Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

ShortTail: taming tail latency for erasure-code-based in-memory systems*

Yun TENG[1,3], Zhiyue LI[2,4], Jing HUANG[1,3], Guangyan ZHANG[†‡2,4]

[1]*College of Computer Science and Technology, Jilin University, Changchun 130012, China*

[2]*Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China*

[3]*Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun 130012, China*

[4]*Beijing National Research Center for Information Science and Technology (Tsinghua University), Beijing 100084, China*

[†]E-mail: gyzh@tsinghua.edu.cn

Received Dec. 8, 2021; Revision accepted Apr. 1, 2022; Crosschecked May 5, 2022; Published online June 1, 2022

Abstract: In-memory systems with erasure coding (EC) enabled are widely used to achieve high performance and data availability. However, as the scale of clusters grows, the server-level fail-slow problem is becoming increasingly frequent, which can create long tail latency. The influence of long tail latency is further amplified in EC-based systems due to the synchronous nature of multiple EC sub-operations. In this paper, we propose an EC-enabled in-memory storage system called ShortTail, which can achieve consistent performance and low latency for both reads and writes. First, ShortTail uses a lightweight request monitor to track the performance of each memory node and identify any fail-slow node. Second, ShortTail selectively performs degraded reads and redirected writes to avoid accessing fail-slow nodes. Finally, ShortTail posts an adaptive write strategy to reduce write amplification of small writes. We implement ShortTail on top of Memcached and compare it with two baseline systems. The experimental results show that ShortTail can reduce the P99 tail latency by up to 63.77%; it also brings significant improvements in the median latency and average latency.

Key words: Erasure code; In-memory system; Node fail-slow; Small write; Tail latency
https://doi.org/10.1631/FITEE.2100566 **CLC number:** TP302

1 Introduction

In-memory systems are widely adopted in a number of areas such as in-memory databases (Dragojević et al., 2015; Kalia et al., 2016) and in-memory key-value (KV) stores (Dragojević et al., 2014; Kalia et al., 2014). Compared with their on-disk counterparts, these systems can provide high bandwidth and low access latency. In addition, recently introduced non-volatile memory, such as Intel Optane Persistent Memory (Intel, 2015), can provide memory-like access latency and high bandwidth while guaranteeing durability, thus promising its further integration into in-memory systems. However, despite optimization for high performance, data durability and availability are also very important for in-memory systems. For example, data in dynamic random access memory (DRAM) will be lost when the power goes off unexpectedly. Even for non-volatile memory (NVM) based systems, server-level failures, such as node shutdown or network interface card (NIC) failure, can still make the corresponding data inaccessible. Therefore, fault tolerance is of great importance in in-memory systems.

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 62025203) and the Changchun Key Scientific and Technological Research and Development Project, China (No. 21ZGN30)

ORCID: Yun TENG, https://orcid.org/0000-0001-5425-5111; Guangyan ZHANG, https://orcid.org/0000-0002-3480-5902

There are two techniques that are widely used to ensure fault tolerance: multi-replica technology and erasure coding (EC). Compared with the multi-replica technology, EC has the advantage of low redundancy overhead with the same fault tolerance level. In today's distributed storage systems, there are many kinds of erasure codes, among which the most commonly used is Reed–Solomon code (RS code) (Reed and Solomon, 1960). RS code calculates m parity blocks with k data blocks according to certain rules, and forms an erasure-coded stripe with data blocks and parity blocks. The $k+m$ blocks of each stripe are distributed in $k+m$ different nodes to tolerate the loss of at most m blocks at the same time, saving a lot of storage overhead compared with multi-replica technology.

However, deploying EC in an in-memory system does not come without cost. The server-level fail-slow can degrade system performance and increase data access latency on the server. For example, a recent work (Gunawi et al., 2018) systematically studied the fail-slow problem of hardware such as CPU, memory, and network components, and pointed out that lack of power, a loose dual inline memory module (DIMM) connection, or NIC buffer corruption might cause these problems. The fail-slow problem is sometimes more difficult to address than a fail-stop problem. Most fault-tolerant systems assume that a component is either working or stopped, and neglect severe performance degradation, even though the system appears to be working normally, which was categorized as gray failure in the previous work (Huang P et al., 2017). It should be noted that even though the hardware fail-slow problem is not as frequent as in the past, it is becoming more and more common as larger-scale systems are deployed along with more hardware and more intricate operations.

This performance problem becomes more severe in applications when a memory-access operation touches more than one single part: due to the synchronous nature of multi-partition requests, the whole request is deemed to be finished only after the slowest request is done. In other words, it is the tail latency of request completions that matters in application performance and user experience, especially in a large distributed in-memory system. In this study, therefore, we focus on addressing the tail latency problem in EC-based in-memory systems.

Although there have been some previous works targeting the tail latency problem in EC systems, none of them can effectively cut the tail latency of both read and write requests. For example, LLF (Hu et al., 2017) can proactively perform degraded reads to avoid forming read hotspots, and therefore reduce the tail latency. However, this approach may trigger extra loads due to unnecessary degraded reads. It also does not perform optimization on write tail latency. EC-Cache (Rashmi et al., 2016) cuts read tail latency by sending read requests to multiple blocks of an EC stripe. The drawback of this approach is that the number of read requests often exceeds the actual needs, which results in extra overheads. It also does not optimize the latency of write requests. Another approach called EC-Store (Abebe et al., 2018) assumes that tail latency comes only from load imbalance, and balances the load with data migration. This approach will create significant estimated cost overhead for data access, especially under a high-throughput workload. It is also difficult to balance the profit and cost of data migration. LBM (Hu and Niu, 2016) optimizes data placement to a target layout by partially migrating blocks. LBM has lower data migration costs than EC-Store, but the storage overhead is pretty high to maintain the access information for each block. In addition, LBM ignores the fail-slow problem.

