Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication

HAIBO CHEN, HENG ZHANG, MINGKAI DONG, ZHAOGUO WANG, YUBIN XIA, HAIBING GUAN, and BINYU ZANG, Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

In-memory key/value store (KV-store) is a key building block for many systems like databases and large websites. Two key requirements for such systems are efficiency and availability, which demand a KV-store to continuously handle millions of requests per second. A common approach to availability is using replication, such as primary-backup (PBR), which, however, requires $M + 1$ times memory to tolerate M failures. This renders scarce memory unable to handle useful user jobs.

This article makes the first case of building highly available in-memory KV-store by integrating erasure coding to achieve memory efficiency, while not notably degrading performance. A main challenge is that an in-memory KV-store has much scattered metadata. A single KV *put* may cause excessive coding operations and parity updates due to excessive small updates to metadata. Our approach, namely Cocytus, addresses this challenge by using a hybrid scheme that leverages PBR for small-sized and scattered data (e.g., metadata and key), while only applying erasure coding to relatively large data (e.g., value). To mitigate well-known issues like lengthy recovery of erasure coding, Cocytus uses an online recovery scheme by leveraging the replicated metadata information to continuously serve KV requests. To further demonstrate the usefulness of Cocytus, we have built a transaction layer by using Cocytus as a fast and reliable storage layer to store database records and transaction logs. We have integrated the design of Cocytus to Memcached and extend it to support in-memory transactions. Evaluation using YCSB with different KV configurations shows that Cocytus incurs low overhead for latency and throughput, can tolerate node failures with fast online recovery, while saving 33% to 46% memory compared to PBR when tolerating two failures. A further evaluation using the SmallBank OLTP benchmark shows that in-memory transactions can run atop Cocytus with high throughput, low latency, and low abort rate and recover fast from consecutive failures.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Hardware → Fault tolerance**; • **Software and its engineering → Distributed memory**;

Additional Key Words and Phrases: Primary-backup replication, erasure coding, KV-store, transactions

This work is supported in part by National Key Research & Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61572314, 61525204 and 61672345), the Top-notch Youth Talents Program of China, Shanghai Science and Technology Development Fund (No. 14511100902), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012). This paper is an extended version of a a paper appeared at 2016 USENIX Conference on File and Storage Technologies (Zhang et al. 2016). The source code of Cocytus is available via http://ipads.se.sjtu.edu.cn/pub/projects/cocytus.
Authors' addresses: H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang (corresponding author), Room 3402, Software Building, 800 Dongchuan Rd., Shanghai China, 200240; emails: {haibochen, shinedark}@sjtu.edu.cn, {mingkaidong, tigerwang1986}@gmail.com, {xiayubin, hbguan, byzang}@sjtu.edu.cn.

ACM 1553-3077/2017/09-ART25 $15.00
https://doi.org/10.1145/3129900

ACM Reference format:
Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. 2017. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Trans. Storage* 13, 3, Article 25 (September 2017), 30 pages.
https://doi.org/10.1145/3129900

1 INTRODUCTION

The increasing performance demand of large-scale Web applications has stimulated the paradigm of placing large datasets within memory to satisfy millions of operations per second with sub-millisecond latency. This new computing model, namely in-memory computing, has been emerging recently. For example, large-scale in-memory key/value systems like Memcached (Fitzpatrick 2004) and Redis (Zawodny 2009) have been widely used in Facebook (Nishtala et al. 2013), Twitter (Twitter Inc. 2012), and LinkedIn. There have also been considerable interests of applying in-memory databases (IMDBs) to performance-hungry scenarios (e.g., SAP HANA (Färber et al. 2012), Oracle TimesTen (Lahiri et al. 2013), and Microsoft Hekaton (Diaconu et al. 2013)).

Even if many systems have a persistent backing store to preserve data durability after a crash, it is still important to retain data in memory for instantaneously taking over the job of a failed node, as rebuilding terabytes of data into memory is time-consuming. For example, it was reported that recovering around 120GB data from disk to memory for an in-memory database in Facebook took 2.5–3h (Goel et al. 2014). Traditional ways of providing high availability are through replication, such as standard primary-backup (PBR) (Budhiraja et al. 1993) and chain-replication (van Renesse and Schneider 2004), by which a dataset is replicated $M + 1$ times to tolerate M failures. However, this also means dedicating M copies of CPU/memory without producing user work, requiring more standby machines, and thus multiplying energy consumption.

This article describes Cocytus, an efficient, available, and strongly consistent in-memory replication scheme. Cocytus aims at reducing the memory consumption for replicas while keeping similar performance and availability of PBR-like solutions, though at additional CPU cost for update-intensive workloads and more network bandwidth during recovery. The key of Cocytus is efficiently combining the space-efficient erasure coding scheme with PBR.

Erasure coding is a space-efficient solution for data replication, which is widely applied in distributed storage systems, including Windows Azure Store (Huang et al. 2012) and Facebook storage (Muralidhar et al. 2014). However, though space-efficient, erasure coding is well-known for its lengthy recovery and transient data unavailability (Huang et al. 2012; Silberstein et al. 2014).

In this article, we investigate the feasibility of applying erasure coding to in-memory key/value stores (KV-stores). Our main observation is that the abundant and speedy CPU cores make it possible to perform coding online. For example, a single Intel Xeon E3-1230v3 CPU core can encode data at 5.26GB/s for Reed-Solomon(3,5) codes, which is faster than even current high-end NIC with 40Gb/s bandwidth. However, the block-oriented nature of erasure coding and the unique feature of KV-stores raise several challenges to Cocytus to meet the goals of efficiency and availability.

The first challenge is that the scattered metadata like a hashtable and the memory allocation information of a KV-store will incur a large number of coding operations and updates even for a single KV put. This incurs not only much CPU overhead but also high network traffic. Cocytus addresses this issue by leveraging the idea of separating metadata from data (Wang et al. 2012) and uses a hybrid replication scheme. In particular, Cocytus uses erasure coding for large-sized values while using PBR for small-sized metadata and keys.

The second challenge is how to consistently recover the lost data blocks online with the distributed data blocks and parity blocks.[1] This is especially import for a high-performance in-memory store since a small duration of blocking would impact a huge amount of services. While state-of-the-art protocols for erasure coding block service to some extent during recovery (Huang et al. 2012; Silberstein et al. 2014; Rashmi et al. 2013; Sathiamoorthy et al. 2013; Muralidhar et al. 2014), Cocytus introduces a distributed online recovery protocol that consistently collects all data blocks and parity blocks to recover the lost data, yet without blocking services on live data blocks and with predictable memory consumption.

We have implemented Cocytus in Memcached 1.4.21 with the synchronous model, in which a server sends responses to clients after receiving the acknowledgments from backup nodes to avoid data loss. We also implemented a pure primary-backup replication in Memcached 1.4.21 for comparison.

To further demonstrate the usefulness of Cocytus, we build a transaction layer atop Cocytus by leveraging Cocytus as a fast and reliable storage layer for database records as well as transaction logs. The transaction layer runs on each machine with a transaction coordination service, which coordinates together with a transaction master using the two-phase commit to commit a transaction. Upon a crash, the transaction layer leverages the logs replicated by Cocytus to reliably recover running transactions.

With evaluation using YCSB (Cooper et al. 2010) to issue requests with different key/value distributions, we show that Cocytus incurs little degradation on throughput and latency during normal processing and can gracefully recover data quickly. Overall, Cocytus has high memory efficiency while incurring small overhead compared with PBR, yet at little CPU cost for read-mostly workloads and modest CPU cost for update-intensive workloads. Evaluation using the SmallBank (Alomari et al. 2008) application from the OLTP-Bench set shows that transactions running atop Cocytus achieves high throughput, low latency and low abort rate, yet can be reliably recovered within a short time.

In summary, the main contribution of this article includes:

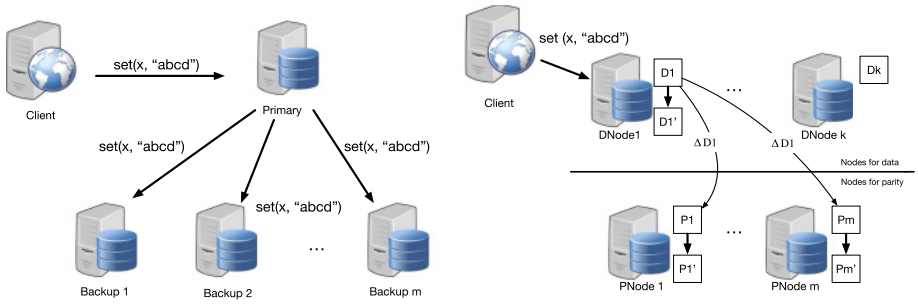
- —The first case of exploiting erasure coding for in-memory KV-store.
- —Two key designs, including a hybrid replication scheme and distributed online recovery that achieve efficiency, availability and consistency.
- —An implementation of Cocytus on Memcached (Fitzpatrick 2004) as well as a showcase of a transaction layer atop Cocytus.
- —A thorough evaluation that confirms Cocytus's efficiency and availability.

The rest of this article is organized as follows. The next section describes necessary background information about primary-backup replication and erasure coding in a modern computing environment. Section 3 describes the design of Cocytus, followed up by the recovery process in Section 4. Section 5 presents a transaction layer built atop Cocytus. Section 6 describes the implementation details. Section 7 presents the experimental data of Cocytus. Finally, Section 8 discusses related work, and Section 9 concludes this paper.

2 BACKGROUND AND CHALLENGES

This section first briefly reviews PBR and erasure coding, and then identifies opportunities and challenges of applying erasure coding to in-memory KV-stores.

[1]Both data blocks and parity blocks are called code words in the coding theory. We term "parity blocks" as those code words generated from the original data and "data blocks" as the original data.



(a) Key/value store with primary-backup replication

(b) Key/value store with erasure coding

Fig. 1. Data storage with two different replication schemes.

2.1 Background

Primary-backup replication: PBR (Bressoud and Schneider 1996) is a widely used approach to providing high availability. As shown in Figure 1(a), each primary node has M backup nodes to store its data replicas to tolerate M failures. One of the backup nodes would act as the new primary node if the primary node failed, resulting in a *view change* (e.g., using Paxos (Lamport 2001)). As a result, the system can still provide continuous services upon node failures. This, however, is at the cost of high data redundancy, for example, M additional storage nodes and the corresponding CPUs to tolerate M failures. For example, to tolerate two node failures, the storage efficiency of a KV-store can only reach 33%.

Erasure coding: Erasure coding is an efficient way to provide data durability. As shown in Figure 1(b), with erasure coding, an N-node cluster can use K of N nodes for data and M nodes for parity ($K + M = N$). A commonly used coding scheme is Reed-Solomon codes (RS-code) (Reed and Solomon 1960), which computes parities according to its data over a finite field by the following formula (the matrix is called a *Vandermonde matrix*):

$$\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_M \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & a_1^1 & \cdots & a_1^{K-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & a_{M-1}^1 & \cdots & a_{M-1}^{K-1} \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_K \end{bmatrix}. \tag{1}$$

An update on a DNode (a node for data) can be achieved by broadcasting its delta to all PNodes (nodes for parity) and asking them to add the delta to parity with a predefined coefficient. This approach works similarly for updating any parity blocks; its correctness can be proven by the following equation, where A represents the *Vandermonde matrix* mentioned in Equation (1).

$$\begin{bmatrix} P_1' \\ \vdots \\ P_i' \\ \vdots \\ P_M' \end{bmatrix} = A * \begin{bmatrix} D_1 \\ \vdots \\ D_i' \\ \vdots \\ D_K \end{bmatrix} = A * \begin{bmatrix} D_1 \\ \vdots \\ D_i + \Delta D_i \\ \vdots \\ D_K \end{bmatrix} = \begin{bmatrix} P_1 \\ \vdots \\ P_i \\ \vdots \\ P_M \end{bmatrix} + \begin{bmatrix} 1 \\ \vdots \\ a_1^{i-1} \\ \vdots \\ a_{M-1}^{i-1} \end{bmatrix} * \Delta D_i.$$

In the example above, we denote the corresponding RS-code scheme as RS(K,N). Upon node failures, any K nodes of the cluster can recover data or parity lost in the failed nodes, and thus

Table 1. The Speed of Coding Data with Different Schemes for a Five-Node Cluster

Scheme	Encoding speed	Decoding speed
RS(4,5)	5.52GB/s	5.20GB/s
RS(3,5)	5.26GB/s	4.83GB/s
RS(2,5)	4.56GB/s	4.24GB/s

RS(K,N) can handle M node failures at most. During recovery, the system recalculates the lost data or parity by solving the equations generated by the above equation.

