BCStore: Bandwidth-Efficient In-memory KV-Store with Batch Coding

Shenglong Li*, Quanlu Zhang*, Zhi Yang* and Yafei Dai*†
*Peking University
†Institute of Big Data Technologies Shenzhen Key Lab for Cloud Computing Technology & Applications
{lishenglong, zql, yangzhi, dyf}@net.pku.edu.cn

***Abstract*—In-memory key-value store KV-store is a crucial building block for many systems including large-scale web services and databases. Recent research works pay close attention to data availability of in-memory KV-store. Replication is a common solution to achieve data availability, but the significant memory consumption makes it not applicable for memory-intensive applications. Integrating erasure coding into in-memory KV-store is an emerging approach to achieve memory efficiency. However, traditional in-place update mechanism of erasure coding incurs significant bandwidth cost for write-intensive workload due to frequent broadcast of data update.**

In this paper, we build BCStore, a bandwidth-efficient in-memory KV-store with erasure coding. We propose a novel *batch coding* mechanism to optimize the bandwidth cost, and verify its feasibility and efficiency through theoretical analysis. To recycle the memory space due to delete and update operations, we design an efficient move-based garbage collection (GC) mechanism with a novel data layout called *virtual coding stripe*. Besides, BCStore guarantees consistent data read and write under these new designs, and supports fast online data recovery. We evaluate the performance of BCStore under various workloads. Experimental results show that BCStore can save up to 41% memory compared with replication, and achieve up to 2.4x throughput improvement and 51% bandwidth saving compared to erasure coding with in-place update.

***Keywords*-Bandwidth-efficient; Availability; Erasure Coding; In-memory KV-store**

I. Introduction

Recent years have witnessed an increasing demand on in-memory computing for large-scale web services and databases, which requires storing large-scale dataset in memory to serve millions of requests per second. For example, large-scale in-memory KV-Stores such as Memcached [1] and Redis [2] have been widely deployed in Facebook [3], Twitter [4] and LinkedIn [5] for improving service performance.

Recent research works pay close attention to data availability of in-memory KV-store. Even though many services have persistent storage to guarantee data durability, recovering large amount of data from persistent storage into memory is time-consuming. For example, Facebook reports that it takes 2.5-3 hours to recover 120GB data of an in-memory database from disk to memory [6], which degrades service performance seriously. Serving requests during recovery also incurs serious overhead because of inefficient random access from persistent storage. Thus, data redundancy in memory is essential for fast failover.

Replication is a traditional way to provide availability for disk-based storage, but it is too costly for memory-based storage due to high replication factor. Integrating erasure coding into in-memory KV-store is an emerging approach to achieve memory efficiency. As the increase of CPU speed, erasure coding can be computed fast enough to handle online services [7].

However, traditional in-place update mechanism of erasure coding [8, 9] incurs bandwidth amplification problem for write requests. When handling a write request, the overall bandwidth cost involves one update on data block and m updates on all the parity blocks (m is the number of parity nodes). The bandwidth cost is amplified by $m + 1$ times. Unlike replication which usually maintains three replicas, erasure coding could have more parity blocks for different availability requirements. As the number of parity nodes increases [10, 11], the bandwidth amplification problem becomes more serious.

Moreover, this problem becomes increasingly severe as the number of write requests grows [12–14]. For example, the workload of Yahoo KV-store has shifted from 80%-90% reads in 2010 to only 50% reads in 2012 [13]. For large-scale web services, peak load can easily run out of network bandwidth. Thus, the bandwidth amplification problem could render serious network congestion. It also makes the monetary cost of bandwidth several times higher, especially under the commonly used peak-load pricing model [15]. Besides, in workload-sharing clusters, the budget of bandwidth resource for each running application is usually limited [16].

To address the abovementioned problem, we propose BCStore, a bandwidth-efficient in-memory KV-store with erasure coding. The key design of BCStore is to do erasure coding in a batch manner, which is referred to as *batch coding*. Specifically, instead of updating parity blocks for each request, we aggregate write requests in a small time window and organize the data of write requests into new stripes for erasure coding. Then new data blocks and parity blocks are appended to original stored blocks. With the batch coding, the bandwidth cost for write requests becomes several times smaller than in-place update. More importantly, we formally analyze the induced latency of batch coding, and verify that it is usually ignorable under modest throughput.

Due to the append mode of batch coding, we need to

recycle the memory space of data blocks which are deleted or updated. To recycle the space quickly, we propose an efficient move-based garbage collection (GC) mechanism. To make GC equally efficient for variable-sized data, we further design a novel data layout called *virtual coding stripe*, to arrange variable-sized data in well-aligned stripes. We also formally prove the bandwidth efficiency of GC mechanism.

Under the batch coding and GC mechanism, BCStore guarantees consistent data read and write through maintaining *stable states* for successful requests. Moreover, the stable state facilitates consistent data recovery after failures. Online data recovery is also designed in our system for serving requests quickly during recovery.

We have implemented BCStore based on an asynchronous communication framework [17] and use Memcached as back-end storage. For comparison, we also deploy Cocytus [7], an in-memory KV-store which employs in-place update for erasure coding. We evaluate BCStore under various workloads with different key distributions and value sizes, and also evaluate the systems with different configurations of erasure coding. The results show that BCStore can save up to 41% memory compared with replication, and achieve up to 2.4x throughput improvement and 51% bandwidth saving compared to Cocytus with little latency overhead.

Our contribution can be summarized as follows:

- We are the first to design batch coding in in-memory KV-store, which achieves high bandwidth efficiency.
- We devise a novel move-based GC mechanism for quickly memory recycling, and propose a new data layout to arrange variable-sized data for efficient GC.
- We prove the feasibility and efficiency of batch coding and GC mechanism through analyzing latency cost and bandwidth cost in theory.
- We design new strategies for consistent read and write and fast data recovery based on batch coding.
- We build an in-memory KV-store integrating above designs. Comprehensive evaluation shows the efficiency of our system.

The rest of this paper is organized as follows. The next section describes necessary background about replication and erasure coding and explains the bandwidth amplification problem. Section III describes our key designs and theoretical analysis. Section IV demonstrates our system and implementation. Section V shows experimental results. Finally, Section VI discusses related work, and Section VII concludes the paper.

II. Background and Motivation

In this section, we first introduce two classic redundant schemes for data availability. Then we pinpoint the bandwidth amplification problem of *in-place update* used in erasure coding.

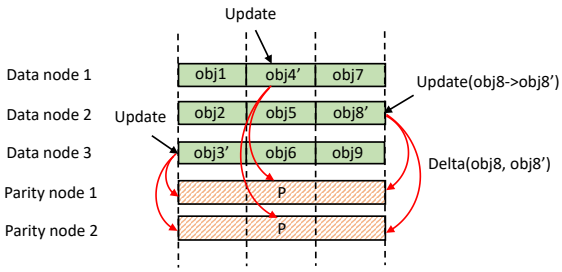


Figure 1. In-place update for small objects.

A. Two Redundant Schemes

Replication. Replication is widely used in storage system to achieve high data availability. In replication scheme, each data node has n backup nodes to store data replications to tolerate n failures. However, replication incurs significant memory overhead for achieving data availability of memory, the memory usage is amplified by $n+1$ times. For many memory-intensive applications, memory is a scarce resource [18]. And in workload-sharing clusters, the memory budget for each application is limited, which makes the situation even worse. Thus, replication is not suitable to be applied for large-scale in-memory applications.

