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

Heng Zhang, Mingkai Dong, and Haibo Chen, ***Shanghai Jiao Tong University***

https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-heng

This paper is included in the Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16).

February 22–25, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-28-7

Open access to the Proceedings of the 14th USENIX Conference on File and Storage Technologies is sponsored by USENIX

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

Heng Zhang, Mingkai Dong, Haibo Chen*
Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

ABSTRACT

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 paper 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 numerous 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 serving KV requests. We have applied Cocytus to Memcached. 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, yet saves 33% to 46% memory compared to PBR when tolerating two failures.

1 INTRODUCTION

The increasing 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 [13] and Redis [47] have been widely used in Facebook [24], Twitter [38] and LinkedIn. There have also been considerable interests of applying in-memory databases (IMDBs) to performance-hungry scenarios (e.g., SAP HANA [12], Oracle TimesTen [18] and Microsoft Hekaton [9]).

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 120 GB data from disk to memory for an in-memory database in Facebook took 2.5-3 hours [14]. Traditional ways of providing high availability are through replication such as standard primary-backup (PBR) [5] and chain-replication [39], 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 paper describes Cocytus, an efficient and available in-memory replication scheme that is strongly consistent. 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. The key of Cocytus is efficiently combining the space-efficient erasure coding scheme with the PBR.

Erasure coding is a space-efficient solution for data replication, which is widely applied in distributed storage systems, including Windows Azure Store [15] and Facebook storage [23]. However, though space-efficient, erasure coding is well-known for its lengthy recovery and transient data unavailability [15, 34].

In this paper, 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.

*Corresponding author

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 [42] and uses a hybrid replication scheme. In particular, Cocytus uses erasure coding for application data while using PBR for small-sized metadata.

The second challenge is how to consistently recover lost data blocks online with the distributed data blocks and parity blocks[1]. Cocytus introduces a distributed online recovery protocol that consistently collects all data blocks and parity blocks to recover lost data, yet without blocking services on live data blocks and with predictable memory.

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. By using YCSB [8] to issue requests with different key/value distribution, 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.

In summary, the main contribution of this paper 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 [13] and a thorough evaluation that confirms Cocytus's efficiency and availability.

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

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

2 Background and Challenges

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

2.1 Background

Primary-backup replication: Primary-backup replication (PBR) [3] 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 [19]). As a result, the system can still provide continuous services upon node failures. This, however, is at the cost of high data redundancy, e.g., 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%.

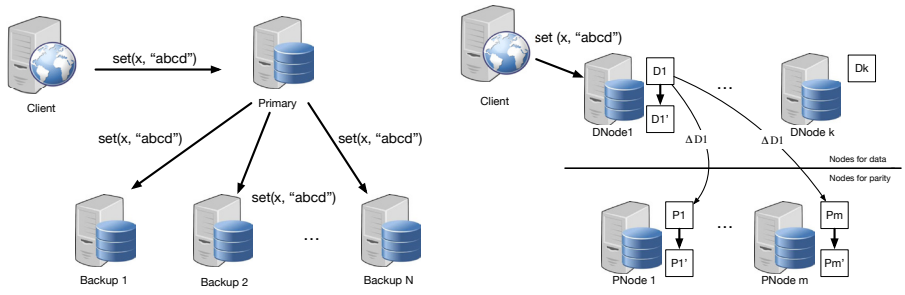
Erasure coding: Erasure coding is an efficient way to provide data durability. As shown in Figure 1(b), with erasure coding, an N-node cluster can use K of N nodes for data and M nodes for parity ($K + M = N$). A commonly used coding scheme is Reed-Solomon codes (RS-code) [30], which computes parities according to its data over a finite field by the following formula (the matrix is called a *Vandermonde matrix*):

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

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 formula (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 RS(K,N) can handle M node



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

(b) Key/value store with erasure coding

Figure 1: Data storage with two different replication schemes.

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.

2.2 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 [14].

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 [4]. 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 5-node cluster with an average CPU core (2.3 GHz Xeon E5, detailed configurations in section 6.1). 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 [31]). 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.

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

Table 1: The speed of coding data with different schemes for a 5-node cluster

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 nature. However, an update operation in a KV-store may result in a number of small updates, which would introduce notable coding operations and network traffic. 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 the four pointers of the double linked list. Some statistics like that for LRU replacement 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 updates. For the allocation metadata, as Memcached uses a slab allocator, an allocation operation commonly changes four variables and a free operation changes six to seven variables.

The second challenge is that a data update involves updates 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, this may easily cause inconsistent recovery of data.