As only M of N nodes are used for storing parities, the memory efficiency can reach K/N. For example, an RS(3,5) coding scheme has storage efficiency of 60% while tolerating up to two node failures.

Unlike PBR, where the backup data can be used directly, erasure coding necessities after-failure recovery, during which consistent data and parities are collected from available nodes to calculate the lost data correctly.

2.2 Motivation: Opportunities and Challenges

The emergence of in-memory computing significantly boosts the performance of many systems. However, this also means that a large amount of data needs to be placed in memory. As memory is currently volatile, a node failure would cause data loss for a large chunk of memory. Even if the data has its backup in persistent storage, it would require non-trivial time to recover the data for a single node (Goel et al. 2014).

However, simply using PBR may cause significant memory inefficiency. Despite an increase of the volume, memory is still a scarce resource, especially when processing the "big-data" applications. It was frequently reported that memory bloat either significantly degraded the performance or simply caused server crashes (Bu et al. 2013). This is especially true for workload-sharing clusters, where the budget for storing specific application data is not large.

Opportunities: The need for both availability and memory efficiency makes erasure coding a new attractive design point. The increase of CPU speed and the CPU core counts make erasure coding suitable to be used even in the critical path of data processing. Table 1 presents the encoding and decoding speed for different Reed-Solomon coding scheme on a five-node cluster with an average CPU core (2.3GHz Xeon E5, detailed configurations in Section 7.1). Both encoding and decoding can be done at 4.24–5.52GB/s, which is several hundreds of times compared to 20 years ago (e.g., 10MB/s (Rizzo 1997)). This means that an average-speed core is enough to handle data transmitted through even a network link with 40Gb/s. This reveals a new opportunity to trade CPU resources for better memory efficiency to provide high availability.

Challenges: However, trivially applying erasure coding to in-memory KV-stores may result in significant performance degradation and consistency issues.

The first challenge is that coding is done efficiently only in a bulk-oriented manner. However, an update operation in a KV-store may result in a number of small updates, which would introduce a notable amount of coding operations and network traffics. For example, in Memcached, both the hashtable and the allocation metadata need to be modified for a *set* operation. For the first case, a KV pair being inserted into a bucket will change four pointers of the double linked list. Some statistics like those for LRU replacement algorithm need to be changed as well. In the case of a hashtable expansion or shrinking, all key/value pairs may need to be relocated, causing a huge amount of memory updates. For the allocation metadata, as Memcached uses a slab allocator,

an allocation operation usually changes four variables and a free operation changes six to seven variables.

The second challenge is that a data update involves changes to multiple parity blocks across machines. During data recovery, there are also multiple data blocks and parity blocks involved. If there are concurrent updates in progress, then this may easily lead to inconsistent data recovery. A simple solution is freezing all related blocks until the completion of recovery, which ensures the correctness of recovered data but hurts the performance during recovery. The performance impact is even grater in a distributed system where network communications are involved during recovery.

3 DESIGN

3.1 Interface and Assumption

Cocytus is an in-memory replication scheme for key/value stores (KV-stores) to provide high memory efficiency and high availability with low overhead. It assumes that a KV-store has two basic operations: $Value \leftarrow get(Key)$ and $set(Key, Value)$, where Key and Value are arbitrary strings. According to prior large-scale analysis on key/value stores in commercial workloads (Atikoglu et al. 2012; Nishtala et al. 2013), Cocytus assumes that the value size is much larger than the key size.

Cocytus handles only omission node failures where a node is fail-stop and won't taint other nodes; commission or Byzantine failures are not considered. It also does not consider a complete power outage that crashes the entire cluster. In such cases, it assumes that there is another storage layer that constantly stores data to preserve durability (Nishtala et al. 2013). Alternatively, one may leverage battery-backed RAM like NVDIMM (Technology 2014; SNIA 2015) to preserve durability.

Cocytus is designed to be synchronous, that is, a response of a *set* request returned to the client guarantees that the data has been replicated/coded and can survive node failures.

Cocytus works efficiently for read-mostly workloads, which are typical for many commercial KV-stores (Atikoglu et al. 2012). For update-intensive workloads, Cocytus would use more CPU resources due to the additional calculations caused by the erasure coding to achieve similar latency and throughput compared to a simple primary-backup replication.

3.2 Architecture

Cocytus separates data from metadata and leverages a hybrid scheme: metadata and keys are replicated using primary-backup while values are erasure coded.

One basic component of Cocytus is the coding group, as shown in Figure 2. Each group comprises K data processes handling requests to data blocks and M parity processes receiving update requests from the data processes. A *get* operation only involves one data node, while a *set* operation updates metadata in both primary and its backup node, and generates diffs to be patched to the parity codes.

Cocytus uses sharding to partition key/value tuples into different groups. A coding group handles a key *shard*, which is further divided into P *partitions* in the group. Each partition is handled by a particular data process, which performs coding at the level of virtual address spaces. This makes the coding operation neutral to the changes of value sizes of a KV pair as long as the address space of a data process does not change. There is no data communication among the data processes, which ensures fault isolation among data processes. When a data process crashes, one parity process immediately handles the requests for the partition that belongs to crashed nodes and recovers the lost data, while other data processes continuously provide services without disruption.

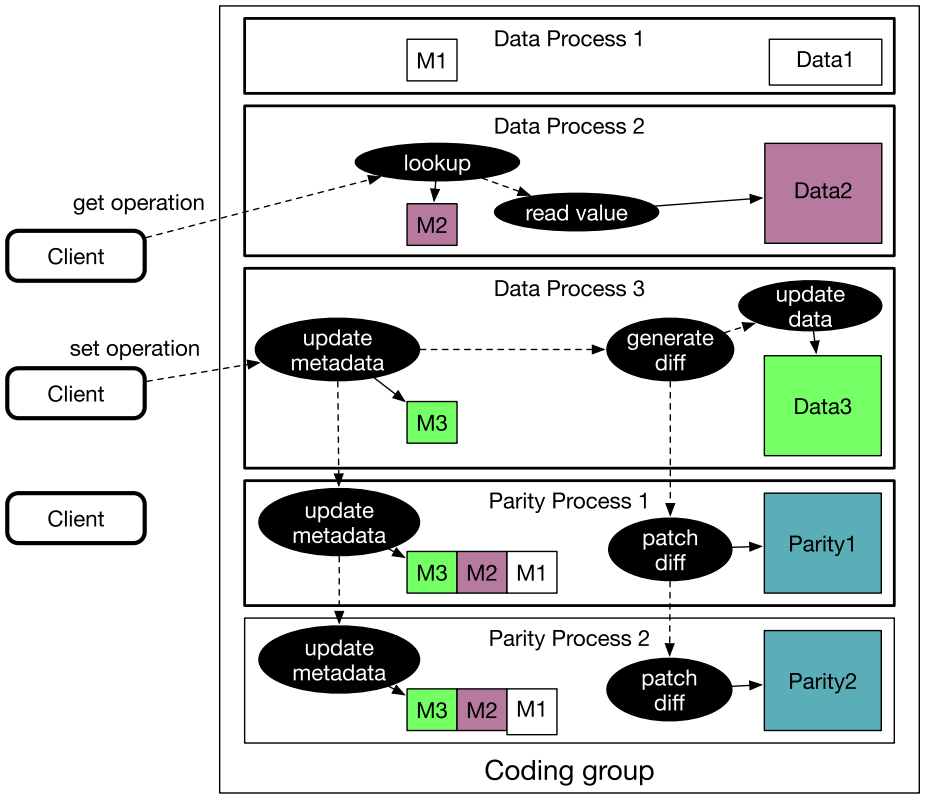


Fig. 2. Requests handled by an coding group in Cocytus, where $K = 3$, $M = 2$.

Cocytus is designed to be strongly consistent, which never loses data or recovers inconsistent data. However, strict ordering on parity processes is not necessary for Cocytus. For example, two data processes update their memory at the same time, which involves two updates on the parity processes. However, the parity processes can execute the updates in any order as long as they are notified that the updates have been received by all of the parity processes. Thus, in spite of the update ordering, the data recovered later are guaranteed to be consistent. Section 4.1.2 will show how Cocytus achieves consistent recovery when a failure occurs.

3.3 Separating Metadata from Data

For a typical KV-store, there are two types of important metadata to handle requests. The first is the mapping information, such as a (distributed) hashtable that maps keys to their value addresses. The second one is the allocation information. As discussed before, if the metadata is erasure coded, there will be a large number of small updates and lengthy unavailable duration upon crashes.

Cocytus uses primary-backup replication to handle the mapping information. In particular, the parity processes save the metadata for all data processes in the same coding group. For the allocation information, Cocytus applies a slab-based allocation for metadata allocation. It further relies on an additional deterministic allocator for data such that each data process will result in the same memory layout for values after every operation.

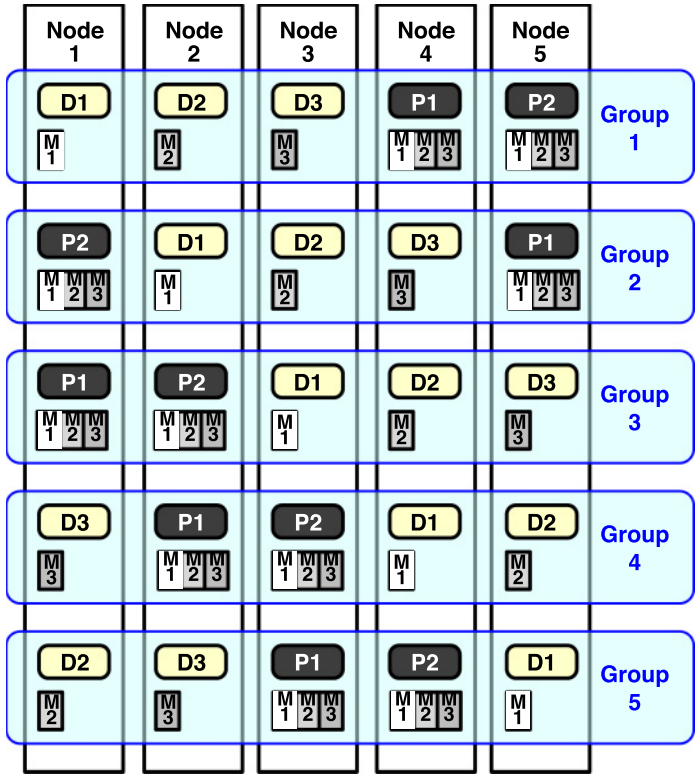


Fig. 3. Interleaved layout of coding groups in Cocytus. The blocks in the same row belong to one coding group.

Interleaved layout: One issue caused by this design is that parity processes save more metadata than those in the data processes, which may cause memory imbalance. Further, as parity processes only need to participate in *set* operations, they may become idle for read-mostly workloads. In contrast, for read-write workloads, the parity processes may become busy and may become a bottleneck of the KV-store.

To address these issues, Cocytus interleaves coding groups in a cluster to balance workload and memory on each node, as shown in Figure 3. Each node in Cocytus runs both parity processes and data processes; a node will be busy on parity processes or data processes for update-intensive or read-mostly workload accordingly.

