2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00091

F-Write: Fast RDMA-supported Writes in Erasure-coded In-memory Clusters

Bin Xu, Jianzhong Huang*, Qiang Cao*
Huazhong University of Sci.& Tech
Wuhan,Hubei,China
{binxu,hjzh,caoqiang}@hust.edu.cn

Xiao Qin
Auburn University
Auburn, AL 36849, USA
xqin@auburn.edu

Ping Xie
Qinghai Normal University
Xining,Qinghai,China
xieping@qhnu.edu.cn

***Abstract*—To satisfy high reliability accompanied by space efficiency requirements, erasure coding is elected to substitute replication as a redundancy mechanism of in-memory clusters. More often than not, the use of erasure coding is limited to read-intensive applications, in which data in erasure-coded clusters are rarely updated. The essential rationale is that update penalty incurred by parity-synchronizations makes long write latency compared with read latency counterpart.**

In this paper, we propose F-Write: a fast RDMA-supported write optimization scheme for erasure-coded in-memory clusters. It entails two distinct features: 1) an extended version of consistency protocol called Fast2PC is created. It directly modifies remote memory regions using one-sided WRITE verb provided by RDMA to implement the operations of transaction log, and records multiple transactions in the log to submit together, effectively curtailing network latency; 2) a speculative update approach is given to substitute immediate update. Aggregated undo transactions are handled speculatively to synchronize parity blocks in the background when parity blocks are needed. For multiple writes to an identical data block at different timestamps, only the original and the latest data blocks are involved in calculating parity blocks, thus mitigating encoding latency. Experimental results indicate that F-Write has lower latency, higher throughput compared to the candidate write schemes. Moreover, the impact on recovery time is negligible. Specifically, under the update-intensive workloads, F-Write cuts down write latency by more than 61%, thereby boosting system throughput by a factor of at least $2.6\times$.

***Index Terms*—In-memory cluster, Erasure encoding, Remote direct memory access, Write optimization**

I. Introduction

Thanks to high-performance random accesses, in-memory computing has been applied to a growing number of data-intensive applications to fulfill low-latency and high-throughput requirements [1] [2]. Owing to limited memory capacity (e.g., tens to hundreds GB) of a single node, there is a pressing demand to build in-memory clusters to cope with large-scale data computing scenarios. Data loss and temporary unavailability inevitably occur in clusters, replication is commonly deployed to guarantee high data availability but suffering excessive storage cost. Erasure coding (EC) is becoming an attractive alternative which exhibits high space efficiency offering even better fault tolerance. Currently, EC has been adopted by in-memory key-value stores, such as Cocytus [3], MemEC [4], and Ring [5].

Although EC is widely adopted in the companies (e.g., Google, Facebook, and Microsoft) bringing down storage cost [6] [7] [8], application scenarios of EC are surprisingly limited. Data maintained by EC is meant to be modified infrequently, because write latency is a whole lot longer than read latency in erasure-coded clusters. The write penalty mainly comes from the parity-synchronization which consists of two parts: 1) multiple additional reads are involved to encode new parity blocks, the process of which induces local CPU and I/O latency. 2) governed by complicated distributed protocols to maintain stripe consistency, multiple round-trip communications give rise to long network latency.

In the era of big data, users' access patterns are dynamically changing. In the cloud systems, the percentage of write operations is gradually increasing [9] [10]. Regardless of write types, write penalties impose an adverse impact on performance, hindering the widely use of EC. Therefore, cutting down write penalties becomes an urgent demand in developing modern erasure-coded systems. Enormous researches have been devoted to reducing local I/Os [11] [12] and considering non-uniform network bandwidths [13] [14]. However, for erasure-coded in-memory clusters currently built in the high-performance networks, they are no longer major concerns. Rather, both multiple round-trip communications and encoding latency issues become critical challenges to be addressed.

In pursuit of more extreme performance, Remote Direct Memory Access (RDMA) is becoming an attractive technology which can bypass the kernel to suppress data copies. Distinctively, RDMA provides unique one-sided verbs to directly read or write remote memory regions without involving remote CPUs. Inspired by the exquisite features of RDMA, we propose *F-Write* - a fast write optimization scheme - for erasure-coded in-memory clusters. F-Write embraces a protocol called Fast2PC, an extension of the traditional 2PC. Compared to 2PC forcing participants to reply statuses after receiving and processing requests, Fast2PC uses WRITE verbs to directly modify remote transaction logs and polls the statuses of verbs from the local NIC, thereby reducing network communications; meanwhile, given parity blocks not servicing read requests generally, Fast2PC records multiple synchronization transactions in the parity nodes' logs temporarily. F-Write applies speculative updates rather than immediate updates and aggregated undo transactions are submitted together at the appropriate time and handled in the background. A merge algorithm is leveraged to consolidate multiple updates of the

978-1-6654-4066-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00091

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.

identical data block modified at different timestamps. Parity increment is calculated only using the original and the latest data blocks based on the linearity of erasure codes, reducing encoding latency. When failures occur, a log tail checking phase is added before the reconstruction phase to guarantee stripe consistency.

We implement a F-Write prototype that applies one-sided verbs into erasure-coded in-memory clusters. The prototype is built on top of the InfiniBand (IB) network and can handle basic read, write and failure recovery. For the purpose of fair comparisons, we also implement traditional write schemes using TCP/IP sockets and two-sided RDMA verbs. Through multiple groups of experiments, we demonstrate that F-Write is adroit at shortening write latency and improving system throughput. Furthermore, evidence shows that F-Write has little adverse impact on recovery time.

The contributions of the paper are summarized as follows.

- We propose *F-Write* - a fast RDMA-supported write optimization scheme - for erasure-coded in-memory clusters. A distinctive protocol called Fast2PC is devised which leverages WRITE verbs to simplify the communication process and consolidates multiple transactions to submit together, reducing network latency.
- We apply the speculative update and give some timings of submission to mitigate encoding latency. Meanwhile, a merge algorithm is included to merge multiple writes to an identical data block, the parity nodes calculate parity increments only using the original and latest data blocks.
- We implement a F-Write prototype in an erasure-coded in-memory cluster. Then, we verify the effectiveness of F-Write and conduct sensitivity analyses.

The rest of the paper is organized as follows. Section.II outlines the background and motivation. The design of F-Write is detailed in Section.III including the Fast2PC protocol and speculative update. We present quantitative evaluations and analyses in Section.IV. Finally, Section.V gives some related works and Section.VI concludes this paper.

