replication systems); or (ii) the total number of replica faults and problematic connections between the primary (the client for client-directed replication case) and backups (chunk servers) is smaller than the number of replicas. Note that a crashed client does not affect data durability. The client has either committed the request before crashing, which indicates the request is already executed by sufficient replicas; or the client has not yet committed the request. In the second case, the (rebooted) new client will fix the inconsistency problem during initialization.

4.2 Ursa Replication Protocol

In this section we detail Ursa's replication protocol, which contains a normal case sub-protocol where no failure occurs, and failure recovery sub-protocol that allocates a new replica to replace the crashed one.

4.2.1 Normal Case

Initialization. Upon opening a virtual disk, the client gets all the locations and view numbers of the chunks from the master, and then asynchronously asks all replicas of each chunk for their version numbers and view numbers. If all replicas of a chunk have the same view number *view* matching the one in the master and all replicas have the same version number *version*, then one of them will be chosen as the primary (preferably the one on an SSD); otherwise, the client first informs the *cluster director* to fix the inconsistency problem (see §4.2.2) and then retries later. Note that the primary in Ursa plays a temporary role for propagating write requests to other replicas and replying read/write requests to the client. The client can choose another replica (in the same view) as the primary without further re-configuration due to the single-client scenario discussed in §4.1.

Read and write. Once the version number *version* and the primary are confirmed, the client sends read and write requests to the primary. Read requests in Ursa are served preferably by primary replicas. For a write request, the client attaches the view number and version number it knows of along with the request. As shown in Fig. 5, upon receiving a write request, the primary (i) checks locally if its view number and version number match the request's numbers; if so (ii) performs the write locally, replicates the write to the backup replicas, increases its version number, and finally (iii) replies to the client. When receiving replication requests, backup replicas perform similar operations and reply to the primary.

However, if the client's view number does not match with the request's number, the primary declines the request and replies to the client. It is the client's responsibility to obtain the current view number from the master. If the client's version number is larger than the one maintained by the primary, the primary tries to update its state by incremental repair — transferring modified data pieces from other replicas. Finally, if the client's version number is one short of the primary's version number, the primary skips the local write as it has already done it, but still forwards the write request to the backups and replies to the client accordingly. The client's version number will not lag behind by more than one due to the single-client scenario discussed in §4.1.

Ursa leverages its hybrid fault model to improve availability when some replica crashes or is temporarily unavailable; at the same time, it strives to sustain data durability. Specifically, write requests are considered committed when all replicas succeed in the normal case. However, if the primary cannot collect responses for write requests from all backups in a timely manner, it compromises durability (usually for a short period of time) and waits for responses from at least half of the backups (which form a majority together with the primary itself), and also waits for a timeout. At the same time, the client notifies the master to fix the inconsistency problem, or finally to allocate a new replica by following the recovery procedure for chunk failures.

Switching of primary replica. If a client sends read/write requests to a failed primary replica, it will resort to one of the backup replicas that holds the newest data as a temporary primary, so as to achieve high availability. Writes are logged to the journal for the temporary primary, and its performance will not be affected when using SSD journals (but the client will limit its rate). Reads are performed after querying the journal index (§3.2), and might experience a brief degradation if the requested data is on HDDs. The cluster director will

in parallel create a new replica on another SSD. Finally, the client will take the new replica on SSD as primary.

Incremental repair. To support incremental repair, each replica (either on SSD or on HDD) maintains another journal in memory called *journal lite*, which caches position, offset, and version number of recent write requests. When a primary/backup recovers from an event of temporary unavailability such as network partition, the recovered replica tries to update its data by sending its current version number to other replicas. Upon receipt, these replicas (i) query their journal lite with the received version number to locate the modified data; (ii) construct a repair message based on indices provided journal lite; and (iii) transfer the message with the new version number to the recovered replica. If the required data indices cannot be obtained by querying journal lite (e.g., because of garbage collection), the whole data chunk is instead transferred.

Client-directed replication. As discussed in §3.2, the processing of writes no larger than 8KB is slightly different: clients directly propagate writes to all replicas instead of only to the primary (but the version numbers are maintained in a similar way). This optimization reduces the latency of small writes at the expense of slightly higher overhead on the client side.

4.2.2 Failure Recovery (View Change)

When inconsistency problems happen, the master asks replicas with the highest version number to carry out incremental repair. If, instead a chunk failure occurs, the master eventually allocates a new replica to replace the failed one. The view number i of the chunk will be increased to $i+1$, and the client will be aware of the new view on the next read/write.

More specifically, the master tries to collect version numbers from a majority of replicas, then selects the highest version number $version_H$ among them as the most recent state. This is in order to ensure that if a write request is executed at $version \leq version_H$ by a majority of replicas in view i, then this write request must be executed in view $i+1$ as well. Then, the master asks the replica with version number $version_H$ to transfer its data to the new allocated replica. If necessary, incremental repair is also executed in order to update existing replicas.

Finally, all replicas update their view number to $i+1$ as long as they maintain the same version number $version_H$ and store the same data. If the master and a replica fail simultaneously, the master is recovered first, and then the chunk is recovered as described above.

The key idea behind failure recovery is that the master tries to select the highest version number from a quorum, i.e., a majority of replicas, as in asynchronous replication systems. The downside of this approach, however, is that if a majority of replica crash data can be permanently lost. In contrast, synchronous replication protocols can make progress with only one survivor. To make full use of the "write-to-all" property of the normal case sub-protocol, Ursa should be able to proceed even if only a minority survive (at least one), without compromising consistency or durability. To this end, the key idea is to detect permanently crashed replicas. If crashed replicas constitute a majority, the master confirms the most recent state with the help of other available replicas (though a minority). One approach to identifying failed replicas is for the master to notify the system administrator, who will manually locate them. A second option is to set a very conservative timeout value, after which the master declares unreachable replicas as having crashed. However, since the second option may still undermine consistency and durability in asynchronous settings, Ursa currently adopts the first approach. More automatic and accurate crash failure detection mechanisms (like FALCON [64], which inspects the OS process table or VMM) will be the subject of future work.