Erasure Coding. Erasure coding is a space-efficient redundancy scheme to provide data availability. We denote a (k, m)-code as an erasure coding scheme. Data is organized into k equal-size blocks called *data blocks*. Data blocks are encoded to generate m coded blocks called *parity blocks*. Data blocks are distributed across k data nodes and parity blocks are distributed across m parity nodes.

More specifically, each parity block is computed by a linear combination of data blocks. For a $(k,\ m)$-code, let $\{a_{ij}\}_{1\le i\le m, 1\le j\le k}$ to be the coefficients of the linear combinations and let $\{D_i\}_{1\le i\le k}$ and $\{P_i\}_{1\le i\le m}$ denote data blocks and parity blocks respectively. Parity blocks can be computed by:

$$P_i = \sum_{j=1}^{k} a_{ij} D_j \tag{1}$$

We consider *Maximum Distance Separable* erasure coding, *i.e.* , the original data can be reconstructed from any k blocks of $k+m$ data and parity blocks. The storage blow up of (k, m) coding scheme is only $(k+m)/k$, which saves much space compared with replication. Meanwhile, the increase of CPU speed enables fast erasure coding (both encoding and decoding can be done at 4.24-5.52GB/s), which makes erasure coding applicable for online in-memory services [7].

B. Bandwidth Amplification Problem

Currently, erasure coding has started to be applied to in-memory KV-store to achieve data availability. As the objects

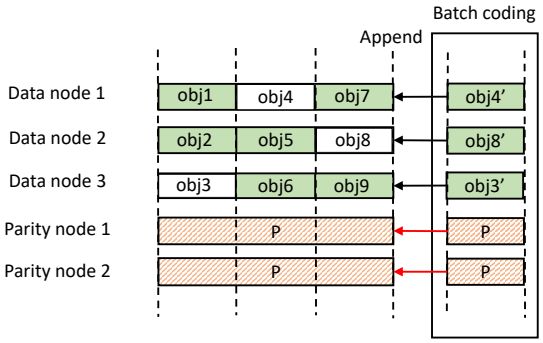


Figure 2. Batch coding mechanism.

in KV-store are usually small, it is not feasible to divide an object into multiple data blocks for coding. Thus, *in-place update* is applied to encode small objects [7]. Figure 1 shows the principle of in-place update. An object is not divided but encoded with other objects. We call the object group for coding as a coding stripe (*e.g.* the group including obj1, obj2 and obj3). In each coding stripe, each object can be seen as a data block. For an object update (*e.g.* obj8), a *delta* between the new object and its old version is computed, then the delta will be sent to all the parity nodes for updating the corresponding parity blocks.

The principle can be formalized by the following formula,

$$P_i^{'}(o) = P_i(o) + a_{it}(D_t^{'}(o) - D_t(o)) \quad (2)$$

Suppose that a word $D_t(o)$ at offset o in data block D_t is updated to a new word $D_t^{'}(o)$, the word $P_i(o)$ at offset o of each parity block P_i needs to be updated to the new word $P_i^{'}(o)$ accordingly.

Obviously, in-place update mechanism incurs bandwidth amplification for write requests. The overall bandwidth cost of handling one write request consists of the traffic on one data node and m parity nodes, which can be computed by adding the size of updated object and m times the size of delta. Since the size of delta is equal to the size of object, the bandwidth cost of handling one write request is amplified by $m+1$ times. This problem would be more serious with the increase of m [10, 11] and with prevalence of the write-intensive workload [12–14].

III. Design and Analysis

A. Basic Batch Coding

To solve the bandwidth amplification problem of in-place update, we resort to batch coding mechanism. Instead of updating all the parity blocks in place for each write request, we aggregate write requests and organize the objects into coding stripes. After batching enough objects for constructing a stripe, new parity blocks are computed in a batch manner based on batched objects. Then new data blocks and parity blocks are sent to corresponding data nodes and parity nodes and appended to original blocks.

Specifically, as shown in Figure 2, when obj3, obj4 and obj8 are updated, new objects including obj3', obj4' and obj8' are batched in a stripe before sending to storage nodes. Then new parity blocks are computed by the new objects. After coding, new data blocks and parity blocks are sent to storage nodes. The updated blocks in the original coding stripe are marked as invalid blocks (white blocks in the Figure 2). With batch coding mechanism, every k data blocks will generate m parity blocks, thus the bandwidth cost of parity blocks is $\frac{m}{k}$ times the bandwidth cost of data blocks. The bandwidth amplification of batch coding is only $\frac{m}{k}+1$ times, while the bandwidth amplification of in-place update is $m+1$ times.

Latency cost analysis. Batch coding induces extra latency cost of requests due to waiting for batch coding. Intuitively, the higher the throughput becomes, the less latency is induced by batching one coding stripe. On the other hand, higher throughput put heavy pressure on network capacity, thus, batch coding is more urgently required under heavy load.

We formally analyze this latency problem to find out the condition that keeps batch coding's bandwidth benefit. The expectation waiting time for filling up one coding stripe can be represented by $E(t)$ (the deduction can be seen in Appendix IX-A):

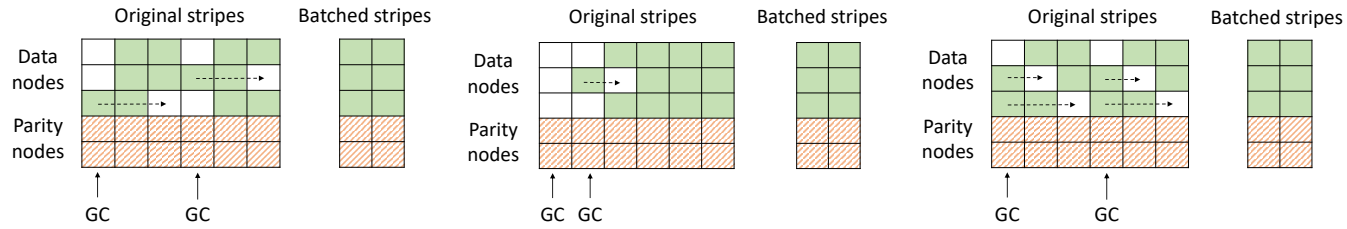
$$E(t) = \int_0^{\infty} t * (T * (1 - 1/e^{\frac{T*t}{k}})^{(k-1)})/(e^{\frac{T*t}{k}})\, dt \quad (3)$$

T is the request throughput and k is the number of data nodes. Figure 3 shows the relation between throughput and expectation of latency cost. With the increase of throughput, the expectation of latency cost decreases. When request throughput exceeds 10^4 ops/s, the expectation of latency cost for waiting one coding stripe is less than 550us. When request throughput reaches 10^6 ops/s, the expectation of latency cost for waiting one coding stripe is only 5.5us, which is far less than the network latency.

To limit the latency cost of batch coding, we allow applications to set a latency bound ϵ. When real-time throughput can meet the condition $E(t) < \epsilon$, we use batch coding to deal with write requests. Otherwise, we use in-place update instead of batch coding to meet the latency requirement. Within the latency bound, multiple coding stripes can be encoded in a batch to improve the efficiency of batch coding.