In this study, we propose ShortTail, an approach that can effectively tame tail latency of read and write requests for EC-enabled memory storage systems. ShortTail realizes this goal by detecting and sidestepping fail-slow nodes and optimizing small writes. First, ShortTail keeps track of request latency on each node and marks those nodes with abnormally high latency as fail-slow nodes. Then, it uses degraded reads and redirected writes to avoid accessing fail-slow nodes. Second, ShortTail also stores data, which are sent in replication with small writes, to shorten the input/output (I/O) path of write operations. Unlike previous approaches (Hu et al., 2017; Abebe et al., 2018), ShortTail can reduce tail latency of both read and write requests.

We implement ShortTail by modifying ECWide-H (Hu et al., 2021). We compare ShortTail with ECWide-H, which has no extra tail latency reducing optimization, and the LLF approach mentioned before. The experimental results show that compared with ECWide-H and LLF, ShortTail reduces the P99 latency by up to 57.40% and 55.07% on average and

the median latency by up to 78.69% and 76.04% on average, respectively.

2 System overview

2.1 Architecture

Fig. 1 presents the ShortTail architecture. ShortTail is an EC-enabled distributed in-memory storage system that uses the memory of multiple nodes to serve memory accesses from upper-level applications. Like previous works based on in-memory systems, such as ECWide-H (Hu et al., 2021) and EC-Cache (Rashmi et al., 2016), we use a centralized architecture with one coordinator node and multiple storage nodes. Each rack contains several memory nodes. The data blocks and parity blocks of each EC stripe are distributed on different memory nodes.

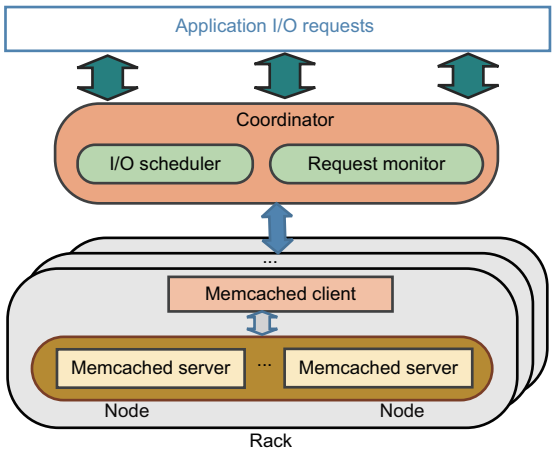

Fig. 1 ShortTail architecture

The coordinator node is responsible for serving application requests and monitoring memory nodes to detect fail-slow nodes. Internally, these two processes are done by an I/O scheduler and a request monitor.

The I/O scheduler schedules requests to achieve low tail latency. For each request, the I/O scheduler evaluates the performance of its destinations with the metadata on the coordinator node and sends remote procedure calls (RPCs) to the target memory nodes for data access. The I/O scheduler prevents requests from accessing fail-slow nodes by performing degraded reads and redirected writes. In addition, the I/O scheduler stores small writes with the multi-replica technology to reduce write amplification.

The request monitor, on the other hand, monitors the state of memory nodes to detect any fail-slow nodes. It maintains the average response time of requests to each memory node in timeslice units, and marks a memory node as fail-slow if its average response time is too long. When a memory node is judged to be slow, the request monitor will periodically probe it with small requests to inspect whether it has returned to a normal state and is ready to serve requests.

We explain how ShortTail works using only a single coordinator node here. If a single coordinator node becomes a performance bottleneck as cluster scale increases, ShortTail divides the coordinator tasks to multiple nodes. It also periodically merges the degradation information of memory nodes among the coordinator nodes to provide a precise performance view of the memory nodes.

There is one Memcached client on each rack, which receives requests from the coordinator and transfers them to Memcached servers on the same rack. Here, each Memcached server runs on a memory node. To distinguish between two types of requests, the requests that the coordinator sent to the Memcached client are called application requests, while the requests that the Memcached client sent to the memory nodes are called EC requests in the rest of the paper. The memory nodes receive EC requests and access the relevant range of local memory according to the address, and then respond to the read and write requests.

2.2 Request handling

When an application request arrives at the coordinator node, the I/O scheduler will first extract the target memory nodes and refer to node degradation counters to see if any one of the target memory nodes is currently fail-slow. If none of the memory nodes is fail-slow, this application request can safely be split into multiple EC requests and sent to the corresponding memory nodes. Otherwise, ShortTail will trigger its optimization to avoid slow EC requests. More specifically, ShortTail adopts different strategies for read and write requests as follows:

1. For an EC read request that falls on a fail-slow node, ShortTail will perform a degraded read, which reads the rest of the data and parity blocks within the same stripe and reconstructs the desired data blocks.

2. For an EC write request that touches a fail-slow node, ShortTail can redirect it to a normal memory node in a load-balanced manner.

By not accessing fail-slow nodes, ShortTail can effectively reduce the tail latency of I/O requests.

The write scheme in ShortTail is covered writing. If a write request targets a normal node, ShortTail writes data onto the original location and covers the old data. If a write request is directed to a fail-slow node, ShortTail reconstructs the old data in the fail-slow node for parity generation and redirects the write request to a replaced node.

ShortTail further optimizes small writes by adaptively selecting the write strategy. According to the write request length, ShortTail will decide to store a request in either EC or replication format. The latter can mitigate the write amplification problem of small EC writes.

In the ShortTail architecture, an EC stripe occupies some memory nodes in at least one rack. During encoding, each memory node sends data blocks to a memory node that holds parity blocks, and the latter performs data encoding. While decoding, the memory node that will store the recovered data receives related data from other memory nodes, and then reconstructs the lost data.