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

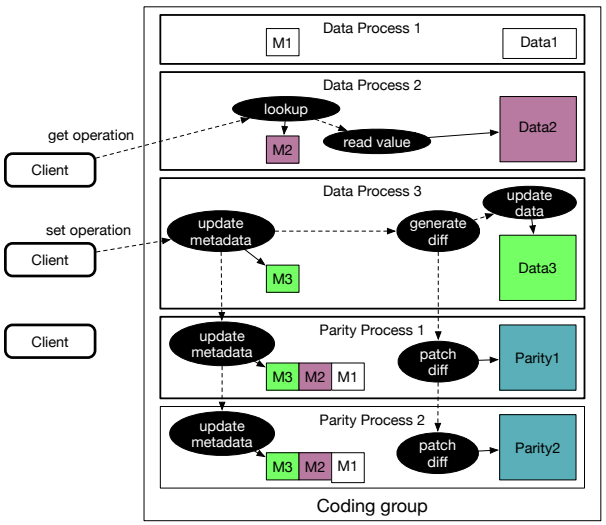


Figure 2: Requests handled by an coding group in Cocytus, where K=3, M=2.

and Value are arbitrary strings. According to prior large-scale analysis on key/value stores in commercial workloads [1, 24], Cocytus assumes that the value size is usually 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 [24]. Alternatively, one may leverage battery-backed RAM like NVDIMM [37, 35] to preserve durability.

Cocytus is designed to be synchronous, i.e., 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 [1]. For update-intensive workloads, Cocytus would use more CPU resources due to the additional calculations caused by the erasure coding, and achieve a 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 key 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.

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 larger 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.

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.

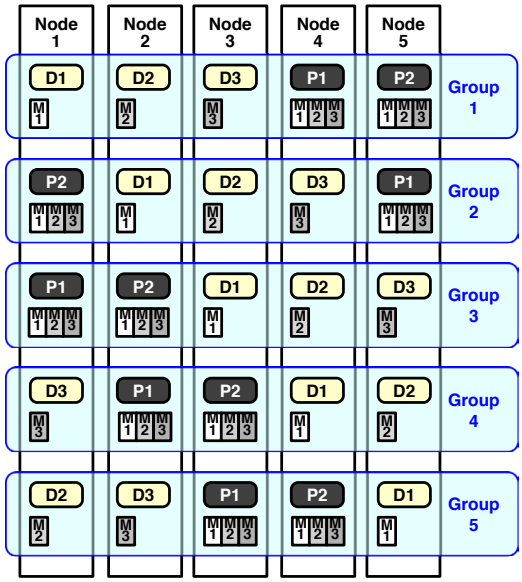


Figure 3: Interleaved layout of coding groups in Cocytus. The blocks in the same row belong to one coding group.

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.

这里和fault isolation说法是矛盾吗？

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 acknowledgements to its data process. After the data process receives acknowledgements 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.

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 is 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}, ..., T_K and the logical timestamps of parity blocks are VT_1,

VT_2, ..., VT_F. If $VT_1 = VT_2 = ... = VT_F$ and $VT_1[F+1..K] = \langle T_{F+1}, T_{F+2}, ..., T_K \rangle$, 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 7ms to 13 ms 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 online 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.

As shown in Figure 4, 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.

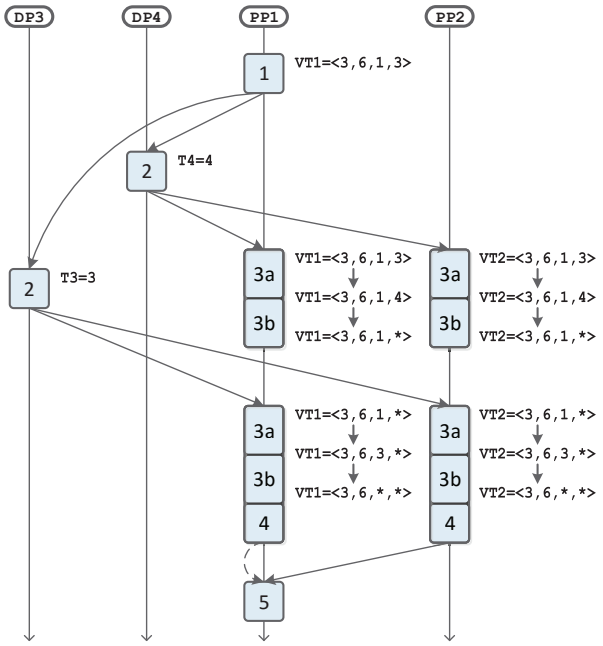


Figure 4: Online recovery when DP1 and DP2 crash in an RS(4, 6) coding group

- 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 no longer associated with the ith data process. It stops being updated by the ith data process. Hence, the rest of parity units on this recovery process are still associated with the ith data process.