B. Garbage Collection

Based on batch coding mechanism, data blocks and parity blocks are appended to the storage, without updating original coding stripes in place. When a request updates an existing object in original stripe, we still allocate new coding stripe to store the new object and compute new parity blocks. Then



(a) Heuristic GC. *The first stripe and fourth stripe are chosen to GC.*

(b) Heuristic GC with Zipfian workload. *The first stripe and the second stripe are chosen to GC.*

(c) Worst case of GC. *Each stripe has an invalid block.*

Figure 4. GC performance for different coding spaces.

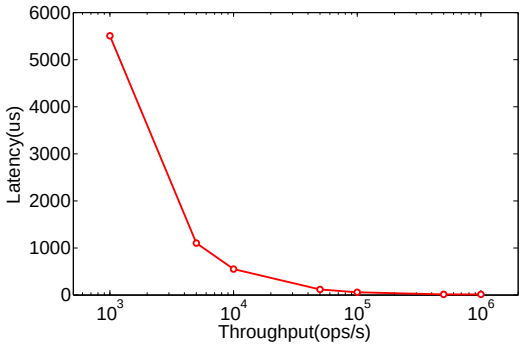


Figure 3. The relation between throughput and waiting time for filling up one coding stripe (k = 3).

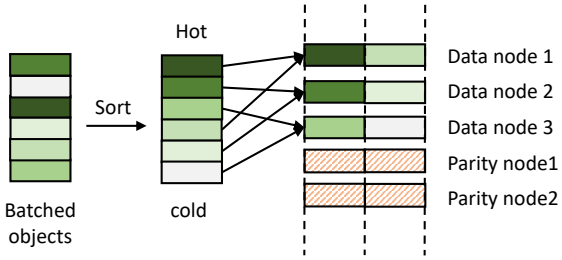


Figure 5. Data arrangement based on key popularity. *Different color represents different popularity.*

the object in the original coding stripe becomes invalid. However, we can not delete the invalid object directly, because it is needed to guarantee the redundant degree of other objects in the same stripe. With the increase of the number of update requests, invalid data blocks would degrade memory efficiency.

Thus, we need to design a garbage collection (GC) algorithm to recycle the space of invalid block. Our key solution is moving valid blocks to replace the invalid ones among coding stipes in order to fill up the coding stripes with valid blocks, then the empty coding stripes can be released. The efficiency of move-based garbage collection depends on the number of moved blocks, because moving a data block causes a delta broadcast to update all the parity blocks.

To achieve bandwidth efficiency, we propose a heuristic garbage collection algorithm. The key idea is to move the valid blocks from the stripes with the most invalid blocks to the stripes with the least invalid blocks to replace the invalid blocks. The block moving strategy is illustrated in Figure 4(a). In this case, white blocks are updated by the requests in batched stripes and becomes invalid. We select the valid blocks in the first stripe, and moves the blocks to the third stripe. Similarly, we move the valid blocks from the fourth stripe to the sixth stripe. Then invalid data blocks and parity blocks in the first and fourth stripe can be released.

Note that given any situation of coding stripes, our GC can free all the invalid blocks with the least moved blocks. Because the number of released stripes is fixed in any GC method, moving blocks in the stripes with the most invalid blocks can minimize the number of moves. Then bandwidth cost for updating parity blocks can be minimized. Though garbage collection algorithm adds CPU cost for handling update workload, we assume that computation will not be the bottleneck with the speedy CPU cores.

Popularity-based data arrangement. We further improve our GC's efficiency by leveraging the characteristic of common workloads with skewed key distribution (*e.g.* Zipf [19]). We aim to make the invalid blocks concentrate on few stripes, which can reduce the number of moved blocks.

We propose a popularity-based block arrangement method to improve GC efficiency. Specifically, as shown in Figure 5, batched requests are sorted according to key popularity, then we place the sorted request objects into the coding stripe in the order of key popularity. Because hot objects are more easily accessed together, more invalid blocks will be generated in few coding stripes. As Figure 4(b) shows, with this data arrangement method, the following requests in batched stripes can be concentrated in the first three stripes based on Zipfian workload. Thus, only one data block needs to be moved to release two stripes.

Bandwidth analysis. Because GC induces extra bandwidth cost, we need to analyze the overall bandwidth cost of batch coding compared with in-place update to verify the bandwidth efficiency of batch coding. Here, we propose a theorem to give a upper bound of the overall bandwidth cost of batch coding.

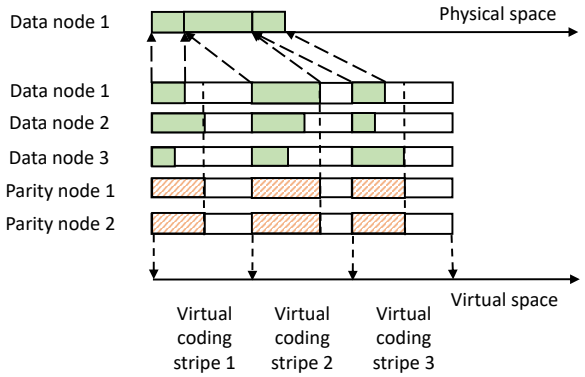


Figure 6. Virtual coding stripe. *Green blocks in data nodes are the batched variable-size objects.*

Theorem 1. *Given uniform write requests over data nodes, the overall bandwidth of batch coding plus garbage collection can not exceed that of in-place update, even in the worst case.*

Here we take an example to understand the worst case intuitively. As Figure 4(c) shows, each stripe has only one invalid block. Based on our GC method, each GC stripe needs to move $k-1$ blocks (k is the number of data nodes), which costs the most GC bandwidth. Through adding the GC bandwidth and the bandwidth cost of coding blocks, we can obtain the overall bandwidth of batch coding. With the guarantee of this theorem, batch coding can always save bandwidth compared with in-place update, even in the worst case. The detailed proof can be seen in Appendix IX-B.

C. Batch Coding for Variable-sized Objects

In particular, objects in the coding stripe are not aligned well due to variable object sizes, which makes it difficult to do garbage collection. Because the objects with variable sizes make coding stripe size different, invalid blocks can not be replaced by other blocks easily. Moving a block needs to match the block size to find an appropriate position.

To solve this problem, we make objects aligned in *virtual coding stripes* as shown in Figure 6. Each virtual coding stripe has a large fixed-length space and is aligned in virtual address. The objects with different sizes can be loaded into the virtual data blocks in each coding stripe. The size of parity block is the same as the biggest object in the virtual coding stripe. We append zero to the other objects for coding alignment. Note that there is little memory waste for this data arrangement, because the data arrangement demonstrated above is in virtual space, only the real data is mapped to physical space as shown in Figure 6. With virtual coding stripe, the blocks can always be moved between stripes, which can facilitate garbage collection.