5 Discussion

5.1 Richly-Featured Clients

Different from storage clients of other systems including key-value [101], object [32], and blob [76] stores, Ursa places rich features (like tiny write replication, striping, snapshot, and client-side caching) in its clients rather than in its chunk servers, so as to improve I/O performance by conducting client-directed operations. The extra maintenance cost for Ursa clients is moderate and acceptable, mainly because the Ursa block store and the QEMU virtual machine monitor are tightly coupled, making their cooperation relatively easy. The rich features of Ursa clients are designed as pluggable modules, following the `decorator` pattern [90], where all the modules implement a common abstract interface of `read()`/`write()`. The clients may evolve over time without affecting system availability by adopting an online upgrade mechanism (§5.2).

5.2 Online Component Upgrade

The various components of Ursa evolve over time, and it is important to keep high availability during upgrades. Ursa prefers online component upgrade to conventional VM migration, because migration takes at least a few seconds for an unloaded VMs and even longer for loaded VMs, which may, e.g., crash the VMs connected to the upgrading clients or degrade the performance of services running on the VMs.

Client. Since the VMM (e.g., QEMU in Fig. 3) will not re-establish the connection when the socket to the client is disconnected, it is challenging to upgrade client components without affecting the guest OS. An intuitive solution is to place as much as possible code into a shared library that can be upgraded using dynamic reloading. However, this solution has many limitations. For instance, neither the main program nor the interface of the shared library can be upgraded, and

a large proportion of existing static libraries may not fit into the shared library model. To address this problem, we realize upgrading at the granularity of processes instead of libraries. We split the client into two processes (core and shell), both of which share the connection to the VMM. When upgrading, the core process (i) stops receiving new I/O requests from the VMM and completes pending requests; (ii) saves its status into a temporary file; and (iii) exits with a specific code. The shell process receives the exit code and starts the new version of the core, which reads its status from the temporary file and resumes the service.

Master. Upgrading the master component is relatively simple, since it is not on the regular I/O paths. We gracefully shut down the old master processes and immediately start the new ones. During the upgrade, operations like virtual disk creation and space allocation may fail; if so, then can be retried after the upgrade completes.

Chunk server. Upgrading the server component is slightly more complex, since server restarts may confuse the failure handling routines if real server failures occur simultaneously with the upgrade. Therefore, we design a graceful hot-upgrade mechanism. We send a specific signal to the chunk server, which (i) closes service ports and stops receiving new I/O requests; (ii) waits for all in-flight requests to complete; (iii) starts the new version of chunk server in a new process; and (iv) checks whether the new chunk server process works correctly. If the hot upgrade succeeds, the old chunk server closes all connections and exits, and the clients re-connect to the new server. If the hot upgrade fails (e.g., because of wrong configurations, missing libraries, etc.), the old chunk server kills the new process, re-opens the service port, and continues its service.

Incremental upgrade. An Ursa cluster is composed of a large number of service processes. We deliberately upgrade one process at a time, and confirm that the current upgrade behaves as expected before performing the next one. It may take days to upgrade an entire cluster. Therefore, we have developed all components with backward compatibility to allow new instances to run together with old ones. Ursa has upgraded four major versions of the replication protocol during its over two-year deployment. In each upgrade, we extended the protocol by adding new operations and keeping all old operations unchanged, so as to guarantee backward compatibility.

5.3 Exploiting Disk Parallelism

Ursa empirically runs two processes for a SATA SSD and four processes for a PCIe SSD in our production clusters. In contrast, a backup HDD runs only one single-threaded process both for journal replay (of small writes) and for replication (of large writes). Our evaluation (omitted because of lack of space) shows that a single-threaded process adopting the elevator algorithm [8] is enough to saturate an HDD even without using `libaio`. Multiple backup server processes or threads for one HDD interfere with each other, confuse the elevator algorithm, and thus degrade the backup performance.

	HDD	SSD	RAM	Power	CPU	Other
%	69.1	4.0	6.2	3.0	2.6	15.1

Table 1. Failure ratios in Ursa deployment.

5.4 Hardware Reliability

According to our statistics (Table 1), HDDs contribute nearly 70% of the total failures, an order of magnitude higher than SSDs. Wear leveling [9] ensures the frequently-written journals not to affect the lifetime of SSDs.

Although SSDs have lower average failure rate than HDDs, a potential risk is that SSD firmware bugs [25] may simultaneously fail many SSDs from the same batches and vendors, because the same bug trigger conditions could happen on multiple SSDs that have similar usage in a block storage cluster. For example, Tencent QCloud [20] recently lost valuable metadata of one of its tenants due to some firmware bug of SSDs, and it has been required to pay $1.6 million in compensation [21].

To avoid correlated failures [53], we increase diversity by purchasing SSDs of different batches/vendors, and store primary chunks and backup journals on SSDs of different batches/vendors. Unlike SSD-only storage, Ursa hybrid storage can rely on the backup HDDs to avoid data loss even if a firmware bug affects many SSDs.

5.5 Limitations

The hybrid structure is optimized to efficiently handle the workload pattern of Ursa's virtual disk service. While it satisfies our needs well, there are a few limitations.

First, during the failure recovery of a primary replica, the client might get exposed to the SSD-HDD performance gap since the replicas are heterogeneous. This requires us to recover quickly to reduce the time window for such degradation.

