PARIX: Speculative Partial Writes in Erasure-Coded Systems

Huiba Li, *mos.meituan.com;* Yiming Zhang, *NUDT;* Zhiming Zhang, *mos.meituan.com;*
Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng, *NUDT*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/li-huiba

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

PARIX: Speculative Partial Writes in Erasure-Coded Systems

Huiba Li
mos.meituan.com

Yiming Zhang
NUDT

Zhiming Zhang
mos.meituan.com

Shengyun Liu
NUDT

Dongsheng Li
NUDT

Xiaohui Liu
NUDT

Yuxing Peng
NUDT

Abstract

Erasure coding (EC) has been widely used in cloud storage systems because it effectively reduces storage redundancy while providing the same level of durability. However, EC introduces significant overhead to small write operations which perform *partial write* to an entire EC group. This has been a major barrier for EC to be widely adopted in small-write-intensive systems such as virtual disk service. Parity logging (PL) appends parity changes to a journal to accelerate partial writes. However, since previous PL schemes have to perform a time-consuming write-after-read for each partial write, i.e., read the current value of the data and then compute and write the *parity delta*, their write performance is still much lower than that of replication-based storage.

This paper presents PARIX, a speculative partial write scheme for fast parity logging. We transform the original formula of parity calculation, so as to use the *data* deltas (between the current/original data values), instead of the *parity* deltas, to calculate the parities during journal replay. For each partial write, this allows PARIX to *speculatively* log only the current value of the data. The original value is needed only once in a journal when performing the first write to the data. For a series of n partial writes to the same data, PARIX performs pure write (instead of write-after-read) for the last $n-1$ ones while only introducing a small penalty of an extra network RTT (round-trip time) to the first one. Evaluation results show that PARIX remarkably outperforms state-of-the-art PL schemes in partial write performance.

1 Introduction

Failures are common in large-scale cloud storage systems [22, 34, 35]. For example, more than 1000 server failures occur in one year in Google's 1800-server clusters [5]. To maintain data durability against failures, storage systems usually have two options, namely, replication [24] and erasure coding (EC) [25]. In replication, the storage system uses multiple replicas for each piece of data, while EC encodes the original data to generate new parities such that the original data can be recovered from a subset of the data and parities. EC has less storage overhead than replication while providing the same or even higher level of durability [32], and thus has been widely adopted in not only RAID systems [10, 30, 28, 20] but also modern cloud storage systems [13, 16, 27].

In cloud storage systems like Amazon Dynamo [12] and Windows Azure [7], small write operations [29] (which perform *partial write* to an entire EC group) are dominant for many real-world workloads. For erasure-coded storage systems that frequently perform small writes, it is important to efficiently support EC partial writes. Usually there are two ways to perform writes [8], namely, in-place update which directly updates the new data, and log-based update which appends the writes to a journal [26]. The logs are asynchronously *replayed* to update the data with the latest values when the system is idle.

Logging improves the write performance but degrades the read performance [32]. Parity logging (PL) [29] adopts a hybrid approach. Since normally only the data is read and the parities will only be read when the data is not available, PL respectively performs in-place update and log-based update for writes of the data and of the parities, so as to achieve a balance between reads and writes. However, state-of-the-art PL schemes [29, 17, 8] have to perform a time-consuming write-after-read for each partial write to compute the *parity delta* (which will be used to "patch" the parity during journal replay), and thus their write performance is still significantly lower than that of replication [32].

This paper presents PARIX, a speculative partial write scheme for fast parity logging. We transform the original formula of parity calculation, so as to use the *data* deltas (between the current and original values), instead of the *parity* deltas, to update the parities during journal replay. For each partial write, this allows PARIX to *speculatively* log only the new value of the data without reading its original value, which is needed only once in a journal when performing the first write to the data. For a series of n partial writes to the same data, PARIX performs pure write (instead of write-after-read) for the last $n-1$ ones while only introducing a small penalty of an extra network RTT (round-trip time) to the first one.