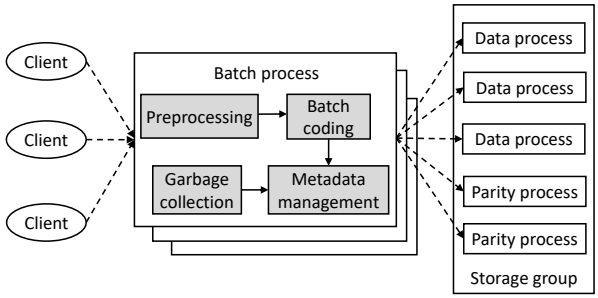


Figure 7. System architecture.

IV. System and Implementation

In this section, we demonstrate BCStore in detail. We first illustrate our system architecture to give an overview of our system. Then we describe how to handle read and write in BCStore. At last we elaborate the mechanisms for consistency and data recovery.

A. System Architecture

As shown in Figure 7, the key component in BCStore is batch process. Each batch process is associated with a group of storage processes including data processes and parity processes, which are distributed on different nodes. Batch process can be deployed on front-end web servers [3] to collect requests and do batch coding before the requests are sent to storage nodes.

During the batch processing, batch process receives write requests from clients and batches them together within a small time window. During preprocessing, the requests are deduplicated according to key for reducing bandwidth cost and sorted based on key popularity for improving the efficiency of garbage collection. Then the data of write requests are arranged in virtual coding stripes and are encoded to compute parity blocks. At last, data blocks and parity blocks are sent to corresponding data processes and parity processes.

Besides, batch process maintains metadata of coding stripes, such as the mapping from key to corresponding coding stripes. Through metadata lookup, batch process can recycle memory space for garbage collection.

In BCStore, we only apply erasure coding on values, while keys and metadata are replicated on multiple machines for redundancy, because they are usually much smaller than values [7]. We use "object" to denote the value of a key-value pair in the following subsections.

B. Handle Read and Write

To support read and write requests based on batch coding, we introduce our metadata design. BCStore leverages the address of the coding stripe as key to index blocks, which is represented by a monotonous incremental id cid.

A read request can be executed as follows. When batch process receives a read request, it first finds out the data process that takes charge of the key by computing the key's hash (*e.g.* consistent hash). Meanwhile, it retrieves the key's coding stripe address. Then it sends the address to that data process to ask for the object.

For write requests, BCStore allocates new coding stripes to load request objects and records the mapping of key to coding stripe. After encoding, the data blocks and parity blocks with the coding stripe address are sent to corresponding storage processes based on key's hash. When write requests update existing keys, batch process updates the key index from original coding stripe to new one and marks the old data blocks in the original stripe as invalid. The metadata of each coding stripe is also leveraged by garbage collection.

C. Consistency

BCStore supports strong consistency when handling read and write requests. On one hand, read requests should always read the newest data which are written successfully. On the other hand, write request should be returned successfully only if all the redundant blocks are written into back-end storage.

To guarantee successful writes, BCStore proposes *stable state* of batch. Specifically, each batch is assigned with an *xid*, which increases monotonously at each batch process. After coding, data blocks and parity blocks carry the *xid* and are sent to corresponding storage processes. When receiving blocks from batch process, storage processes first write the blocks into a buffer, and then send back the responses with *xid* to batch process. After batch process receives all the responses with the same *xid*, the batch with the *xid* is regarded as stable batch. Then batch process records *xid* as stable batch id and sends responses of the requests in the stable batch to clients. At last, the stable batch id is sent to data processes and parity processes, then the blocks in the buffer with smaller or equal *xid* are stored to back-end storage.

When batch process receives a read request, it first embeds the stable batch id to the request and sends the request to corresponding data process. Data process first flushes the buffered data with smaller or equal *xid* compared with stable batch id into back-end storage, then retrieves the object from back-end storage based on coding address and returns it to client. Thus, read requests can always read the latest writes, which avoids read inconsistency when failure occurs.

D. Recovery

When a process fails, BCStore needs to reconstruct lost data while serving client's requests. Here we introduce the recovery approaches to handle storage process failure and batch process failure. We also assume no more than m storage processes crash simultaneously. Some data would become unavailable if more than m storage processes crash.

Storage process recovery. When a storage process fails, each batch process first broadcasts its stable batch id to corresponding alive storage processes. Storage processes apply the buffered data which *xid* is less than or equal to the stable batch id. After reaching a stable state among all storage processes, batch process communicates with any k storage processes on different storage nodes to request the coding data. Specifically, it looks up the coding stripe addresses of the keys in each coding stripe, then sends them to corresponding storage processes. After receiving k blocks of each coding stripe, batch process recovers the lost blocks. When a new storage process restarts, it migrates the recovered data with corresponding index to the new one.

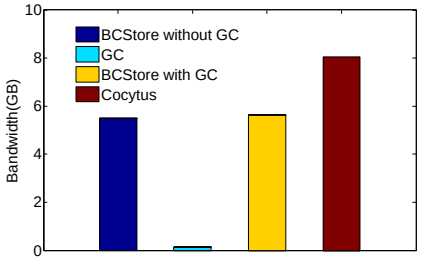
BCStore allows the batch process to handle requests during recovery to maintain the performance of service. To handle read requests on the lost data, batch process first locates the coding stripe based on the request key. Then the lost data on that coding stripe can be recovered first and returned to client timely. When handling write requests onto the failed process, the request data can be stored in the buffer of batch process temporarily. After a new storage process restarts, the request data will be migrated to the new storage process.

Batch process recovery. When batch process fails, a new batch process is launched on another machine to take over the jobs of failed batch process. The requests to the failed batch process will be redirected to the new batch process. First, it copies the metadata replication of failed batch process from other batch processes. Then, new batch process communicates with the storage processes that are associated with the failed batch process to collect the latest *xid*s. The minimum number Min_{xid} among those *xid*s is chosen as the stable batch id. Because the requests in the stripes with *xid* $\leq Min_{xid}$ may have been successfully returned to clients, while those in the stripes with *xid* $> Min_{xid}$ are not returned. After reconstructing the metadata, new batch process can continue to serve client's requests.

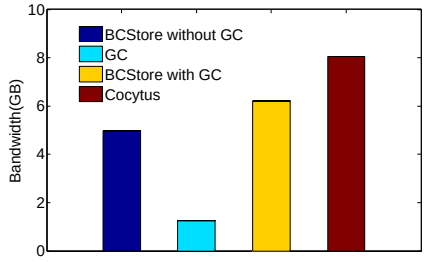
The uncompleted batches during batch process failure can be safely discarded without violating linearizability (*xid*s of these batches are larger than Min_{xid}), because the requests in these batches have not been replied to the clients. The data in these batches is also not visible to the following read requests due to our specific approach of handling read requests (§ IV-C). The new batch process informs the associated storage processes to discard the data with batch id larger than Min_{xid}, and then starts to generate new coding batches with the batch id from $Min_{xid} + 1$.

V. Evaluation

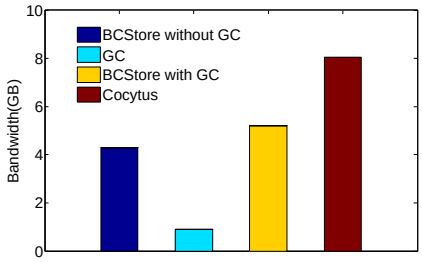
In this section, we evaluate the performance of BCStore and compare to erasure coding with in-place update and replication in different aspects including bandwidth cost, throughput performance, memory consumption and latency cost.