Second, Ursa's SSD journals do not support continuous peak loads of random small writes for a long period of time (a rare event for our virtual disk service), since the total journal size on all SSDs of a storage machine is bounded. We work around this limitation using secondary HDD journals and rate-limited clients.

Third, in hybrid storage all replicas on SSDs are primaries while in traditional SSD-only storage some replicas on SSDs are backups, so the recovery of SSD failures is more urgent for Ursa than for SSD-only storage.

Fourth, Ursa requires its total SSD capacity to be at least as large as its data volume, while traditional cache-based block storage is adaptive to any SSD capacity.

6 Evaluation

This section presents evaluation results from micro benchmarks, trace-driven experiments, and real-world use of Ursa. We build two testbeds. The first small testbed has four machines for most micro-benchmarks and trace-driven experiments, and the second large testbed has 45 machines (from a production cluster) for scalability tests. Each machine has dual 8-core Xeon E5-2650 2.30GHz CPU, 128GB RAM, eight 7200RPM 1TB HDDs, and two Intel 750 PCIe 400GB SSDs, connected to a 10GbE network. The OS is CentOS 7.2. We test three replication modes, namely, SSD-HDD-hybrid, SSD-only, and HDD-only. The master and lock module (§4.1) are co-located on one machine.

We compare Ursa with Ceph [98] and Sheepdog [17]. Measurements are conducted on VMs mounting the virtual disks. If not specified, each result is the mean of 20 runs. In most experiments the variances to the mean are relatively small, so (if not specified) we omit the error bars for clarity.

We seek to answer the following questions: Is the performance of Ursa's hybrid mode in IOPS, latency, and throughput comparable to the expensive SSD-only mode of Ursa, Ceph and Sheepdog (§6.1)? What is the impact of Ursa's various designs, including journal index, journal expansion, and failure recovery (§6.2)? How scalable is Ursa (§6.3)? What is the performance of Ursa in the trace-driven experiments compared to Ceph and Sheepdog (§6.4)? And how does Ursa perform in real-world use compared to Amazon AWS and Tencent QCloud (§ 6.5)?

6.1 I/O Performance

We use three machines to run chunk servers and one to run the client. We evaluate the performance of Ursa's hybrid storage in random IOPS, random I/O latency, and throughput, and compare it to the expensive SSD-only mode of Ursa, Ceph and Sheepdog. The I/O size is 4KB in the IOPS and latency tests, which is within Ursa's client-directed replication threshold (§3.2), so in Ursa the backup writes are directly performed by the client. In comparison, Ceph is designed to rely on the primary to perform backup writes while Sheepdog always has the client issue all primary/backup writes in parallel. The I/O size is 1MB in the test of throughput. The queue depth (qd) is 16 (the maximum depth supported by QEMU's NBD driver) for IOPS tests, and 1 for latency and throughput tests. For Ursa's SSD-HDD-hybrid, the SSD journal has a maximum size of 1/10 the SSD capacity. SSD-only Ursa does not use journals. OS page cache is disabled.

The result is shown in Fig. 6, where `Ursa-Hybrid` (SSD-HDD-hybrid) has I/O performance similar to `Ursa-SSD` (SSD-only) and outperforms `Sheepdog` (SSD-only) and `Ceph` (SSD-only) in all tests except for write throughput. The block size (BS) in the IOPS and latency tests is set to 4KB. This is smaller than the 8KB threshold for client-direct replication (§3.2), and thus the backup writes are directly handled by the client (instead of the primary server). The I/O read latencies of all systems are similar because read requests are handled by primary SSDs. `Ursa-Hybrid` and `Ursa-SSD` perform better in read IOPS than `Sheepdog` and `Ceph`, partially because they exploit multi-level parallelism (§3.4). The write performance of `Ursa-Hybrid` is similar to that of `Ursa-SSD` in IOPS and latency: the journals of `Ursa-Hybrid` transform random backup HDD writes into sequential SSD writes whose performance is comparable to that of random backup SSD writes in `Ursa-SSD`, even when the journals are simultaneously replayed.

In Fig. 6, `Ursa-Hybrid` has the worst write throughput. This is because we *deliberately* set the block size to be 1MB, which is larger than the 64KB threshold for journal bypassing (§3.2). This setting, in which all backup writes go directly to HDDs (instead of using journals), demonstrates the worst-case throughput of Ursa.

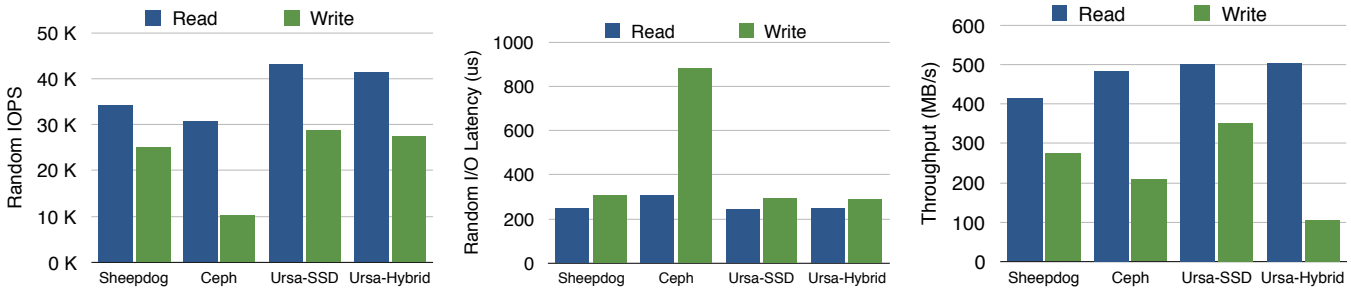
Besides I/O performance, we also measure the CPU efficiency (defined as the I/O performance divided by the CPU utilization) while focusing on the critical path between the client and servers. We set the size of all tested data to be small enough (4MB) that it could fit within a single data chunk for all tested systems, and all reads and writes could actually access the cache of the chunk. Fig. 7 shows the read/write IOPS efficiency for the client and server in Ursa, Sheepdog and Ceph. Ursa outperforms Sheepdog and Ceph by orders of magnitude in the efficiency tests, owing not only to Ursa's exploitation of multi-level parallelism (§3.4) but also to its highly optimized implementation. Ceph lacks client-side results because data cannot be easily extracted from its in-QEMU client.

