Copyright © 2019 Tech Science Press CMC, vol.58, no.1, pp.153-167, 2019

# GFCache: A Greedy Failure Cache Considering Failure Recency and Failure Frequency for an Erasure-Coded Storage System

**Mingzhu Deng[1], Fang Liu[2, *], Ming Zhao[3], Zhiguang Chen[2] and Nong Xiao[2, 1]**

**Abstract:** In the big data era, data unavailability, either temporary or permanent, becomes a normal occurrence on a daily basis. Unlike the permanent data failure, which is fixed through a background job, temporarily unavailable data is recovered on-the-fly to serve the ongoing read request. However, those newly revived data is discarded after serving the request, due to the assumption that data experiencing temporary failures could come back alive later. Such disposal of failure data prevents the sharing of failure information among clients, and leads to many unnecessary data recovery processes, (e.g. caused by either recurring unavailability of a data or multiple data failures in one stripe), thereby straining system performance.

To this end, this paper proposes GFCache to cache corrupted data for the dual purposes of failure information sharing and eliminating unnecessary data recovery processes. GFCache employs a greedy caching approach of opportunism to promote not only the failed data, but also sequential failure-likely data in the same stripe. Additionally, GFCache includes a FARC (Failure ARC) catch replacement algorithm, which features a balanced consideration of failure recency, frequency to accommodate data corruption with good hit ratio. The stored data in GFCache is able to support fast read of the normal data access. Furthermore, since GFCache is a generic failure cache, it can be used anywhere erasure coding is deployed with any specific coding schemes and parameters.

Evaluations show that GFCache achieves good hit ratio with our sophisticated caching algorithm and manages to significantly boost system performance by reducing unnecessary data recoveries with vulnerable data in the cache.

**Keywords:** Failure cache, greedy recovery, erasure coding, failure recency, failure frequency.

## 1 Introduction

In recent years, unstoppable data explosion [Wu, Wu, Liu et al. (2018); Sun, Cai, Li et al. (2018)] generated by wireless sensors [Yu, Liu, Liu et al. (2018)] and terminals [Liu and

[1] College of Computer, National University of Defense Technology, No. 109 Deya Road, KaiFu District, Changsha, 410073, China.

[2] School of Data and Computer Science, Sun Yat-Sen University, No. 132 East Outer Ring Road of University City Road, Guangzhou, 510006, China.

[3] Arizona State University, BYENG 460, 699 S Mill Ave, Tempe AZ, 85281, USA.

[*] Corresponding Author: Fang Liu. Email: liufang25@mail.sysu.edu.cn.

Li (2018); Guo, Liu, Cai et al. (2018)] keeps driving up the demand for larger space in various big data storage systems. Due to the ever-growing data volume and ensuing space overhead concern, erasure coding, blessed with its capability to provide higher levels of reliability at a much lower storage cost, are gaining popularity [Wang, Pei, Ma et al. (2017)]. For instance, Facebook clusters employ a RS (10, 4) code to save money [Rashmi, Shah, Gu et al. (2015); Rashmi, Chowdhury, Kosaian et al. (2016)] while Microsoft invents and deploys its own LRC code in Azure [Huang, Simitci, Xu et al. (2012)]. Conversely, countless commercial components of storage systems are inherently unreliable and susceptible to failures [Zhang, Cai, Liu et al. (2018)]. Moreover, as the system aggressively scales up to compensate for the influx of data [Liu, Zhang, Xiong et al. (2018)], data corruptions, either temporary or permanent, become a normal daily occurrence. For an erasure-coded storage system, a reconstruction operation is called to recover the failed data blocks, with the help of parity blocks. Unlike the permanent data failure, which is fixed through a background job, temporarily unavailable data is recovered on-the-fly, in order to serve the ongoing read request. This is because, while permanent failures are under system surveillance by monitoring mechanisms like heart-beat, temporary data failure is unknown until it is accessed. In big data storage system like Hadoop, an I/O exception will occur upon accessing unavailable data. Insistence on such exceptions after several repeated attempts leads to the situation of de-graded read. Although data recovery process is triggered immediately to fulfill a degraded read request, system performance is still degraded. This is due to disproportionate amounts of I/O and network bandwidth [Cai, Wang, Zheng et al. (2013)] consumed by each recovery process. For example, given a (6, 4) RS code and the block size to be 16 MB, a corrupted data block in a stripe needs a block of 16MB read and then downloaded from each of other six healthy nodes. In general, given a (k, m) MDS eras-ure code, k times of overhead is incurred to reconstruct one block.