II. Background and Motivation

A. Erasure-coded Clusters

Failures commonly triggered by hardware or software errors, power outrages, or the like in clusters may cause data loss. Replication, relatively simple, is adopted to prevent it. A N-way replication distributes N copies of data on N nodes, which tolerates $N-1$ faults suffering high storage overhead. For example, the space efficiency of triplication is $1/3 = 33.3\%$. In clusters equipped with expensive media such as DRAM or NVM, EC is an attractive alternative to slash cost by the virtue of high space efficiency [3] [4] [5]. There are two configurable parameters in an erasure-coded scheme, k and r. A erasure-coded stripe contains k data blocks and r parity blocks which are calculated by an encoding procedure using data blocks. They are distributed to $(k+r)$ nodes to accomplish fault tolerance. Any k out of total $(k+r)$ blocks are enough to recover the original data. Under the $(6,4)$ scheme, the space efficiency reaches $4/6 = 66.7\%$

When any data block in a stripe are updated, all the parity blocks in the stripe will have to be recalculated (i.e, parity synchronization) to furnish stripe consistency. For the most linear codes, there is no need to recalculate new parity blocks through encoding all the data blocks. After obtaining a product of the data block increment and the corresponding predefined coefficient, the product is added to each old parity block to obtain a new parity block, thus reducing I/O and network overheads. This method is referred to as *incremental update*, which is more suitable of small-write intensive workloads. Its correctness has been proven in numerous systems [11] [12] [15].

B. Issues of Long Write Latency

In erasure-coded clusters, write latency - being considerably larger than read latency - severely limits the widely use of EC. In data centers such as those operated by Google and Facebook, EC is applied to store warm/cold data rather than hot data; the rationale behind this common practice is that warm/cold data are hardly modified in the life cycle [16] [17].

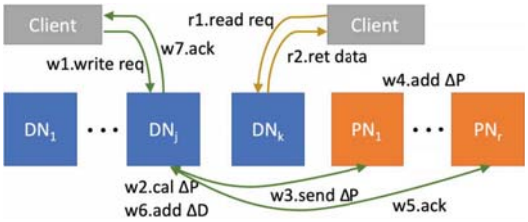


Fig. 1. A normal-read and a normal-write process in an erasure-coded cluster.

As shown in Fig.1, a client first issues a request to a node where target data are located. For a read request, the target node returns requested data to the client simply. For a write request, the target node - acting as a coordinator - embarks on the synchronizations among all the parity blocks. Within the parity-synchronization process, the coordinator first reads old data blocks and calculates parity increments delivered to corresponding parity nodes, and then each parity node reads the old parity block and adds the corresponding parity increment to it, returning a response to the coordinator. Upon receiving all the responses, the coordinator updates its local data block, followed by sending back an acknowledgement (ACK) to the client. Such a parity-synchronization process is the root cause of massive write latency that entails both calculations and multiple round-trip network communications for consistency.

C. Consistency Protocol

The parity-synchronization process is a typical distributed transaction scenario which should be guaranteed to be atomic. It means that the data block and all the parity blocks should succeed in being updated together. In case of a synchronization failure, all the parity nodes will force to cancel the parity-synchronizations, and therefore, the local data block is also unable to be updated. If breaking the consistency of data blocks and parity blocks in a stripe, unexpected data will

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.

be reconstructed. Numerous protocols has been put forward to handle distributed transactions. Two-phase commit protocol (2PC) [18] is the most commonly used one thanks to its simplicity. However, it is plagued by long network latency because of the requirement of a round-trip network communication in both preparation and submission phases (more details in Section.III-B1). Many versions of 2PC were proposed to alleviate its lacks [3] [19]. It should be noted that some other popular consistency protocols such as Paxos and Raft are commonly used to ensure consistency among multiple replications of the same data partition in primary backup systems. Therefore, they are not suitable for parity-synchronizations in erasure-coded systems where data blocks are unequal to parity blocks.

D. Opportunities Brought by RDMA

With the development of high-performance networks, *RDMA* is gradually becoming a promising technology and widely used in in-memory clusters. Bypassing the operating system kernel and fulfilling the transport in NIC, RDMA facilitates low-latency and high-throughput memory accesses with little CPU involvement on hosts. Similar to TCP/IP, RDMA provides SEND and RECV - referred to as two-sided verbs appeared in a pair. Additionally, RDMA furnishes unique one-sided verbs such as READ and WRITE. With one-sided verbs in place, applications are enabled to directly read or write remote registered memory regions without the participation of remote CPUs. Each RDMA connection is represented by a set of work queues called queue pair (QP), which is equivalent to a socket connection in TCP/IP. Each QP includes a send queue and a receive queue, both of which are maintained by NIC. When a verb is initiated by an application, it is wrapped into a work queue element (WQE) sent to the queue associated with NIC, where SEND, WRITE and READ verbs are placed into send queue and RECV verb is delivered to receive queue. NIC asynchronously executes WQEs in the queue. To notify the completion of verbs, each QP is connected with a completion queue (CQ). Once completed, the verb's completion status is packaged into a completion queue element (CQE) pushed into CQ. Then, the application polls the CQE from the corresponding CQ to gauge the verb's status.

The unique one-sided verbs and working mechanism bring opportunities to streamline and simplify consistency protocols. For example, to modify a remote memory region via a remote procedure call, a local host first issues a request to a remote host through SEND, then the remote host receives and processes the request through RECV. After modifying the requested memory, the remote host replies a response to the local host through SEND. When the local host acquires the response through RECV, the latency of process is measured as one round-trip time (RTT). The functions of SEND and RECV can be substituted by WRITE in RDMA-supported clusters, the local host makes use of WRITE to directly modify the remote memory, then polling the local completion queue to obtain the verb's status. The process causes little CPU overhead and its latency is curbed to a half RTT. The idea can also be adopted to operate remote transaction logs of 2PC in in-memory clusters.

III. Design of F-Write

A. Overview

To tackle the problem of excessive write latency in erasure-coded in-memory clusters, we propose a RDMA-supported write optimization scheme called F-Write, which is adept at optimizing write performance from the angles of consistency protocol and computation mitigation.

Parity-synchronization is a typical distributed transaction scenario, where 2PC is commonly adopted. Although 2PC is simple to implement, it is mandatory to generate at least two round-trip network latency. F-Write utilizes one-sided WRITE verbs rather than SEND and RECV verbs to add and revoke transaction logs; F-Write records multiple transactions in the log to submit together. Such a new RDMA-supported protocol is referred to as *Fast2PC*.

In erasure-coded systems, parity blocks are only accessed during degraded reads or reconstructions. Therefore, the synchronization calculation can be performed at an appropriate time speculatively, using aggregated undo transactions in Fast2PC. A speculative update mitigate encoding latency during write requests. Meanwhile, similarly to PARIX [12], F-Write applies the linearity of erasure codes to reduce calculations, suppressing CPU overhead.

B. Fast2PC

Fast2PC - an RDMA-supported distributed protocol - entails two prominent changes over the conventional 2PC: a) it replaces two-sided SEND and RECV verbs with one-sided WRITE verb to implement remote memory writes; b) it submits a single consolidated transaction aggregated from multiple synchronization transactions.