We also evaluate the sequential IOPS of these systems with different queue depths. For sequential reads (Fig. 8) and writes (Fig. 9), Ursa outperforms others mainly because it exploits network parallelism and pipelining. In all tested systems, the sequential write IOPS is lower than the sequential read IOPS, because writes frequently cause lock contentions which complicate parallelism exploitation.

6.2 Impact of Various Design Components

This section evaluates Ursa's various design components including journal index, journal expansion, and failure recovery.

Ursa has extremely high requirements on the performance of range queries over its journal index (§3.3), which cannot be satisfied by existing LSMT-based techniques. We evaluate the performance of Ursa's range-query-optimized index, and compare it with state-of-the-art PebblesDB [87], which supports range query by locating the first key (using `seek()`) and traversing all keys in the range (using `next()`). We insert 700,000 *random ranges* respectively to Ursa and PebblesDB, each with an integer start randomly chosen from $[0, 2^{20})$ and a random length ranging from $[1, 2^6]$. For Ursa, we have 100,000 ranges stored in the red-black-tree and



(a) Rand. IOPS (BS=4KB, QD=16) **(b)** Rand. I/O Latency (BS=4KB, QD=1) **(c)** Seq. TP (BS=1MB, QD=1)

Figure 6. Performance of URSA (SSD-HDD-hybrid and SSD-only), Sheepdog and Ceph (SSD-only). Cache is disabled.

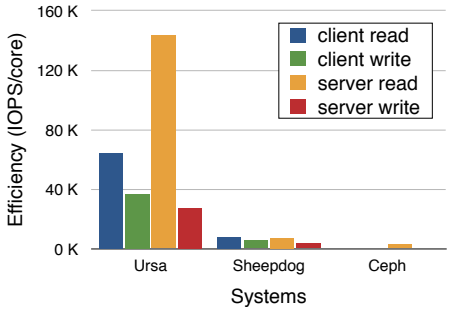


Figure 7. IOPS efficiency.

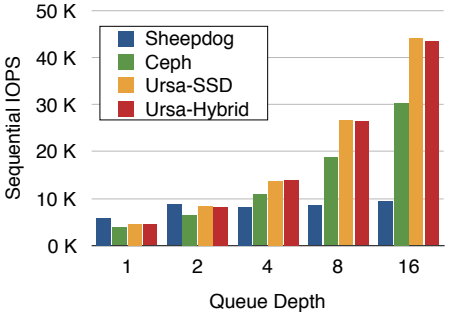


Figure 8. Sequential read IOPS.

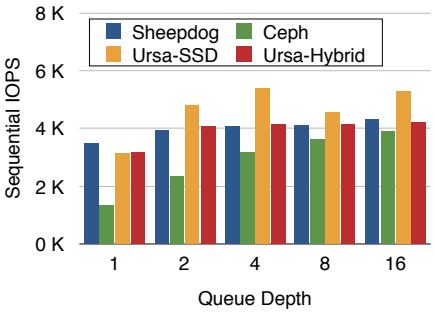


Figure 9. Sequential write IOPS.

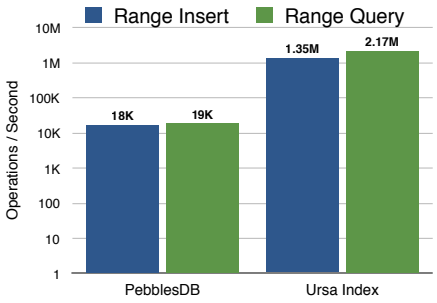


Figure 10. Ursa vs. PebblesDB.

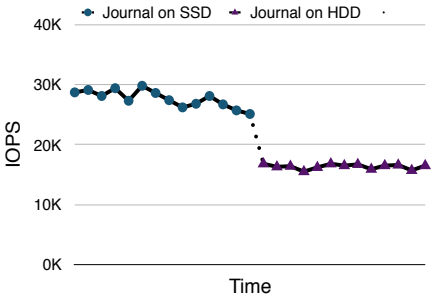


Figure 11. Journal expansion.