- 4. When a recovery process has received and handled all data units from alive 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 $fn_1, fn_2, ..., fn_F$ indicate the numbers of F failure

data processes and the rn_1, rn_2, ..., rn_F indicate the numbers of F parity processes chosen to be the recovery processes.

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

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 which 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.

4.1.3 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, the recovery process checks whether the data blocks needed for the value have been recovered. If the data blocks have not been recovered, 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, the recovery process calls the recovery function for them. After recovery, the recovery process handles the operation like a normal data process.

4.2 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, 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, 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.

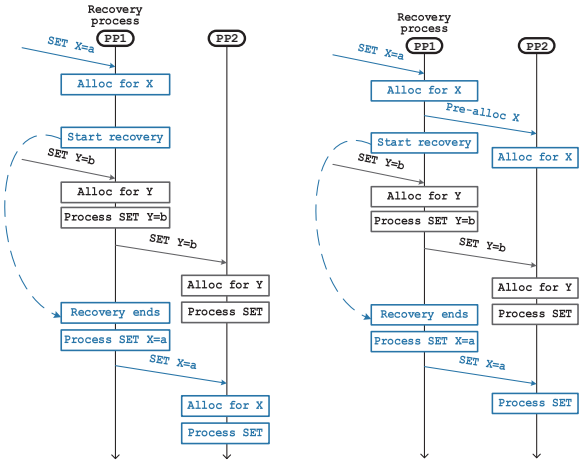
Parity process recovery: The parity process recovery is straightforward. After a parity process crashes, the data process marks all data blocks with a *miss* bit for that parity process. The data processes first send the metadata to the recovering parity process. Once the transfer of metadata completes, 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, the recovery fails and Cocytus starts to recover the failed data process.

5 Implementation

We first built a KV-store with Cocytus from scratch. To understand its performance implication on real KV-stores, we also implemented Cocytus on top of Memcached 1.4.21 with the synchronous model, by adding about 3700 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 which

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 write-intensive workloads, which we leave as future work. We use Jerasure [27] and GF-complete [26] 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 [33, 28]. 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.



(a) without pre-alloc (b) with pre-alloc

Figure 5: 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 procesess.

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 5(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, 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 5(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 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, 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 3 data blocks and 1 parity block, otherwise it should be recovered in the way that involves 2 data blocks and 2 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 [17] 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.

6 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.

6.1 Experimental Setup

Hardware and configuration: Due to our hardware limit, we conduct all experiments on a 6-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 5 out of the 6 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 that 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 [8] benchmark to generate our workloads. We generate each key by concatenating the 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 [10], 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 [24] 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, i.e., the field sizes are 0.1KB/0.4KB/1.6KB accordingly. To limit the total data size to be 64GB, the item numbers for 1/4/16 KB are 64/16/1 million respectively. However, due to the object compression, 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.

6.2 Memory Consumption

As shown in Figure 6, Cocytus achieves notable memory saving compared to PBR, due to the use of erasure coding. With a 16KB value size, Cocytus achieves 46%

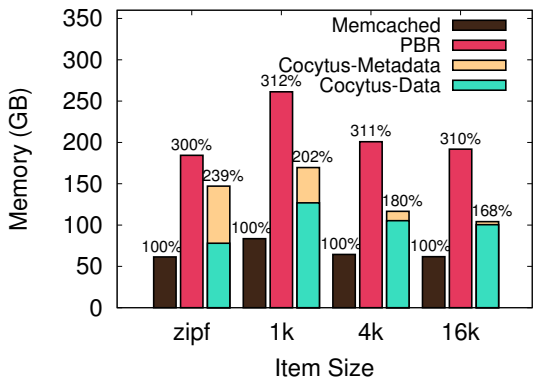


Figure 6: Memory consumption of three systems with different value sizes. Due to the compression in YCSB, the total memory cost for different value sizes differs a little bit.

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, e.g., 25%, 9.5% and 4% of all consumed memory for value sizes of 1KB, 4KB 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 6. 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.

6.3 Performance

As shown in Figure 7, 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 followings 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 similarly 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 *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.