The interleaved layout can also benefit the recovery process by exploiting the cluster resources instead of one node. Because the shards on one node belong to different groups, a single node failure leads a process failure on each group. However, the first parity nodes of these groups are distributed across the cluster, all nodes will work together to do recovery.

To extend Cocytus in a large scale cluster, there are three dimensions to consider, including the number of data processes (K) and the number of parity processes (M) in a coding group, as well as the number of coding groups. A larger K increases memory efficiency but makes the parity process suffer from higher CPU pressure for read-write workloads. A larger M leads to more failures to be tolerated but decreases memory efficiency and degrades the performance of *set* operations. A neutral way to extend Cocytus is deploying more coding groups.

3.4 Consistent Parity Updating with Piggybacking

Because an erasure-coding group has multiple parity processes, sending the update messages to such processes needs an *atomic broadcast*. Otherwise, a KV-store may result in inconsistency. For example, when a data process has received a *set* request and is sending updates to two parity processes, a failure occurs and only one parity process has received the update message. The following recovery might recover incorrect data due to the inconsistency between parities.

A natural solution to this problem is using two-phase commit (2PC) to implement atomic broadcast. This, however, requires two rounds of messages and doubles the I/O operations for *set* requests. Cocytus addresses this problem with a piggybacking approach. Each request is assigned with an *xid*, which monotonously increases at each data process like a logical clock. Upon receiving parity updates, a parity process first records the operation in a buffer corresponding with the *xid* and then immediately send acknowledgments to its data process. After the data process receives acknowledgments from all parity processes, the operation is considered stable in the KV-store. The data process then updates the *latest stable xid* as well as data and metadata, and sends a response to the client. When the data process sends the next parity update, this request piggybacks on the *latest stable xid*. When receiving a piggybacked request, the parity processes mark all operations that have smaller *xid* in the corresponding buffer as *READY* and install the updates in place sequentially. Once a failure occurs, the corresponding requests that are not received by all parity processes will be discarded.

3.5 Request Handling

After introducing the basic design of Cocytus, this section describes the process of request handling in detail.

When a request comes, a data process first checks whether the key belongs to the shard it manages. If the request is wrongly dispatched, then the data process sends a response of *WRONG-SHARD*. After passing the validation, for a *get* request, the data process finds the key/value pair in the hashtable. If found, then the value is sent back to the client. Otherwise, the data process sends a *NOTFOUND* back to the client.

For a *set* request, the data process tries to find the key/value pair in the hashtable. If the key does not exist in the hashtable, then the data process allocates a space to save the key/value pair in the hashtable. The data process also allocates a free space for the data region, which is previously allocated during system initialization. If the key has been mapped to an old value, then the data process frees the old value but does not reset the memory, because resetting memory causes parity updates. After finding the space, the data process generates the diffs between the value and the old data in that place. The data process also appends an automatically incremental ID, called *xid* for the request. Then, the data process sends the allocated address and the request to the parity process, in which the value is replaced with the diffs between the value and the original data on the allocated space. In fact, the allocated address is not necessary thanks to the deterministic allocator. Hence, the network traffic of Cocytus is similar to primary-backup replication.

When a parity process received a request from a data process, it records the request in a buffer and sends a response to the data process. After the data process received all responses from parity processes, it sends a response to the client, updates the memory in place and updates the *latest stable xid*. When the parity process received a request with the piggybacked *xid*, it marks the buffered request with smaller *xid* as *READY* and applies the updates sequentially. The parity processes deal the requests with the same way as the data processes. Because the allocation and hashtable are both deterministic, the states of both types of processes keep the same.

4 RECOVERY

When a node crashes, Cocytus needs to reconstruct lost data online while serving client requests. Cocytus assumes that the KV-store will eventually keep its fault tolerance level by assigning new nodes to host the recovered data. Alternatively, Cocytus can degenerate its fault tolerance level to tolerate fewer failures. In this section, we first describe how Cocytus recovers data in-place to the parity node and then illustrate how Cocytus migrates the data to recover the parity and data processes when a crashed node reboots or a new standby node is added.

4.1 Data Recovery

Because data blocks are only updated at the last step of handling *set* requests which are executed sequentially with *xid*. We can regard the *xid* of the latest completed request as the logical timestamp (T) of the data block. Similarly, there are K logical timestamps ($VT[1..K]$) for a parity block, where K is the number of the data processes in the same coding group. Each of the K logical timestamps is the *xid* of the latest completed request from the corresponding data process.

Suppose data processes 1 to F crash at the same time. Cocytus chooses all alive data blocks and F parity blocks to reconstruct the lost data blocks. Suppose the logical timestamps of data blocks are $T_{F+1}, T_{F+2}, \ldots, T_K$ and the logical timestamps of parity blocks are $VT_1, VT_2, \ldots, VT_F$. If $VT_1 = VT_2 = \cdots = VT_F$ and $VT_1[F+1..K] = \langle T_{F+1}, T_{F+2}, \ldots, T_K \rangle$, then theses data blocks and parity blocks agree with formula (1). Hence, they are consistent.

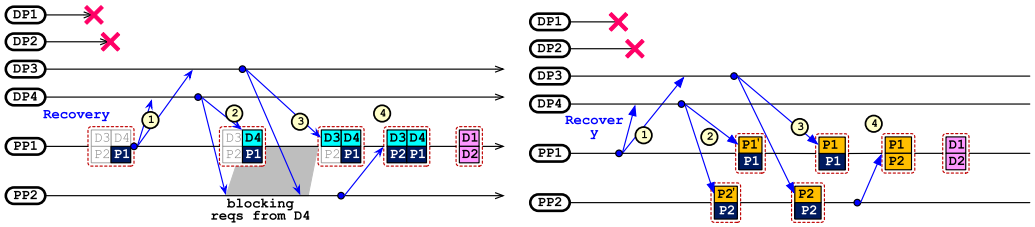
The recovery comprises two phases: preparation and online recovery. During the preparation phase, the parity processes synchronize their request buffers that correspond to the failed processes. Once the preparation phase completes, all parity blocks are consistent on the failed processes. During online recovery, alive data process send their data blocks with its logical timestamp, so the parity processes can easily provide the consistent parity blocks.

4.1.1 Preparation. Once a data process failure is detected, a corresponding parity process is selected as the recovery process to do the recovery and to provide services on behalf of the crashed data process. The recovery process first collects latest *xid*s, which correspond to failed data processes from all parity processes. Hence, a parity process has a latest *xid* for each data process, because it maintains an individual request buffer for each data process. The minimal latest *xid* is then chosen as the stable *xid*. Requests with greater *xid* received by the failed data process haven't been successfully received by all parity processes and thus should be discarded. Then, the stable *xid* is sent to all parity processes. The parity processes apply the update requests in place of which the *xid* equal to or less than the stable *xid* in the corresponding buffer. After that, all parity processes are consistent in the failed data process, because their corresponding logical timestamps are all the same with the stable *xid*.

The preparation phase blocks key/value requests for a very short time. According to our evaluation, the blocking time is only 7 to 13ms even under a high workload.

4.1.2 Online Recovery. The separation of metadata and data enables online recovery of key/value pairs. During recovery, the recovery process can leverage the replicated metadata to reconstruct lost data online to serve client requests, while using idle CPU cycles to proactively reconstruct other data. During the recovery, data blocks are recovered in a granularity of 4KB, which is called a recovery unit. According to the address, each recovery unit is assigned an ID for the convenience of communication among processes.

For each recovery unit, the key issue is to get all corresponding blocks from other data/parity processes in the coding group in a consistent way. For example, suppose Cocytus uses RS(4,6) to encode the data. There are six processes, that is, DP1, DP2, DP3, DP4, PP1, PP2 (DP is data process



(a) a blocking delivery way (b) a non-blocking way

Fig. 4. Semi-blocking vs. non-blocking based recovery.

and PP is parity process), in a coding group, which contains data D1, D2, D3, D4, P1, P2. The relationships between the data are

$$P1 = D1 + D2 + D3 + D4,$$

$$P2 = D1 + 2 * D2 + 4 * D3 + 8 * D4.$$

Once DP1 and DP2 crash, PP1 and PP2 become the recovery processes for DP1 and DP2 accordingly, where PP1 is the recovery leader and PP2 is the recovery worker. When the PP1 wants to reconstruct a block, it needs to collect all corresponding blocks from D3, D4, P2 to calculate D1 and D2 with the following formulas:

$$D1 = 2 * P1 - P2 + 2 * D3 + 6 * D4,$$

$$D2 = -P1 + P2 - 3 * D3 - 7 * D4.$$

To ensure D1 and D2 are reconstructed in a consistent way, it is critical to ensure that all P1, P2, D3, and D4 are collected consistently. In the following, we first describe a naive approach using Chandy-Lamport's distributed snapshot protocol (Chandy and Lamport 1985), which, however, requires a number of memory buffers as well as blocking requests from data processes to the coding group during the decoding process. Then, we present a refined, yet still simple protocol that releases this guarantee. Finally, we use an example to illustrate the whole process of recovery.

Partial-blocking Based Recovery. Figure 4(a) illustrates a simplified version varied from this protocol. The recovery process first broadcasts snapshot requests to all data processes (step 1) and continuously receives the parity update requests from its coding group. The request from PP1 to PP2 is unnecessary, because the parity processes will not send requests that could lead to inconsistent results. Based on Chandy-Lamport's protocol, once a parity process (i.e., PP1) receives the first data block from DP4, it stops serving any requests from DP4 (step 2). When receiving all data blocks, PP1 starts to serve requests from all data processes again (step 3). After receiving P2 from PP2, PP1 can reconstruct D1 and D2 (step 4).

Although this approach prevents freezing all related blocks at the beginning, it still incurs service suspension and consumes a lot of memory buffer: First, if one data process is busy on dealing with *get* requests, the long suspension time may lead to performance degradation on another shard. Second, it would also hold a lot of memory buffers for a long time. For example, if N blocks are recovered in parallel, to buffer the data, the parity process needs memory about $(K - 1) * N * blocksize$ at most, where K is the number of data processes and N is the number of simultaneously recovered blocks.

Non-blocking Based Recovery. Cocytus uses a refined solution to recovery, which requires less memory buffer and never suspends message delivery. The key of our solution is to reconstruct the data with a streaming-based way.

Figure 4(b) shows how the recovery process managing P1 recovers the data block when DP1 and DP2 crash on a RS(4,6) coded Cocytus. Suppose PP1 wants to recover D1 and D2, which are lost. It broadcasts *recovery* requests to DP3 and DP4 (step 1). When a data process receives a *recovery* request, it sends a response with the requested blocks to each involved parity process. When PP1 receives the first data block from DP4 (step 2), it copies its code block to $P1'$. At that moment,

$$P1' = D1 + D2 + D3 + D4.$$

PP1 executes a *subtraction* operation on $P1'$ by D4, then $P1'$ becomes

$$P1' = D1 + D2 + D3.$$

If an update request from D3 process on the same block comes before receiving the data block from D3 process, then the update operation are done on both P1 and $P1'$. After receiving the data block from DP3 (step 3), $P1'$ becomes $P1''$:

$$P1'' = D1 + D2.$$

Since DP1 and DP2 have crashed, so no update operations from their recovery processes on the same block are accepted before the data block is completely recovered. Similarly, PP2 also finally get a code block $P2''$, which are

$$P2'' = D1 + 2 * D2.$$

Then PP2 sends the $P2'$ block to P1 process (step 4). Hence, the PP1 can recover D1 and D2 with fewer data blocks, which is faster and needs less memory buffer.

$$D1 = 2 * P1' - P2',$$

$$D2 = P2' - P1'.$$

In this way, Cocytus requires much fewer memory buffers and is completely non-blocking during the recovery process.