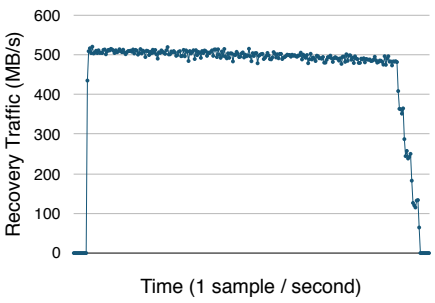


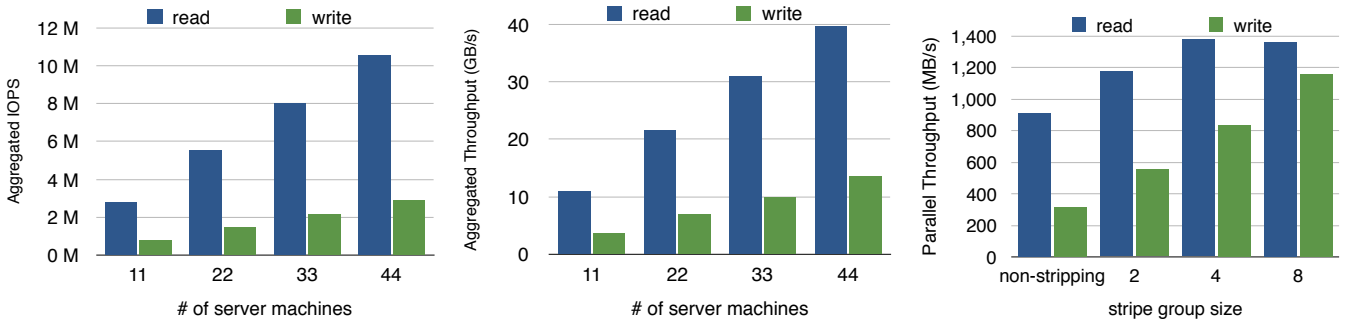
Figure 12. Failure recovery.

600,000 in the array. We perform 100,000 random range queries respectively to URSA's index and PebblesDB.

The result is shown in Fig. 10: URSA outperforms PebblesDB by two orders of magnitude in both range insertions and range queries. This is mainly because URSA's in-memory journal index is designed specifically for range mappings (`{offset, length}` → `j_offset`), while PebblesDB's FLSM (Fragmented Log-Structured Merge Trees) structure is only for normal key-value mappings (`offset` → `j_offset`).

In the micro-benchmark tests (§6.1), the secondary HDD journals have not been used since no backup workloads are long enough to make all SSDs exhaust their quotas. To verify the design of journal expansion (§3.2) for rarely-long workloads of random small writes, we emulate an overflow of an SSD journal and evaluate the IOPS before and after the overflow.

The result is shown in Fig. 11, where each data point represents the mean IOPS of 10 successive seconds. Clearly, the backup load is successfully redirected from the SSD journal to the HDD journal, demonstrating the effectiveness of URSA's journal expansion mechanism. Performance degradation is not significantly high, because HDDs perform much better in sequential journal appends than in random small writes. As discussed in §3.2, HDD journals are seldom used for replication in URSA's practical deployment, mainly because URSA's SSD journals already hold most of the random small write workloads.



(a) IOPS (BS=4KB, QD=16) **(b)** Seq. Throughput (BS=1MB, QD=1) **(c)** Striping (BS=1MB).

Figure 13. Scalability test.

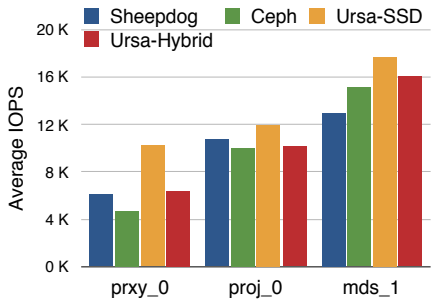


Figure 14. Trace-driven comparison.

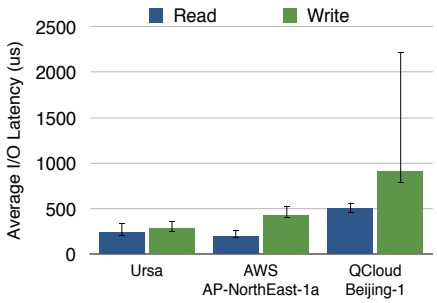


Figure 15. Public cloud latency.

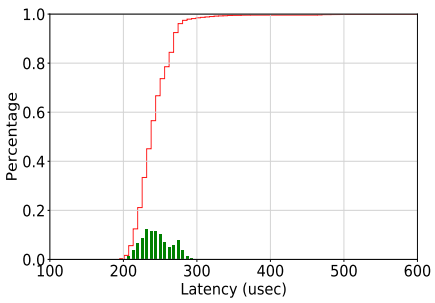


Figure 16. Latency PDF & CDF.

To evaluate the recovery of Ursa hybrid storage where the backup data might be located both on backup HDDs and in SSD journals, we fill a chunk server's SSD on a target machine, and then disable it to evaluate the recovery procedure. Since in the small testbed we have only three chunk server machines (each having two SSDs and eight HDDs), for simplicity Ursa recovers data to the other SSD co-located with the failed SSD on the target machine. Leveraging the LSMT-based index (§3.3), the data is recovered from both backup HDDs and SSD journals in the other two machines.

We measure the failure recovery procedure, and the result is shown in Fig. 12. The recovery speed keeps about 500 MB/sec during the recovery. The speed is bounded by the inbound network bandwidth (10Gbps). The journal on the failed SSD is asynchronously recovered.

6.3 Scalability

We measure the aggregate I/O performance of Ursa, as the number of storage machines increases from 11 to 44. Chunk servers and clients run on all machines to saturate the system. Fig. 13a and 13b show that both IOPS and throughput increase linearly with the number of machines. Ursa's I/O latency changes little (not shown because of lack of space) as the number of machines increases from 11 to 44.

We evaluate the effect of striping (§3.4) on throughput in the 44-machine cluster, running a client on a dedicated machine equipped with two 10GbE NICs. The queue depth is set to 16 instead of 1 for exploiting more parallelism. Fig. 13c shows that both read and write throughput increase as the stripe group size increases. The throughput of reads is relatively stable (900MB/sec ~ 1.4GB/sec). The throughput of writes is lower than that of reads, because multiple replicas need to be written and journals are bypassed in backup operations (block size = 1MB).