Based on PARIX, we have built a prototype of an erasure-coded block store [1] providing virtual disks that can be mounted by cloud-oblivious applications with strong consistency guarantees. Evaluation on the PARIX block store shows that PARIX not only achieves similar or even higher I/O performance compared to replication (with much higher storage efficiency), but also remarkably outperforms state-of-the-art PL schemes in partial write performance by up to orders of magnitude.

This paper makes the following contributions.

- We propose a novel speculative partial write scheme (PARIX) for fast parity logging in erasure-coded storage systems.
- We apply PARIX and implement an erasure-coded block store supporting efficient journal replay and fast failure recovery.
- We report evaluation results of PARIX's I/O performance from prototype measurement to demonstrate the effectiveness of our designs.

The rest of this paper is organized as follows. §2 discusses the background and related work. §3 introduces PARIX partial writes. §4 describes the prototype of a block store using PARIX-backed EC. §5 presents the evaluation results. And §6 concludes the paper.

2 Background

Erasure coding (EC) introduces less storage overhead than replication while providing the same level of durability [8]. Essentially, EC calculates linear combinations of the original data in the Galois Field [25] $GF(2^\wedge w)$, where encoding is performed in the unit of w-bit words (usually $w = 8$). For EC(m,k), we have k parity stripes $p_j, j = 1,2,\cdots,k$, for m original data stripes $d_i, i = 1,2,\cdots,m$, and the $m+k$ stripes are called an EC group which ensures durability under any k failures. The parity stripes $p_j, j = 1,2,\cdots,k$, is calculated by

$$(p_1, p_2, \cdots, p_k)^T = A \times (d_1, d_2, \cdots, d_m)^T, \qquad (1)$$

where $A = [a_{ij}]_{m\times k}$ is the encoding coefficient matrix.

Small writes are dominant for many real-world workloads in cloud storage systems, so it is important to efficiently support EC *partial writes*, i.e., writes on some part of an entire EC group. Early EC storage systems applies in-place update [6] to both data and parity, which leads to frequent disk seeks on hard-disk drives (HDDs).

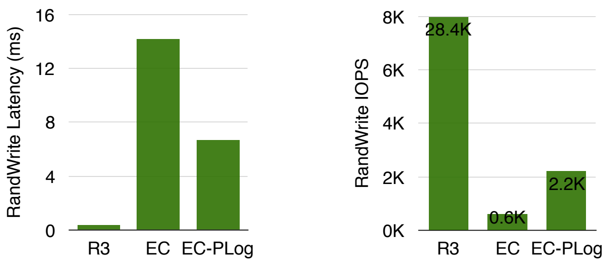


Figure 1: EC vs. replication in (cached) write latency and IOPS. R3: 3× replication with backup logging. EC: erasure coding with in-place update (no logging). EC-PLog: erasure coding with parity logging.

We compare EC (using in-place update) to replication in a small testbed of three machines, demonstrating EC suffers from poor write performance (Fig. 1).

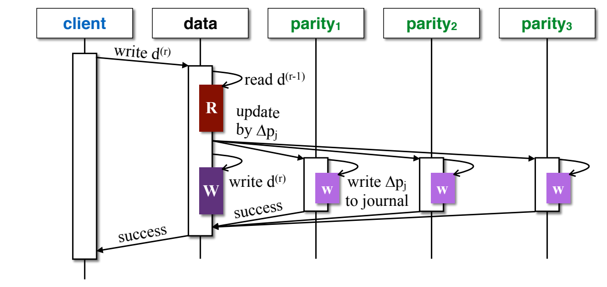
Log-based EC storage systems [16, 14, 8] improve small writes by appending the writes to a journal. Logging transforms random small writes into sequential writes to the journal, and thus (for HDDs) it avoids frequent disk seeks and boosts the write performance compared with in-place update. However, log-based approach suffers from poor read performance since the data is scattered in the journal. Parity logging (PL) [29] adopts a hybrid approach to alleviate this problem. It adopts in-place update to write the data and uses logging to write the parities. Since the parities will be read only when some data is unavailable, it improves small write performance without affecting normal reads.