Combining with Vector Timestamps. The following shows how Cocytus combines the above streaming-based recovery with vector timestamps to derive a consistent recovery. As shown in Figure 5, there are five steps in our online recovery protocol:

- —1. To reconstruct a recovery unit, a recovery process becomes the recovery initiator and sends messages consisting of the recovery unit ID and a list of involved recovery processes to alive data processes.
- —2. When the ith data process receives the message, it sends the corresponding data unit to all recovery processes along with its logical timestamp T_i.
- —3(a). When a recovery process receives the data unit and the logical timestamp T_i, it first applies the requests whose *xid* equals to or less than T_i in the corresponding buffer. At this time, the ith logical timestamp on this recovery process equals to T_i.
- —3(b). The recovery process *subtracts* the corresponding parity unit by the received data unit with the predefined coefficient. After the subtraction completes, the parity unit is detached from the ith data process and will not receive updates from that data process.
- —4. When a recovery process has received and handled all data units from the surviving data processes, it sends the final corresponding parity unit to the recovery initiator, which is only associated with the failed data processes.
- —5. When the recovery initiator has received all parity units from recovery processes, it decodes them by solving the following equation, in which the $fn_1, fn_2, \ldots, fn_F$ indicate the numbers of F failure data processes and the $rn_1, rn_2, \ldots, rn_F$ indicate the numbers of F parity processes chosen to be the recovery processes:

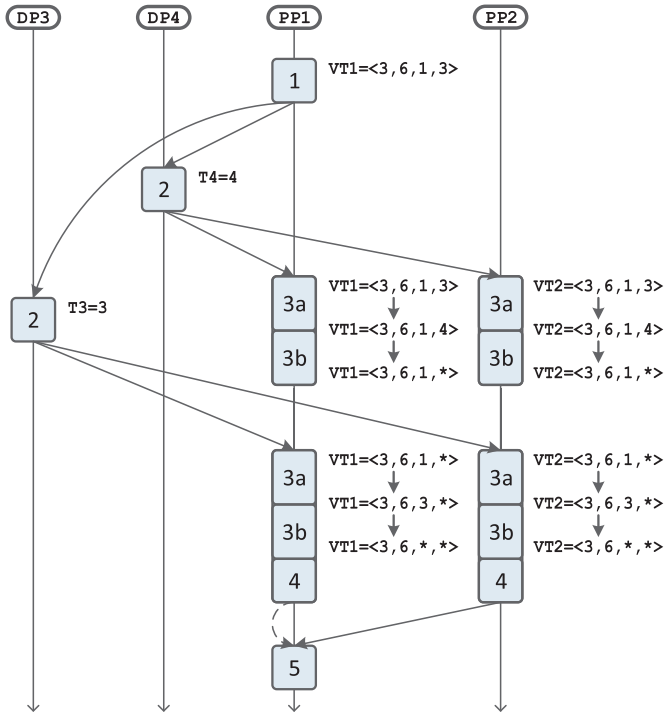


Fig. 5. Online recovery when DP1 and DP2 crash in an RS(4, 6) coding group.

$$\begin{bmatrix} P_{rn_1} \\ P_{rn_2} \\ \vdots \\ P_{rn_F} \end{bmatrix} = \begin{bmatrix} a_{rn_1-1}^{fn_1-1} & a_{rn_1-1}^{fn_2-1} & \cdots & a_{rn_1-1}^{fn_F-1} \\ a_{rn_2-1}^{fn_1-1} & a_{rn_2-1}^{fn_2-1} & \cdots & a_{rn_2-1}^{fn_F-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{rn_F-1}^{fn_1-1} & a_{rn_F-1}^{fn_2-1} & \cdots & a_{rn_F-1}^{fn_F-1} \end{bmatrix} * \begin{bmatrix} D_{fn_1} \\ D_{fn_2} \\ \vdots \\ D_{fn_F} \end{bmatrix}. \quad (2)$$

4.2 Correctness Argument

Here, we briefly argue the correctness of the protocol. Because when a data block is updated, all parity processes should have received the corresponding update requests. Hence, in step 3(a), the parity process must have received all required update requests and can synchronize its corresponding logical timestamp with the received logical timestamp. Since the received data block and parity block have the same logical timestamps, the received data block should be the same as the data block that is used to construct the parity block. Because a parity block is a *sum* of data blocks with the individual predefined coefficients in the *Vandermonde* matrix, after the subtraction in step 3(b), the parity block is only constructed by the rest of data blocks. At the beginning of step 4, the parity block is only constructed by the data blocks of failed data processes, because the parity process has done step 3 for each alive data process. Finally, with the help of stable *xid* synchronization in the preparation phase, the parity blocks received in step 5 are all consistent and should agree with Equation (2).

4.2.1 Request Handling on Recovery Process. Cocytus allows a recovery process to handle requests during recovery. For a *get* request, it tries to find the key/value pair through the backup

hashtable. If it finds the pair, then the recovery process checks whether the data blocks needed for the value have been recovered. If the data blocks have not been recovered, then the recovery process initiates data block recovery for each data block. After the data blocks are recovered, the recovery process sends the response to the client with the requested value.

For a *set* request, the recovery process allocates a new space for the new value with the help of the allocation metadata in the backup. If the allocated data blocks are not recovered, then the recovery process calls the recovery function for them. After recovery, the recovery process handles the operation like a normal data process.

4.3 Data Migration

Data process recovery: During the data process recovery, Cocytus can migrate the data from the recovery process to a new data process. The recovery process first sends the keys as well as the metadata of values (i.e., sizes and addresses) in the hashtable to the new data process. While receiving key/value pairs, the new data process rebuilds the hashtable and the allocation metadata. After all key/value pairs are sent to the new data process, the recovery process stops providing services to clients.

When metadata migration completes, the data (i.e., value) migration starts. At that moment, the data process can handle the requests as done in the recovery process. The only difference between them is that the data process does not recover the data blocks by itself. When data process needs to recover a data block, it sends a request to the recovery process. If the recovery process has already recovered the data block, then it sends the recovered data block to the data process directly. Otherwise, it starts a recovery procedure. After all data blocks are migrated to the data process, the migration completes.

If either the new data process or the corresponding recovery process fails during data migration, then both of them should be killed. This is because having only one of them will lead to insufficient information to provide continuous services. Cocytus can treat this failure as a data process failure.

Parity process recovery: The parity process recovery is straightforward. After a parity process crashes, the data process marks all data blocks with a *miss* bit for that parity process. The data processes first send the metadata to the recovering parity process. Once the transfer of metadata completes, the logical timestamps of new parity processes are the same with the metadata it has received. After the transfer of metadata, the data processes migrate the data that may overlap with *parity update* requests. Before sending a *parity update* request, which involves data blocks marked with a *miss* bit, the data process needs to send the involved data blocks to the new parity process. In this way, data blocks sent to the new parity process have the same logical timestamps with the metadata sent before. After the new parity process receives all data blocks, the recovery completes. If either of the data processes fails during the recovery of the parity process, then the recovery fails and Cocytus starts to recover the failed data process.

5 SUPPORTING HIGH-AVAILABLE TRANSACTIONS

Many in-memory transaction systems (Wang et al. 2014a; Wei et al. 2015; Chen et al. 2016) rely on a key/value store to store database records. To showcase how the scheme of Cocytus can help reduce memory consumption of in-memory transaction systems while preserving high availability, this section presents a design that builds a transaction layer atop the KV-store of Cocytus to form an in-memory transaction system.

5.1 Transaction Layer

As shown in Figure 6, the transaction layer comprises a set of transaction coordination (TC) services running on each machine, each with a unique ID to manage data stored in that machine

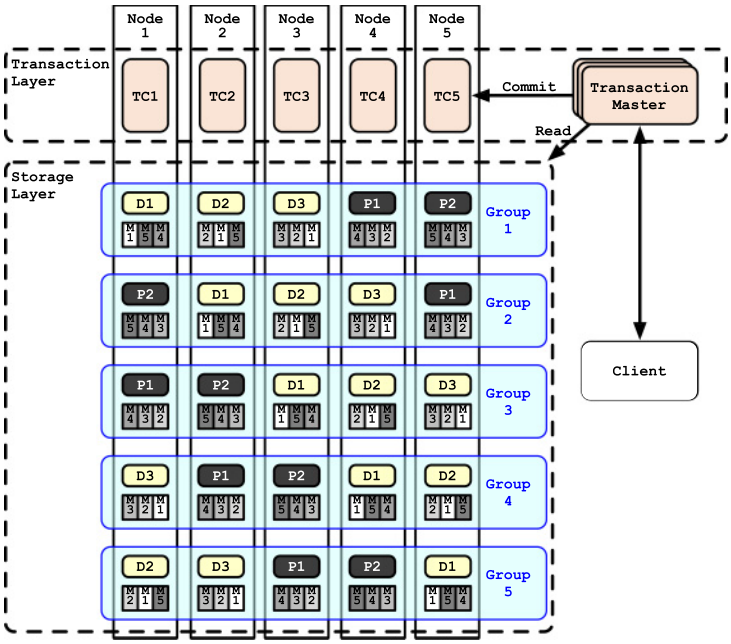


Fig. 6. Transaction Layer.

and to coordinate transactions. Cocytus uses the two-phase commit (2PC) protocol to commit a distributed transaction.

To assist transaction execution, Cocytus maintains three types of key/value pairs, which are directly stored in the underlying key/value store: (1) data fields, which are used to store data used by a transaction; (2) log fields, which are used to store transaction logs; (3) system fields, which are used to store the system variables used by the transaction layer itself. Cocytus extends each data field stored by the underlying KV-store with a version number to validate a transaction.

Each transaction maintains a read set and a write set. A transaction read first searches through the read set and write sets to see if the key/value pair has already been read or written. If not, then the transaction will get the pair from the underlying KV-store and record the pair into the read set. Each write by a transaction will be buffered into the write set.

As Cocytus mainly targets read-mostly scenarios and many transactional workloads contain many read-only transactions (e.g., six read-only vs. four read-write transactions in TPC-E (Chen et al. 2011)), Cocytus differentiates read-only transactions from read-write transactions. A read-only transaction only requires one phase to commit, as shown in Figure 7. To commit a read-only transaction, the transaction master in the client first sends a commit request to the transaction coordination services managing the corresponding key/value pairs. The request contains the corresponding key as well as the version number. Each coordination service will first check if there is any locked key, which indicates whether there is a concurrent read-write transaction in progress; it will then check if the version number matches. If the checks pass successfully, then the coordination service will send "AGREE" to the transaction master. The read-only transaction can commit successfully if all coordination service replies with "AGREE," otherwise the transaction will abort.

Cocytus uses the standard two-phase commit protocol to commit a read-write transaction, as shown in Figure 8. The first step is to prepare a transaction commit, where the transaction master sends a set of information to each coordination service. The information includes: (1) transaction

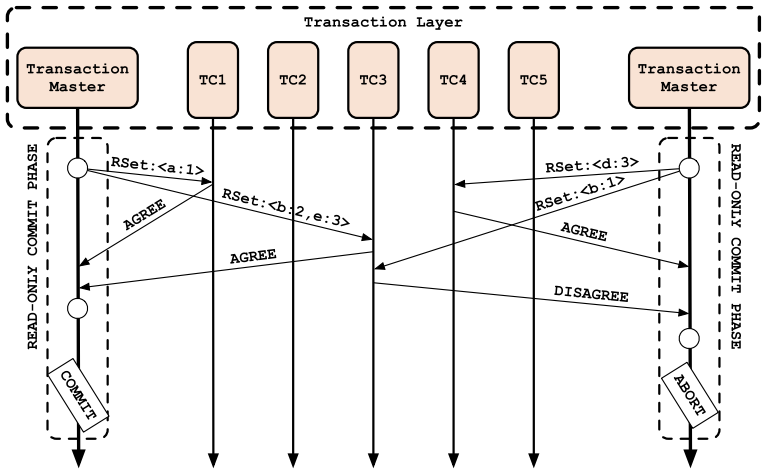


Fig. 7. Read-only Transaction Commit.