6.4 Trace Driven Evaluation

We evaluate the IOPS of `Ursa-Hybrid` using MSR traces [1] that record block-level reads and writes of traditional desktop and server applications gathered below the filesystem cache, and compare it with `Ceph`, `Sheepdog` and `Ursa-SSD` (all in SSD-only mode). We write a custom benchmark tool that ignores the timestamps in the traces and issues I/O requests with qd16. We run the 36 MSR traces, which have various combinations of read/write ratios. Fig. 14 shows three representative traces (`prxy_0`, `proj_0`, and `mds_1`) demonstrating different I/O patterns. In all experiments `Ursa-SSD` is the best performer, and `Ursa-Hybrid` is comparable to or better than `Ceph` and `Sheepdog` (in their SSD-only mode).

6.5 Production Evaluation

Ursa has been deployed for over two years [14]. We compare Ursa block storage service with other public SSD-only block

storage services including Amazon AWS [3] and Tencent QCloud [20]. We do not measure IOPS or throughput, because in commercial clouds both metrics are limited by the SLA. For each block storage service, we use a VM (with 2-vCPU, 4GB-RAM) to continuously measure the I/O latency every 2 seconds for two days.

Fig. 15 depicts the mean, 1st percentile, and 99th percentile I/O latency of each block storage service, showing Ursa's SSD-HDD-hybrid storage has I/O latency comparable with that of other commercial SSD-only storage services. Note that it is not practical to conduct a completely fair comparison between Ursa and AWS/QCloud, because the block storage services may adopt different architecture/hardware and all tests are affected by background workloads due to, e.g., overselling. Fig. 16 shows the probability density function (PDF) and cumulative distribution function (CDF) of Ursa's I/O latency, including both read and write operations.

7 Related Work

In this section, we briefly discuss related work on block storage, EC storage, hybrid storage, file systems, object storage, and consistency issues.

Block storage. Distributed block storage [59, 63, 73, 97] provides a block interface to remote clients using protocols like iSCSI [26] and AoE [39]. For example, Petal [63] uses redundant network storage to provide virtual disks. Salus [96] leverages HDFS [11] and HBase [56] to provide virtual disk service with ordered-commit semantics using a two-phase commit protocol. pNFS (parallel NFS) [59] exports a block-/object/file interface to local cloud storage. Blizzard [74] is built on FDS [79] and exposes disk parallelism to virtual disks with crash consistency guarantees. Strata [46] designs a block store using only SSDs. Ursa shares a few similar design points with these systems (e.g., striping [74] and replication [29]). Compared to them, Ursa implements a production block store with the SSD-HDD-hybrid structure and exploits multi-level parallelism.

EC storage. Erasure coding (EC) is more efficient than replication [88]. The key challenge for EC is to improve write performance. For example, Sheepdog [17] does not directly support partial write, instead it emulates partial write by reading unmodified data, re-encoding them together, and writing them as a full write. Parity logging approaches [40, 60, 93] log deltas of parity chunks, which cannot efficiently support partial writes since every write requires a read at the data chunk. PariX [67] performs speculative partial writes to alleviate the inability of EC [40] to support random small writes. Compared to replication, EC optimizes for capacity at the expense of I/O performance. Since (HDD) capacity is the least valuable resource in a hybrid architecture, we prefer Ursa to PariX for providing our block storage service.

Hybrid storage. Previous SSD-HDD hybrid designs mainly focus on using SSD as a cache layer. For example, Nitro [66] designs a capacity-optimized SSD cache for primary storage. Solid State Hybrid Drives (SSHD [51]) integrate an SSD inside a traditional HDD and realize SSD cache in a way similar to Nitro. Griffin [92] designs a hybrid storage device that uses HDDs as a write cache for SSDs to extend SSD lifetimes. Compared to these studies, Ursa uses SSDs directly for primary storage instead of as a cache, so as to boost I/O performance.

File systems. Distributed file systems spread the data of a file across many storage servers [45, 65]. For instance, GFS [54] is a large-scale fault-tolerant file system for data-intensive cloud applications. Zebra [57] uses striping on RAID [42] and logs [89] for high disk parallelism. BPFS [45] focuses on persistent memory hardware and uses epoch barrier to provide an in-memory file system with ordering guarantees. OptFS [43] improves the journaling file system [85] by decoupling durability from ordering. Compared to them, Ursa targets different scenarios and workloads and consequently makes different design decisions. For instance, GFS has random/sequential reads and append writes from cloud applications, while Ursa has (dominant) random and (non-dominant) sequential I/O workloads, but no appends from its mounted VMs running desktop/server applications.

Object storage. Object storage stores data as objects with a unique ID in a flat address space [4, 32, 76]. For example, Haystack [32] stores billions of photos, and Ceph [98] supports not only objects but also traditional files. Object storage usually provides append-only and replacement operations. In contrast, Ursa targets VMs running unmodified applications that often randomly modify some parts of a virtual disk.

Consistency. Some storage systems like HBase [12] and BigTable [41] focus on strong consistency [5, 12, 41], but at the expense of reduced availability or durability [96]. To address this problem, some systems relax the consistency guarantees for better availability and durability [34, 50, 61], while others focus on providing end-to-end protection against faulty nodes [52, 68, 71]. Ursa exploits the usage scenario of block storage (where one virtual disk is commonly mounted by one single VM) to provide strong consistency guarantees without sacrificing performance in most cases.

8 Conclusion

This paper describes Ursa, a block storage system which designs the SSD-HDD-hybrid structure by proposing an efficient journal mechanism to bridge the performance gap between SSDs and HDDs. Evaluation shows that Ursa outperforms state-of-the-art open-source and commercial block storage systems. Our future work includes (i) utilization of RDMA/DPDK/SPDK for achieving extremely low latency and (ii) improvement of the IOPS performance for (non-dominant) sequential writes. The key components of Ursa are available at [2].