State-of-the-art PL schemes [29, 17, 8] log the *parity delta* for each partial write, which will be used to "patch" the parity during journal replay. When updating a data stripe d_i, the delta Δp_j of parity stripe $p_j, j = 1,2,\cdots,k$, is calculated by

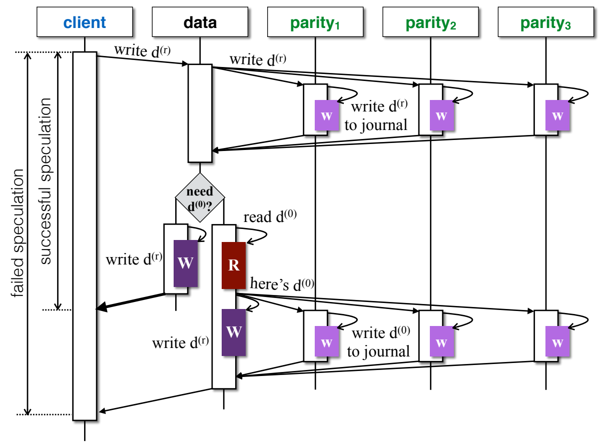
$$\Delta p_j = a_{ij} \times \Delta d_i, \qquad (2)$$

where Δd_i is the delta of data stripe d_i and $a_{ij} \in A$ is the encoding coefficient.

According to Eq. (2), for the r^{th} write on a data stripe d_i (denoted as $d_i^{(r)}$), we first have to read $d_i^{(r-1)}$, the current value of d_i before this write, and we have $\Delta p_j = a_{ij} \times (d_i^{(r)} - d_i^{(r-1)}), j = 1,2,\cdots,k$. Then we write the new data on the data server and send the k parity deltas to the parity servers. The entire procedure is illustrated in Fig. 2a. Our test shows that the latency of the write-after-read operation on 7,200 RPM HDDs is about 8.3 milliseconds, which is higher than that of pure write (due to one more disk seek). This contributes most to the performance degradation of partial writes in current PL schemes and results in significantly lower small write performance compared to replication (especially for cached writes), as shown in Fig. 1.



(a) Traditional partial writes in previous parity-logging read $d^{(r-1)}$ to compute Δp_j, which is appended to the parity journal.



(b) Speculative partial writes append $d^{(1)}$ and $d^{(0)}$ to the parity journal for the 1st write, and $d^{(r)}$ for the rth writes ($r \neq 1$), so as to avoid disk reads for *overwrites*.

Figure 2: Parity-logging vs. PARIX (in partial writes).

3 Speculative Partial Write

As discussed in §2, previous PL schemes have to perform disk reads for partial writes to compute the parity deltas, which will be used to update the parities during journal replay. Clearly, the key to improve the performance of partial writes is to reduce the number of reads.

Consider a series of r writes to the same data stripe d_i, say, $d_i^{(1)}, d_i^{(2)}, \cdots, d_i^{(r)}$, the parity stripes ($p_j, j = 1,2,\cdots,k$) of which have not yet been replayed. Let $d_i^{(0)}$ and $p_j^{(0)}$ be the original values of the data d_i and parity p_j, respectively. By Eq. (2), we could update a parity stripe p_j by $\Delta p_j^{(1)} = a_{ij} \times \Delta d_i^{(1)}$, $\Delta p_j^{(2)} = a_{ij} \times \Delta d_i^{(2)}$, $\cdots$, $\Delta p_j^{(r)} = a_{ij} \times \Delta d_i^{(r)}$, and thus we have

$$\begin{aligned} p_j^{(r)} &= p_j^{(0)} + \sum_{x=1}^{r} \Delta p_j^{(x)} = p_j^{(0)} + \sum_{x=1}^{r} a_{ij}(d_i^{(x)} - d_i^{(x-1)}) \\ &= p_j^{(0)} + a_{ij} \times (d_i^{(r)} - d_i^{(0)}). \end{aligned} \quad (3)$$

By Eq. (3), the current parity stripes $p_j^{(r)}, j = 1,2,\cdots,k$, could be calculated by $p_j^{(0)}$, $d_i^{(0)}$, and $d_i^{(r)}$.