1) 2PC: it is envisioned as the premise foundation of Fast2PC. 2PC has two key roles - coordinator and participant. Each participant has a log to record transactions. As to parity-synchronization in erasure-coded clusters, a data node is dedicated as the coordinator, and the corresponding parity nodes serve as the participants. 2PC includes two phases, namely, the preparation and submission phase. As shown in Fig.2, in the preparation phase, the data node originates a *Prepare* request delivering the synchronization data to each parity node, each parity node accepts and adds the transaction to its local log, followed by passing a *Vote* response carrying a self-generated state. In the submission phase, the data node makes a *Commit* or *Rollback* request to each parity node if all Votes are YES or there is a NO Vote. After all the parity nodes accept and commit/retract the synchronization data in the logs, parity nodes reply final ACK responses. In a nutshell, 2PC is a simple yet effective protocol maintaining consistency of data blocks and parity blocks in a stripe.

2) Optimizations: The first optimization of Fast2PC benefits from the feature that one-sided verbs offer direct read or write to remote memory regions. As exemplified in Section II-D, the modification of transaction logs can be

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.

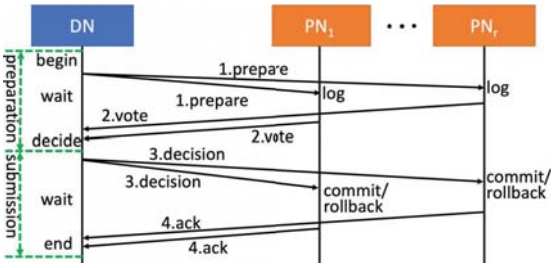


Fig. 2. The detailed process of 2PC.

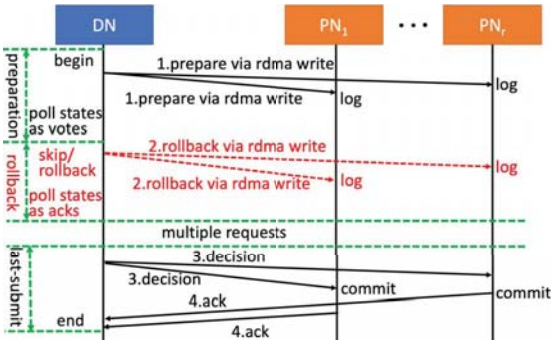


Fig. 3. The detailed process of Fast2PC.

implemented via WRITE verbs. Fig. 3 demonstrates that in the preparation phase, the data node directly writes *Prepare* requests to parity nodes' transaction logs with WRITE, polling its local completion queue to obtain WRITEs' states as *Votes*. If a *Rollback* is decided, the data node triggers WRITE again to remove the written transaction in each parity node, then polling its local completion queue to obtain states as ACKs. Otherwise, the preparation phase ends. This optimization can shortens the latency of preparation phase significantly.

The second optimization of Fast2PC lies in the submission phase. Remote CPUs do not perceive WRITE verbs in RDMA technology. Thus, SEND and RECV verbs are still required to notify remote CPUs to process transaction logs. The submission phase leads to non-negligible latency (i.e., 1RTT) in 2PC. Instead of submitting each transaction immediately, Fast2PC aggregates multiple transactions in a way that a consolidated single transaction is submitted to cut back latency. Fig. 3 shows that after the preparation phase, the parity-synchronization process for each write ends and data node can update local data blocks and response to clients. It is ensured that logs kept in the parity nodes are in a consistent state before the last-submit phase. After a few consecutive write requests, the last-submit phase is invoked at an appropriate time, then SEND and RECV verbs are employed to notify participants to submit undo transactions together. The timing of submission will be given in SectionIII-C2

3) Complexity Analysis: We put 2PC and Fast2PC under the microscope to analyze performance in term of time and message complexity. The time complexity is gauged in form of latency caused by message transfer, whereas the message complexity is measured by the number of passed messages, because a message is usually short. In Table.I, n indicates the number of requests, m is the number of participants and α is the probability of rollbacks. Each 2PC results to $2RTT$ latency, so the time complexity of 2PC is $2nRTT$. In the preparation phase, the coordinator sends m *Prepare* requests and receives m *Votes*. In the submission phase, the coordinator sends m *Commit/Rollback* requests and receives m ACKs, so the message complexity of 2PC is $4mn$.

The preparation phase gives rise to $1/2$ RTT for each write without rollbacks. Considering the probability of rollbacks, for n requests, $0.5n*(\alpha+1)RTT$ is measured prior to the last-submit phase; the submission in Fast2PC is identical to that in 2PC, which causes $1RTT$. The overall time complexity of Fast2PC is $0.5n*(\alpha+1)+1RTT$. In the preparation phase, the coordinator originates m *Prepare* requests. Then, the coordinator invokes m *Rollback* requests according to the probability α of rollbacks. After n write requests, the coordinator sends m *Commit* requests and receives m ACKs in the last-submit phase. That is, the message complexity of Fast2PC is $(1+\alpha)*mn+2m$. More often than not α is small, theoretically speaking, Fast2PC slashes the time and message complexities of 2PC by approximately 75%.

TABLE I
COMPLEXITY COMPARISONS BETWEEN FAST2PC AND 2PC.

Protocol	Time complexity	Message complexity
2PC	$2nRTT$	$4mn$
Fast2PC	$0.5n*(\alpha+1)+1RTT$	$(1+\alpha)*mn+2m$

4) Transaction Log: In in-memory clusters, a contiguous memory region in each storage node is dedicated to recording transactions which is called transaction log. During the RDMA initialization phase, an application is obligated to create a protection domain, register a contiguous memory region as a log, and then bind a queue pair to the registered region. In Fast2PC, the data node leverages WRITE verbs to add transactions into the log, which are packaged with synchronization data. Since WRITE verbs are polled from the send queue and executed by NIC in a FIFO manner, the order of transactions preserved in the log matches the order of writes. It means new data will not be overwritten accidentally.

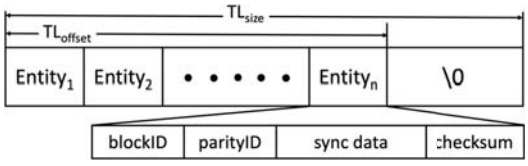


Fig. 4. The structure of a transaction log.