Acknowledgments

We would like to thank Lorenzo Alvisi, our shepherd, and the anonymous reviewers for their insightful comments. We thank Wei Bai and Peng Chen for the discussion at MSRA. We also thank Xiaohui Liu for carefully drawing Fig. 16. Yiming Zhang is the corresponding author. This research is supported by the National Key Research and Development Program of China (2016YFB1000101), and the National Natural Science Foundation of China (61772541 and 61872376).

Appendix A Proofs of Linearizability

We sketch the proof that Ursa guarantees per-chunk linearizability. The proof includes two parts: (1) in normal case where no failure recovery (view-change) occurs, Ursa replication protocol ensures linearizability; (2) during view change from i to $i+1$, Ursa replication protocol ensures that every write request committed at version number n in view i is committed at version number n in view $i+1$.

Note that we assume the clocks between the Master and clients are loosely synchronized so as to ensure the single-client property for each virtual disk (and chunk), i.e., timing assumption holds. Interested readers may refer to, e.g., [36, 54] for the detail of how primary lease works. The single-client property or client lease is ensured in a similar way.

We say a write request is committed if the client who issued the request has confirmed its execution, i.e., the client has executed step 6 in Fig 5.

A.1 Normal Case

In normal case where no failure occurs, we prove that per-chunk linearizability is guaranteed. Specifically,

Lemma .1. *In normal case, if a write request req_w is committed and a read request req_r is issued thereafter, then req_r will see the data modified by req_w.*

At first we argue that in normal case, the request executed at each version number is unique:

Corollary .2. *In normal case, if replica R_i has executed req_w at version number n, replica R'_i has executed req'_w at version number n, then $req_w = req'_w$.*

Upon opening a virtual disk, the client confirms all metadata by the help of the master and chunk servers before proceeding. Hence, the client and all chunk servers start with a consistent state, i.e., with matching view number and version number. If the old client is detached and a new client is connected, the new client will be following the same procedure.

Suppose R_i has executed req_w at n in view i, then a client c should have issued req_w in view i, and c will not issue another request at n. So if req'_w is also issued by c, then $req_w = req'_w$. Without loss of generality, suppose a new client c' is connected after c is detached. At initial time, either c' should have obtained a matching version number n' and $n' \geq n$, or the replicas are inconsistent in version number and the master will fix the problem by incremental repair. In either case, c' will start with a version number $n' \geq n$ since R_i has executed req_w at n. So if R'_i has executed req'_w at version number n, $req_w = req'_w$.

Corollary .2 is true.

Upon a write request req_w is committed at version number n, req_w must be executed by a majority of replicas, and these replicas and the client has updated their version number to $n+1$. Based on Corollary .2, no other request can be executed by any replica at n.

Upon client c issues a read request req_r after req_w is committed, c must be aware of the highest version number $n' \geq n$. Either c is the one who committed req_w, or c is a new client and started with a matching version number $\geq n$. In either case, the client can accept a reply of req_r only if the version number it maintains n' is matching with the one replied, no matter which replica served the read request. Then based on Corollary .2, req_r will see the data modified by req_w.

A.2 Failure Recovery

In failure recovery where a new replica is allocated, we focus on the proof that a data modified by a committed write request is durable.

Lemma .3. *If req_w is committed at version number n in view i, after failure recovery (view change) to view $i+1$, req_w is executed at n by every replica in view $i+1$.*

Since req_w is committed in view i, req_w is executed by a majority of replicas, say by set m_i. During view change to $i+1$, the master tries to collect version numbers from a majority of replicas in view i, say from set m'_i. (1) If it succeeds, then there exists a chunk server $s_i \in m_i \cap m'_i$ that has executed req_w at version number n, so s_i provides version number $n' \geq n$ and transfers its data to new replica. (2) Otherwise, we assume failure recovery can still proceed with a majority of replicas in view i failed, as discussed in Sec. 4.2.2. Then, the master collects version numbers from the rest normal replicas, say m^r_i. Since the total number of replica faults and problematic connections between the primary (the client for client-directed replication case) and backups (chunk servers) is less than the number of all replicas, there exists replica $s_r \in m_i$ that does not crash or be partitioned during the propagation of req_w in view i. Hence, s_r has executed req_w in view i. s_r also must be in m^r_i as it is normal during failure recovery. Hence, s_r provides version number $n' \geq n$ and transfers its data to the new replica. In either case, Lemma .3 is true.

With Lemma .1 and Lemma .3, if req_w is committed before req_r is issued, then req_r will see the data modified by req_w.