However, those newly revived data are either discarded immediately or not tracked after serving the request. Such disposal is somewhat reasonable due to the assumption that data experiencing temporary failures could later come back alive. On the other hand, such design overlooks the importance of keeping recovered data. For instance, due to the uncertainty of causal factors, like hardware glitches, which data is to be unavailable as well as when it occurs, conforms to no particular distribution. This makes failure pattern difficult to find, and to follow amid the various failure statistics. In other words, repeated temporary data unavailability is likely to occur for reasons such as persisting system hot spots, recurring software upgrades and so forth. Therefore, the existing disposal of a recovered data would inevitably lead to repeated recoveries due to its recurring unavailability, thereby straining system resources and performance. Furthermore, statistics [Subedi, Huang, Liu et al. (2016)] indicate that multiple-failure scenarios occur, and multiple blocks of a stripe can be unavailable simultaneously or incrementally. Given the current data recovering practice of one reconstruction operation per degraded read, multiple data corruptions on a stripe inevitably require multiple data recovery processes. However, those failed data can be produced by using only one recovery process. Therefore, such redundant recoveries on a single stripe lead to wasted system resources [Li, Cai and Xu (2018)] and degraded performance. Additionally, big data storage systems like Azure, often support multiple clients' access. Without a central storing recorder (medium) of recovered data to enable the sharing of failure information among multiple-clients, those repeated and redundant data reconstruction operation happening to one client would unnecessarily reoccur among different clients. Therefore, we argue buffering and sharing failure information is of instrumental importance in the avoidance of unnecessary data reconstruction, and will improve system performance.

To this end, this paper considers a typical distributed setting of an erasure-coded storage system and proposes GFCache to cache corrupted data to serve those purposes. GFCache employs a greedy caching approach of opportunity to promote not only the failed data, but also sequential failure-likely data in the same stripe. Additionally, GFCache includes a FARC (Failure ARC) cache replacement algorithm, which features a balanced consideration of failure recency, frequency to accommodate data corruption with good hit ratio. The stored data in GFCache is able to support fast read of the normal data access. Furthermore, since GFCache is a generic failure cache, it can be used anywhere erasure coding is deployed with any coding schemes and parameters. Evaluations show that GFCache achieves good hit ratio with our sophisticated caching algorithm and manages to significantly boost system performance by reducing unnecessary data recoveries with vulnerable data in the cache. For instance, compared to the current system without a failure cache, GFCache manages to reduce the average latency of a request to 12.19%. Also, GFCache achieves 24.62% hit ratio than 22.74% of CoARC in Subedi et al. [Subedi, Huang, Liu et al. (2016)] under workload h3.

The rest of the paper is organized as follows: Section II presents the background information. Section III illustrates the design of GFCache whereas Section IV evaluates with the experiments. Section V reviews the related work and Section VII concludes the paper.

## 2 Backgrounds and motivations

### *2.1 Erasure coding*

In comparison to replication, erasure coding essentially employs mathematical computation in the production of redundancy to provide data protection [Plank, Simmerman and Schuman (2008)]. In general, a stripe consisted of k +m partitions is used as the smallest independent unit to preserve the capability to reconstruct itself. Such reconfigurable capability is formed through an encoding process, where m parity partitions are produced by k data partitions through a matrix multiplication. In practice, as expressed in $\mathrm{D} \times G = \begin{pmatrix} D \\ P \end{pmatrix}$, a generator matrix G is used to form a mathematical relationship between the original data D and the generated parity P partitions, and any one of the k data partitions is protected. Furthermore, if a matrix formed by any k rows of the generator matrix G is invertible, any combination of no more than m partition failures can be restored through a similar reverse matrix multiplication process with k surviving partitions. Such desirable property is called MDS (maximum distance separable) property [Plank (2013)] and the process to regenerate corrupted data partitions are called data recovering. So far the most widely-used MDS code is RS code. In comparison, non-MDS codes feature non-uniform numbers of participants involved in the encoding and decoding processes. For example, in LRC codes [Huang, Simitci, Xu et al. (2012);

Sathiamoorthy, Asteris, Papailiopoulos et al. (2013)], fewer data blocks are needed to generate local parity blocks than that of global parity blocks.

### *2.2 Failure scenarios*

Data can be corrupted due to various reasons, ranging from software glitches, and hardware wear-out to possible human mistakes. Although a large school of studies focuses on gathering failure statistics, the time and place of a data corruption are near impossible to find beforehand. Therefore, before actual data loss occurs, the process to somehow restore the corrupted data is of great importance. In order to prioritize data recovery in different failure scenarios, a clear failure classification is used to divide failures into permanent and temporary, according to the lasting of an event. Permanent failures refer to the permanently lost data, like a node breakdown or a malfunctioning disk. Since a permanent failure often involves a large amount of data and may cause heavy damage, it is always under system surveillance to be alarmed in a timely manner. For example, in Hadoop, each node comprising the cluster reports its health to the metadata server through a periodical heartbeat mechanism. Once a permanent failure is confirmed, repairing efforts are to be scheduled at less busy hours to revive data in a new replacement. This is because such repairing work requires a great deal of cooperation from other nodes and takes a long time (e.g. days) to complete [Rashmi, Shah, Gu et al. (2015)].