(a) Bandwidth cost for uniform workload.



(b) Bandwidth cost for moderate-skewed Zipfian workload.



(c) Bandwidth cost for heavy-skewed Zipfian workload.

Figure 8. Bandwidth cost for different workloads.

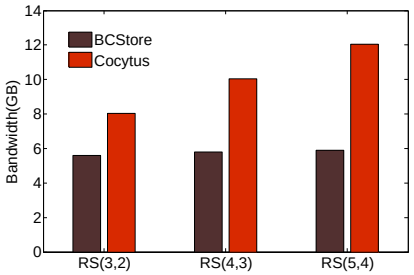


Figure 9. Bandwidth cost for different coding schemes.

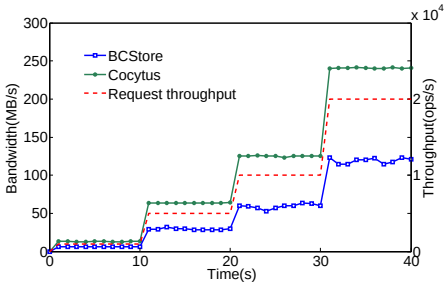


Figure 10. Bandwidth cost for different throughput.

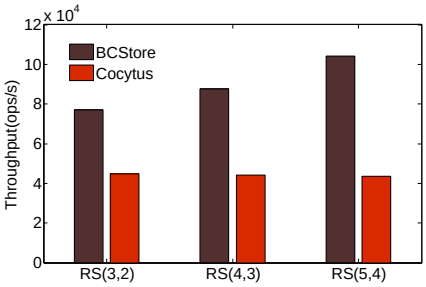


Figure 11. Throughput performance for different coding schemes.

A. Setup

Cluster configuration and system parameters. Our experiments are conducted on 10 machines running SUSE Linux 11 containing 12 * AMD Opteron Processor 4180 CPUs. The machines are distributed over multiple racks and connected via 1Gb Ethernet. By default, we implement RS(3,2) erasure coding on five nodes which can tolerate two node failures. Each node deploys a storage process to handle requests. Each node also acts as front-end web server to issue requests to other nodes. We deploy 10 batch processes on each node and each batch process is responsible for a range of key. We set the number of batched requests to be 100 on each batch process. We implement real-time GC in the experiments to improve memory efficiency.

We build BCStore based on an asynchronous communication framework [17]. We deploy Memcached [1] as the back-end storage and leverage libMemcached [20] to communicate with Memcached server on storage nodes. We use Jerasure [21] and GF-complete [22] for the Galois-Field operations in RS coding.

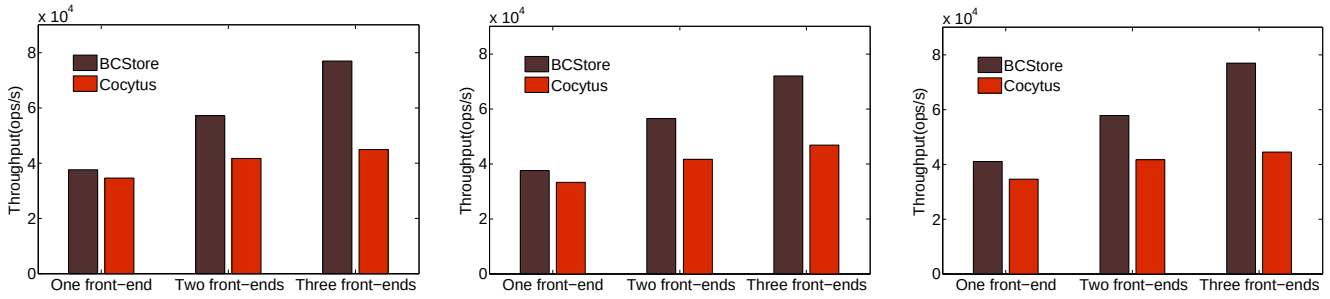
Targets of comparison. We compare BCStore with Cocytus [7], which employs in-place update for erasure coding. By default, we configure Cocytus with RS(3,2) erasure coding on five nodes. Each node also acts as front-end web server to issue requests as the configuration of BCStore. Besides, we compare with replication for evaluating memory consumption in Section V-D.

Workload. We use YCSB [23] benchmark to generate our workload. We generate three workloads through adjusting the parameter of Zipfian distribution, including uniform distribution (random keys), moderate-skewed Zipfian distribution, heavy-skewed Zipfian distribution. Keys in the workload range from zero to ten million. Since the median of the value sizes from Facebook [3] are 4.34KB for Region and 10.7KB for Cluster, we test different systems with similar value sizes, including 1KB, 4KB and 16KB. By default, value size is set to be 4KB in the following experiments. We evaluate the systems with 50%:50% read/write ratio workload based on Yahoo KV-store workload [13].

B. Bandwidth Cost

First, we evaluate overall bandwidth cost of BCStore and Cocytus with one front-end node. The overall bandwidth cost of BCStore consists of the traffic between batch nodes and storage nodes, and GC traffic from data nodes to parity nodes. The overall bandwidth cost of Cocytus includes the traffic of data nodes for read and write, and the traffic from data nodes to parity nodes for data update. Figure 8(a) - 8(c) show the bandwidth cost for different workloads. From the results, BCStore achieves notable bandwidth saving compared with Cocytus for all the three workloads. Since GC can be executed in idle time, BCStore can save more bandwidth cost during service peak hours.

In Figure 8(a), BCStore costs little GC bandwidth because there are few update operations in uniform workload.



(a) Throughput performance for uniform workload.

(b) Throughput performance for moderate-skewed Zipfian workload.

(c) Throughput performance for heavy-skewed Zipfian workload.

Figure 12. Throughput performance for different workloads.

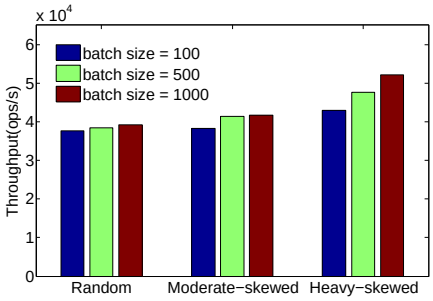


Figure 13. Throughput performance for different batch sizes.

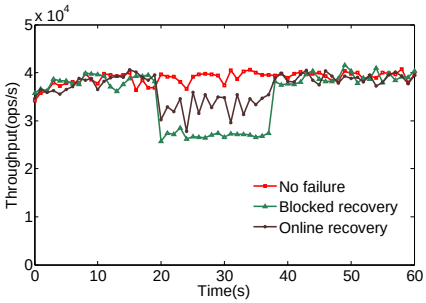


Figure 14. Throughput for recovery.

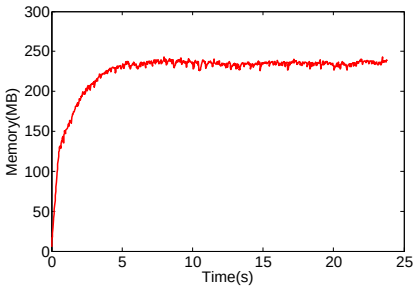


Figure 15. Real-time memory consumption for moderate-skewed Zipfian workload.