Fig.4 shows the structure of a transaction log, each entity documents information relevant to parity-synchronization of a write request. An entity contains data block ID, parity block ID, synchronization data and checksum. Data block ID and parity block ID respectively identify the data block and the

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.

parity block need to be synchronized; the synchronization data is used to calculate new parity block; the checksum furnishes integrity checking for entities. The data node maintains a variable TL_{offset} representing the length of written entities. It determines the starting address to next write, and is passed as a parameter each time when a WRITE verb is executed. After each verb is successfully executed, TL_{offset} is updated to the length of current entities. Another variable TL_{size} denotes transaction log size. The last-submission will be triggered when TL_{offset} approaches TL_{size} indicating that the space of transaction log is insufficient. TL_{size}, of course, can be readily configured to record enough entities according to various application scenarios. Data blocks are usually small in in-memory clusters (e.g., the block size in MemEC is 4KB [4]). In this paper, it is set to a default value of 8MB.

5) Failure Recovery: The failure handling methods in various scenarios adopting 2PC have been proposed to ensure consistency after recovery [20]. As an extension of 2PC, Fast2PC can inherit 2PC's solutions natively. Unfortunately, the use of WRITE verbs may cause hurdles [21]. As shown in Fig.5, failures may occur when a data node is executing WRITE verbs, the transaction logs in the parity nodes may have three states: 1) The transaction is completely written. 2) The transaction is partially written. 3) The verb is still in the send queue and the transaction has not yet been written. If the reconstruction is directly accomplished when participants keep an inconsistent state, unexpected data may be recovered.

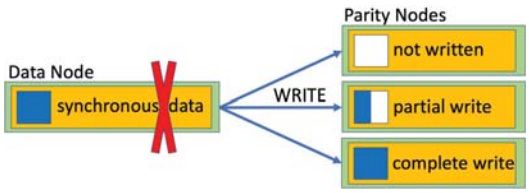


Fig. 5. Inconsistent states of parity nodes in case of the failure of WRITE.

To address the above-mentioned case, we advocate for an approach to unifying the states of participants. More specifically, the following four pre-process steps should be employed. 1) An application selects a new node as a reconstructed node, which locates parity nodes and sends *Commit* requests. 2) Each parity node is reliant on checksum to check the tail entity in the log, deletes the incomplete data, and delivers the valid length of transaction log to the reconstructed node. 3) After accepting the valid lengths sent by all the parity nodes, the reconstructed node decides the smallest as a unified valid length, which is sent to each parity node. 4) Each parity node submits the transaction entities in the valid range to synchronize parity blocks and sends ACK to the reconstructed node, then the reconstructed node embarks on the reconstruction phase, after receiving all the ACKs.

Checking integrity and unifying the states of parity nodes tend to bring forth an additional burden on recovery time. Nevertheless, network traffic caused by the pre-process steps is fairly small and; therefore, the above four steps have little adverse impact on the overall recovery time (more details in Section.IV-C1).

C. Speculative Update

1) Art of Speculation: In erasure-coded clusters, data blocks preserve raw data serving normal reads that dominate the majority of accesses; parity blocks - stored to ensure data availability - are infrequently accessed except for degraded reads or reconstructions. Such an access pattern gives ample opportunities for speculative update. As mentioned in the second optimization of Fast2PC, instead of submitting each transaction immediately, parity node just records transaction entities of multiple writes in the log and submits them together speculatively at an appropriate time. The old parity blocks will be synchronized using aggregated transaction entities in the background. For a single write request, the data node can response to clients after the preparation phase ends, so speculative update can hide the encoding latency in the parity node and the latency of submission phase. Furthermore, the data node judiciously packages both original and updated data blocks into synchronization transactions which are then sent to each parity node. As a result, the calculation of the parity increment in the data node can also be skipped.

2) Timing of Submission: Recall that parity blocks are accessed in the cases of degraded reads or reconstructions. It means that before degraded reads or reconstructions, parity blocks should be synchronized to the latest to avert decoding unexpected data through submitting aggregated transactions passively. Intuitively, the synchronization prior to decoding not only slows down the process of degraded reads or reconstructions, but also accumulates numerous transactions in the log to enlarge subsequent latencies. To alleviate this problem, we introduce active submissions - a trade-off between immediate and passive submission. Three conditions are given as below.

- Passively submit transactions prior to degraded reads or reconstructions.
- The space of log is limited. When the size of written contents reaches a pre-defined threshold, the transactions will be actively submitted.
- When nodes are in an idle state. An interval event is triggered to actively submit transactions.

3) Background Synchronization: When speculative update is triggered, the log is monopolized until transaction entities are processed, so subsequent batch of write requests will await to write entities in the log. In order to alleviate wait latency, a simple producer-consumer model is illustrated in Fig.6 through copying logs during the background synchronization process.

The model mainly orchestrates three components, namely, a data receiver, a transaction log queue, and an update-thread group. 1) The data receiver acts as a producer, which receives synchronization data and listens for submissions from data nodes. After receiving a submission, the parity node copies a transaction log as production data and resets the original log. A parity node accepts messages from multiple data nodes corresponding to multiple queue pairs (QPs), in which each

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.

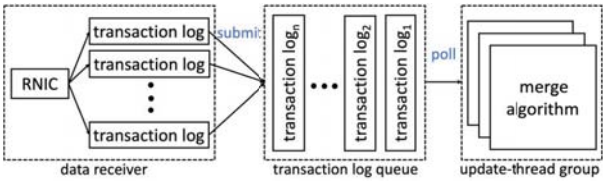


Fig. 6. The producer-consumer model for background synchronizations.

QP binds a registered memory region used as a transaction log. 2) The transaction log queue is focused on storing production data. The data receiver appends a copy of the transaction log to the queue's tail. 3) The update-thread group performs as a consumer. If the transaction log queue is non-empty, the consumer polls transaction logs from the queue to work. The consumer contains one or more update threads, which are responsible for processing transaction logs in parallel by using the merge algorithm elaborated in Section III-C4.

4) Merge Algorithm: A PARIX-like merge algorithm is applied to speed up the background synchronization [12]. For a single-write request to data block D_i, a new parity block (e.g., P_j) can be obtained via formula $P_j^{'} = P_j + \alpha(D_i^{'} - D_i)$. For u-time-write requests to data block D_i, new parity P_j is expressed as formula $P_j^u = P_j^0 + \alpha(D_i^1 - D_i^0) + \alpha(D_i^2 - D_i^1) + ... + \alpha(D_i^u - D_i^{u-1})$, where a value in set {0,1,2,...,u} is regarded as an update timestamp. The formula is further simplified as $P_j^u = P_j^0 + \alpha(D_i^u - D_i^0)$. Therefore, for multiple updates of an identical data block, only the original and latest data block need to be retrieved from the transaction log for calculations. Compared to calculating the parity increment for each update, this method conserves CPU resources and shortens encoding latency. In the worst case, a one-shot write may lead to extra network overhead for transferring the original data block. Nevertheless, such an overhead is negligible in the high performance networks.