3 ShortTail approach

3.1 Identifying performance degradation

Because ShortTail relies on the degradation information of memory nodes to optimize EC request access latency, we first design a lightweight performance degradation identification algorithm to identify fail-slow nodes.

For each memory node, ShortTail divides the physical time into timeslices with a fixed length t and records the EC request latency information in each timeslice. The timeslice to which an EC request belongs is determined by its start time. When all EC requests in a timeslice finish and the timeslice is gone, ShortTail calculates the average request latency in that timeslice and reclaims the statistics space of the timeslice. The corresponding node is judged as a fail-slow node when the average latency is higher than a preset threshold x. Then this node is marked fail-slow to avoid subsequent requests to access this node.

We also maintain a thread in the coordinator node that periodically sends probe requests to see whether the fail-slow nodes have returned to a normal state and are ready to serve EC requests.

Fig. 2 presents an example of this detection for a single node. Requests marked with the same color denote that their start time falls in the same timeslice. When all requests in timeslice 0 return, their average latency av_0 is calculated. Because av_0 is smaller than threshold x, ShortTail thinks this memory node is not under degradation currently. When calculating the average latency in timeslice 1, the result av_1 is larger than threshold x. Accordingly, ShortTail sets the flag as one, indicating that this node is now fail-slow.

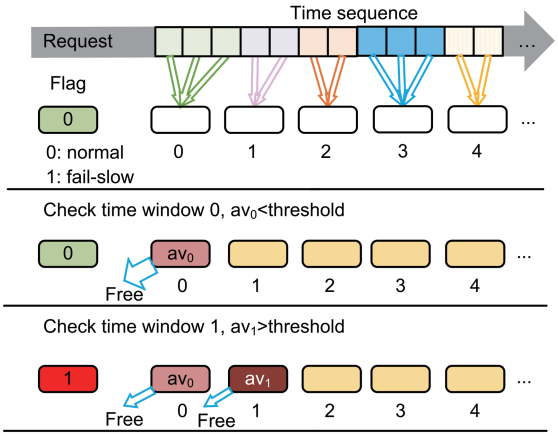

Fig. 2 Fail-slow node identification

References to color refer to the online version of this figure

Because the optimal values of t and x are workload-dependent, they cannot be immediately known when ShortTail performs a cold start. ShortTail first collects request latency information and calculates t and x before turning on the function of performance degradation detection. During runtime, ShortTail also keeps track of application requests and adjusts t and x if the workload characteristics change.

A satisfactory timeslice length t should be as small as possible, with the condition that there are enough requests falling in each timeslice on average. If t is too small, ShortTail can misjudge the degradation information by a single straggler request.

Because memory nodes may suffer from performance fluctuation, requests to a normal node can also see increased latency, which may mislead the judgment of fail-slow nodes. ShortTail avoids this

by accommodating enough requests in each timeslice and averaging their latency when deciding states of the nodes. To choose threshold x, ShortTail first calculates the average request latency of the normal memory nodes within a timeslice, and then sets x at about three times of the average request latency.

The main memory overhead of identifying fail-slow nodes comes from two parts: request metadata and node metadata. First, request metadata consist of index, issue time, and return time of each request. Because the judgment of fail-slow nodes is performed in a timeslice, the memory space of request metadata related to previous timeslices can be reclaimed dynamically. Second, node metadata include only Boolean variables for each node to denote if it is a fail-slow node. Therefore, the memory overhead of ShortTail is limited.

When monitoring memory nodes, the coordinator records the latency of each request without additional network traffic. After a node has been detected as a fail-slow node, the coordinator sends extra probe requests to the fail-slow node periodically, but this will not incur much network traffic because the probing frequency is low.

3.2 Sidestepping fail-slow nodes

ShortTail optimizes access latency of EC requests by sidestepping fail-slow nodes. This is accomplished using two key techniques: degraded reads and redirected writes. We show how ShortTail handles read and write requests on fail-slow nodes below.

3.2.1 Read handling

ShortTail performs degraded reads for read requests that fall on fail-slow nodes. When any EC read requests fall on a fail-slow node, ShortTail will try to collect other blocks of the corresponding stripe by issuing additional requests to the relevant nodes. After this, ShortTail can reproduce the desired data block by EC reconstruction without accessing the fail-slow node. This read handling strategy is feasible only when the number of fail-slow nodes that the stripe touched is smaller than the number of parity blocks m. If this condition is not satisfied, ShortTail will still perform reads on fail-slow nodes because data reconstruction cannot be finished by accessing those normal nodes. However, this will not impair the performance too much because the simultaneous appearance of multiple fail-slow nodes is rather rare.

Fig. 3 presents an example of the read handling process. Briefly, we assume that the stripe width is five. Because the read request falls on node 2, which has already been identified as fail-slow, ShortTail will avoid accessing this node by reading the blocks in the other four nodes and performing data reconstruction.

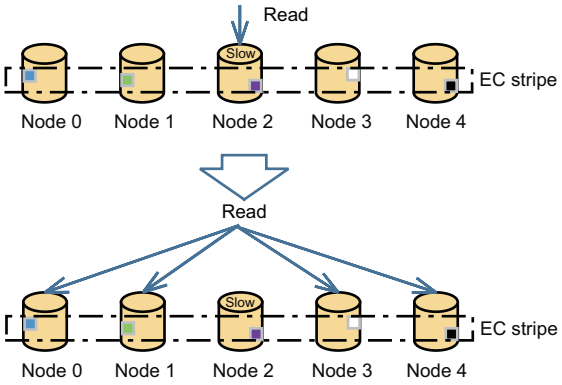

Fig. 3 Fail-slow node sidestepping for read requests

3.2.2 Write handling

Write handling in ShortTail is more complicated because it modifies both data blocks and parity blocks. When a write request is located on a fail-slow memory node, ShortTail will redirect it to a normal node and record this mapping in the coordinator node. The relevant parity blocks are also updated.