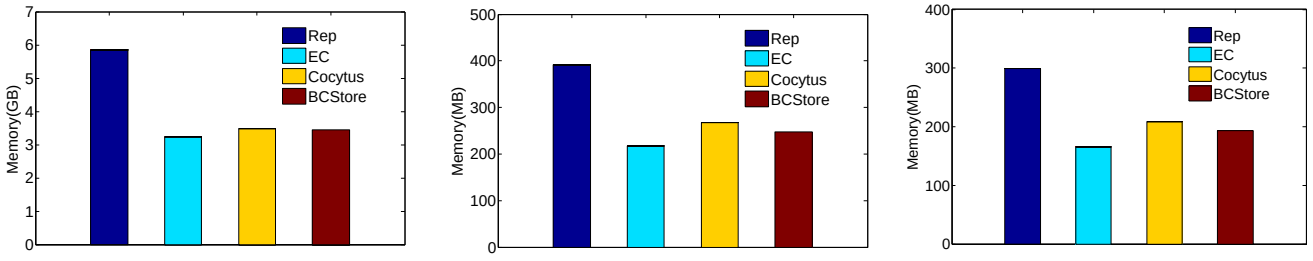
When key distribution becomes skewed in moderate-skewed Zipfian distribution, as shown in Figure 8(b), BCStore costs more GC bandwidth. That is because the number of update requests increases, which causes more data moves and delta broadcast to parity nodes. Besides, taking advantage of key deduplication in batch coding, BCStore costs less bandwidth for Zipfian workload than uniform workload without GC. For the workload of heavy-skewed Zipfian distribution, as shown in Figure 8(c), GC bandwidth cost reduces because more hot keys concentrate on hot stripes, which can reduce the number of data moves and delta broadcast. Besides, key duplication during batch coding can save more bandwidth for heavy-skewed Zipfian workload.

Then we test the overall bandwidth cost for different coding schemes including RS(3,2), RS(4,3) and RS(5,4) on uniform workload. Figure 9 shows that with the increase of the number of data nodes and parity nodes, BCStore saves more bandwidth than Cocytus. This is because, with the increase of parity nodes, Cocytus has to send the delta to more parity nodes, thus, the bandwidth amplification increases linearly, *i.e.* $m+1$ times. In contrast, the bandwidth cost for data update in BCStore is proportional to the ratio between the number of parity nodes and the number of data nodes due to batch coding. The overall bandwidth of BCStore can save up to 51% bandwidth cost compared with Cocytus.

Next, we evaluate the overall bandwidth cost under different throughput with RS(5,4) coding scheme. We run the uniform workload continuously and change request throughput per 10 seconds. Then we compare the overall bandwidth cost per unit time of BCStore and Cocytus. Figure 10 shows that when request throughput increases, BCStore can save more bandwidth than Cocytus. When the application sets a latency bound of batch coding, we can switch from batch coding to in-place update when throughput is below the theoretical bound (according to Formula 3). Note that the bandwidth cost of in-place update is acceptable under low throughput (*e.g.* 5000ops/s). When the throughput increases, we can switch to batch coding to reduce the bandwidth cost during service peak hours.

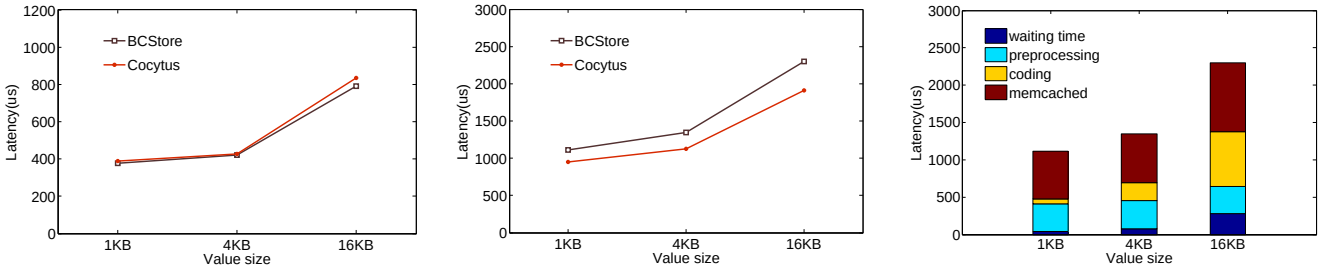
C. Throughput

Then, we compare throughput performance between BCStore and Cocytus with different number of front-end nodes for different workloads. As shown in Figure 12(a) - 12(c), with the increase of the number of front-end nodes, BCStore can achieve higher throughput than Cocytus. For one front-end node, the throughput of two systems is similar because the bandwidth of front-end node becomes bottleneck, even though BCStore can save more bandwidth than Cocytus. When we add front-end nodes, the throughput of Cocytus does not increase significantly, because the bandwidth of



(a) Memory consumption for uniform workload. (b) Memory consumption for moderate-skewed Zipfian workload. (c) Memory consumption for heavy-skewed Zipfian workload.

Figure 16. Memory consumption for different workloads.



(a) Read latency. (b) Write latency. (c) Write latency performance and analysis.

Figure 17. Latency cost and analysis.

storage nodes becomes bottleneck. In contrast, BCStore can improve throughput without the bandwidth bottleneck of storage nodes and take full advantage of the bandwidth of multiple front-end nodes. With the key distribution becomes skewed, BCStore can achieve high throughput because of key deduplication.

Besides, we add storage nodes and evaluate throughput performance with different RS coding schemes including RS(3,2), RS(4,3) and RS(5,4) with three front-end nodes. As shown in Figure 11, with the increase of the number of storage nodes, BCStore performs more throughput advantage over Cocytus. This is because when the number of data nodes increases, each parity node receives more update requests from data nodes, which causes the parity node to be bottleneck easily. In contrast, BCStore can take advantage of cluster bandwidth efficiently to achieve higher throughput. According to the results, the throughput of BCStore can outperform Cocytus by up to 2.4x improvement.

Next we evaluate throughput performance of BCStore with different numbers of batched requests for different workloads. As shown in Figure 13, with the batch size increases, the throughput becomes higher, because large batch size can facilitate the deduplication efficiency and GC efficiency. With the workload becomes skewed, the benefit of large batch size becomes obvious, because more requests are deduplicated during batch coding.

Then we evaluate the throughput performance during data recovery. We run the uniform workload continuously with RS(3,2) and emulate a data node failure at 20s. We compare the performance of blocked recovery which blocks the requests onto the failed node, and online recovery which can handle requests during recovery. As Figure 14 shows, throughput drops after one data node fails. With blocked recovery, BCStore recovers the throughput performance after all the data of the failed data is recovered. Compared with blocked recovery, online recovery can improve the throughput performance because the request data can be recovered first and returned to clients timely.

D. Memory Consumption

Then, we test memory consumption of BCStore for different workloads. As Figure 16 shows, BCStore achieves remarkable memory saving compared with 3-replication (Rep) for all the workloads, because of the space efficiency of erasure coding. The memory saving can reach to 41% compared with replication. Moreover, BCStore can achieve similar memory consumption to Cocytus, because BCStore leverages real-time GC to reduce memory overhead of invalid data blocks and extra parity blocks. Both BCStore and Cocytus need store extra metadata and metadata replication, so the memory consumption is a little higher than erasure coding for values (EC).