This enables us not to use the delta of the parity, but to use the delta of the data itself, i.e., the difference between the data's latest and original values ($d^{(r)}$ and $d^{(0)}$, where for conciseness we omit the subscripts), to calculate the parities. Consequently, $d^{(0)}$ only needs to be read once when writing $d^{(1)}$. As shown in Fig. 2b, for each write the data server *speculatively* sends the latest value ($d^{(r)}$) to the parity servers without reading $d^{(0)}$. The data server reads $d^{(0)}$ only when the parity servers explicitly request it by returning an error code `NEED_DO`.

Note that in Fig. 2b the data server does not know whether $d^{(0)}$ is needed before receiving responses from parity servers. This is because $d^{(0)}$ is needed every time after the log gets merged into the parity chunk, which is performed independently by every parity server. It is too expensive to maintain the consensus about whether $d^{(0)}$ is needed for every chunk on every parity server, as it introduces overwhelming communication cost, memory footprint and design complexity.

For a series of r writes, $d^{(1)} \sim d^{(r)}$, the speculation will succeed for $r-1$ writes ($d^{(2)}, d^{(3)}, \cdots, d^{(r)}$) and will only fail once ($d^{(1)}$). Consequently, PARIX avoids disk reads (on the data server) for the last $r-1$ writes while only introducing a small penalty of an extra network RTT to the first one.

A partial write to an EC group performs both random writes to the *data* and sequential appends to the *parity*. Although PARIX and previous PL schemes have similar overhead in performing appends on the parity servers, PARIX remarkably outperforms them in performing writes on the data server: for a non-cached (resp. cached) *overwrite*, PARIX's overhead is a disk write (resp. a memory write), while previous PL schemes' overhead is a disk write after a disk read (resp. a memory write after a disk read) assuming the read is cache-missed.

If the speculation fails, $d^{(0)}$ needs to be sent from the data chunk to the parity chunk, introducing an extra network RTT of about 0.1 $\sim$ 0.2 milliseconds. The failed speculation also wastes extra network bandwidth, which is negligible for modern networks as the partial writes are small. It is the parity servers' responsibility to track whether $d^{(0)}$ is already in the log for its EC group.

Compared to existing PL techniques, the speculation-based scheme usually reduces the amount of reads and slightly increases the amount of writes when missing $d^{(0)}$. In the worst case (of large sequential one-shot writes), speculation might double the amount of writes. Besides, a few more extra bytes will be transferred between data/parity servers when missing $d^{(0)}$. Clearly, large sequential one-shot write workload is not suitable for the speculative partial write scheme, and could be recognized by an additional cache layer (which will be studied in our future work) and handled as full writes.

Full writes. Workloads in real-world applications per-

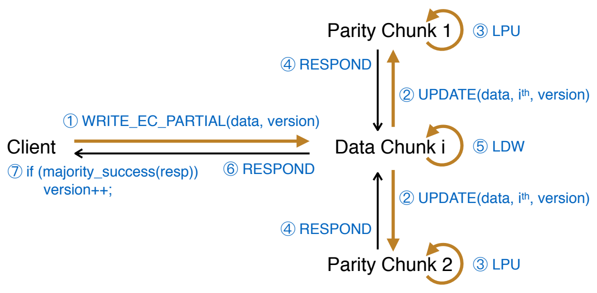

Parity Chunk 1
③ LPU
④ RESPOND
② UPDATE(data, ith, version)
① WRITE_EC_PARTIAL(data, version)
Client
Data Chunk i
⑤ LDW
⑦ if (majority_success(resp)) version++;
⑥ RESPOND
② UPDATE(data, ith, version)
④ RESPOND
Parity Chunk 2
③ LPU

Figure 3: Partial writes with strong consistency guarantee. LDW: local data write. LPU: local parity update.

form not only random small writes but also large sequential writes, which induce *full writes* on the entire EC groups. For a full write $d = (d_i)$, $i = 1, 2, \cdots, m$, we first compute the parity $p = (p_j)$, $j = 1, 2, \cdots, k$ by Eq. (1) and write the parity into the corresponding parity servers. We then invalidate previous logs for the EC group in the journal by appending a special mark I. Therefore, the logs for data d on the parity journal are in the form of "$d^{(1)}$, $d^{(0)}$, $d^{(2)}$, $\cdots$, I, $d^{(1)}$, $d^{(0)}$, $d^{(2)}$, $\cdots$, I, $\cdots$". Note that $d^{(1)}$ is ahead of $d^{(0)}$ due to the speculation (Fig. 2b).