If ShortTail meets a fail-slow node when it writes a parity block, ShortTail redirects it to another normal node. The target node of write redirection is chosen according to two rules:

1. The fault tolerance rule. All the blocks of an EC stripe fall on different memory nodes.

2. The load-balancing rule. The number of blocks written to each node in the redirection mode should be balanced across all memory nodes.

To abide the second rule, ShortTail maintains a redirection count for each memory node to record how many write requests have been redirected to the corresponding node. It will choose the target node of write redirection from the normal nodes with the smallest redirection count. By following this rule, ShortTail can avoid overloading certain memory nodes with many more redirected blocks than the other nodes. Write redirection in ShortTail differs

from those in many previous works (Wilkes et al., 1996; Wu et al., 2016), in which small writes were redirected to a memory buffer for I/O aggregation. Moreover, they did not consider the problem of load imbalance, which may cause latency increment.

Fig. 4 shows an example of write handling. Assume that there are eight memory nodes in the system. An EC stripe with a width of five blocks scatters its data and parity blocks on the first five nodes. We suppose that an incoming write request arrives in node 2, which is currently fail-slow. To avoid a slow write while maintaining load balance, ShortTail will redirect this request to node 5, because it has the smallest redirection count. Then, the redirection count of node 5 is increased by one.

3.3 Optimizing small writes

ShortTail creates further optimization on small writes. This is motivated by the fact that a write operation will be padded to a full stripe when its size is obviously smaller than the EC stripe width, which will incur severe write amplification and waste both storage space and I/O bandwidth. ShortTail optimizes this by selectively performing EC writes according to the request size. If the request size is larger than a given threshold s, the request will be stored in EC form with possible padding of zero. Otherwise, the request is regarded as a small write and stored in replication to save space and to mitigate the write amplification problem. The optimization also causes small writes involving fewer nodes, which reduces the chance of accessing fail-slow nodes.

The most desirable threshold s can be determined by a simple calculation. As shown in Fig. 5, we suppose that a write request contains c blocks. If the data are stored in EC form, they will occupy k data blocks and m parity blocks, with total space overhead of $k+m$ blocks. If they are stored in replication form, the space consumption will be $p \times c$ for the p-replica strategy. When EC storage and multi-replica storage have the same storage efficiency, we have $k+m = p \times c$. To obtain the same level of fault tolerance between EC and multi-replica strategy, we have $p = m+1$. So, we obtain $c = \frac{k+m}{m+1}$. Therefore, we set threshold s as $\lfloor \frac{k+m}{m+1} \rfloor$ in our deployment. If c is larger than s, the write request should be stored in EC form to save space. Otherwise, replicating c blocks with p replicas is more desirable.

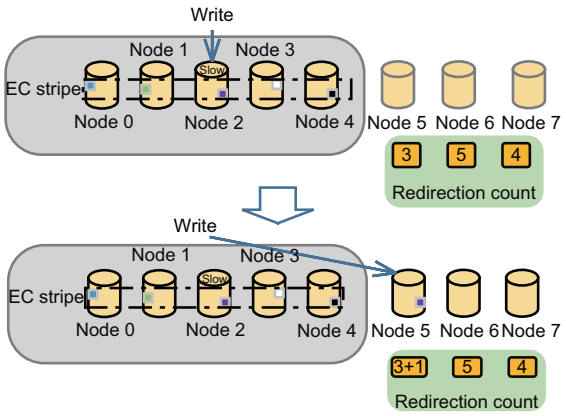

Fig. 4 Fail-slow node sidestepping for write requests

4 Performance evaluation

4.1 Evaluation methodology

We implement the ShortTail approach on the Linux system by modifying the source code of ECWide-H (a version of ECWide (Hu et al., 2021)), with about 1200 lines of code added or modified.

ECWide-H is an in-memory storage system that optimizes wide stripe EC. For each application request, the coordinator of ECWide-H splits the request into multiple EC requests and sends them to the corresponding memory nodes. If an EC request tries to read a failed node, the coordinator will perform a degraded read to reconstruct the data. For write requests, ECWide-H performs in-place updates for both data and parity blocks.

As in ECWide-H, ShortTail uses RS code to calculate the global parities, and accelerates repairing one-disk failure by maintaining local parities for part of a stripe; this design is also a local recoverable code (LRC). We modify mainly the coordinator in

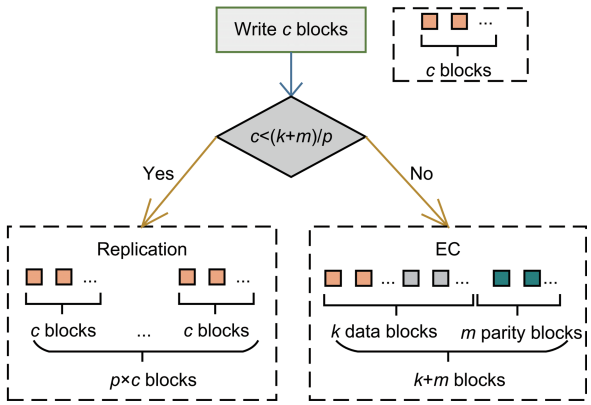

Fig. 5 Optimization for small writes

ECWide-H to implement ShortTail. First, we create the request monitor by adding data structures and functions to count the latency of each memory node and determine if any node is under fail-slow. Second, we change the request processing logic in the coordinator and add redirected write and small write optimization to reduce the latency of write requests.

Due to the limited number of machines available, we simulate each rack with a machine. Each machine starts a number of processes, and each process represents a memory node. Our testbed uses six machines, with each machine running four processes, thus forming a simulation of a 24-node EC storage system. Because it is relatively rare that a node becomes fail-slow in such a small-scale cluster, we simulate it by randomly choosing a memory node and adding additional operations to the requests targeting this node.