Algorithm 1: merge algorithm

Input: TL - a transaction log
Output: None

initialize two tables $T_{original}$ and T_{latest}, with mapping $blockid$ to $Blk_{original}$ and Blk_{latest};
$Offset = 0$;
while $OffSet < sizeof(TL)$ **do**
  parse a transaction $Entity$ from $Offset$;
  if $T_{original}.get(Entity.blockid) == NULL$ **then**
    $T_{original}.put(Entity.blockid, Entity.data)$;
  else
    $T_{latest}.put(Entity.blockid, Entity.data)$;
  $Offset += sizeof(Entity)$;
foreach $blockid \in T_{original}/T_{latest}$ **do**
  $Blk_{original} = T_{original}.get(blockid)$;
  $Blk_{latest} = T_{latest}.get(blockid)$;
  $\Delta_{parity} = \alpha(Blk_{latest} - Blk_{original})$;
  $Parity_{new} = \Delta_{parity} + Parity_{old}$;
clear the tables $T_{original}$ and T_{latest};
free the TL;

Algorithm 1 outlines the pseudo-code of merge algorithm which retrieves the original and latest data blocks from the transaction log for multiple updates with respect to an identical data block. The algorithm consists of several steps:

- Initialize two tables containing mappings of the block id to the original and latest data block respectively;
- Iterate the transaction log, parse transaction entities for each write, and query the table according to the block id. If the original block is null, set the data from entity as the original one; otherwise, set it as the latest one;
- Traverse the tables and obtain the original and latest block according to block ids. Then, calculate the parity increments using two data blocks, and add the parity increments to old corresponding parity blocks;
- Clear tables and free the transaction log after the synchronization is completed.

IV. Experiments and Results

A. Prototype Design

To verify the effectiveness of F-Write, we implement a F-Write prototype with approximately 3500 lines of C programming code. It supports basic read, write and recovery functions. For the fair-comparison purpose, traditional write schemes implemented using TCP/IP sockets and two-sided RDMA verbs are incorporated into the prototype.

The Fig.7 shows the overall architecture of the prototype. A classic master-slave pattern is employed to facilitate horizontal expansion. The coordinator includes a metadata manager manipulating the metadata of blocks, stripes, nodes, and others in a cluster, and a heartbeat monitor gauging the states of storage nodes. Each storage node supplies pre-allocated memory used for storage service. Additionally, each storage node is deployed with a transaction module that takes consistency protocols to ensure atomicity of distributed transactions, and a recovery module applied to perform reconstruction tasks. The client is used to replay traces, imitating concurrent user accesses. It first queries the coordinator for location information about blocks and stripes. Then, it sends requests to the target node and wait for responses. For a read request, the storage node directly return the requested data. For a write request, with the help of transaction module, the storage node updates local data after succeeding in updating all the parity blocks, followed by responding to clients.

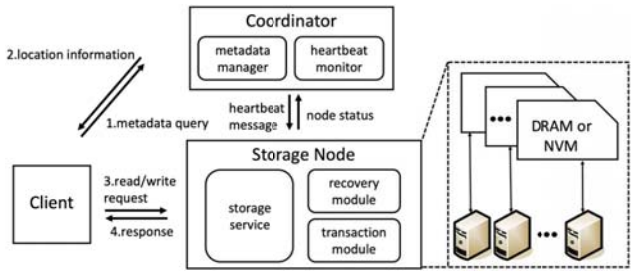


Fig. 7. The overall architecture of the F-Write prototype.

In the prototype, RDMA communication is asynchronous based on event mechanism, so I/O multiplexing-epoll is

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.

used as the communication mechanism of TCP/IP. One-sided WRITE verb only works in the reliable connection mode, so all the communications choose the reliable connection as transport mode. In order to foster adaptability and flexibility, we advocate for the native InfiniBand verbs and RDMA management libraries offered by Mellanox to implement the communications among nodes [22]. The efficient Intel's ISA-L is chosen as the RS encoding library, which optimizes the encoding at the machine instruction level [23].

B. Experimental Environment

1) System Setup: Our prototype system is built upon a high-performance InfiniBand network. It is comprised of a client, a coordinator node, and seven storage nodes. All the nodes are equipped with the Mellanox adapters and connected through a Mellanox switch. The client and coordinator are Dell R720 servers, each of which is equipped with 2 Xeon. E5-2609 @2.40GHz CPUs and 4GB DDR3 memory. Each storage node contains an Intel.i3-2120 @3.30GHz CPU and 32GB DDR3 memory. On the servers we install Ubuntu16.04 (x86-64) in which the kernel version is 4.4.0-62-generic.

2) Testing Methodology: Table.II lists three write schemes for comparisons. Both BRW and BTW endorse 2PC as the consistency protocol, along with which two-sided verbs (i.e., SEND and RECV) are incorporated to implement parity-synchronizations. They adopt immediate updates, where parity blocks are immediately updated in a write request. It is intuitive that BRW achieves higher performance than BTW thanks to low-latency transmission provided by RDMA. Our F-Write adopt Fast2PC which utilizes the one-sided WRITE verbs in a full swing to synchronize parity blocks. It temporarily records undo transactions in the log for a consolidated submission at an appropriate time, which are also known as speculative update. It is expected that these improvements make F-Write exceed BRW in term of write performance.

TABLE II
COMPARISONS OF TESTING SCHEMES.

scheme	protocol	communication	update method
F-Write	Fast2PC	one-sided verb	speculative update
BRW (basic rdma write)	2PC	two-sided verb	immediate update
BTW (basic tcp write)	2PC	two-sided socket	immediate update

We use workloads generated from the YCSB benchmark [24] to evaluate the three schemes. We focus on update-intensive workloads. The number of unique data blocks is 5K; the client issues total 500K requests; and the access pattern follows a *zipf* distribution with a high shape parameter μ.

In the validation tests, we set up the default configuration as follows. a) RS parameters: $k = 4$, $r = 2$; b) the distribution parameter $\mu = 0.8$; c) data block size is 4KB; d) transaction log size is 8MB; e) when the trace is replayed to 100K, the coordinator stops processing the subsequent trace; then a storage node is disabled to mimic failures, the system kicks in the recovery process.

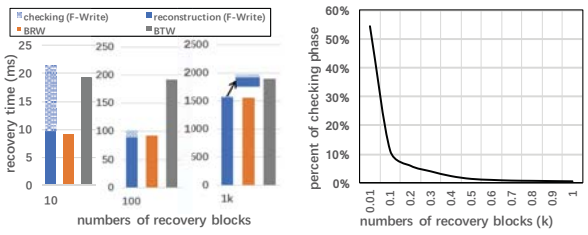
Our primary metrics for performance comparisons are latency, throughput and recovery time. The latency reflects the processing efficiency of a single write request, whereas the throughput measures the overall processing capacity of a system. A short recovery time means that a storage system exhibits high reliability. The additional checking phase prior to the reconstruction phase in F-Write enlarges latency to some extent and hence, we ought to verify that the adverse impact of F-Write on reliability is marginal and desirable.