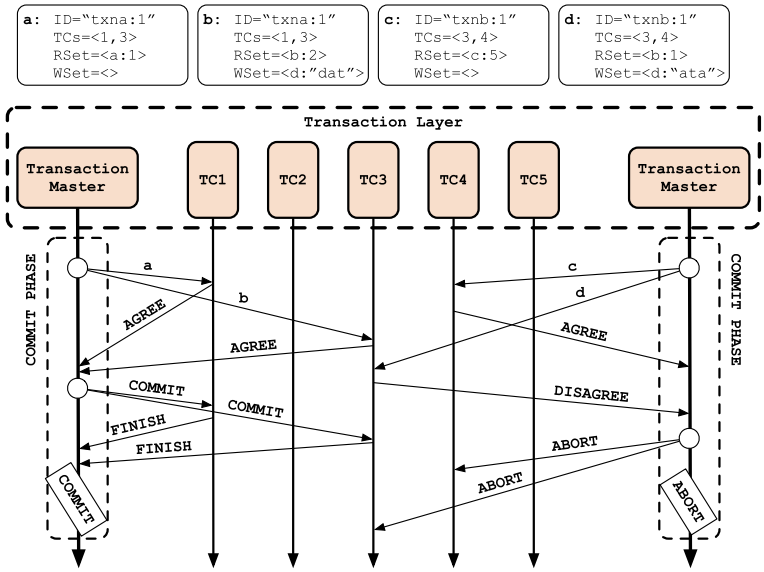


Fig. 8. Read-Write Transaction Commit.

ID; (2) the IDs of all involved coordination services; (3) keys and the corresponding versions in the read set for each coordination service; (4) keys and values in the write set for each coordination services. Each coordination service will use such information to determine: (1) if there is a concurrent yet conflicting transaction in the process of commit (i.e., if the key is locked or not); or (2) the versions for keys in the read set have not changed. After the validation passes, the coordination service will lock the corresponding keys, log the related information to the underlying KV-store and send an "AGREE" the transaction master. Then, if all coordination services replied "AGREE," then the transaction can be committed and log the related transaction information; yet the changes are still not visible to other transactions. Next, the master sends a "COMMIT" message to all coordination services to ask them to commit the changes; each of the coordination service

will install the changes, unlock the keys and reply to the master with "FINISH" once the changes have been committed. Finally, the master, upon collecting all "FINISH" confirmations from all involved coordination services, marks the transaction as committed.

5.2 Transaction Recovery

Cocytus follows a layered design by exploiting the high availability provided by the KV-store as a building block to tolerate possible machine failures. As discussed above, Cocytus logs critical information regarding a transaction during the committing process, which can be leveraged to do recovery upon failures, similar to standard recovery to the two-phase commit. The key issue here is to provide proper crash recovery to the transaction master and the coordination services.[2]

If a transaction master failed, then another machine (a survival or a standby machine) will take the job of the failed master. The transaction information can be recovered from the underlying KV-store. For each transaction, the master then uses the logged transaction status to determine whether to abort or commit a transaction. If a transaction has sent all "COMMIT" but received no confirmations, or a transaction has received all "FINISH" confirmations from coordination services, then the transaction can be restarted or marked as "FINISH" accordingly. If the transaction is in a state in between, then Cocytus needs to reply the coordination services to decide to commit or not. Specifically, an elected coordination service in a pending transaction will send messages to other services to decide if a transaction should be committed or not and commit or abort accordingly. Each coordination service will rely on a timeout mechanism to decide whether a transaction master is alive or not.

If a coordination service failed, then the underlying KV-store will first recover all KV pairs. Then, a new service node will recover necessary states for the coordination service from the recovered logs in the KV pairs. Currently, each coordination service uses a monotonically increasing number to identify logs. However, it does not serialize all logs among concurrent transactions to maximize performance. Hence, it may be possible that log 10 is available but log 9 is missing. Besides, since the writing of transaction data and the "FINISH" record is done concurrently, it is possible that the "FINISH" record exists yet the related transactional data is missing.

To address this issue, each coordination service maintains two system variables, stable log number (SLN) and maximum log number (MLN). The former indicates all transactions whose log numbers are no larger than MLN have been persisted (including both data and log). If there is a "FINISH" log for a transaction but the data has not been durable, then the transactional actions will be redone on this service. If there is no "FINISH" log for a transaction, then the service then locks related keys in the write set. If another transaction B currently holds the lock required by this transaction, then this indicates that transaction B must have finished according to the serialization order. Hence, the service finishes the related action to confirm the commit for transaction B. Since all data is stored in memory, it is fairly fast to recover transactions.

6 IMPLEMENTATION

To understand its performance implication on real KV-stores, we have implemented Cocytus on top of Memcached 1.4.21 with the synchronous model, by adding about 3,700 SLoC to Memcached. Currently, Cocytus only works for single-thread model and the data migration is not fully supported. To exploit multicore, Cocytus can be deployed with sharding and multi-process instead of multi-threading. In fact, using multi-threading has no significant improvement for data processes that may suffer from unnecessary resource contention and break data isolation. The parity processes could be implemented in a multi-threaded way to distribute the high CPU pressure under

[2]Note that the master may be colocated with one of the coordination service.

write-intensive workloads, which we leave as future work. We use Jerasure (Plank et al. 2008) and GF-complete (Plank et al. 2013) for the Galois-Field operations in RS-code. Note that Cocytus is largely orthogonal with the coding schemes; it will be our future work to apply other network or space-efficient coding schemes (Shah et al. 2012; Rashmi et al. 2015). The transaction layer is built from scratch and runs directly atop the modified Memcached. To study the impact of Cocytus on the transaction performance, we also implement the transaction layer atop the Memcached with primary-backup replication. This section describes some implementation issues.

Deterministic allocator: In Cocytus, the allocation metadata is separated from data. Each data process maintains a memory region for data with the *mmap* syscall. Each parity process also maintains an equivalent memory region for parity. To manage the data region, Cocytus uses two AVL trees, of which one records the free space and the other records the allocated space. The tree node consists of the start address of a memory piece and its length. The length is ensured to be multiples of 16 and is used as the index of the trees. Each memory location is stored in either of the trees. An *alloc* operation will find an appropriate memory piece in the free-tree and move it to the allocated-tree and the *free* operations do the opposite. The trees manage the memory pieces in a way similar to the buddy memory allocation: large blocks might be split into small ones during *alloc* operations and consecutive pieces are merged into a larger one during *free* operations. To make the splitting and merging fast, all memory blocks are linked by a list according to the address. Note that only the metadata is stored in the tree, which is stored separately from the actual memory managed by the allocator.

Pre-alloc: Cocytus uses the deterministic allocator and hashtables to ensure all metadata in each node is consistent. Hence, Cocytus only needs to guarantee that each process will handle the related requests in the same order. The piggybacked two-phase commit (Section 3.4) can mostly provide such a guarantee.

One exception is shown in Figure 9(a). When a recovery process receives a *set* request with X=a, it needs to allocate memory for the value. If the memory for the value needs to be recovered, then the recovery process first starts the recovery for X and puts this *set* request into a waiting queue. In Cocytus, the recovery is asynchronous. Thus, the recovery process is able to handle other requests before the recovery is finished. During this time frame, another *set* request with Y = b comes to the recovery process. The recovery process allocates memory for it and fortunately the memory allocated has already been recovered. Hence, the recovery process directly handles the *set* request with Y = b without any recovery and sends requests to other parity processes for fault-tolerance. As soon as they receive the request, other processes (for example, PP2 in the figure) allocate memory for Y and finish their work as usual. Finally, when the recovery for X is finished, the recovery process continues to handle the *set* request with X = a. It also sends fault-tolerance requests to other parity processes, on which the memory is allocated for X. Up to now, the recovery process has allocated memory for X and Y successively. However, on other parity processes, the memory allocation for Y happens before that for X. This different allocation ordering between recovery processes and parity processes will cause inconsistency.

Cocytus solves this problem by sending a pre-allocation request (shown in Figure 9(b)) before each *set* operation is queued due to recovery. In this way, the parity processes can pre-allocate space for the queued set requests and the ordering of memory allocation is guaranteed.

Recovery leader: Because when multiple recovery processes want to recover the two equivalent blocks simultaneously, both of them want to start an online recovery protocol, which is unnecessary. To avoid this situation, Cocytus assigns a recovery leader in each group. A recovery leader is a parity process responsible for initiating and finishing the recovery in the group. All other parity processes in the group will send recovery requests to the recovery leader if they need to recover data, and the recovery leader will broadcast the result after the recovery is finished. A

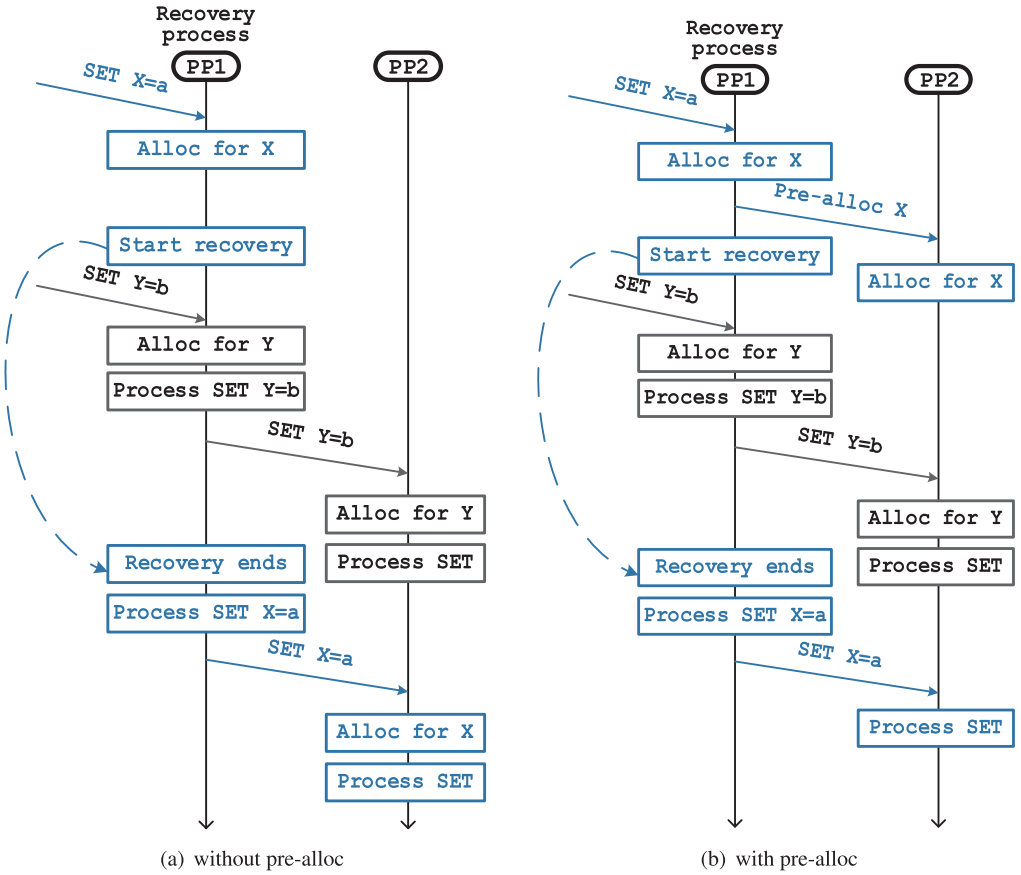


Fig. 9. In (a), the memory allocation ordering for X and Y is different on PP1 and PP2. In (b), thanks to the pre-alloc, the memory allocation ordering remains the same on different processes.

recovery leader is not absolutely necessary but such a centralized management of recovery can prevent the same data from being recovered multiple times and thus reduce the network traffic. Considering the interleaved layout of the system, the recovery leaders are uniformly distributed on different nodes and won't become the bottleneck.