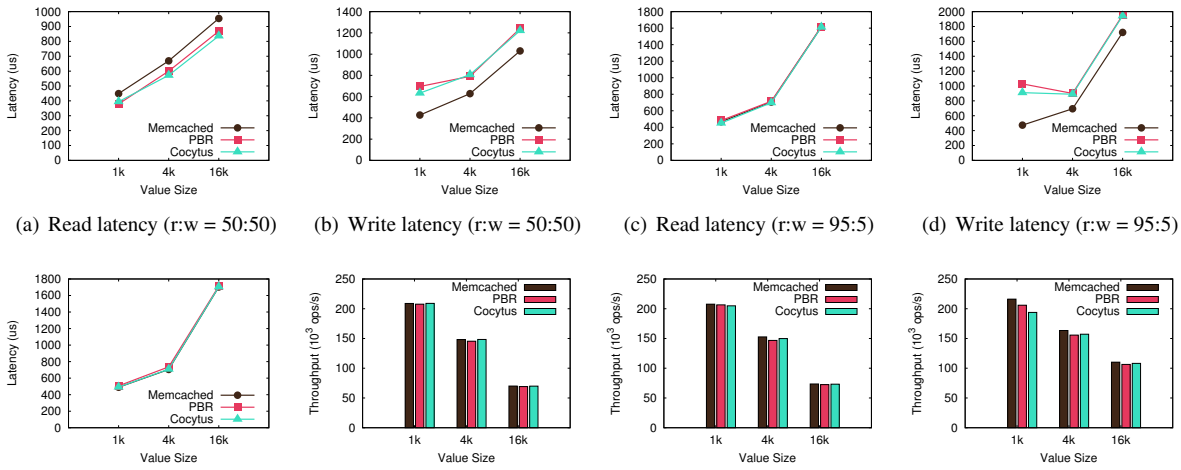
Lower read latency of PBR and Cocytus: There is an interesting phenomenon is that higher write latency causes lower read latency for PBR and Cocytus under update-intensive case (i.e., r:w = 50:50). This may be because when the write latency is higher, more client threads are waiting for the *set* operations 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 done faster because there would be fewer client threads that could block this operation. As a result, the latency of *get* is lower.

6.4 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 at 60s 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 8 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 8(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%, as shown in Figure 8(b). In the case with frequent *set* requests, as shown in Figure 8(c), the recovery affects the throughput of normal request handling modestly. The reason



(a) Read latency (r:w = 50:50) (b) Write latency (r:w = 50:50) (c) Read latency (r:w = 95:5) (d) Write latency (r:w = 95:5)

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

Figure 7: Comparison of latency and throughput of the three configurations.

Read : Write	Memcached	PBR		Cocytus	
	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

Table 2: CPU utilization for 1KB value size

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 8, we can observe that the throughput of the blocked version collapses even when there is only one node failure and 5% of *set* operations.