C. Experimental Results

1) Validation Tests: Under the default configuration, a group of experiments is conducted to examine the validness of F-Write. The results are presented in terms of latency, throughput and recovery time.

Improvement of Performance - Figs.8(a), 8(b), and 8(c) show median, $99th$ percentile, and average write latencies as functions of the number of clients. Among all the tested schemes, F-Write has the lowest median, $99th$ percentile and average latencies. Because Fast2PC adopts one-sided verbs to implement the operations of transaction log and aggregates multiple transactions to submit together, effectively mitigating network latency; and aggregated undo transactions are processed speculatively in the background to skip partial encoding latency. Regardless of tested schemes, the latency continues to surge along with an increasing number of clients. This trend is attributed to that when the capacity of system fails to keep the pace with the increasing concurrency degree, an excessive number of requests cannot be processed immediately. In particular, BTW has the largest increment. Because F-Write and BRW schemes offload transport to NIC and bypass the kernel, achieving excellent transmission performance. More system resources can be used to service requests.

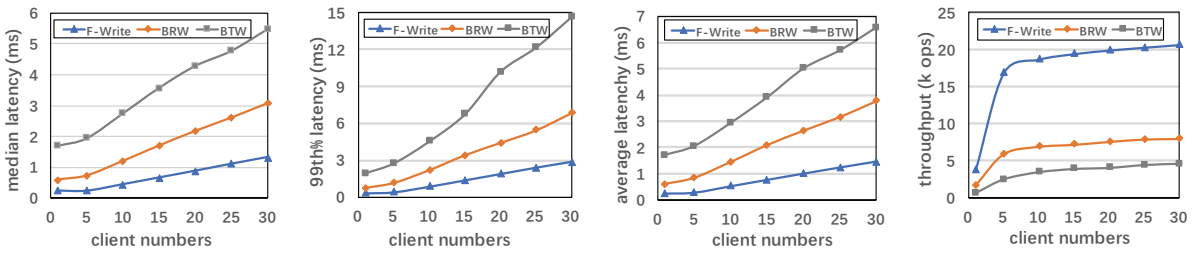
Fig. 8(d) shows the impacts of the number of client on throughput. Compared with two candidate schemes, F-Write exhibits the highest throughput performance. Furthermore, the system enjoys an upward trend in throughput readings, followed by a flattened progression. This trend is expected, because the system has not yet reached the pressure limit in an initial stage of pushing the concurrency degree high. This set of experiments are evident that F-Write greatly enhance the write performance in erasure-coded in-memory clusters. Specifically, F-Write improves system throughput by $2.6\times$ and $4.5\times$ and curbs write latency by 61% and 78%, compared against two candidate schemes.



(a) Recovery Time (b) Impact for Recovery in F-Write

Fig. 9. Recovery time comparisons of the three write schemes using the default configurations. Parameters k, r, μ, S_{blk} and S_{log} are set to 4, 2, 0.8, 4KB, and 8MB, respectively.

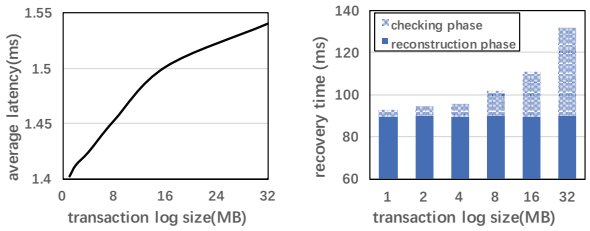
Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.



(a) Median Latency (b) 99th Percentile Latency (c) Average Latency (d) Throughput

Fig. 8. Performance comparison of the three write schemes using the default configurations. Parameters k, r, μ, S_{blk} and S_{log} are set to 4, 2, 0.8, 4KB, and 8MB, respectively.

Impact for Recovery - Besides optimizing performance, ensuring reliability is equally important. Fig. 9(a) shows the time spent in recovering 10, 100 and 1k data blocks governed by all the tested schemes. We can observe that the recovery time of F-Write is slightly higher than that of BRW, because Fast2PC needs to check and unify the states of transaction log in each parity node, then submit to synchronized all the parity blocks before reconstructions, resulting in extra latency. The recovery time of F-Write is lower than that of BTW thanks to low-latency transmission provided by RDMA. Moreover, as to the number of data blocks to be recovered increases, the proportion of checking phase time to total recovery time decreases. Because more recovery blocks lead to longer reconstructions phase naturally, but the checking phase is not affected by the number of recovery blocks, which brings forth the same latency. Fig.9(b) unravels a more detailed trend of the proportion with respect to the number of recovered data blocks. We can find that when the number of recovery blocks increases gradually, the proportion of checking phase shrinks to a small value. For example, when the number of blocks is 300, the proportion is 4.1%. When the number of blocks is 1000, the proportion shrinks to merely 0.65%. In a production environment, the number of data blocks in a failed node is usually large. Therefore, we conclude that F-Write makes a negligible impact on system reliability.



(a) Average Latency (b) Recovery Time

Fig. 10. Comparisons of the three write schemes using the default configurations. Parameter S_{log} is set to $\{1, 2, 4, 8, 16, 32\}MB$.

2) Sizes of Transaction Logs - S_{log}: Fig. 10(a) unveils the trend of average latency as the size of transaction log increases. We can find that an increasing value of S_{log} gives rise to enlarged write latency of F-Write. Such a performance degradation is attributed to the fact that a large transaction log leads to a huge overhead in registering and copying data in the adapter, thereby deteriorating the overall write latency. This phenomenon is consistent with the findings reported in a prior study [25], in which RDMA is considered to be suitable for applications exhibiting long connections and small I/Os. Additionally, Fig. 10(a) shows that the increment of write latency in F-Write is sharp when S_{log} is small and vice versa, because the number of registrations is reduced when S_{log} exceeds a specific threshold (e.g., around 15 MB in this study).

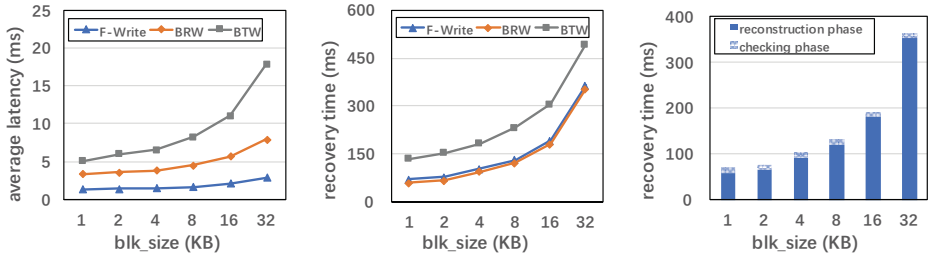
Fig. 10(b) reveals the trend of recovery time as the size of transaction log increases. We can observe that the latency of reconstruction phase almost remains unchanged; the latency of checking phase gradually surges along with the increasing value of S_{log}. These trends are explained as follows. During the checking phase, F-Write unifies the states of transaction logs and submits undo transaction entities to synchronize the parity blocks. A larger log may contains more transaction entities to process, prolonging the latency of checking phase.