Conversely, temporary failures indicate a brief unavailability of the data caused by non-destructive reasons, such as a persisting system hot spot, software upgrade. In essence, a temporary failure is transient and may revive later by itself. Instead of causing a damaging impact of data loss, a temporary failure often slows down the current access request. This is because a temporary failure cannot be found until the data is accessed. Due to such essence, a temporary failure needs to be dealt with right away to finish serving the ongoing request. In systems like Hadoop, a healthy node is randomly chosen to initiate a corresponding recovery by data read from other surviving nodes on the fly.

A large school of studies gathers failure statistics and show that temporary failures are more common, and account for the majority in distributed storage systems. Moreover, among all levels of failures, single failure accounts for the majority (99.75%), and multiple failures are not impossible [Pinheiro (2007); Ma (2015)].

## 3 Design

This section details the design of the proposed GFCache for an erasure-coded storage system.

### *3.1 Architecture*

Although big data storage systems feature good scalability with a distributed data storage cluster, clients still need to contact the centralized Metadata Server for metadata queries. For example, in Hadoop, an access request from the client is sent to the master NameNode, before the client contacts corresponding DataNode(s) for actual data access. This paper argues that providing a failure cache, which is installed in the Metadata Server, will not greatly impact the traditional flow of data access, but can significantly improve performance by avoiding unnecessary data recovery processes. Fig. 1 demonstrates the proposed architecture, with our GFCache installed upon a distributed storage system. In the system, two Metadata Servers manage the metadata and supervise a cluster of DataNodes. GFCache is implemented on top of an NVM storage device, such as an inexpensive, small-sized, solid-state drive (SSD). A failure cache management module controls both the promotion and the eviction of the data. The GFCache can be directly plugged into the Metadata Server, such as the NameNode in Hadoop, and can be shared by all clients. GFCache can also be installed as a standalone node, independent of the Metadata Server. Upon receiving queries from a client, the Metadata Server is able to check GFCache for speed access. Otherwise, a client continues to contact corresponding Data Servers to fetch the requested data. In terms of data recovery, recently failed data is stored in GFCache after being recovered from a designated Data Server as a background job with lower priority. In return, before performing a data recovery, the designated node (a Data Server) can check GFCache to gather participating data needed for reconstruction. As there is already a module for erasure-coding that maintains information of erasure-coded data (e.g. RaidNode in Hadoop), the Cache management module can communicate and coordinate for failure data eviction, such as the placement of the evicted data, and the update of the stripe metadata.

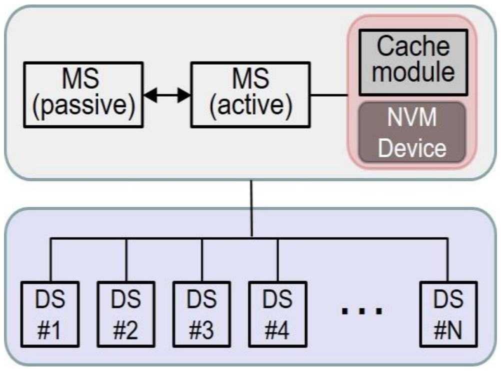

**Figure 1:** Architecture of adding GFCache

### *3.2 Greedy caching*

Storing the failed data in a cache enables the sharing of failure information, which thereby avoids the process of repeatedly regenerating the same data. However, simply caching currently unavailable data does not reduce redundant recovery processes caused by multiple unavailable data in one stripe. This is because, unlike node failure that can be monitored via the heart-beat mechanism, temporary data failure is unknown until attempts to access it are made. Each failed data will incur one degraded read, which must recover the data before it can be promoted to the failure cache, as shown in subplot (a) of Fig. 2. In this figure, unavailable data blocks B and C in stripe 1, experience a degraded read and is cached after their recovery.

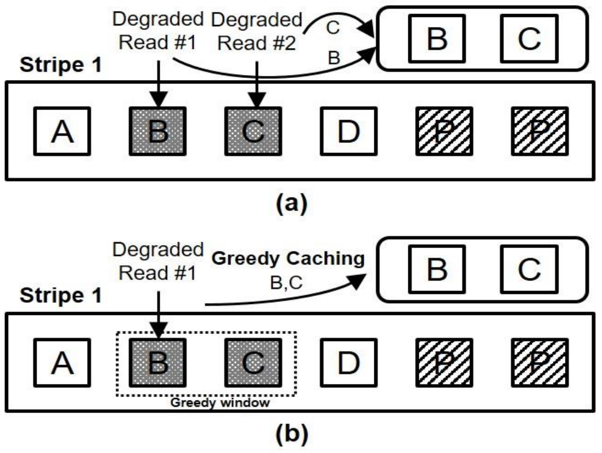

**Figure 2:** (a) The architecture and flow of a traditional big data storage system. (2) Our proposed architecture installs a NVM cache as a plug-in within the Metadata Server, which captures newly recovered data and supports fast data access

Therefore, aside from recurrences of data unavailability, GFCache also strives to diminish redundant data recovery processes of multiple data in a stripe. In regards to the uncertain status of data pre-access, GFCache employs greedy caching. In essence, this is a means of opportunistically pre-fetching increased amounts of data, needed for a better hit ratio. Such greedy practice is based upon the key insight, in which the whole information of a stripe can be recovered by a single reconstruction process, given any combination of no more than m failures under a (k, m) MDS erasure code. In other words, data produced by many recovery processes due to multiple failures happening in one stripe can actually be done in one single recovery, thus leading to a good save. The subplot (b) of Fig. 2 illustrates our greedy caching upon each degraded read recovery. We can see that upon the recovery of data block B, all the blocks of stripe 1 can be produced. GFCache greedily caches block B and block C, and thus resulting a failure cache hit when unavailability of data block C is encountered upon its access. In this way, the redundant recovery of block C is avoided.