Short-cut Recovery for Consecutive Failures: When there are more than one data process failures and the data of some failed processes are already recovered by the recovery process, the further recovered data might be wrong if we do not take the recovery process into consideration.

In the example given in Figure 4(b), suppose DP1 (data process 1) fails first and PP1 (parity process 1) becomes a recovery process for it. After PP1 recovered a part of data blocks, DP2 fails and PP2 becomes a recovery process for DP2. At that moment, some data blocks on PP1 have been recovered and others haven't. To recover a data block on DP2, if its corresponding data block on DP1 has been recovered, it should be recovered in the way that involves three data blocks and one parity block, otherwise it should be recovered in the way that involves two data blocks and two parity blocks. The procedures of the two kinds of recovery are definitely different.

Primary-backup replication: To evaluate Cocytus, we also implemented a primary-backup (PBR) replication version based on Memcached-1.4.21 with almost the same design as Cocytus, like synchronous write, piggyback, except that Cocytus puts the data in a coded space and needs

to decode data after a failure occurs. We did not directly use Repcached (KLab Inc. 2011) for two reasons. One is that Repcached only supports one slave worker. The other one is that *set* operation in Repcached is asynchronous and thus does not guarantee crash consistency.

7 EVALUATION

We evaluate the performance of Cocytus by comparing it to primary-backup replication (PBR) and the vanilla Memcached. The highlights of our evaluation results are the followings:

- —Cocytus achieves high memory efficiency: It reduces memory consumption by 33% to 46% for value sizes from 1KB to 16KB when tolerating two node failures.
- —Cocytus incurs low overhead: It has similar throughput with PBR and vanilla KV-store (i.e., Memcached) and incurs small increase in latency compared to vanilla KV-store.
- —Cocytus can tolerate failures as designed and recover fast and gracefully: Even under two node crashes, Cocytus can gracefully recover lost data and handle client requests with close performance with PBR.

7.1 Experimental Setup

Hardware and configuration: Due to our hardware limit, we conduct all experiments on a six-node cluster of machines. Each machine has two 10-core 2.3GHz Intel Xeon E5-2650, 64GB of RAM and is connected with 10Gb network. We use five of the six nodes to run as servers and the remaining one as client processes.

To gain a better memory efficiency, Cocytus could use more data processes in a coding group. However, deploying too many data processes in one group increases the burden on parity processes, which could be a bottleneck of the system. Because of the limitation of our cluster, we deploy Cocytus with five interleaved EC groups, which are configured as RS(3,5) so the system can tolerate two failures while maximizing the data processes. Each group consists of three data processes and two parity processes. With this deployment, each node contains three data processes and two parity processes of different groups.

Targets of comparison: We compare Cocytus with PBR and vanilla Memcached. To evaluate PBR, we distribute 15 data processes among the five nodes. For each data process, we launch 2 backup processes so that the system can also tolerate two node failures. This deployment launches more processes (45 processes) compared to Cocytus (25 processes), which could use more CPU resource in some cases. We deploy the vanilla Memcached by evenly distributing 15 instances among the five nodes. In this way, the number of processes of Memcached is the same as the data processes of Cocytus.

Workload: We use the YCSB (Cooper et al. 2010) benchmark to generate our workloads. We generate each key by concatenating a table name and an identifier, and a value is a compressed HashMap object, which consists of multiple fields. The distribution of the key probability is Zipfian (Egghe 2005), with which some keys are hot and some keys are cold. The length of the key is usually smaller than 16B. We also evaluate the systems with different read/write ratios, including equal-shares (50%:50%), read-mostly(95%:5%), and read-only (100%:0%).

Since the median of the value sizes from Facebook (Nishtala et al. 2013) are 4.34KB for *Region* and 10.7KB for *Cluster*, we test these caching systems with similar value sizes. As in YCSB, a value consists of multiple fields, to evaluate our system with various value sizes, we keep the field number as 10 while changing the field size to make the total value sizes be 1KB/4KB/16KB, that is, the field sizes are 0.1KB/0.4KB/1.6KB accordingly. To limit the total data size to be 64GB, the item numbers for 1/4/16KB are 64/16/1 million, respectively. However, due to the object compression,

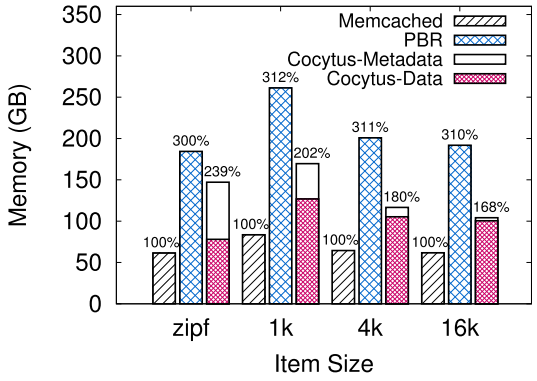


Fig. 10. Memory consumption of three systems with different value sizes. Due to the compression in YCSB, the total memory cost for different value sizes differs a little bit.

we cannot predict the real value size received by the KV-store and the values may not be aligned as well; Cocytus aligns the compressed values to 16 bytes to perform coding.

7.2 Memory Consumption

As shown in Figure 10, Cocytus achieves notable memory saving compared to PBR, due to the use of erasure coding. With a 16KB value size, Cocytus achieves 46% memory saving compared to PBR. With RS(3,5), the expected memory overhead of Cocytus should be 1.66X while the actual memory overhead ranges from 1.7X to 2X. This is because replicating metadata and keys introduces more memory cost, for example, 25%, 9.5%, and 4% of all consumed memory for value sizes of 1, 4, and 16KB. We believe such a cost is worthwhile for the benefit of fast and online recovery.

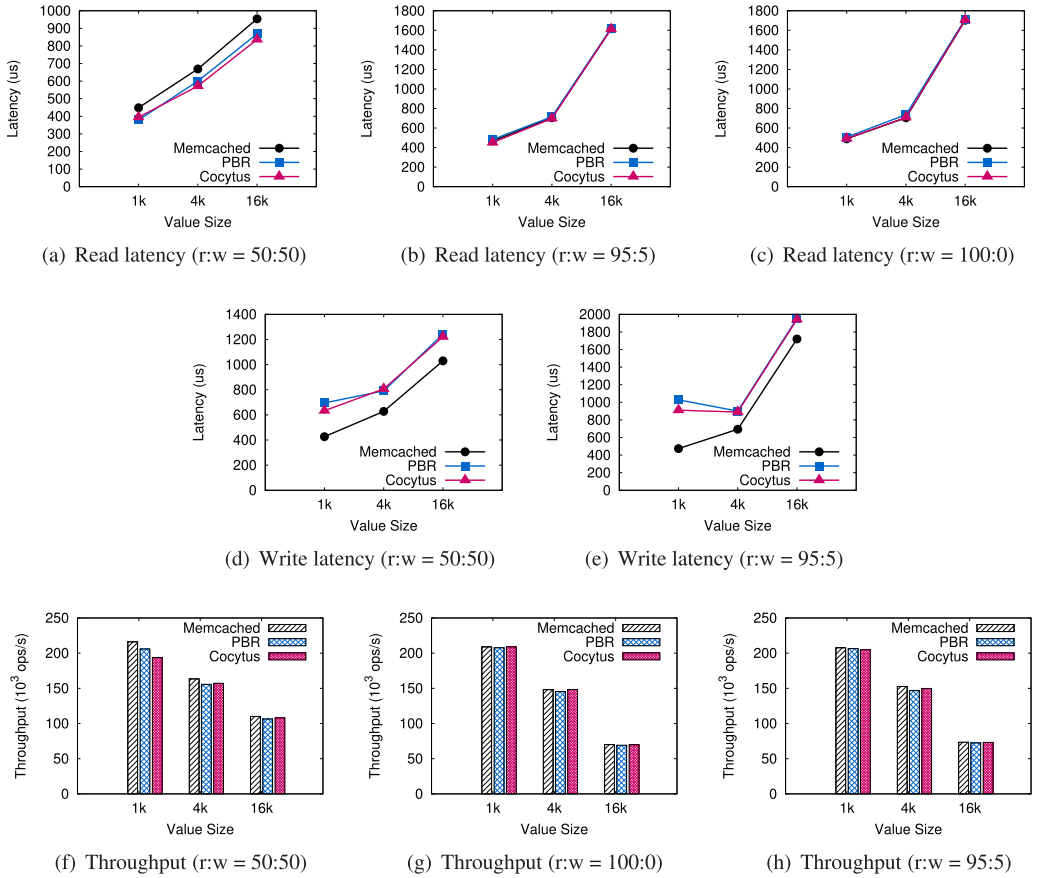
To investigate the effect of small- and variable-sized values, we conduct a test in which the value size follows the Zipfian distribution over the range from 10B to 1KB. Since it is harder to predict the total memory consumption, we simply insert 100 million such items. The result is shown as *zipf* in Figure 10. As expected, more items bring more metadata (including keys), which diminishes the benefit of Cocytus. Even so, Cocytus still achieves 20% memory saving compared to PBR.

7.3 Performance

As shown in Figure 11, Cocytus incurs little performance overhead for read-only and read-mostly workloads and incurs small overhead for write-intensive workload compared to vanilla Memcached. Cocytus has similar latency and throughput with PBR. The following use some profiling data to explain the data.

Small overhead of Cocytus and PBR: As the three configurations handle *get* request with similar operations, the performance is similar in this case. However, when handling *set* requests, Cocytus and PBR introduce more operations and network traffic and thus modestly higher latency and small degradation of throughput. From the profiled CPU utilization (Table 2) and network traffic (Memcached:540Mb/s, PBR: 2.35Gb/s, Cocytus:2.3Gb/s, profiled during 120 clients insert data), we found that even though PBR and Cocytus have more CPU operations and network traffic, both of them were not the bottleneck. Hence, multiple requests from clients can be overlapped and pipelined. Hence, the throughput is similar with the vanilla Memcached. Hence, both Cocytus and PBR can trade some CPU and network resources for high availability, while incurring small user-perceived performance overhead.

Higher write latency of PBR and Cocytus: The latency is higher when the read-write ratio is 95%:5%, which is a quite strange phenomenon. The reason is that *set* operations are preempted by



(a) Read latency (r:w = 50:50) (b) Read latency (r:w = 95:5) (c) Read latency (r:w = 100:0)

(d) Write latency (r:w = 50:50) (e) Write latency (r:w = 95:5)

(f) Throughput (r:w = 50:50) (g) Throughput (r:w = 100:0) (h) Throughput (r:w = 95:5)

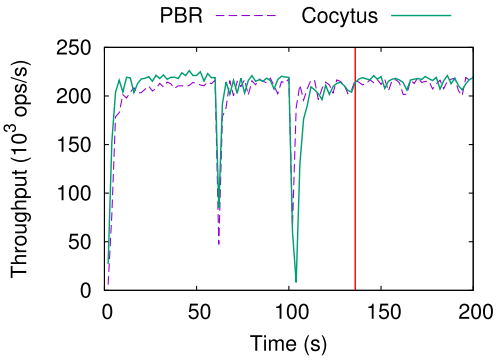
Fig. 11. Comparison of latency and throughput of the three configurations.