To verify the effectiveness of our approach, we compare ShortTail with two EC-enabled in-memory storage systems under Yahoo! Cloud Serving Benchmark (YCSB) (Cooper et al., 2010) workload and Usr_1 workload (Narayanan et al., 2008). The first system that we compared is an ECWide-H system (Hu et al., 2021), which is an EC storage system with no optimization of tail latency. The second system is an LLF-enabled system, which was proposed by Hu et al. (2017). The main idea of LLF is to proactively perform degraded reads to avoid forming data hotspots. For each read request, LLF determines whether to perform normal reads or degraded reads by inspecting the load on the relevant nodes. If the destination nodes of normal reads suffer from heavier loads than any of the destination nodes of degraded reads, LLF will choose to use the degraded read for more balanced loads and shorter response time. We choose ECWide-H (Hu et al., 2021) because it is a new in-memory storage system based on erasure code. Using this system for comparison, we can evaluate the effectiveness of ShortTail in reducing tail latency. LLF is significantly related to ShortTail because it also targets tail latency reduction, but it does not optimize write requests, and our experiments show that write optimization is also very important.

4.2 Overall performance

We compare three evaluated approaches concerning average latency, median latency, and P99 latency. To make our results more convincing, we test the results under the YCSB workload and Usr_1 workload, as shown in Figs. 6 and 7, respectively. The results under both workloads show that ShortTail can reduce the P99 latency efficiently compared with the ECWide-H approach, with around 51.03% and 63.77% reduction on the YCSB and Usr_1 workloads, respectively. ShortTail reduces the P99 latency by 57.40% on average on these two workloads. Even compared with LLF, which performs optimization on request latency, ShortTail still reduces the P99 latency by 48.15% and 62.00% under the YCSB and Usr_1 workloads, respectively. ShortTail reduces the P99 latency by 55.07% on average on these two workloads. As for the median and average latency results, ShortTail also outperforms the ECWide-H and LLF significantly. Compared to ECWide-H, ShortTail reduces the median latency and the average latency by 78.69% and 76.59% on average, respectively. Compared to LLF, ShortTail reduces the median latency and the average latency by 76.04% and 72.94% on average, respectively.

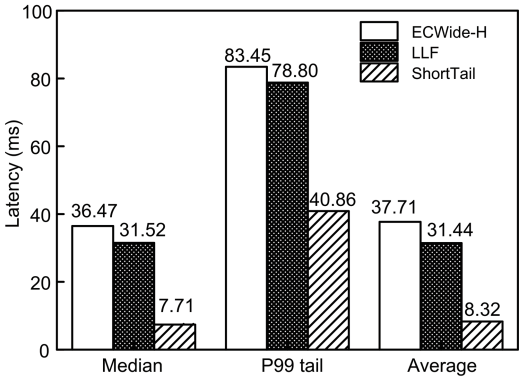

Fig. 6 Performance comparison under the YCSB workload

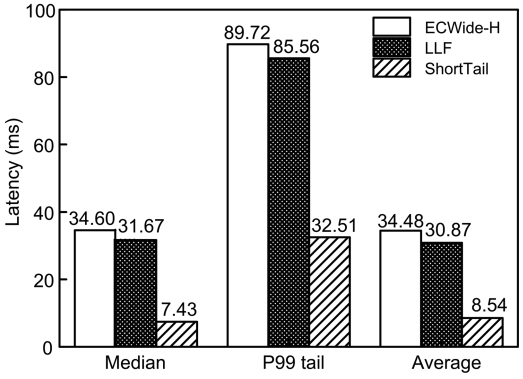

Fig. 7 Performance comparison under the Usr_1 workload

This is mainly because our design for performance degradation detection can precisely identify the fail-slow nodes, and we avoid the straggler read requests by performing degraded reads. Our optimization of write requests also contributes to this positive result.

4.3 Effects of individual techniques

This subsection measures the impact of three individual techniques proposed by this paper. We implement three versions of ShortTail, each adding an individual technique:

1. "+Read Tail Opt" adds the technique of avoiding reading fail-slow nodes (read handling in Section 3.2) to ECWide-H;

2. "+Write Tail Opt" adds the technique of avoiding writing fail-slow nodes (write handling in Section 3.2) to the "+Read Tail Opt" version;

3. "+Small Write Opt" adds the technique of optimizing small writes (Section 3.3) to the "+Write Tail Opt" version. In other words, the "+Small Write Opt" version is a full version of ShortTail.

Fig. 8 shows the performance comparison of these three versions of ShortTail and the two baseline approaches under the YCSB workload. We can see that detecting and sidestepping fail-slow nodes have a significant effect on latency reduction. The read tail optimization cuts the P99 latency by up to 8.45% and 3.05% respectively, compared with ECWide-H and LLF. Although LLF also targets read optimization, we can see that ShortTail still achieves lower latency than LLF because the radical fail-slow detection algorithm in LLF can incur many unnecessary degraded reads, which will burden the network and fill request handling queues in the memory nodes, further increasing request latency.