Fundamentally, the greedy caching is opportunistic and how and how much of the greed is actually matters. In plain words, over-greed may lead to the abusive use of cache space and the eviction of more important failure data, while under-greed does not serve the purpose of reducing redundant recoveries. To this end, GFCache accordingly adopts three adjustments on respectively which data, and how much data to pre-fetch with greed, and how to manipulate such data in the cache. Firstly, GFCache caches sequential data after the current vulnerable block of the same stripe, e.g. the data block C behind data block B. This is due to the assumption that access locality may lead to failure locality. Additionally, GFCache maintains a greedy window of size m to reduce the abusive use of cache space. This is due to the upper bound of the failure tolerance of a (k, m) MDS erasure code. Adjustment of the window size can be made if using a non-MDS erasure code, like LRC codes. Last but not least, except for the actually corrupt data (e.g. block B), other data promoted to GFCache with greed (e.g. block C) is assumed with an unconfirmed possibility to failure in the near future. Therefore, GFCache treats such data as failure-likely and accordingly keeps them at the closest place to eviction, such that they will not occupy the cache space long if the prediction does not result in a hit. Pseudo codes of greedy caching are included in the replacement algorithm.

### *3.2 Failure caching replacement algorithm*

Essentially, the significance of a cache depends on how data is managed within the device. As the core of caching, various innovative and powerful cache replacement algorithms are proposed with regards to the incoming normal workload accesses. GFCache differs itself by caching newly recovered data, which undergoes temporary unavailability recently. In other words, instead of interacting with the normal access of healthy data, GFCache only functions when failure happens and functions as a static ROM for normal access without dynamic adjustments.

Since data failure statistics are quite random and do not follow a certain distribution, GFCache considers a comprehensive failure caching replacement algorithm with respect to both failure recency and failure frequency to aim for a higher hit ratio. In other words, this paper assumes that recently failed data is likely to fail again in the near future and a data which fails often is prone to failure again. By keeping more recently and frequently failed data longer in GFCache, more time is allowed for such temporarily unavailable data to revive healthy. The general idea behind this combined consideration can be expressed as $C=W\times R+(1-W)\times F$, in which, R stands for failure recency and F for failure frequency. F will increment by one if a data is promoted into cache due to actually corruption. For the failure-likely data which is cached by greed, the R is set to zero and the F does not increment for its corruption is not confirmed yet. Data with smallest C is evicted if the cache is full. Since the weights of failure recency and failure frequency are dynamically changing, an adaptive update of W is vital to maintaining a good hit ratio. Algorithm 3.1 provides details of our comprehensive caching algorithm with dynamic tuning, which gains inspiration from ARC algorithm [Megiddo and Modha (2003)].

In Algorithm 3.1, different treatments are applied separately to corrupted data and data cached by greed. If data is cached by its own corruption, Failure ARC (FARC) algorithm is used for adjustment. In comparison, data cached by greed is directly put into the place of GFCache for earliest eviction if it is a miss. For any evicted data, if its original copy comes back alive, it can be discarded. Otherwise, it is written back to its original node or a designated node. After that, the metadata information of the corresponding stripe is updated accordingly. Note that if the original residing node of the evicted happens to undergo a permanent node failure, the data can be directly written to the replacement node and save some reconstructing resources.

**Algorithm 3.1** Cache replacement algorithm of GFCache

```
Input: Newly recovered data blocks $d_1, d_2, ..., d_t$
Initialization: set $p = 0$ and set four LRU lists $T_1, B_1, T_2, B_2$ to empty.
for any $d_t$ do
    if $GC(d_t) \neq -1$ then
        $d_t$ is recovered due to its own unavailability
        FARC($d_t$)
    else
        $d_t$ is recovered due to Greedy Caching
        if Cache_Miss($d_t$) then
            Promote $d_t$ to the cache;
            Replace($d_t, p$);
        end if
    end if
end for

procedure REPLACE($d_t$, p)
    if $|T_1| > 0$ && ( $|T_1| > p$ || ( $d_t \in B_2$ && $|T_1| = p$ ) then
        pop_front($T_1$);
        Evict($d_t$);
        $B_1$.push_back($d_t$);
    else
        pop_front($T_2$);
        Evict($d_t$);
        $B_2$.push_back($d_t$);
    end if
end procedure

procedure FARC($d_t$)
    Apply ARC algorithm;
end procedure

procedure EVICT($d_t$)
    Write_Back($d_t$)
    Update_Stripe_Info($d_t$)
end procedure
```

## 4 Experiments

This section experiments with real-world traces to compare GFCache with other approaches. The other approaches are: (1) Without a failure cache (No Cache), which is common in current big-data storage systems, as shown in Fig. 1; (2) FARC (failure ARC), which represents a simple adoption of the classic ARC [Megiddo and Modha (2003)]; (3) CoARC from related work [Subedi, Huang, Liu et al. (2016)]. Note that, GFCache differs from FARC with our proposed greedy caching. As opposed to CoARC, GFCache features a greedy caching and the consideration of both failure recency and failure frequency.