Table 2. CPU Utilization for 1KB Value Size

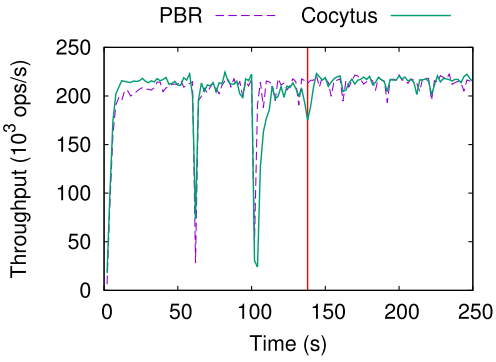
	Memcached	PBR		Cocytus	
Read : Write	15 processes	15 primary processes	30 backup processes	15 data processes	10 parity processes
50%:50%	231%CPUs	439%CPUs	189%CPUs	802%CPUs	255%CPUs
95%:5%	228%CPUs	234%CPUs	60%CPUs	256%CPUs	54%CPUs
100%:0%	222%CPUs	230%CPUs	21%CPUs	223%CPUs	15%CPUs

get operations. In Cocytus and PBR, *set* operations are FIFO, while *set* operations and *get* operations are interleaved. Especially in the read-mostly workload, the *set* operations tend to be preempted, as *set* operations have longer path in PBR and Cocytus.

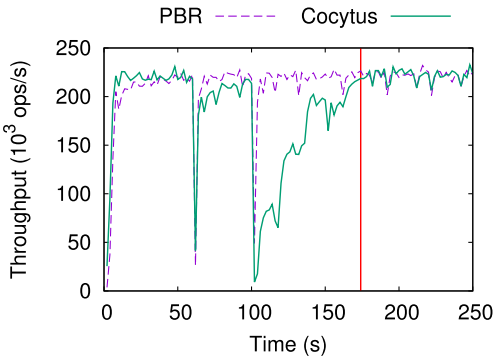
Lower read latency of PBR and Cocytus: There is an interesting phenomenon is that higher write latency causes lower read latency for PBR and Cocytus under update-intensive case (i.e., r:w = 50:50). This may be because when the write latency is higher, more client threads are waiting for the *set* operations at a time. However, the waiting on *set* operation does not block the *get* operation from other client threads. Hence, the client threads waiting on *get* operation could be



(a) Throughput on recovery (r:w = 100:0)



(b) Throughput on recovery (r:w = 95:5)



(c) Throughput on recovery (r:w = 50:50)

Fig. 12. Performance of PBR and Cocytus when nodes fail. The vertical lines indicate all data blocks are recovered completely.

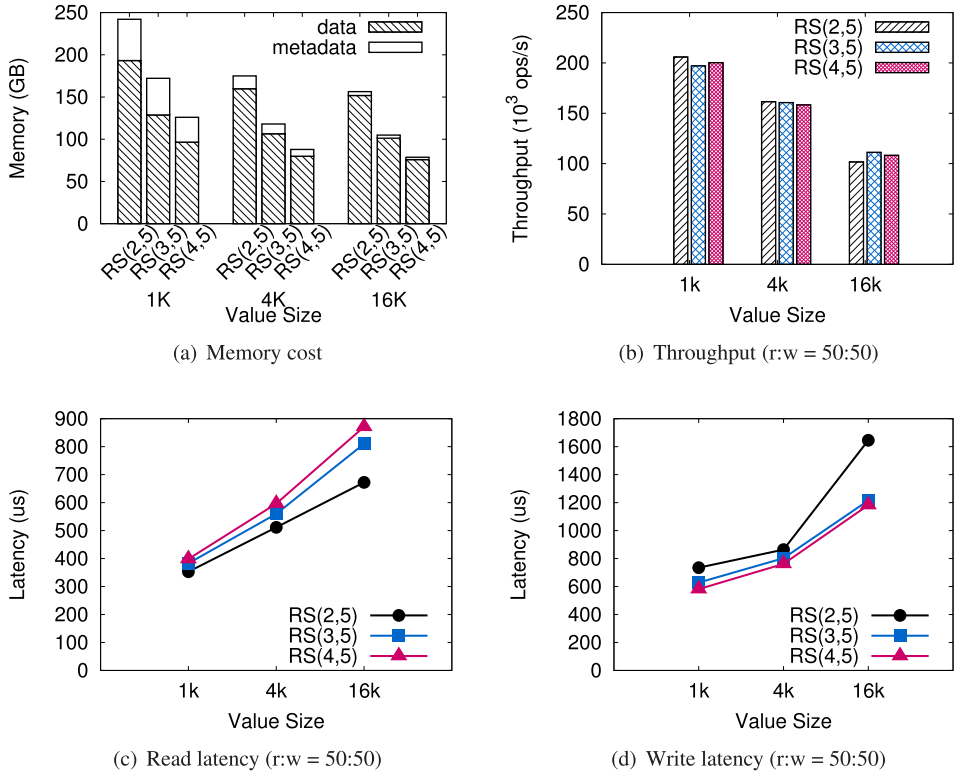
done faster, because there would be fewer client threads that could block this operation. As a result, the latency of *get* is lower.

7.4 Recovery Efficiency

We evaluate the recovery efficiency using 1KB value size for read-only, read-mostly and read-write workloads. We emulate two node failures by manually killing all processes on the node. The first node failure occurs on the 60th second after the benchmark starts. And the other node failure occurs at 100s, before the recovery of the first failure finishes. The two throughput collapses in each of the subfigures of Figure 12 are caused by the TCP connection mechanism and can be used coincidentally to indicate the time a node fails. The vertical lines indicate the time that all the data has been recovered.

Our evaluation shows that after the first node failure, Cocytus can repair the data at 550MB/s without client requests. The speed could be much faster if we use more processes. However, to achieve high availability, Cocytus first does recovery for requested units and recovers cold data when the system is idle.

As shown in Figure 12(a), Cocytus performs similarly as PBR when the workload is read-only, which confirms that data recovery could be done in parallel with read requests without notable overhead. The latencies for 50%, 90%, 99% requests are 408us, 753us, and 1117us in Cocytus during recovery. Similar performance can be achieved when the read-write ratio is 95%:5%, as shown in



(a) Memory cost

(b) Throughput (r:w = 50:50)

(c) Read latency (r:w = 50:50)

(d) Write latency (r:w = 50:50)

Fig. 13. Performance under different coding schemes.

Figure 12(b). In the case with frequent *set* requests, as shown in Figure 12(c), the recovery affects the throughput of normal request handling modestly. The reason is that to handle *set* operations Cocytus needs to allocate new blocks, which usually triggers data recovery on those blocks. Waiting for such data recovery to complete degrades the performance. In fact, after the first node crashes, the performance is still acceptable, since the recovery is relatively simpler and not all processes are involved in the recovery. However, when two node failures occur simultaneously, the performance can be affected more notably. Fortunately, this is a very rare case and even if it happens, Cocytus can still provide services with reasonable performance and complete the data recovery quickly.

To confirm the benefit of our online recovery protocol, we also implement a blocked version of Cocytus for comparison. In the blocked version of Cocytus, the *set* operations are delayed if there is any recovery in progress and the *get* operations are not affected. From Figure 12, we can observe that the throughput of the blocked version collapses even when there is only one node failure and 5% of *set* operations.

7.5 Different Coding Schemes

To understand the effect under different coding schemes, we evaluate the Cocytus with RS(4,5), RS(3,5) and RS(2,5). As shown in Figure 13, the memory consumption of RS(2,5) is the largest and the one of RS(4,5) is the least. All the three coding schemes benefit more from larger value sizes. Their throughput is similar, because there are no bottlenecks on servers. However, the write latency of RS(2,5) is a little bit longer, since it sends more messages to parity processes. The reason

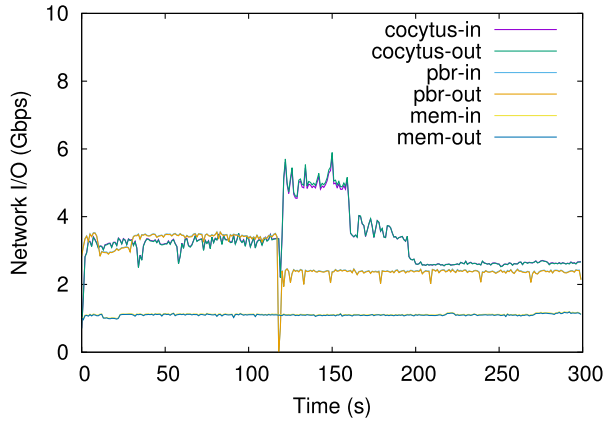
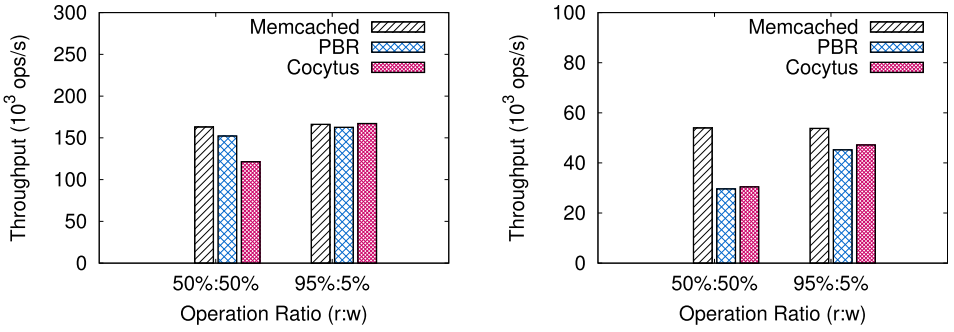


Fig. 14. Network cost of three systems with 1KB value size and 50%:50% read-write ratio.



(a) Lower CPU frequency (b) Smaller network bandwidth

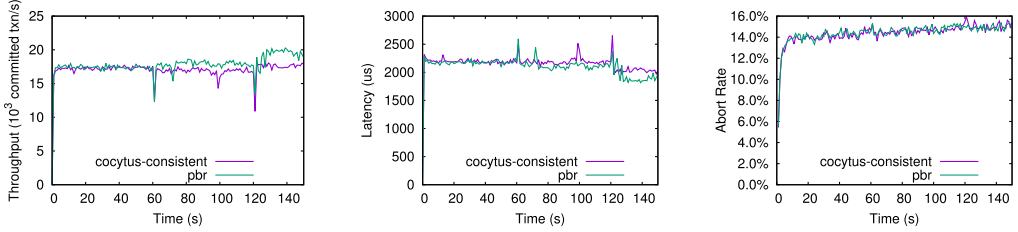
Fig. 15. Performance under lower CPU frequency and smaller network bandwidth. The value size is 1KB.

why RS(2,5) has lower read latency should be a longer write latency causes lower read latency (similar to the case described previously).

7.6 Performance Under Limited Resources

To understand Cocytus' performance under limited resources, we first collect the network bandwidth cost of three systems in 1KB value size and 50%:50% read-write ratio scenario. The data that are shown in Figure 14 is the total network cost of five servers. In this evaluation, we crash one of the five servers (for both Cocytus and PBR) at the 120th second, so there is a period of network fluctuation starting at that point. Before the crash, the network cost of PBR and Cocytus is around 3Gbps, which is close to PBR and is about 3× of that of Memcached. After the crash, PBR′ s network cost drops, because one server is down on which there is no more network traffic. Cocytus′s network cost reaches 5Gbps–{6}Gbps during data recovery. After recovery, the network cost of PBR and Cocytus are close again.

Next, we limit the network bandwidth and CPU resources to test the three systems in resource-limited environments. As Figure 15 shows, when we limit the CPU frequency to 1.2GHz, which is 2.3GHz in the previous test, Cocytus's performance is worse than other two systems on 50%:50% read-write ratio due to encoding work. Nevertheless, the performance of three systems is close when the read-write ratio is 95%:5%. When we limited the outside network bandwidth to 100Mbps



(a) Throughput of transactions (b) Latency of transactions (c) Abort rate of transactions

Fig. 16. Performance of transaction layer.

for each server with the traffic controller (tc) utility, the performance of PBR and Cocytus is half of Memcached's on 50%:50% read-write ratio; the performance difference is smaller on 95%:5% read-write ratio.