Replay. The replay of parity journals is asynchronously performed when the disk is idle. The (basic) replay procedure is straightforward. A process traverses the journal from the beginning, and for each parity *block* (the minimum unit of a disk sector) it records the original and latest data blocks ($d^{(0)}$ and $d^{(r)}$) in RAM. When encountering a mark I, it invalidates the records that are ahead of I. Finally it updates all the parities using the recorded original and latest values by Eq. (3).

4 PARIX Block Store

We have implemented a prototype of PARIX block store (PBS), which utilizes PARIX to provide virtual disks [18, 31, 21] that can be mounted by virtual machines (VMs) running cloud-oblivious POSIX applications. The design of PBS is similar to Blizzard [21] and URSA [1], except that PBS uses PARIX-backed EC (instead of *replication* in Blizzard and URSA) to achieve data durability.

PBS organizes its data and parity into fixed-size (normally 64MB) data/parity *chunks*. Like URSA [1], PBS leverages MySQL [4] and Redis [2] to implement a global master [9], which can be configured into the high-availability (HA) mode [33]. The master manages metadata [23] such as chunk ID/size and performance statistics, coordinates services like volume creation and recovery [19], and detects errors like missing servers and inconsistent chunks. Clients retrieve chunk information from the master, and read/write data through the chunk servers. PBS adopts no *nested striping* [21], because EC has essentially achieved the same effect.

Fig. 3 shows the partial write procedure in PBS. A client sends a write request to the data server, which forwards it to *all* relevant parity chunks on different parity servers. When receiving the write, the parity servers perform *local parity update* (LPU) to the *per-chunk* journal (Fig. 2b) and respond to the data server. Note that the data server cannot perform in-place *local data write* (LDW) for updating the data to its disk until this point, since if the parity servers request the initial value ($d^{(0)}$) in their response it will need to perform read-after-write (instead of pure write) and send $d^{(0)}$ to them.

PBS extends the basic replay procedure (§3). We maintain an index structure in RAM recording the positions of $d^{(0)}$ and $d^{(r)}$ for each parity block, so that in replay we could update a parity block by reading only the two blocks in the journal without traversing the journal. For EC(m,k), in the worst case the size of logs needed to be read from the journal for replaying a parity chunk is $2m$ times the parity chunk size, because calculating a parity block requires at most m data blocks each of which requires its own $d^{(0)}$ and $d^{(r)}$. Since the journal is replayed whenever the disk is idle, in practice its size is much smaller than $2m$ times the parity chunk size.

In the worst case the index structure keeps 2 addresses in RAM for each data block (of 512B), but the actual index size is much less than that, because: (i) the sizes of most small writes are at least 4KB (a page), instead of 512B (a block), which only requires to keep in RAM the first index and the size of each write; and (ii) a large write will immediately free all the in-RAM indices for the corresponding blocks. Assuming an average write size of 64KB, the in-RAM index size is at least three orders of magnitude smaller than the size of the data.

Recovery. When a data/parity chunk fails, the healthy data/parity blocks in the corresponding EC groups are read to perform the recovery. The unplayed parity logs in the journal are first replayed, similar to the aforementioned normal replay procedure. The small difference is that the recovery is pipelined: each parity block is used to calculate the failed block right after it is replayed.

Consistency. PBS uses a lease to ensure a virtual disk has at most one active client at any time and leverages (chunk-level) versioning [15] to guarantee per chunk strong consistency [11] (Fig. 3). The versioning mechanism is similar to that of parity logging [17], the details of which are omitted here due to lack of space.