### *4.1 Environments and workloads*

**Simulator** This paper adopts a trace-driven simulation method for evaluation purposes. Our simulator simulates a distributed storage system with a cluster of storage nodes and bases on PFSsim [Liu, Figueiredo, Xu et al. (2013)], which is widely used in various research works [Liu, Cope, Carns et al. (2012); Li, Dong, Xiao et al. (2012a, 2012b); Li, Xiao, Ke et al. (2014)]. Our simulator runs on node 19 of the computing cluster of VISA lab at ASU. The node has 2X Intel Xeon E5-2630 2.40 GHz processor and 62GiB of RAM with 2X1TB disk of Seagate 7200 and the model number is ST1000NM0033-9ZM. The operation system of the machine is Ubuntu 14.04.1 with Linux version 3.16.0-30. By default, all simulations emulate a cluster of 12 storage nodes with an RS (8, 4) code employed.

**Datasets** The real-world traces we adopt come from Chen et al. [Chen, Luo and Zhang (2011)], which are widely used in academic studies and prototype implementations. In detail, the CAFTL traces come from three representative categories, ranging from typical office work-loads (Desktop), big data queries (Hadoop) to transaction processing on PostgreSQL (Transaction). More detail on the collecting of the CAFTL traces can be found in Chen et al. [Chen, Luo and Zhang (2011)].

**Failure Creation** Since there are no existing failure traces of CAFTL workloads, randomization is used to generate data corruption to emulate degraded read. We set the total failure rate to be around 1% of the whole working data set. In detail, we corrupt random size of data in a random stripe to cause unavailability. Results are averaged of 20 runs, in which the total failure rate varies slightly but manages to conform to an expected normal distribution.

**Metrics** This paper adopts the latency and the hit ratio of the failure cache as the metrics to compare different approaches. Note that latency is averaged across all the requests and then normalized by that of GFCache to rule out the difference of units.

### *4.2 Effectiveness of a failure cache*

Fig. 3 shows the significant difference made on the average latency of a request between having and having not a failure cache. We can clearly see that with a failure cache, the average latency of a request has been reduced drastically. For instance, an 88.56% latency reduction is seen under workload d1, whereas the gap grows to almost 94% under workload h1. Although the performance gap varies, the system performance is significantly improved as a whole. The reason behind the boost is that with failed data cached, fast data access becomes possible with cache hits in GFCache in the following two situations. (1) Normal data access is boosted with a shortcut to check GFCache during its communication with the Metadata servers in the first place. Therefore, a degraded read can be bypassed with a cache hit. (2) During a degraded read, helper data participating the data recovery can be fast downloaded from the GFCache, instead of from the corresponding node. In other words, the fact that buffering corrupted data in a cache manages to make a boosting contribution to the system performance justifies the installment of a failure cache, considering the decreasing cost of a storage device.

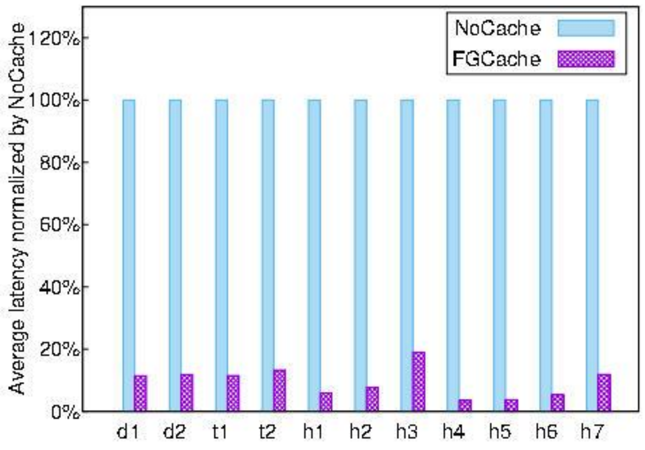

**Figure 3:** Effectiveness of a failure cache

### *4.3 Effectiveness of greedy caching*

Although it is straightforward to add a failure cache to an existing system, it fails to reduce redundant data reconstruction operations occurred in a stripe without our proposed greedy caching technique. In order to compare the difference made by the greedy caching, we implement a baseline failure cache called FARC (Failure ARC), which does not use the greedy caching technique.