In a nutshell, extending S_{log} will cut down write performance and system reliability, so it needs to be designed elaborately to record enough transaction entities according to the size of data blocks.

3) Sizes of Data Blocks - S_{blk}: Fig. 11(a) illustrate that given a specific S_{blk}, F-Write outperforms other candidate schemes in term of average latency. As S_{blk} increases, the time of data block transmission becomes longer, so the latency measures of all the tested schemes gradually go up. Moreover, the latency discrepancy between F-Write and two candidate schemes (i.e., BRW and BTW) becomes widened. On one hand, when S_{blk} is extended, there exists a surge in the latency caused by multiple copies and protocol encapsulation of data in TCP, but F-Write benefits from the zero copying feature inherited from RDMA. On the other hand, speculative update in F-Write can skip more encoding latency, because the larger the data block, the longer the encoding time.

We can observe from Fig. 11(b) that the recovery time of BTW is larger than those of F-Write and BRW under a specific S_{blk}; F-Write and BRW share similar recovery time measures. The reason has been explained in IV-C1. Regardless of the tested schemes, the recovery time gradually enlarged along with an increasing S_{blk}. Because a larger block needs more time to be transported and decoded. Fig. 11(c) demonstrates that the time consumed by the checking phase in F-Write is nearly unchanged, but the time spent in the reconstruction phase rises when S_{blk} goes up, because F-Write's checking phase involves no transmission of data blocks. Therefore, the total recovery time gradually increases. This performance

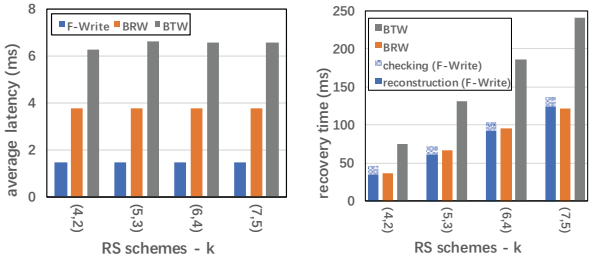
Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.



(a) Average Latency (b) Recovery Time (c) Recovery Time in F-Write

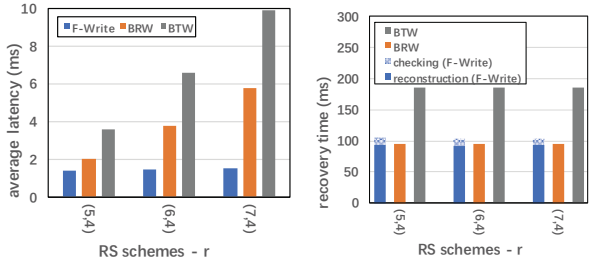
Fig. 11. Comparisons of the three write schemes using the default configurations. Parameter S_{blk} is set to $\{1, 2, 4, 8, 16, 32\}KB$.

trend is also confirmed by the results plotted in Fig. 11(b).



(a) Average Latency (b) Recovery Time

Fig. 12. Comparisons of the three write schemes using the default configurations. Parameter k is set to $\{2, 3, 4, 5\}$ and $r = 2$.



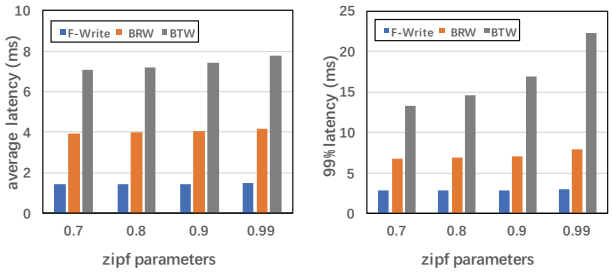
(a) Average Latency (b) Recovery Time

Fig. 13. Comparisons of the three write schemes using the default configurations. Parameter r is set to $\{1, 2, 3\}$ and $k = 4$.

4) Coding Parameters - $(k + r, k)$: Figs.12(a) and 12(b) unravel that as the value of k increases, the average latencies of all the tested schemes remain basically unchanged, but the recovery time gradually increases. Because parameter k is closely associated with reconstruction phase, a larger k indicates that a recovery node should retrieve more blocks from surviving nodes prior to decoding, prolonging the recovery time. We discover from Figs. 13(a) and 13(b) that a large value of r gives rise to a long write latency, but unacts on recovery time. Such a trend is applicable to all the schemes, because the reduction of r means that the consistency of less parity blocks in a stripe should be ensured. Moreover, there are some common findings that F-Write offers the best write performance among all the tested schemes and F-Write exhibits a slightly larger recovery time than BRW. These results have been explained in IV-C1.

In a nutshell, a large value of k cuts down recovery time, but improves space efficiency; a large value of r hurts write performance, but improves system fault tolerance.

5) Access Distribution - μ: The zipf distribution parameter stipulates an access concentration in traces. A large value of parameter μ prescribes an uneven access distribution, thereby affecting user access latency. The experimental results reveal that μ has a marginal impact on recovery time. Fig. 14 only shows the trends of the three write schemes in terms of average latency and 99 percentile latency under the various values of parameter μ. We can find that regardless of the value of μ, F-Write always outperforms two candidate schemes in terms of average latency and $99th$ latency. As the value of parameter μ increases, the average and $99th$ latencies of BTW noticeably rise, whereas the latencies of F-Write and BRW surge in a subtle way. Because a large value of μ leads to high imbalance of requests, resulting in long average waiting latency. However, thanks to the feature of zero copying, F-Write and BRW have a more powerful processing capability than BTW to deal with uneven accesses easily.



(a) Average Latency (b) 99th Percentile Latency

Fig. 14. Comparisons of the three write schemes using the default configurations. Parameter μ is set to {0.7,0.8,0.9,0.99}.

V. Related Work

There are a hoard of studies aiming to boost write performance by reducing network traffic. For example, CAU slashes cross-rack traffic by selecting appropriate update paths according to the number of updated data and parity blocks [14]; TA-Update builds an update tree according to the network layout, and tasks are distributed to nodes in the update tree [13]; CASO organizes stripes with highly-correlated data blocks, so the reconstruction-write method is applied to multi-block updates to suppress network traffic [26].

In on-disk storage systems, some write methods have been developed to shorten disk-seek time. PLR and PARIX enable the parity increments to be appended to the operation logs and; therefore, parity blocks are asynchronously updated [11] [12]; BCStore manages new stripes with updated blocks to avoid parity synchronization, and periodically uses garbage collection to reclaim useless blocks [27].