This result also addresses the concern that the reconstruction calculation may affect system performance. The rationale behind this is that the calculation overhead is fairly low compared to other overhead of a request. For a stripe of n data blocks that are encoded with RS code, the decoding complexity is $O(n^3)$. Because it is rare that multiple nodes are simultaneously fail-slow, in ShortTail, most of the degraded reads can be realized with local parity blocks. We also use Intel Intelligent Storage Acceleration Library (ISA-L) to accelerate this process, so the computational complexity can be further reduced. We have measured the computational time

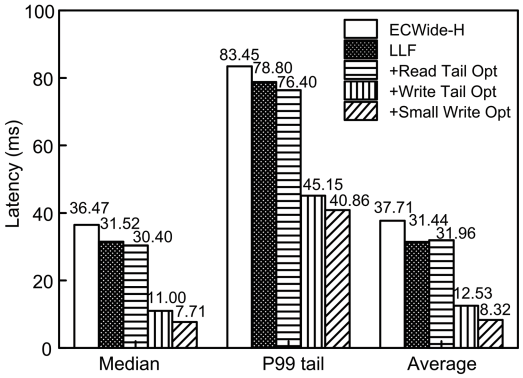

Fig. 8 Performance contributions of individual techniques

of reconstructing data using erasure codes. The experimental results show that the average latency of degraded reads is 8.46 ms, while the computational time is only about 0.01 ms. This shows that the computational overhead is negligible in degraded reads.

When write tail optimization is added, ShortTail further reduces the P99 latency by up to 37.45% and 39.66% compared with ECWide-H and LLF, respectively. The results show that both read and write requests should be considered when optimizing tail latency in a distributed EC storage system.

Finally, the addition of small write optimization can further decrease the P99 latency by 5.14% and 5.44% compared with ECWide-H and LLF, respectively. This is because our small write optimization saves the storage bandwidth of small writes by minimizing the accessed data volume; therefore, it reduces the resource contention level and access latency. It should be noted that the results regarding the median and average latencies exhibit trends similar to the analysis above.

4.4 Sensitivity to internal parameters

Finally, we study the impact of ShortTail's key parameters. ShortTail has two important system parameters used in the process of detecting fail-slow nodes: timeslice length t and threshold value x (Section 3.1). Fig. 9 shows how the tail latency of ShortTail changes with different values of t and x under the YCSB workload.

When the timeslice length t increases from 10 to 30 ms in a 5 ms step, the tail latency does not change too much (no more than 20%). This is because a 10 ms timeslice length is already large enough to resist an occasional straggler and accurately detect

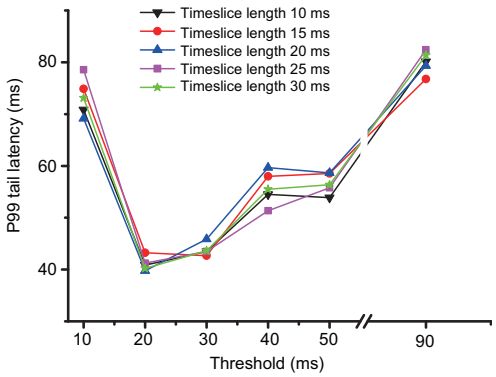

Fig. 9 Influence of changing parameters

fail-slow nodes in the system. A larger timeslice length does not influence the detection results.

As for threshold x, when it increases from 20 to 90 ms, the tail latency rises accordingly. This is because larger thresholds cause the fail-slow detection to be less sensitive, and leave the system in a long period of degraded access before the fail-slow nodes are detected. However, if x is smaller (e.g., 10 ms in our experiments), the tail latency will rise. This is because nodes can be frequently misjudged as fail-slow nodes in this circumstance. If there are too many fail-slow nodes, our sidestepping techniques (i.e., degraded read and redirected write) will not work, and in turn increase the tail latency.

5 Related works

In this section, we introduce some existing works that target the most relevant fields of this paper, including building EC systems, designing in-memory storage systems, and optimizing storage system latency.

5.1 Erasure coding in storage system

EC has been widely studied in distributed storage systems (Plank and Huang, 2013; Balaji et al., 2018) because it is efficient in achieving reliability with low storage overhead. Among various kinds of EC, RS codes (Reed and Solomon, 1960) have been widely deployed today (Weil et al., 2006; Wilcox-O'Hearn and Warner, 2008; Ford et al., 2010; Ovsiannikov et al., 2013; Li XL et al., 2019; Lin et al., 2021), mainly for their deterministic coding to tolerate the loss of any m parity blocks. Our work also employs this property to provide data availability by building EC-based storage systems.

EC-based distributed systems put significant demands on a network, especially in the recovery process. Some works design various kinds of minimum-storage regenerating (MSR) codes (Dimakis et al., 2010) to minimize the repair bandwidth and accelerate the reconstruction process, like PM-RBT codes (Rashmi et al., 2015), Butterfly (Pamies-Juarez et al., 2016), and Clay (Vajha et al., 2018). Unfortunately, MSR codes have low encoding speed. In contrast, ShortTail targets the optimization data services instead of the reconstruction process.

ShortTail uses RS code, and also uses locally recoverable code by calculating local parities for part of a stripe, but ShortTail still considers how to reduce network traffic by performing only degraded reads if necessary.

5.2 In-memory storage

In-memory storage has gained great popularity in recent years due to its high bandwidth and low latency compared with disks or solid state drives (SSDs). For example, the Redis Memory Store (https://redis.io/) has been widely used and can store many kinds of data structures such as strings, hashes, lists, sets, bitmaps, and streams. FaRM (Dragojević et al., 2014) is a distributed in-memory system that uses fast RDMA (Poke and Hoefler, 2015) to build distributed shared memory (DSM) on top of multiple memory nodes and provides memory access with transactional support. However, these works all provide fault tolerance using replication, which consumes more precious memory space.

ShortTail is implemented based on Memcached, and uses EC to save storage overhead to guarantee data reliability. In addition to ShortTail, some previous works optimize Memcached but target different aspects. For instance, MemC3 (Fan et al., 2013) improves the hash collision handling and solves the concurrency issue in Memcached using concurrent Cuckoo hashing (Pagh and Rodler, 2004) and optimistic locking. Nishtala et al. (2013) optimized Memcached to enhance scalability and deployed it in Facebook, but the multi-replica technology is still used in this work to accelerate data reads.