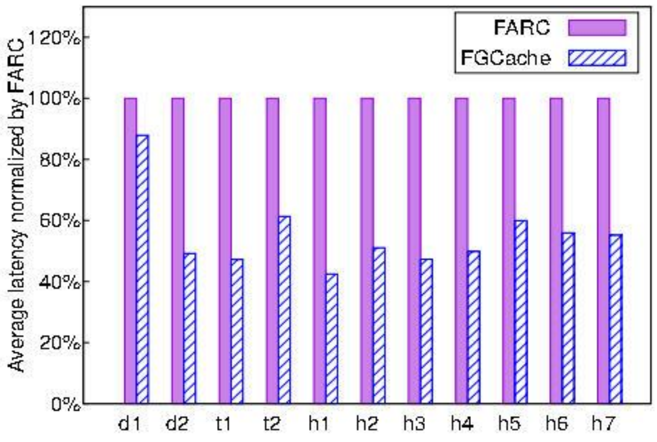

**Figure 4:** Effectiveness of the greedy caching

Fig. 4 provides the comparison of the average latency of a request between FARC and GFCache. In general, a shorter latency is experienced by a request under GFCache throughout all the workloads. The range starts from around 13% to 58% depending on the workload. For example, GFCache outperforms FARC by 13.06% under the workload of h1. This is because, with greedy caching, failure-likely data are aggressively cached in GFCache. If such opportunistic gambling with greed produces a cache hit in the near future, the performance is to be boosted without suffering redundant repairs of an otherwise de- graded read. If data cached with greed is a cache miss, little overhead is caused due to its early eviction from the GFCache.

### *4.4 Comparison with CoARC*

The most related work to our GFCache is CoARC from Subedi et al. [Subedi, Huang and Liu (2016)], which features an LRF (least-recently-failed) failure caching algorithm and an aggressive recovery of all other temporarily unavailable blocks in the same stripe. Fig. 5 and Fig. 6 respectively compare our GFCache with CoARC in the latency and the hit ratio.

In Fig. 5, a close performance in latency is witnessed under some workloads (e.g. h6), whereas GFCache contributes to larger latency reduction as opposed to CoARC in other cases. For instance, GFCache experiences a 5.74% smaller latency under d1 than that of CoARC. Under h1, CoARC is 8.46% slower than GFCache.

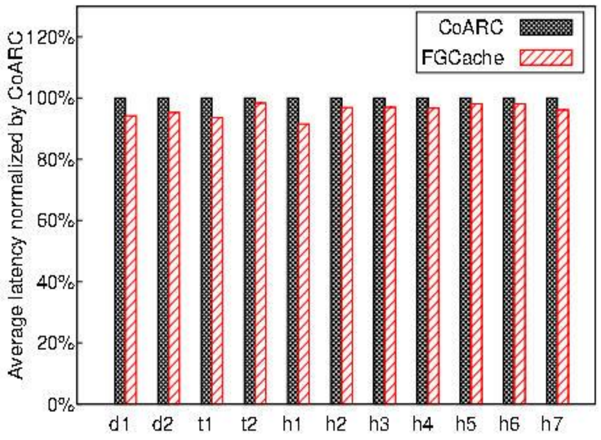

**Figure 5:** Latency comparison to CoARC

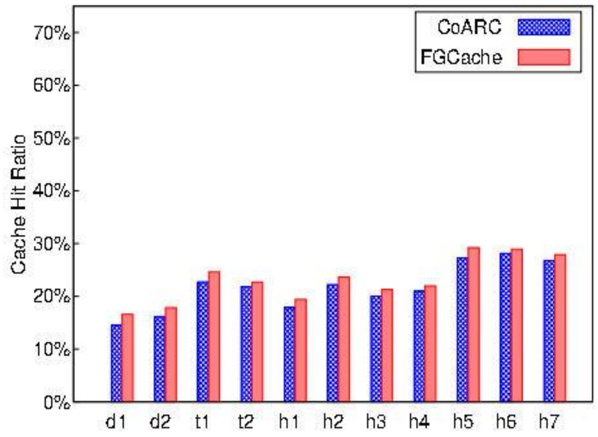

**Figure 6:** Hit ratio comparison to CoARC

In Fig. 6, unlike a normal cache, the hit ratio of a failure cache is generally low (no more than 30%) regardless of a specific replacement algorithm. This is largely due to two reasons. Firstly, as an input source, data failures are comparatively in much smaller amount than the normal data access. Secondly, the failure pattern of corrupted data is hard to capture even with caching algorithms which prove effective in a normal cache.

Regarding the contrast between GFCache and CoARC, the gap in between is not very big in general. For example, CoARC and GFCache exhibit a resembling hit ratio to be respectively 28.14% and 28.92% under h6. However, GFCache surpasses CoARC in other cases. Under d1, GFCache achieves a hit ratio of 16.59% in opposition to 14.51% of CoARC.

We argue those disparity gaps result from two aspects. One is that GFCache considers both failure recency and failure frequency to manage the cached data while CoARC's LRF only focuses on failure recency, thus leading to higher hit ratios in some cases. The other is that the aggressive approach in CoARC to recover all the failed data in a stripe needs to wait for the completion of identification of the last data, thus leading to idle wait time. In contrast, greedy caching adopted in GFCache causes no idle wait. However, greedy caching do incur cache miss due to its speculative opportunism.