References

[1] http://iotta.snia.org/traces/388.
[2] http://nicexlab.com/ursa/.
[3] https://aws.amazon.com/ebs/.
[4] https://aws.amazon.com/s3/.
[5] https://azure.microsoft.com/en-us/services/storage/.
[6] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-attaching-volume.html.
[7] https://en.wikipedia.org/wiki/Advanced_Format.
[8] https://en.wikipedia.org/wiki/Elevator_algorithm.
[9] https://en.wikipedia.org/wiki/Wear_leveling.
[10] https://git.fedorahosted.org/cgit/libaio.git.
[11] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[12] https://hbase.apache.org/.
[13] https://intl.aliyun.com/.
[14] https://mos.meituan.com/.
[15] https://nbd.sourceforge.io/.
[16] https://oss.oracle.com/projects/ocfs/.
[17] https://sheepdog.github.io/sheepdog/.
[18] https://wiki.openstack.org/cinder.
[19] https://www.microsoft.com/en-us/cloud-platform/desktop-virtualization.
[20] https://www.qcloud.com/.
[21] https://www.technode.com/2018/08/06/tencent-cloud-user-claims-1-6-million-compensation-for-data-loss/.
[22] https://www.zadarastorage.com/blog/tech-corner/hdd-versus-ssd-head-head-comparison/.
[23] http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919/.
[24] http://www.storagereview.com/ssd_vs_hdd.
[25] www.storagereview.com/how_upgrade_ssd_firmware.
[26] Stephen Aiken, Dirk Grunwald, Andrew R Pleszkun, and Jesse Willeke. A performance analysis of the iscsi protocol. In *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 123–134. IEEE, 2003.
[27] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large cams for high performance data-intensive networked systems. In *NSDI*, pages 433–448. USENIX Association, 2010.
[28] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 1–14. ACM, 2009.
[29] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.
[30] Mary G Baker, John H Hartman, Michael D Kupfer, Ken W Shirriff, and John K Ousterhout. Measurements of a distributed file system. In *Proc. Symposium on Operating Systems Principles*, pages 198–212, 1991.
[31] Oana Maria Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. Triad: creating synergies between memory, disk and log in log structured key-value stores. In *Annual Technical Conference*. USENIX Association, 2017.
[32] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 47–60, Berkeley, CA, USA, 2010. USENIX Association.
[33] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 141–154, Berkeley, CA, USA, 2011. USENIX Association.
[34] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.
[35] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.
[36] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.
[37] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.
[38] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *2007 Linux Storage & Filesystem Workshop, LSF 2007, San Jose, CA, USA, February 12-13, 2007*, 2007.
[39] Ed L Cashin. Kernel korner: Ata over ethernet: putting hard drives on the lan. *Linux Journal*, 2005(134):10, 2005.
[40] Jeremy CW Chan, Qian Ding, Patrick PC Lee, and Helen HW Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 163–176, 2014.
[41] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.
[42] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.
[43] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.
[44] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Consistency without ordering. In *FAST*, page 9, 2012.
[45] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.
[46] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX conference on File and Storage Technologies (FAST)*, pages 17–31, 2014.
[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[48] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36, New York, NY, USA, 2011. ACM.

[49] Biplob K. Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *PVLDB*, 3(2):1414–1425, 2010.

[50] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[51] Borislav Dordevic, Valentina Timcenko, and Slavica Bostjancic Rakas. Sshd: Modeling and performance analysis. *INFOTEH-JAHORINA*, 15(3):526–529, 2016.

[52] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. Sporc: Group collaboration using untrusted cloud resources. In *OSDI*, volume 10, pages 337–350, 2010.

[53] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, volume 10, pages 1–7, 2010.

[54] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[55] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

[56] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, 2014.

[57] John H Hartman and John K Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.

[58] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[59] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pnfs. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 18–27. IEEE, 2005.

[60] Chao Jin, Dan Feng, Hong Jiang, and Lei Tian. Raid6l: A log-assisted raid6 storage architecture with improved write performance. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE, 2011.

[61] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[62] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[63] Edward K Lee and Chandramohan A Thekkath. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices*, volume 31, pages 84–92. ACM, 1996.

[64] Joshua B Leners, Hao Wu, Wei-Lun Hung, Marcos K Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 279–294. ACM, 2011.

[65] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference*, volume 1, pages 2–5, 2008.

[66] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference*, pages 501–512, 2014.

[67] Huiba Li, Yiming Zhang, Zhiming Zhang, Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng. Parix: Speculative partial writes in erasure-coded systems. In *Annual Technical Conference*, pages 581–587. USENIX Association, 2017.

[68] Jinyuan Li, Maxwell N Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *OSDI*, volume 4, pages 9–9, 2004.

[69] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[70] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[71] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):12, 2011.

[72] Sidney R Maxwell. Experiments with a coroutine execution model for genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 413–417. IEEE, 1994.

[73] Dutch T Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J Feeley, Norman C Hutchinson, and Andrew Warfield. Parallax: virtual disks for virtual machines. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 41–54. ACM, 2008.

[74] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 257–273, 2014.

[75] C Mohan. Disk read-write optimizations and data integrity in transaction systems using write-ahead logging. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 324–331. IEEE, 1995.

[76] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. F4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 383–398, Berkeley, CA, USA, 2014. USENIX Association.

[77] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[78] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. Ssd failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*. ACM, 2016.

[79] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, , and Yutaka Suzue. Flat datacenter storage. In *OSDI*, 2012.

[80] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[81] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[82] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, pages 29–41, 2011.

[83] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review*, 43(4):92–105, 2009.

[84] Patrick OâĂŹNeil, Edward Cheng, Dieter Gawlick, and Elizabeth OâĂŹNeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[85] Juan Piernas, Toni Cortes, and José M García. Dualfs: a new journaling file system without meta-data duplication. In *Proceedings of the 16th international conference on Supercomputing*, pages 137–146. ACM, 2002.

[86] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 8:1–8:14, New York, NY, USA, 2015. ACM.

[87] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[88] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM computer communication review*, 27(2):24–36, 1997.

[89] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[90] Jason M Smith and David Stotts. Spqr: Flexible automated design pattern extraction from source code. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 215–224. IEEE, 2003.

[91] Jon A Solworth and Cyril U Orji. Write-only disk caches. In *ACM SIGMOD Record*, volume 19, pages 123–132. ACM, 1990.

[92] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[93] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 64–75. ACM, 1993.

[94] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*, pages 1–14, 1996.

[95] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[96] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the salus scalable block store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 357–370, 2013.

[97] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing storage for a million machines. In *HotOS*, 2005.

[98] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[99] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late:meeting deadlines in datacenter networks. In *ACM SIGCOMM*, pages 50–61, 2011.

[100] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 1–14, 2012.

[101] Yiming Zhang, Chuanxiong Guo, Dongsheng Li, Rui Chu, Haitao Wu, and Yongqiang Xiong. Cubicring: Enabling one-hop failure detection and recovery for distributed in-memory storage systems. In *NSDI*, pages 529–542, 2015.