5.3 Latency reduction

There are some works that target request latency optimization. Some replication-based systems

use hedging requests to reduce latency, in which redundant requests are sent (Andersen et al., 2005; Huang C et al., 2012; Stewart et al., 2013) to multiple replicas, and pick up the one with the fastest response. To prevent hedging request from adding too many network loads, Shah et al. (2016) presented a statistics model to capture the system features and send redundant requests only when it deems that the benefit justifies the cost. However, these works are oriented to replication storage, which is different from our EC-based strategies.

Other works target scenarios that are different from ShortTail. For example, PANDO (Uluyol et al., 2020) is a geo-distributed storage system (Li C et al., 2012) that uses Paxos (Lamport, 1998) to ensure data consistency. PANDO optimizes request latency by selecting a proxy that is closer to the target data centers and letting it access the data before the result is sent back to the client side. This huge access latency difference does not exist in our data center-oriented scenario.

Reducing tail latency has gained more and more attention with the increasing size of storage systems and stricter requirements for application quality of service (QoS). For example, C3 (Ganjam et al., 2015) uses a cubic function to estimate the queue length on each replica to guide low-latency access, but C3 still targets multi-replica scenarios. LLF is an approach that can proactively perform degraded reads to avoid forming read hotspots, thereby reducing tail latency, but it does not optimize the write requests. EC-Cache (Rashmi et al., 2016) cuts read tail latency by sending read requests to multiple blocks of an EC stripe. When the number of returned requests is sufficient to reconstruct the entire EC stripe, the read is deemed to be completed. The drawback is that EC-Cache is a black box approach that does not inspect the state of server nodes, so the number of read requests often exceeds the actual needs, resulting in extra overhead. It also does not optimize the latency of write requests. EC-Store assumes that tail latency comes only from load imbalance, and balances the load with data migration. Unlike EC-Store, ShortTail assumes that tail latency is caused mainly by fail-slow rather than load imbalance, and identifies fail-slow nodes. LBM (Hu and Niu, 2016) is a min-k-partition model that optimizes data placement in a target layout, and achieves this by partially migrating blocks. LBM has lower data migration costs than EC-Store, but the storage overhead of LBM is fairly high to maintain the access information for each block. In addition, LBM considers only load imbalance and ignores the fail-slow problem.

6 Conclusions

This paper proposes ShortTail, an EC-based distributed in-memory storage system that can efficiently cope with the fail-slow problem of memory nodes and maintains low tail latency for application requests. ShortTail uses a low-overhead request monitor to detect fail-slow nodes, and uses two optimizations on read and write requests to avoid accessing fail-slow nodes. ShortTail also uses a small-write optimization to reduce the write amplification. Together, these techniques significantly reduce tail latency compared with peer systems.

With ShortTail, applications can benefit from fast distributed memory with high performance and data availability. This is particularly promising for future applications with large-scale deployments and higher performance and QoS demands.

Contributors

Yun TENG designed the research and implemented the prototype system. Yun TENG and Zhiyue LI drafted the paper. Jing HUANG and Guangyan ZHANG helped organize the paper. Yun TENG and Guangyan ZHANG revised and finalized the paper.

Compliance with ethics guidelines

Yun TENG, Zhiyue LI, Jing HUANG, and Guangyan ZHANG declare that they have no conflict of interest.

References

Abebe M, Daudjee K, Glasbergen B, et al., 2018. EC-Store: bridging the gap between storage and latency in distributed erasure coded systems. Proc IEEE 38th Int Conf on Distributed Computing System, p.255-266. https://doi.org/10.1109/ICDCS.2018.00034

Andersen DG, Balakrishnan H, Kaashoek MF, et al., 2005. Improving web availability for clients with MONET. Proc 2nd Symp on Networked Systems Design and Implementation, p.115-128.

Balaji SB, Krishnan MN, Vajha M, et al., 2018. Erasure coding for distributed storage: an overview. *Sci China Inform Sci*, 61(10):100301. https://doi.org/10.1007/s11432-018-9482-6

Cooper BF, Silberstein A, Tam E, et al., 2010. Benchmarking cloud serving systems with YCSB. Proc 1st ACM Symp on Cloud Computing, p.143-154. https://doi.org/10.1145/1807128.1807152

Dimakis AG, Godfrey PB, Wu YN, et al., 2010. Network coding for distributed storage systems. *IEEE Trans Inform Theory*, 56(9):4539-4551. https://doi.org/10.1109/TIT.2010.2054295

Dragojević A, Narayanan D, Hodson O, et al., 2014. FaRM: fast remote memory. Proc 11th USENIX Conf on Networked Systems Design and Implementation, p.401-414.

Dragojević A, Narayanan D, Nightingale EB, et al., 2015. No compromises: distributed transactions with consistency, availability, and performance. Proc 25th Symp on Operating Systems Principles, p.54-70. https://doi.org/10.1145/2815400.2815425

Fan B, Andersen DG, Kaminsky M, 2013. MemC3: compact and concurrent MemCache with dumber caching and smarter hashing. Proc 10th USENIX Conf on Networked Systems Design and Implementation, p.371-384.

Ford D, Labelle F, Popovici FI, et al., 2010. Availability in globally distributed storage systems. Proc 9th USENIX Conf on Operating Systems Design and Implementation, p.61-74.

Ganjam A, Jiang JC, Liu X, et al., 2015. C3: Internet-scale control plane for video quality optimization. Proc 12th USENIX Conf on Networked Systems Design and Implementation, p.131-144.

Gunawi HS, Suminto RO, Sears R, et al., 2018. Fail-slow at scale: evidence of hardware performance faults in large production systems. Proc 16th USENIX Conf on File and Storage Technologies, p.1-14.