7.7 Transaction Performance

We evaluate the transaction performance atop Cocytus by studying the raw throughput, latency, abort rate as well as the crash recovery performance. In this evaluation, we set the number of transaction masters the same as the number of data nodes. Each transaction master and its corresponding data node are located on the same machine. In this way, there is no cross machine traffic between transaction master and data node. We implemented SmallBank (Cahill et al. 2009) in C++ with an extension operation from the OLTP-Bench (Difallah et al. 2013). The benchmark contains six operations that are amalgamate, balance, deposit checking, send payment, transact saving and write check. We set the frequency of the six operations as 15%:15%:15%:25%:15%:15%, which is the same as the config in OLTP-Bench. Hence, only 15% transactions are read-only. There are 100,000 accounts in the bank, of which 4,000 accounts are hotpot and are accessed in 90% probability.

Figure 16(a) illustrates the performance in normal mode and in crash mode. The transactions atop Cocytus can achieve around 17,000 transactions per second, with a mean latency of 2.3ms (in Figure 16(b)) and a transaction abort rate between 12% to 16% (in Figure 16(c)). There are three reasons why performance difference between the K/V operations and transaction operations are large. First, each transaction involves 3 to 8 KV operations. Second, 85% transactions are read-write that involve more transaction-related traffic on commit. Third, the transactions have a hotspot, and they would be serialized if they access the same key.

The high abort rate are mostly caused by clients when they found some data is unexpected, for example, there is no enough money on withdrawing.

To study how Cocytus handles crash recovery, we shut down a machine running both the data node and the transaction coordination services on the 60th second, and then shut down another machine on the 120th second. From Figure 16(a), we can see that the time to recover the data node as well as the transaction service is in 2s, and then the performance goes back to the normal level.

We also run the transaction layer on top of Memcached with PBR for comparison. As shown in Figure 16(a), both systems experience similar performance in normal execution and during recovery except that PBR reaches a little higher performance after two machine crash. This further confirms the efficiency and availability of Cocytus without sacrificing performance in an in-memory transaction system.

8 RELATED WORK

Separation of work: The separation of metadata/key and values is inspired by prior efforts on separation of work. For example, Wang et al. (2012) separate data from metadata to achieve

efficient Paxos-style asynchronous replication of storage. Yin et al. (2003) separate execution from agreement to reduce execution nodes when tolerating Byzantine faults. Lu et al. (2016) separate keys from value in a persistent LSM-tree-based KV-store to minimize I/O amplification for SSD-conscious storage. Clement et al. (2009) distinguish omission and Byzantine failures and leverage redundancy between them to reduce required replicas. In contrast, Cocytus separates metadata/key from values to achieve space-efficient and highly-available key/value stores.

Erasure coding: Erasure coding has been widely adopted in storage systems in both academia and industry to achieve both durability and space efficiency (Huang et al. 2012; Silberstein et al. 2014; Rashmi et al. 2013; Sathiamoorthy et al. 2013; Muralidhar et al. 2014). Generally, they provide a number of optimizations that optimize the coding efficiency and recovery bandwidth, like local reconstruction codes (Huang et al. 2012), Xorbas (Sathiamoorthy et al. 2013), piggyback codes (Rashmi et al. 2013), and lazy recovery (Silberstein et al. 2014). PanFS (Welch et al. 2008) is a parallel file system that uses per-file erasure coding to protect files greater than 64KB, but replicates metadata and small files to minimize the cost of metadata updates. Atlas (Lai et al. 2015) is also a KV storage system that uses EC to replace three-copy replication and separates the management of metadata and data. Atlas is designed for HDD and it splits a patch of values for EC. In contrast, Cocytus is designed for memory storage and splits entire storage space. A followup work of Cocytus, EC-Cache (Rashmi et al. 2016) applies online erasure coding within a single large object to provide dynamic load balancing in memory caching systems under skewed workloads. Yet, EC-cache only targets immutable objects while Cocytus targets both immutable and mutable objects.

Replication: Replication is a standard approach to fault tolerance, which may be categorized into synchronous (Budhiraja et al. 1993; Bressoud and Schneider 1996; van Renesse and Schneider 2004) and asynchronous (Lamport 2001; Bolosky et al. 2011). Mojim (Zhang et al. 2015) combines NVRAM and a two-tier primary-backup replication scheme to optimize database replication. Cocytus currently leverages standard primary-backup replication to provide availability to metadata and key in the face of omission failures. It will be our future work to apply other replications schemes or handle commission failures.

RAMCloud (Ongaro et al. 2011) exploits scale of clusters to achieve fast data recovery. Imitator (Wang et al. 2014b) leverages existing vertices in partitioned graphs to provide fault-tolerant graph computation, which also leverages multiple replicas to recover failed data in one node. However, they do not provide online recovery such that the data being recovered cannot be accessed simultaneously. In contrast, Cocytus does not require scale of clusters for fast recovery but instead provide always-on data accesses, thanks to replicating metadata and keys.

Key/value stores: There have been a considerable number of interests in optimizing key/value stores, leveraging advanced hardware like RDMA (Mitchell et al. 2013; Stuedi et al. 2012; Kalia et al. 2014; Wei et al. 2015) or increasing concurrency (Fan et al. 2013; Li et al. 2014; Liu et al. 2014). Cocytus is largely orthogonal with such improvements, and we believe that Cocytus can be similarly applied to such key/value stores to provide high availability.

Fault-tolerant transaction processing: There has been a long thread of providing fast and reliable distributed transactions (Thomson et al. 2012; Corbett et al. 2012, 2013; Dragojević et al. 2015; Chen et al. 2016). Compared to the transactions in Cocytus, prior designs use primary-backup replication, which incurs more memory consumptions. We believe Cocytus can be easily integrated to prior distributed transaction designs and showcase a simple design that achieves good performance and fast recovery.

9 CONCLUSION AND FUTURE WORK

Efficiency and availability are two key demanding features for in-memory key/value stores. We have demonstrated such a design that achieves both efficiency and availability by building

Cocytus and integrating it into Memcached. Cocytus uses a hybrid replication scheme by using PBR for metadata and keys while using erasure-coding for values with large sizes. Cocytus is able to achieve similarly normal performance with PBR and little performance impact during recovery while achieving much higher memory efficiency.

We plan to extend our work in several ways. First, we plan to explore a larger cluster setting and study the impact of other optimized coding schemes on the performance of Cocytus. Second, we plan to investigate how Cocytus can be applied to other in-memory stores using NVRAM (Venkataraman et al. 2011; Coburn et al. 2011; Yang et al. 2015).

REFERENCES

Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The cost of serializability on platforms that use snapshot isolation. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE'08)*. IEEE, 576–585.

Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*. ACM, 53–64.

William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the Conference on Network Systems Design and Implementation (NSDI'11)*.

Thomas C. Bressoud and Fred B. Schneider. 1996. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst. (TOCS)* 14, 1 (1996), 80–107.

Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A bloat-aware design for big data applications. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*. ACM, 119–130.

Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. 1993. The primary-backup approach. *Distrib. Syst.* 2 (1993), 199–216.

Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable isolation for snapshot databases. *ACM Trans. Database Syst. (TODS)* 34, 4 (2009), 20.

K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. (TOCS)* 3, 1 (1985), 63–75.

Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *ACM SIGMOD Rec.* 39, 3 (2011), 5–10.

Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using rdma and htm. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, 26.

Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 277–290.

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 105–118.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, 251–264.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild et al. 2013. Spanner: Googles globally distributed database. *ACM Trans. Comput. Syst. (TOCS)* 31, 3 (2013), 8.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 International Conference on Management of Data*. ACM, 1243–1254.

Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 54–70. DOI: http://dx.doi.org/10.1145/2815400.2815425

Leo Egghe. 2005. Zipfian and lotkaian continuous concentration theory. *J. Amer. Soc. Info. Sci. Technol.* 56, 9 (2005), 935–945.

Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the Conference on Network Systems Design and Implementation (NSDI'13)*, Vol. 13. 385–398.

Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database–An architecture overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux J.* 2004, 124 (2004), 5.

Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhrúv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet Wiener. 2014. Fast database restarts at facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 541–549.

Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, and others. 2012. Erasure coding in windows azure storage. In *Proceedings of the USENIX Annual Technical Conference*. 15–26.

Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference of the Special Interest Group on Data Communications (SIGCOMM'14)*. ACM, 295–306.

KLab Inc. 2011. Homepage. Retrieved from http://repcached.lab.klab.org.

Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. 2013. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13.

Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. IEEE, 1–14.

Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, 27.

Ran Liu, Heng Zhang, and Haibo Chen. 2014. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC*, Vol. 14. 219–230.

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the Conference on File and Storage Technologies (FAST'16)*. 133–148.

Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference*. 103–114.

Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang *et al.* 2014. F4: Facebooks warm BLOB storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 383–398.

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab et al. 2013. Scaling memcache at facebook. In *Proceedings of the Conference on Network Systems Design and Implementation (NSDI'13)*. 385–398.

Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 29–41.

J. S. Plank, E. L. Miller, and W. B. Houston. 2013. *GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic*. Technical Report UT-CS-13-703. University of Tennessee.

J. S. Plank, S. Simmerman, and C. D. Schuman. 2008. *Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications—Version 1.2*. Technical Report CS-08-627. University of Tennessee.

K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. 2015. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 81–94.

K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. *Proceedings of the Conference on USENIX HotStorage* (2013).

K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI'16)*.

Irving S. Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *J. Soc. Industr. Appl. Math.* 8, 2 (1960), 300–304.

Luigi Rizzo. 1997. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Comput. Commun. Rev.* 27, 2 (1997), 24–36.

Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the Very Large Data Base Endowment*. VLDB Endowment, 325–336.

Nihar B. Shah, K. V. Rashmi, P. Vijay Kumar, and Kannan Ramchandran. 2012. Distributed storage codes with repair-by-transfer and nonachievability of interior points on the storage-bandwidth tradeoff. *IEEE Trans. Info. Theory* 58, 3 (2012), 1837–1852.

Mark Silberstein, Lakshmi Ganesh, Yang Wang, Lorenzo Alvisi, and Mike Dahlin. 2014. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proceedings of International Conference on Systems and Storage*. ACM, 1–7.

SNIA. 2015. NVDIMM Special Interest Group. Retrieved from http://www.snia.org/forums/sssi/NVDIMM (2015).

Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. Wimpy nodes with 10GbE: Leveraging one-sided operations in Soft-RDMA to boost memcached. In *Proceedings of the USENIX Annual Technical Conference*. 347–353.

Viking Technology. 2014. ArxCis-NV (TM): Non-Volatile DIMM. Retrieved from http://www.vikingtechnology.com/arxcis-nv.

Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. ACM, 1–12. DOI: http://dx.doi.org/10.1145/2213836.2213838

Twitter Inc. 2012. Twemcache is the Twitter Memcached. Retrieved from https://github.com/twitter/twemcache (2012).

Robbert van Renesse and Fred B. Schneider. 2004. Chain replication for supporting high throughput and availability. In *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI'04)*, Vol. 4. 91–104.

Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H. Campbell, and others. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the Conference on File and Storage Technologies (FAST'11)*. 61–75.

Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. 2014b. Replication-based fault-tolerance for large-scale graph processing. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*. IEEE, 562–573.

Yang Wang, Lorenzo Alvisi, and Mike Dahlin. 2012. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proceedings of the USENIX Annual Technical Conference*. 413–424.

Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014a. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, 26.

Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 87–104.

Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable performance of the panasas parallel file system. In *Proceedings of the Conference on File and Storage Technologies (FAST'08)*, Vol. 8. 1–17.

Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. USENIX Association, 167–181.

Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'03)*. ACM, 253–267.

Jeremy Zawodny. 2009. Redis: Lightweight key/value store that goes the extra mile. *Linux Mag.* 79 (2009).

Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 167–180.

Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 3–18.

Received September 2016; revised March 2017; accepted March 2017