5 Evaluation

This section presents evaluation results of the PBS prototype. Our testbed consists of 10 machines, each with dual 10-core Xeon E5-2630v4 2.20GHz CPU, 128GB RAM, one 10GbE NIC port, and 10 7200RPM HDDs. The machines connect to a non-blocking 10GbE network. The

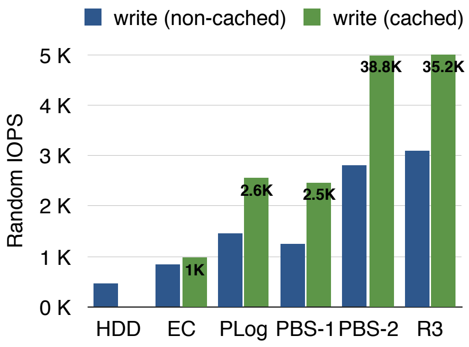


Figure 4: IOPS test. Background log flushing is omitted since PARIX only introduces a very small amount of extra data for logging compared to existing PL schemes.

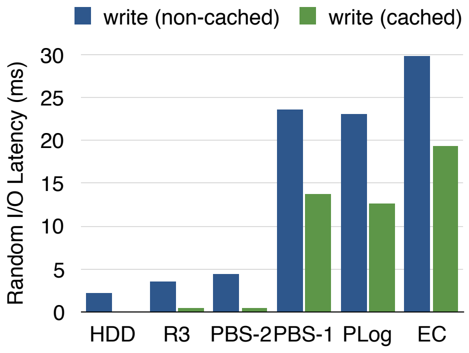


Figure 5: Latency test. Note that PARIX tries to eliminate unnecessary reads when performing speculative partial writes, so we measure random I/O latency so as to avoid the impact of prefetching and caching.

sizes of data/parity chunks and EC stripes are 64MB and 16KB, respectively. The virtual disk size is 100GB. The performance is measured by micro benchmarks, namely, small writes of 4KB block size (`fio --rw=randwrite bs=4KB`). §5.1 measures the performance of PBS in IOPS and latency, and §5.2 shows the recovery performance of PBS with different journal sizes.

5.1 PARIX Block Store

This section evaluates PBS. All measurements are performed on the VMs that mount virtual disks. For non-cached write, we turn off the cache in the OS and RAID cards, but keep the on-disk cache (otherwise the tests would not be able to get stable results). The queue depth is 1 and 32 for latency and IOPS tests, respectively.

Figs. 4 and 5 show the results in IOPS and random latency, respectively, where `HDD` represents the baseline performance of an HDD, `R3` uses 3× replication, `PBS-1` and `PBS-2` use EC(4,2) respectively with *failed* and *successful* speculation, `EC` is the standard EC(4,2) mode (no journal), and `PLog` uses traditional parity logging.

First, PARIX remarkably outperforms `PLog` when write-after-read is avoided in successful speculation. Second, PARIX is comparable to `PLog` even when the speculation fails, since the penalty is as small as an extra network RTT. Note that in order to compare different EC partial write schemes we must exclude the influence of read caching and prefetching, which exist in multiple layers in the I/O stack. Therefore, in Fig. 5 we measure the *random* (instead of *sequential*) I/O latency for all EC schemes, which may be up to more than 20 milliseconds unless the speculation succeeds (in `PBS-2`).

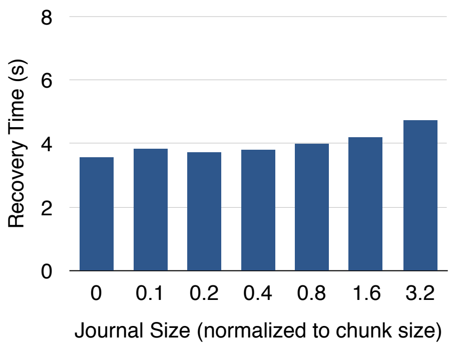


Figure 6: Single chunk failure recovery.

5.2 Recovery

We test the recovery performance of PARIX block store on 3 machines, using EC(4,2) with 64MB chunk size. A client first continuously performs small writes (of 4KB block size) until the (per parity chunk) journal size reaches a pre-defined proportion to the chunk size, which simulates the scenario that some corresponding parity logs in the journal have not yet been replayed before performing the recovery. We then emulate a data chunk failure by killing its service process. Fig 6 depicts the recovery times with respect to the exponentially-increased journal size (ranging from 0 to 3.2× chunk size). We do not test higher journal sizes, since in those cases replication would be even more efficient than EC and thus it might be inappropriate to apply PARIX. The result shows that the recovery overhead introduced by the parity journal is small, owing to the (in-RAM) full index.