## 5 Related works

This paper studies data recovery of an erasure-coded storage system with a failure cache. Therefore, we review related work in the following order.

**Data Recovery of Erasure Coding** A large school of works conducted excellent research in the area of enhancing data recovery of erasure coding, which can be classified into three categories: (1) designing new classes of coding algorithm to essentially reduce data needed for per recovery process [Dimakis, Godfrey, Wu et al. (2010); Huang, Simitci, Xu et al. (2012)]; (2) searching for a more efficient recovery sequence with less data reads [Khan, Burns, Plank et al. (2012)]; (3) proposing optimization in different system and network settings [Fu, Shu and Luo (2014); Shen, Shu, Lee et al. (2017)]. All these works focus on facilitating per recovery process of either single failure or multi-failure. In comparison, this paper treats the data recovery process as a black box and differentiates itself by reducing repeated and redundant recovery of failed data through buffering failed data. Therefore, this paper is perpendicular and complimentary to above work.

**Normal Caching** Caching is one of the oldest and most fundamental and use techniques in modern computing, which has been ubiquitously employed in nearly everywhere in the entire computational stack. Although various caching policies [Mattson, Gecsei, Slutz et al. (1970); Megiddo (2003)] have been proposed with different trade-off, the common purpose of such caches is to accommodate incoming normal data access, rather than failed data. Therefore, this paper contrasts itself with conventional caches by buffering a completely different source of data source, the temporary unavailable data.

**Failure Caching** In terms of failure caching in a setting of an erasure-coded storage system, very few research pays attention to the recurring data recovery problems. Sudedi et al. [Subedi, Huang, Liu et al. (2016)] treats per recovery process as a black box and first proposes CoARC, which essentially features a least-recently-failed (LRF) cache to buffering newly recovered data in order to eliminate repeated recoveries of the same data. This paper falls into the same track of reference [Subedi, Huang, Liu et al. (2016)]. However, this paper distinguishes itself with Subedi et al. [Subedi, Huang, Liu, et al. (2016)] in the following two aspects: (1) our GFCache employs a greedy caching policy to buffer all the data in a stripe upon its first recovery while CoARC waits to confirm all unavailable blocks in the failed strip to start recovery. (2) our GFCache features a more complicated eviction policy considering both failure recency and failure frequency while CoARC employs a simple LRU algorithm on failed data. Behinds, our GFCache is self-adaptive and scan-resistance while CoARC is not. Therefore, GFCache is able to achieve a higher hit ratio in general.

## 6 Conclusion

This paper proposes GFCache to address the repeated and redundancy data recoveries with the classic caching idea to buffer failed data. GFCache features a greedy caching each data recovery process and designs an innovative and self-adaptive caching replacement algorithm with a combined consideration of failure recency and failure frequency. Last but not least, cached data in GFCache provides fast read access to normal access workloads. Evaluations prove that GFCache achieves good hit ratio and manages to significantly boost system performance.

**Acknowledgement:** We would like to greatly appreciate the anonymous reviewers for their insightful comments. This work is supported by The National Key Research and Development Program of China (2016YFB1000302) and The National Natural Science Foundation of China (61433019, U1435217).

## References

**Cai, Z.; Wang, Z.; Zheng, K.; Cao, J.** (2013): A distributed tcam coprocessor architecture for integrated longest prefix matching, policy filtering, and content filtering. *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 417-427.

**Chen, F.; Luo, T.; Zhang, X.** (2011): Caftl: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, vol. 11, pp. 77-90.

**Dimakis, A. G.; Godfrey, P. B.; Wu, Y.; Wainwright, M. J.; Ramchandran, K.** (2010): Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539-4551.

**Fu, Y.; Shu, J.; Luo, X.** (2014): A stack-based single disk failure recovery scheme for erasure coded storage systems. *IEEE 33rd International Symposium on Reliable Distributed Systems*, pp. 136-145.

**Guo, Y.; Liu, F.; Cai, Z.; Xiao, N.; Zhao, Z.** (2018): Edge-based efficient search over encrypted data mobile cloud storage. *Sensors*, vol. 18, no. 4.

**Huang, C.; Simitci, H.; Xu, Y.; Ogus, A.; Calder, B. et al.** (2012): Erasure coding in windows azure storage. *Usenix Annual Technical Conference*, pp. 15-26.

**Huang, M.; Liu, Y.; Zhang, N.; Xiong, N.; Liu, A. et al.** (2018): A services routing-based caching scheme for cloud assisted CRNs. *IEEE Access*, vol. 6, pp. 15787-15805.

**Khan, O.; Burns, R. C.; Plank, J. S.; Pierce, W.; Huang, C.** (2012): Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. *Proceedings of the 10th USENIX Conference on File and Storage*, pp. 20.