Besides, we also monitor the real-time memory consumption of BCStore to evaluate GC timeliness. Figure 15 shows the real-time memory consumption for moderate-skewed Zipfian workload. From the results, we can see that memory reaches the maximum in a short time and becomes stable in the following time. This is because most of the

requests are update requests in the workload. Through real-time GC, BCStore can achieve similar memory efficiency with Cocytus.

E. Latency

At last, we evaluate latency cost of BCStore and Cocytus using uniform workload with RS(3,2) with batch size of one coding stripe. Figure 17 shows the results of latency performance for read and write with specific latency components. From the results of Figure 17(a), BCStore achieves similar read latency as Cocytus for different value sizes.

Figure 17(b) shows the average write latency of BCStore and Cocytus to handle one coding stripe requests. From the result, write latency of BCStore is a little higher than Cocytus. Then we analyze the latency components for handling write requests in BCStore, which is shown in Figure17(c). The write latency consists of the waiting time for filling up one coding stripe, preprocessing time of key deduplication and popularity sorting, coding time and writing time for Memcached. We observe that waiting time for batch coding takes up a small part of the overall latency because of high request throughput. As the value size increases, BCStore consumes more coding time and writing time. Waiting time for batching requests also increases because throughput decreases with the increase of value size.

VI. Related Work

A. Replication

Replication is a traditional approach to achieve data availability in many systems [24–28]. Primary-backup approach [25, 29] replicates data across multiple servers, with one server designated as the primary node, and the rest as backups. Clients only send requests to the primary node. Chain replication [26] can be viewed as an instance of the primary-backup approach, which can achieve high throughput and availability without sacrificing strong consistency. Replication state machine [27, 28, 30] is a widely-used protocol to guarantee the consistent operations across multiple replications. RAMCloud [31] stores redundant copies on disk or flash and leverages large-scale back-end cluster to achieve fast data recovery.

Compared with replication approaches, BCStore leverages primary-backup replication to provide availability of metadata and keys, while using erasure coding for values to achieve space efficiency. Besides, BCStore does not require large-scale cluster for fast data recovery. BCStore can recover data quickly from distributed memory and provides high data availability of in-memory KV-store.

B. Erasure Coding

Erasure coding is widely used in storage systems in both academic work and industry to achieve data availability and space efficiency [10, 11, 32–35]. Local Reconstruction Code [10, 11] adds local parities to reduce the number of coding blocks needed to read during recovery, while keeping low storage overhead. EC-Cache [36] leverages erasure coding to achieve a load-balanced and low latency cluster cache for data-intensive workload. Piggybacking code [32] adds new functions of one byte-level stripe onto the parities, which reduces the amount of data required during recovery. Lazy recovery [34] decreases erasure coding recovery rate to reduce the required network bandwidth. Cocytus [7] applies erasure coding in the in-memory KV-store to achieve fast data recovery and space efficiency.

Different from previous erasure coding applications, BCStore addresses the bandwidth efficiency for applying erasure code in the in-memory KV-store, especially for write-intensive workload. We design batch coding to reduce bandwidth cost, and propose a heuristic garbage collection algorithm to improve memory efficiency.

C. KV-stores

There have been a lot of work on application and optimization of KV-store. DeCandia et al. builds Dynamo [37], a highly available KV-store through data replication with weak consistency. LinkedIn develops Voldemort [5] as distributed KV-store system. Lakshman et al. develops Cassandra [38], a schema-based distributed key-value store. Large-scale in-memory KV-Stores like Memcached [1] and Redis [2] have been widely used in Facebook [3], Twitter [4] as data cache. MICA [39] optimizes parallel data access and network stack of request handling to achieve high throughput for a wide range of workloads. Silt [40] designs three basic key-value stores with different emphasis on memory-efficiency and write-friendliness. Memc3 [41] presents a set of workload inspired algorithms such as optimized cuckoo hashing and optimistic locking to improve Memcached performance. Other research works have proposed using new hardware such as RDMA [42–45], high speed NICs [39], and FPGAs [46] to optimize key-value store. Some other in-memory KV-store databases [47–50] aim to speed up transaction performance.

BCStore focuses on high data availability of in-memory KV-store and bandwidth efficiency of applying erasure code for online services. Our work is complementary with previous KV-store optimizations and can be applied to these KV-Stores to provide high availability and efficiency.

VII. Conclusion and Future Work

Efficiency and availability are two crucial features for in-memory KV-Stores. In this paper, we build BCStore, which applies erasure coding for data availability and space efficiency, and design batch coding mechanism to achieve high bandwidth efficiency for write workload. Besides, we propose a heuristic garbage collection algorithm to improve memory efficiency. We theoretically analyze the bandwidth cost and latency cost of batch coding. At last, through

evaluating different workloads with different system configurations, BCStore can achieve high bandwidth efficiency and memory efficiency with little latency overhead compared to erasure coding with in-place update.

In future work, we first plan to study new replication and erasure coding schemes to optimize performance of BCStore. Then we want to explore advanced hardware like RDMA and NVRAM to extend our work. At last, we consider to handle transactions on BCStore to support in-memory database.

VIII. Acknowledgement

We would like to thank our shepherd Swaminathan Sundararaman for his guidance, and MSST reviewers for their valuable feedback. This work is supported by State Key Program of National Natural Science Foundation of China under Grant No. 61232004, NSFC under Grant No. 61472009, and Shenzhen Key Fundamental Research Projects under Grant No. JCYJ20151014093505032.

References

[1] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[2] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Magazine*, vol. 79, 2009.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[4] M. Rajashekhar and Y. Yue, "Twemcache: Twitter memcached," 2012.

[5] "Project Voldemort," http://www.project-voldemort.com/voldemort/, 2016.

[6] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. Wiener, "Fast database restarts at facebook," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 541–549.

[7] H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory kv-store with hybrid erasure coding and replication," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 167–180.

[8] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 2005, pp. 336–345.

[9] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan, "Parity logging with reserved space: towards efficient updates and recovery in erasure-coded clustered storage." in *FAST*, 2014, pp. 163–176.

[10] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 15–26.

[11] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *Proceedings of the VLDB Endowment*, vol. 6, no. 5. VLDB Endowment, 2013, pp. 325–336.

[12] S. Gokhale, N. Agrawal, S. Noonan, and C. Ungureanu, "Kvzone and the search for a write-optimized key-value store." in *HotStorage*, 2010.

[13] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 217–228.

[14] H. Amur, D. G. Andersen, M. Kaminsky, and K. Schwan, "Design of a write-optimized data store," 2013.

[15] "Improving cdn capacity utilization with peak load pricing," https://www.citrix.com/content/dam/citrix/en_us/documents/products-solutions/improving-cdn-capacity-utilization-with-peak-load-pricing.pdf, 2016.

[16] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end performance isolation through virtual datacenters." in *OSDI*, 2014, pp. 233–248.

[17] "Robust Distributed System Nucleus (rDSN)," https://github.com/Microsoft/rDSN, 2016.

[18] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," in *ACM SIGPLAN Notices*, vol. 48, no. 11. ACM, 2013, pp. 119–130.