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.

Additionally, some schemes improve write performances by offloading the encoding tasks to smart hardwares. UMR-EC provides a unified erasure coding framework to fully utilize multiple hardwares such as CPU, GPU and HCA in parallel, thus reducing encoding latency [28]. TriEC is a new NIC offload paradigm which decomposes a full EC calculation pipeline into three stages to improve the parallelism. It also supports a receive-and-decode operation to match application semantics [29].

As mentioned above, many studies have worked on write optimizations in erasure-coded clusters through reducing network traffic, minimizing disk I/Os, and offloading encoding tasks, and so on. Little work is on RDMA-supported optimizations. Cocytus [3] and MemEC [4] just utilize traditional two-sided SEND/RECV verbs to accelerate the network communication, without fully exploiting the advantages of RDMA verbs. On the contrary, to our knowledge, F-Write is the first attempt to optimize writes using one-sided WRITE verbs for erasure-coded in-memory clusters, with simultaneously considering the characteristics of RDMA communications and the structural properties of erasure codes.

VI. Conclusion

In this study, We developed a write optimization scheme (i.e., F-Write) for erasure-coded in-memory clusters by jointly incorporating RDMA and erasure coding. F-Write embraces two salient features. (1) Fast2PC replaces two-sided SEND/RECV verbs with one-sided WRITE verb to implement the operations of transaction log, where multiple transactions are consolidated to submit together. (2) Speculative updates enable synchronization transactions to be submitted at an appropriate time, at which the updates are processed in the background. The merge algorithm merges multiple updates of an identical data block and generates parity increments only using the original and latest data blocks. After the extensive experiments, we observe that, relative to the two candidate write schemes, F-Write can cut down write latency by more than 61%, thereby boosting system throughput by a factor of at least $2.6\times$ with little adverse impact on recovery time.

Acknowledgment

This work was supported by Creative Research Group Project of NSFC No.61821003, and the National Science Foundation of China under Grants 61702004, 61762075 and 61872413. Xiao Qin's work is supported by the U.S. National Science Foundation under Grants CCF-0845257, CNS-0917137, and CCF-0742187, and the 111 Project under Grant B07038. Huang and Cao are both corresponding authors.

References

[1] M. Zaharia, M. Chowdhury *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[2] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," *ACM SIGMOD Record*, vol. 40, no. 4, pp. 45–51, 2012.

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

[4] M. M. Yiu, H. H. Chan, and P. P. Lee, "Erasure coding for small objects in in-memory kv storage," in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 14:1–14:12.

[5] K. Taranov, G. Alonso, and T. Hoefler, "Fast and strongly-consistent per-item resilience in key-value stores," in *Proceedings of the 13th EuroSys Conference*, 2018, pp. 39:1–39:14.

[6] M. Ovsiannikov, S. Rus, D. Reeves *et al.*, "A the quantcast file system," *PVLDB*, vol. 6, no. 11, pp. 1092–1101, 2013.

[7] K. V. Rashmi, N. B. Shah, D. Gu *et al.*, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster," in *5th USENIX Workshop on Hot Topics in Storage and File Systems3*, 2013.

[8] C. Huang, H. Simitci, Y. Xu *et al.*, "Erasure coding in windows azure storage," in *USENIX Annual Technical Conference*, 2012, pp. 15–26.

[9] G. DeCandia, D. Hastorun, M. Jampani *et al.*, "Dynamo: amazon's highly available key-value store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007, pp. 205–220.

[10] B. Calder *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the 23th ACM Symposium on Operating Systems Principles*, 2011, pp. 143–157.

[11] J. C. W. Chan, Q. Ding, P. P. C. Lee, and H. H. W. Chan, "Parity logging with reserved space: towards efficient updates and recovery in erasure-coded clustered storage," in *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014, pp. 163–176.

[12] H. Li *et al.*, "PARIX: speculative partial writes in erasure-coded systems," in *USENIX Annual Technical Conference*, 2017, pp. 581–587.

[13] Y. Wang, X. Pei, X. Ma *et al.*, "TA-Update: An adaptive update scheme with tree-structured transmission in erasure-coded storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 8, pp. 1893–1906, 2018.

[14] Z. Shen and P. P. C. Lee, "Cross-rack-aware updates in erasure-coded data centers," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 80:1–80:10.

[15] J. Xia, J. Huang, X. Qin *et al.*, "Revisiting updating schemes for erasure-coded in-memory stores," in *International Conference on Networking, Architecture, and Storage*, 2017, pp. 1–6.

[16] F. André, A. Kermarrec, E. L. Merrer *et al.*, "Archiving cold data in warehouses with clustered network coding," in *Proceedings of the 9th EuroSys Conference*, 2014, pp. 21:1–21:14.

[17] M. Subramanian, W. Lloyd, S. Roy *et al.*, "f4: Facebook's warm BLOB storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 383–398.

[18] B. C. Boutros *et al.*, "A two-phase commit protocol and its performance," *Proceedings of 7th International Conference and Workshop on Database and Expert Systems Applications*, pp. 100–105, 1996.

[19] S. Gupta and M. Sadoghi, "Easycommit: A non-blocking two-phase commit protocol," in *Proceedings of the 21th International Conference on Extending Database Technology*, 2018, pp. 157–168.

[20] P. A. Bernstein *et al.*, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981.

[21] Y. Taleb *et al.*, "Tailwind: Fast and atomic rdma-based replication," in *USENIX Annual Technical Conference*, 2018, pp. 851–863.

[22] "RDMA User Manual," http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[23] "ISA-L," https://software.intel.com.

[24] B. F. Cooper, A. Silberstein, E. Tam *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 143–154.

[25] P. W. Frey and G. Alonso, "Minimizing the hidden cost of RDMA," in *29th IEEE International Conference on Distributed Computing Systems*, 2009, pp. 553–560.

[26] Z. Shen *et al.*, "Correlation-aware stripe organization for efficient writes in erasure-coded storage systems," in *36th IEEE Symposium on Reliable Distributed Systems*, 2017, pp. 134–143.

[27] S. Li, Q. Zhang, Z. Yang, and Y. Dai, "Bcstore: Bandwidth-efficient in-memory kv-store with batch coding," in *IEEE 33th Symposium on Mass Storage Systems and Technologies*, 2017.

[28] H. Shi *et al.*, "UMR-EC: A unified and multi-rail erasure coding library for high-performance distributed storage systems," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 219–230.

[29] H. Shi and X. Lu, "Triec: tripartite graph based erasure coding NIC offload," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 44:1–44:34.

Authorized licensed use limited to: Auburn University. Downloaded on November 17,2026 at 23:11:38 UTC from IEEE Xplore. Restrictions apply.