**Li, X.; Dong, B.; Xiao, L.; Ruan, L.; Liu, D.** (2012a): CEFLS: a cost-effective file lookup service in a distributed metadata file system. *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 25-32.

**Li, X.; Dong, B.; Xiao, L.; Ruan, L.; Liu, D.** (2012b): HCCache: A hybrid client-side cache management scheme for I/O-intensive workloads in network-based file systems. *13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 467-473.

**Li, X.; Xiao, L.; Ke, X.; Dong, B.; Li, R. et al.** (2014): Towards hybrid client-side cache management in network-based file systems. *Computer Science and Information Systems*, vol. 11, no. 1, pp. 271-289.

**Li, Y.; Cai, Z. P.; Xu, H.** (2018): LLMP: exploiting LLDP for latency measurement in software-defined data center networks. *Journal of Computer Science and Technology*, vol. 33, no. 2, pp. 277-285.

**Liu, F.; Li, T.** (2018): A clustering-anonymity privacy-preserving method for wearable IoT devices. *Security and Communication Networks*, vol. 2018, pp. 1-8.

**Liu, N.; Cope, J.; Carns, P.; Carothers, C.; Ross, R. et al.** (2012): On the role of burst buffers in leadership-class storage systems. *IEEE 28th Symposium on Mass Storage Systems and Technologies*, pp. 1-11.

**Liu, Y.; Figueiredo, R.; Xu, Y.; Zhao, M.** (2013): On the design and implementation of a simulator for parallel file system research. *IEEE 29th Symposium on Mass Storage Systems and Technologies*, pp. 1-5.

**Ma, A.; Traylor, R.; Douglis, F.; Chamness, M.; Lu, G. et al.** (2015): Raidshield: characterizing, monitoring, and proactively protecting against disk failures. *ACM Transactions on Storage*, vol. 11, no. 4, pp. 1-28.

**Mattson, R. L.; Gecsei, J.; Slutz, D. R.; Traiger, I. L.** (1970): Evaluation techniques for storage hierarchies. *IBM Systems Journal*, vol. 9, no. 2, pp. 78-117.

**Megiddo, N.; Modha, D. S.** (2003): ARC: A self-tuning, low overhead replacement cache. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, vol. 3, pp. 115-130.

**Pinheiro, E.; Weber, W. D.; Barroso, L. A.** (2007): Failure trends in a large disk drive population. *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, vol. 7, pp. 17-23.

**Plank, J. S.** (2013): Erasure codes for storage systems: a brief primer. *Usenix Magazine*, vol. 38, no. 6, pp. 44-50.

**Plank, J. S.; Simmerman, S.; Schuman, C. D.** (2008): Jerasure: a library in C/C++ facilitating erasure coding for storage applications-Version 1.2. *Technical Report CS-08-*

*627*. University of Tennessee.

**Rashmi, K.; Chowdhury, M.; Kosaian, J.; Stoica, I.; Ramchandran, K.** (2016): Ec-cache: load-balanced, low-latency cluster caching with online erasure coding. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 401-417.

**Rashmi, K.; Shah, N. B.; Gu, D.; Kuang, H.; Borthakur, D. et al.** (2015): A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 331-342.

**Sathiamoorthy, M.; Asteris, M.; Papailiopoulos, D.; Dimakis, A. G.; Vadali, R. et al.** (2013): Xoring elephants: novel erasure codes for big data. *Proceedings of the VLDB Endowment*, vol. 6, pp. 325-336.

**Shen, Z.; Shu, J.; Lee, P. P.; Fu, Y.** (2017): Seek-efficient i/o optimization in single failure recovery for xor-coded storage systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 877-890.

**Subedi, P.; Huang, P.; Liu, T.; Moore, J.; Skelton, S. et al.** (2016): Coarc: co-operative, aggressive recovery and caching for failures in erasure coded hadoop. *45th International Conference on Parallel Processing*, pp. 288-293.

**Sun, W.; Cai, Z.; Li, Y.; Liu, F.; Fang, S. et al.** (2018): Security and privacy in the medical internet of things: a review. *Security and Communication Networks*, vol. 2018.

**Wang, Y.; Pei, X.; Ma, X.; Xu, F.** (2017): Ta-update: an adaptive update scheme with tree-structured transmission in erasure-coded storage systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 8, pp. 1893-1906.

**Wu, M.; Wu, Y.; Liu, C.; Cai, Z.; Xiong, N. et al.** (2018): An effective delay reduction approach through portion of nodes with larger duty cycle for industrial WSNs. *Sensors*, vol. 18, no. 5, pp. 1535.

**Yu, S.; Liu, X.; Liu, A.; Xiong, N.; Cai, Z.; Wang, T.** (2018): Adaption broadcast radius-based code dissemination scheme for low energy wireless sensor networks. *Sensors*, vol. 18, no. 5, pp. 1509.

**Zhang, H.; Cai, Z.; Liu, Q.; Xiao, Q.; Li, Y. et al.** (2018): A survey on security-aware measurement in SDN. *Security and Communication Networks*, vol. 2018.