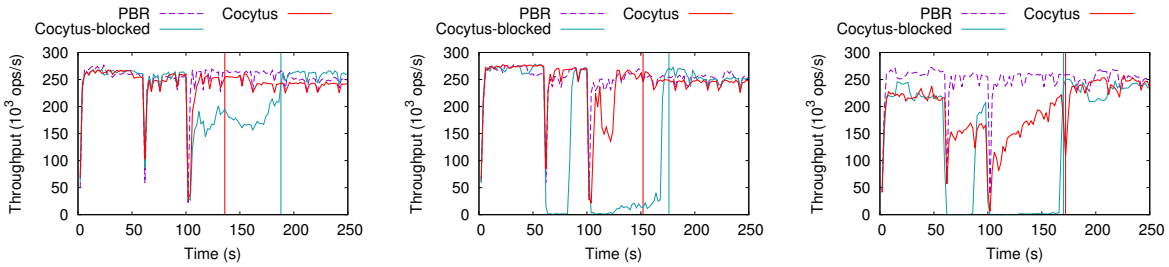
6.5 Different Coding Schemes

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

7 Related Work

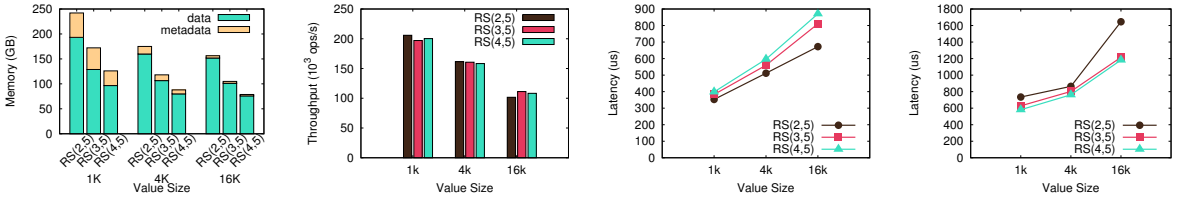
Separation of work: The separation of metadata/key and values is inspired by prior efforts on separation of work. For example, Wang et al. [42] separate data from metadata to achieve efficient Paxos-style asynchronous replication of storage. Yin et al. [46] separate execution from agreement to reduce execution nodes when tolerating Byzantine faults. Clement et al. [6] 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 [15, 34, 29, 32, 23]. Generally, they provide a number of optimizations that optimize the coding efficiency and recovery bandwidth, like local reconstruction codes [15], Xorbas [32], piggyback codes [29] and lazy recovery [34]. PanFS [44] is a parallel file system that uses per-file erasure coding to protect files greater than 64KB, but repli-



(a) Throughput on recovery (r:w = 100:0) (b) Throughput on recovery (r:w = 95:5) (c) Throughput on recovery (r:w = 50:50)

Figure 8: Performance of PBR and Cocytus when nodes fail. The vertical lines indicate all data blocks are recovered completely.



(a) Memory cost (b) Throughput (r:w = 50:50) (c) Read latency (r:w = 50:50) (d) Write latency (r:w = 50:50)

Figure 9: Performance under different coding schemes

cates metadata and small files to minimize the cost of metadata updates.

Replication: Replication is a standard approach to fault tolerance, which may be categorized into synchronous [5, 3, 39] and asynchronous [19, 2]. Mojim [48] 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 [25] exploits scale of clusters to achieve fast data recovery. Imitator [41] 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 [22, 36, 16, 43] or increasing concurrency [11, 20, 21]. 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.

8 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 [40, 7, 45]. Finally, we plan to investigate how to apply Cocytus to replication of in-memory databases.

Acknowledgment

We thank our shepherd Brent Welch and the anonymous reviewers for their constructive comments. This work is supported in part by China National Natural Science Foundation (61572314), 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), and Singapore NRF (CREATE E2S2). The source code of Cocytus is available via `http://ipads.se.sjtu.edu.cn/pub/projects/cocytus`.

REFERENCES

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, pages 53–64. ACM, 2012.

[2] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, 2011.

[3] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[4] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ACM SIGPLAN International Symposium on Memory Management*, pages 119–130. ACM, 2013.

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

[6] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290. ACM, 2009.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118. ACM, 2011.

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

[9] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 international conference on Management of data*, pages 1243–1254. ACM, 2013.

[10] L. Egghe. Zipfian and lotkaian continuous concentration theory. *Journal of the American Society for Information Science and Technology*, 56(9):935–945, 2005.

[11] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.

[12] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[13] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

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

[15] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. In *USENIX Annual Technical Conference*, pages 15–26, 2012.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 295–306. ACM, 2014.

[17] KLab Inc. http://repcached.lab.klab.org, 2011.

[18] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[19] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[20] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[21] R. Liu, H. Zhang, and H. Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC*, volume 14, pages 219–230, 2014.

[22] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

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

[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.

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

[26] J. S. Plank, E. L. Miller, and W. B. Houston. GF-Complete: A comprehensive open source library for Galois Field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, January 2013.

[27] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[28] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. 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*, pages 81–94. USENIX Association, 2015.

[29] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. *Proc. USENIX HotStorage*, 2013.

[30] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

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

[32] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, pages 325–336. VLDB Endowment, 2013.

[33] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Distributed storage codes with repair-by-transfer and nonachievability of interior points on the storage-bandwidth tradeoff. *Information Theory, IEEE Transactions on*, 58(3):1837–1852, 2012.

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

[35] SNIA. Nvdimm special interest group. http://www.snia.org/forums/sssi/NVDIMM, 2015.

[36] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *USENIX Annual Technical Conference*, pages 347–353, 2012.

[37] V. Technology. Arxcis-nv (tm): Non-volatile dimm. `http://www.vikingtechnology.com/arxcis-nv`, 2014.

[38] Twitter Inc. Twemcache is the twitter memcached. https://github.com/twitter/twemcache, 2012.

[39] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[40] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, pages 61–75, 2011.

[41] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. Replication-based fault-tolerance for large-scale graph processing. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 562–573. IEEE, 2014.

[42] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *USENIX Annual Technical Conference*, pages 413–424, 2012.

[43] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104. ACM, 2015.

[44] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.

[45] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 167–181. USENIX Association, 2015.

[46] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, pages 253–267. ACM, 2003.

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

[48] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18. ACM, 2015.