Hu YC, Niu D, 2016. Reducing access latency in erasure coded cloud storage with local block migration. Proc 35th Annual IEEE Int Conf on Computer Communications, p.1-9. https://doi.org/10.1109/INFOCOM.2016.7524628

Hu YC, Wang YS, Liu B, et al., 2017. Latency reduction and load balancing in coded storage systems. Symp on Cloud Computing, p.365-377. https://doi.org/10.1145/3127479.3131623

Hu YC, Cheng LF, Yao QR, et al., 2021. Exploiting combined locality for wide-stripe erasure coding in distributed storage. Proc 19th USENIX Conf on File and Storage Technologies, p.233-248.

Huang C, Simitci H, Xu YK, et al., 2012. Erasure coding in windows azure storage. USENIX Conf on Annual Technical Conf, p.2.

Huang P, Guo CX, Zhou LD, et al., 2017. Gray failure: the Achilles' heel of cloud-scale systems. Proc 16th Workshop on Hot Topics in Operating Systems, p.150-155. https://doi.org/10.1145/3102980.3103005

Intel, 2015. Intel Announces Optane Storage Brand for 3D XPoint Products. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products [Accessed on Nov. 8, 2021].

Kalia A, Kaminsky M, Andersen DG, 2014. Using RDMA efficiently for key-value services. *SIGCOMM Comput Commun Rev*, 44(4):295-306. https://doi.org/10.1145/2740070.2626299

Kalia A, Kaminsky M, Andersen DG, 2016. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. Proc 12th USENIX Symp on Operating Systems Design and Implementation, p.185-201.

Lamport L, 1998. The part-time parliament. *ACM Trans Comput Syst*, 16(2):133-169. https://doi.org/10.1145/279227.279229

Li C, Porto D, Clement A, et al., 2012. Making geo-replicated systems fast as possible, consistent when necessary. Proc 10th USENIX Conf on Operating Systems Design and Implementation, p.265-278.

Li XL, Li RH, Lee PPC, et al., 2019. OpenEC: toward unified and configurable erasure coding management in distributed storage systems. Proc 17th USENIX Conf on File and Storage Technologies, p.331-344.

Lin SY, Gong GW, Shen ZR, et al., 2021. Boosting full-node repair in erasure-coded storage. USENIX Annual Technical Conf, p.641-655.

Narayanan D, Donnelly A, Rowstron A, 2008. Write off-loading: practical power management for enterprise storage. *ACM Trans Storage*, 4(3):10. https://doi.org/10.1145/1416944.1416949

Nishtala R, Fugal H, Grimm S, et al., 2013. Scaling memcache at Facebook. Proc 10th USENIX Symp on Networked Systems Design and Implementation, p.385-398.

Ovsiannikov M, Rus S, Reeves D, et al., 2013. The quantcast file system. *Proc VLDB Endow*, 6(11):1092-1101. https://doi.org/10.14778/2536222.2536234

Pagh R, Rodler FF, 2004. Cuckoo hashing. *J Algor*, 51(2):122-144. https://doi.org/10.1016/j.jalgor.2003.12.002

Pamies-Juarez L, Blagojevic F, Mateescu R, et al., 2016. Opening the chrysalis: on the real repair performance of MSR codes. Proc 14th USENIX Conf on File and Storage Technologies, p.81-94.

Plank JS, Huang C, 2013. Tutorial: erasure coding for storage applications. Proc 11th USENIX Conf on File and Storage Technologies.

Poke M, Hoefler T, 2015. DARE: high-performance state machine replication on RDMA networks. Proc 24th Int Symp on High-Performance Parallel and Distributed Computing, p.107-118. https://doi.org/10.1145/2749246.2749267

Rashmi KV, Nakkiran P, Wang JY, et al., 2015. Having your cake and eating it too: jointly optimal erasure codes for I/O, storage and network-bandwidth. Proc 13th USENIX Conf on File and Storage Technologies, p.81-94.

Rashmi KV, Chowdhury M, Kosaian J, et al., 2016. EC-Cache: load-balanced, low-latency cluster caching with online erasure coding. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.401-417.

Reed IS, Solomon G, 1960. Polynomial codes over certain finite fields. *J Soc Ind Appl Math*, 8(2):300-304. https://doi.org/10.1137/0108018

Shah NB, Lee K, Ramchandran K, 2016. When do redundant requests reduce latency? *IEEE Trans Commun*, 64(2):715-722. https://doi.org/10.1109/TCOMM.2015.2506161

Stewart C, Chakrabarti A, Griffith R, 2013. Zoolander: efficiently meeting very strict, low-latency SLOs. Proc 10th Int Conf on Autonomic Computing, p.265-277.

Uluyol M, Huang A, Goel A, et al., 2020. Near-optimal latency versus cost tradeoffs in geo-distributed storage. Proc 17th USENIX Symp on Networked Systems Design and Implementation, p.157-180.

Vajha M, Ramkumar V, Puranik B, et al., 2018. Clay codes: moulding MDS codes to yield an MSR code. Proc 16th USENIX Conf on File and Storage Technologies, p.139-154.

Weil SA, Brandt SA, Miller EL, et al., 2006. Ceph: a scalable, high-performance distributed file system. Proc 7th Symp on Operating Systems Design and Implementation, p.307-320.

Wilcox-O'Hearn Z, Warner B, 2008. Tahoe: the least-authority filesystem. Proc 4th ACM Int Workshop on Storage Security and Survivability, p.21-26. https://doi.org/10.1145/1456469.1456474

Wilkes J, Golding R, Staelin C, et al., 1996. The HP AutoRAID hierarchical storage system. *ACM Trans Comput Syst*, 14(1):108-136. https://doi.org/10.1145/225535.225539

Wu SZ, Mao B, Chen XL, et al., 2016. LDM: log disk mirroring with improved performance and reliability for SSD-based disk arrays. *ACM Trans Storage*, 12(4):22. https://doi.org/10.1145/2892639