[19] L. Egghe, "Zipfian and lotkaian continuous concentration theory," *Journal of the American Society for Information Science and Technology*, vol. 56, no. 9, pp. 935–945, 2005.

[20] "libMemcached," http://libmemcached.org/, 2016.

[21] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2," 2008.

[22] J. S. Plank, K. Greenan, E. Miller, and W. Houston, "Gf-complete: A comprehensive open source library for galois field arithmetic," *Technical Report UT-C S-13–716, University of Tennessee*, 2013.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[24] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 80–107, 1996.

[25] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[26] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability." in *OSDI*, vol. 4, 2004, pp. 91–104.

[27] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[28] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Symposium on Networked Systems Design and Implementation (NSDI)*, 2011, pp. 141–154.

[29] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976, pp. 562–570.

[30] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[31] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 29–41.

[32] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

[33] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran, "Distributed storage codes with repair-by-transfer and nonachievability of interior points on the storage-bandwidth tradeoff," *IEEE Transactions on Information Theory*, vol. 58, no. 3, pp. 1837–1852, 2012.

[34] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin, "Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage," in *Proceedings of International Conference on Systems and Storage*. ACM, 2014, pp. 1–7.

[35] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebooks warm blob storage system," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 383–398.

[36] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ram-

chandran, "Ec-cache: load-balanced, low-latency cluster caching with online erasure coding," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016, pp. 401–417.
[37] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.
[38] "Apache cassandra," http://cassandra.apache.org/, 2016.
[39] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," *management*, vol. 15, no. 32, p. 36, 2014.
[40] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 1–13.
[41] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing." in *NSDI*, vol. 13, 2013, pp. 385–398.
[42] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 295–306.
[43] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store." in *USENIX Annual Technical Conference*, 2013, pp. 103–114.
[44] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached." in *USENIX Annual Technical Conference*, 2012, pp. 347–353.
[45] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 87–104.
[46] M. Blott, K. Karras, L. Liu, K. A. Vissers, J. Bär, and Z. István, "Achieving 10gbps line-rate key-value stores with fpgas." in *HotCloud*, 2013.
[47] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The sap hana database–an architecture overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.
[48] T. Lahiri, M.-A. Neimat, and S. Folkman, "Oracle timesten: An in-memory database for enterprise applications." *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 6–13, 2013.
[49] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 1243–1254.
[50] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 18–32.
[51] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014, p. 8.

IX. Appendix

In the Appendix, we analyze latency cost in batch coding and bandwidth cost in GC in theory. Table I shows the variables and corresponding meanings used in the following analysis.

A. Latency Cost

Here we compute the latency cost of waiting one coding stripe using probabilistic model. We assume that all the requests are update requests and are distributed to different data nodes uniformly. We use exponential distribution to model the possibility of request arrival in a period of time. The probability distribution function of exponential distribution $P(t)$ can be represented as follows:

variable	meaning
N_r	the number of requests
k	the number of data nodes
m	the number of parity nodes
N_m	the number of moved blocks
N_g	the number of GC stripes
S_v	object size
T	request throughput

Table I
VARIABLE TABLE

$$P(t) = 1 - e^{-\lambda * t} \tag{4}$$

$P(t)$ represents the possibility that one request arrives in time t on one data node. λ represents the average arrival number of requests per unit time. Because overall request throughput is T and requests are distributed among data nodes uniformly, λ for each data node can be represented as $\frac{T}{k}$.

The formula can be expanded as:

$$P(t) = 1 - e^{-\frac{T}{k} * t} \tag{5}$$

We use $G(t)$ to represent the possibility that all the data nodes receive one request individually in time t, which denotes the possibility for waiting one coding stripe. Because the request arrivals are independent among data nodes, $G(t)$ can be represented as:

$$G(t) = (1 - e^{-\frac{T}{k} * t})^k \tag{6}$$

Then we obtain the probability density function $g(t)$ through the derivation of $G(t)$:

$$g(t) = (T * (1 - 1/e^{\frac{T*t}{k}})^{(k-1)})/(e^{\frac{T*t}{k}}) \tag{7}$$

The waiting time for fill up one coding stripe can be represented by the expectation of $g(t)$, that is $E(t)$:

$$E(t) = \int_0^\infty t * (T * (1 - 1/e^{\frac{T*t}{k}})^{(k-1)})/(e^{\frac{T*t}{k}})\, dt \tag{8}$$

The latency model can be extended to other workload patterns (*e.g.* Zipfian distribution). We set different λ for k data nodes. Then the possibility of receiving one request on each data node in time t can be represented as:

$$G'(t) = \prod_{i=1}^{k} (1 - e^{-\lambda_i * t}) \tag{9}$$

Then through deducing the probability density function, we can obtain the expectation of the waiting time for batching a coding stripe under other workload patterns.

B. Bandwidth Cost

Then, we compare the overall bandwidth cost of batch coding and in-place update in theory.

First, we analyze the bandwidth cost of in-place update. We assume that all the requests are write requests. For each write request, data object is sent to data node and the delta is sent to all parity nodes. Delta size is equal to object size. Thus, the overall bandwidth cost of in-place update B_{in} can be represented as follows:

$$B_{in} = N_r * S_v + m * N_r * S_v \tag{10}$$

With batch coding mechanism, every k requests send m parity block updates. Thus, the traffic to parity nodes is proportional to the ratio of the number of parity nodes and the number of data nodes. The bandwidth cost of batch coding B_{bc} can be computed as follows:

$$B_{bc} = N_r * S_v + \frac{m}{k} * N_r * S_v \tag{11}$$

The overall bandwidth cost needs to add GC bandwidth. In GC, each data move triggers a delta broadcast to all the parity nodes. So the overall GC bandwidth B_{gc} can be represented by

$$B_{gc} = m * N_m * S_v \tag{12}$$

Here we analyze the upper bound and lower bound of GC bandwidth. The best case is that the number of moved blocks is 0. For example, all the write requests insert new keys, or all the update requests concentrate on hot stripes so that there are no valid data blocks in the GC stripes. Thus, the lower bound of GC bandwidth cost is 0.

The worst case is that all the requests are update requests and each GC stripe has one invalid data block as shown in Figure 4(c), so the number of moved blocks is the product of $k-1$ and N_g. We assume all the write requests are distributed on data processes uniformly. Thus, the number of GC stripes can be represented as N_r/k. GC bandwidth cost of the worst case B_{gc_w} can be deduced as follows:

$$B_{gc\_w} = \frac{m * (k-1)}{k} * N_r * S_v \tag{13}$$

Through adding B_{bc} and B_{gc_w}, we can obtain the overall bandwidth cost of batch coding of the worst case, which is the same as the bandwidth cost of in-place update,

$$B_{bc\_all\_w} = N_r * S_v + m * N_r * S_v \tag{14}$$

Based on the above analysis, the worst case of overall bandwidth cost of batch coding can not exceed in-place update.

For other unbalanced workload patterns (*e.g.* Zipfian), coding stripes may not be filled up, which causes generating extra parity blocks compared with uniform workload pattern. Under these workload patterns, the worst case of overall bandwidth cost of batch coding may greater than in-place update. In this case, we can leverage load balancing techniques [51] (*e.g.* consistent hashing, hot-content replication) to migrate load imbalance among data nodes.