6 Conclusion

This paper proposes PARIX for fast EC parity logging. We identify the root cause (write-after-read) for the poor performance of current EC partial writes, and speculatively performs pure write instead of write-after-read for small overwrites. We have implemented a prototype of PARIX block store (PBS). Evaluation shows that PBS remarkably outperforms current PL schemes. In the future, we plan to use PBS at the backend of our commercial block store in MOS (Meituan Open Service) [3].

Acknowledgement

This work is supported by the National Key Research and Development Program of China (2016YFB1000100) and the National Natural Science Foundation of China (61379055, 61379053, and 61222205). Yiming Zhang is the corresponding author. We thank Professor Ryan Huang, Huaimin Wang, Yijie Wang, Haibo Mi and Ziyang Li for their help to improve this paper.

References

[1] http://nicexlab.com/ursa/.

[2] http://redis.io/.

[3] https://mos.meituan.com/.

[4] https://www.mysql.com/.

[5] http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/.

[6] AGUILERA, M. K., JANAKIRAMAN, R., AND XU, L. Using erasure codes efficiently for storage in a distributed system. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on* (2005), IEEE, pp. 336–345.

[7] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., 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* (2011), ACM, pp. 143–157.

[8] CHAN, J. C., DING, Q., LEE, P. P., AND CHAN, H. H. 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)* (2014), pp. 163–176.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed storage system for structured data. In *OSDI* (2006), pp. 205–218.

[10] CHEN, P. M., AND LEE, E. K. *Striping in a RAID level 5 disk array*, vol. 23. ACM, 1995.

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

[12] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *SOSP* (2007), pp. 205–220.

[13] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *OSDI* (2010), vol. 10, pp. 1–7.

[14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[15] GLENDENNING, L., BESCHASTNIKH, I., KRISHNAMURTHY, A., AND ANDERSON, T. E. Scalable consistency in scatter. In *SOSP* (2011), pp. 15–28.

[16] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., YEKHANIN, S., ET AL. Erasure coding in windows azure storage. In *Usenix annual technical conference* (2012), Boston, MA, pp. 15–26.

[17] JIN, C., FENG, D., JIANG, H., AND TIAN, L. Raid6l: A log-assisted raid6 storage architecture with improved write performance. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)* (2011), IEEE, pp. 1–6.

[18] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 84–92.

[19] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[20] MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the performance of log-structured file systems with adaptive methods. In *SOSP* (1997), ACM.

[21] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., KHAN, O., AND NAREDDY, K. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 257–273.

[22] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., TANG, L., ET AL. f4: Facebook's warm blob storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 383–398.

[23] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J. K., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *SOSP* (2011), pp. 29–41.

[24] OUSTERHOUT, J. K., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G. M., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review 43*, 4 (2009), 92–105.

[25] RIZZO, L. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM computer communication review 27*, 2 (1997), 24–36.

[26] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 26–52.

[27] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment* (2013), vol. 6, VLDB Endowment, pp. 325–336.

[28] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (1995), USENIX Association, pp. 21–21.

[29] STODOLSKY, D., GIBSON, G., AND HOLLAND, M. Parity logging overcoming the small write problem in redundant disk arrays. In *ACM SIGARCH Computer Architecture News* (1993), vol. 21, ACM, pp. 64–75.

[30] THOMASIAN, A. Reconstruct versus read-modify writes in raid. *Information processing letters 93*, 4 (2005), 163–168.

[31] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 357–370.

[32] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems* (2002), Springer, pp. 328–337.

[33] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 307–320.

[34] ZHANG, Y., LI, D., GUO, C., WU, H., XIONG, Y., AND LU, X. Cubicring: Exploiting network proximity for distributed inmemory key-value store. *IEEE/ACM Transactions on Networking* (2017).

[35] ZHANG, Y., LI, D., TIAN, T., AND ZHONG, P. Cubex: Leveraging glocality of cube-based networks for ram-based key-value store. In *IEEE INFOCOM* (2017).