Parity-Only Caching for Robust Straggler Tolerance

Mi Zhang, Qiuping Wang, Zhirong Shen, and Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong

***Abstract*—Stragglers (i.e., nodes with slow performance) are prevalent and incur performance instability in large-scale storage systems, yet it is challenging to detect stragglers in practice. We make a case by showing how erasure-coded caching provides robust straggler tolerance without relying on timely and accurate straggler detection, while incurring limited redundancy overhead in caching. We first analytically motivate that caching only parity blocks can achieve effective straggler tolerance. To this end, we present POCache, a parity-only caching design that provides robust straggler tolerance. To limit the erasure coding overhead, POCache slices blocks into smaller subblocks and parallelizes the coding operations at the subblock level. Also, it leverages a straggler-aware cache algorithm that takes into account both file access popularity and straggler estimation to decide which parity blocks should be cached. We implement a POCache prototype atop Hadoop 3.1 HDFS, while preserving the performance and functionalities of normal HDFS operations. Our extensive experiments on both local and Amazon EC2 clusters show that in the presence of stragglers, POCache can reduce the read latency by up to 87.9% compared to vanilla HDFS.**

I. Introduction

Large-scale clusters and storage systems often suffer from high performance variability or long tails for various reasons [20], such as hardware slowdown [24], [26], load imbalance [42], resource sharing [50], and workload skewness [15], [55]. Such performance variability and long tails are often caused by the presence of *stragglers* (also known as "gray failures" [30] or "fail-slow faults" [24]), which refer to the nodes that remain operational but with slow performance. Stragglers are problematic, as they easily introduce performance instability that degrades user experience.

Unfortunately, detecting and pinpointing stragglers is non-trivial and may take hours or even months, due to the complexity of root cause analysis and limited knowledge about the full hardware stack [24]. Some systems address straggler tolerance via *selective replication*, which caches replicas for popular objects [11], [18], [27], [47] so as to avoid accessing the stragglers (i.e., hotspots with overloaded requests) under skewed workloads. However, the popularity of objects can sharply change in a short period of time [31], and caching all objects is infeasible due to the high redundancy overhead of replication. Thus, selective replication is arguably ineffective given the limited available cache space [31], [42].

This motivates us to study how to provide *robust* straggler tolerance for distributed storage systems in practice; by robust, we mean that our straggler tolerance design does not rely on accurate detection of stragglers. We explore *erasure-coded caching*, which caches the erasure-coded blocks with limited redundancy penalty. Erasure coding has been widely studied in the literature to provide fault tolerance for distributed storage systems against fail-stop failures (e.g., [23], [29]). Here, we explore how erasure coding, coupled with caching, tolerates stragglers that cause performance variability and long tails.

Although previous studies have also explored erasure-coded caching (e.g., [9], [10], [25], [42]), there remain challenges to make erasure-coded caching feasible in practical distributed storage systems. First, the encoding and decoding operations of erasure coding add non-negligible latency to the I/O requests that access in-memory cache (e.g., 30% in EC-Cache [42]), which can degrade the normal I/O performance that is without coding operations. Second, designing an appropriate cache algorithm specifically for erasure-coded caching remains non-trivial; in particular, we need to address the issue of which erasure-coded data should be cached based on the file access popularity in the presence of stragglers. Finally, we should properly integrate erasure-coded caching into existing distributed storage systems, without changing the functionalities of normal operations.

In this paper, we propose POCache, a parity-only caching scheme that achieves robust straggler tolerance without relying on accurate straggler prediction. We show that caching only a small set of *parity blocks* (i.e., the redundant blocks encoded from file data) can effectively tolerate stragglers with limited caching and bandwidth overhead. We summarize our contributions as follows.

- We show via mathematical analysis that caching only parity blocks is more effective to bypass stragglers than caching only data blocks as in selective replication. Our analysis also provides insights that guide our POCache design.
- We design POCache, a parity-only caching scheme that provides robust straggler tolerance. POCache mitigates erasure coding overhead via two mechanisms, namely *block slicing* and *incremental encoding*, in which we partition blocks into smaller subblocks and parallelize coding operations at the subblock level.
- We design a *straggler-aware cache algorithm* that decides which parity blocks should be cached by taking into account both file access popularity and straggler estimation. Our algorithm takes a best-effort approach to cache the parity blocks of popularly accessed files based on our estimation of which nodes are likely to be stragglers. Note that our straggler estimation may be inaccurate (i.e., stragglers are falsely detected), yet POCache still provides robust straggler tolerance through parity-only caching.
- We implement a POCache prototype atop Hadoop 3.1 HDFS [4]. We show that our implementation preserves the

original I/O workflows, storage layouts, and fault tolerance of HDFS.

- We evaluate POCache on both local and Amazon EC2 clusters. We show that compared to reads in vanilla HDFS, POCache can reduce the read latency by up to 87.9% in the presence of stragglers, and suppress the read latency in the presence of stragglers to almost identical to that in the normal case where no straggler exists. We also conduct various experiments to justify the robustness of straggler tolerance of POCache.

The source code of our POCache prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/pocache**.

The rest of the paper proceeds as follows. Section II introduces the background details of erasure coding, and motivates that parity-only caching can reduce the probability of hitting stragglers. Section III presents the design of POCache. Section IV describes the implementation details of POCache on Hadoop 3.1 HDFS. Section V shows our evaluation results on both local and Amazon EC2 clusters. Section VI reviews related work, and finally Section VII concludes the paper.

II. Background and Motivation

In this section, we first introduce the basics (i.e., erasure coding, data layouts, and straggler impact), and analyze the effect of straggler tolerance by caching only parity blocks. We then pose the challenges of applying parity-only caching to distributed storage systems with different data layouts.

A. Basics

Erasure coding: Erasure coding provably incurs much less redundancy than replication under the same degree of fault tolerance [48]. At a high level, an erasure code is usually configured by two parameters namely n and k, where $n > k$. An (n, k) erasure code encodes, via Galois Field arithmetic [41], k fixed-size uncoded *data blocks* to generate another $m = n-k$ coded *parity blocks* of the same size, such that the collection of the n data and parity blocks forms a *stripe*. An erasure code is said to satisfy the *Maximum Distance Separable (MDS)* property if any k blocks in a stripe suffice to reconstruct (or decode) the original k data blocks. A storage system typically stores multiple stripes, each of which is independently encoded. A well-known family of MDS erasure codes is Reed-Solomon (RS) codes [43], which are extensively employed in current commodity storage systems (e.g., Ceph [49], QFS [40], and HDFS [4]). Erasure coding has been widely used to tolerate fail-stop failures [23], [29]. Also, many repair-efficient techniques (e.g., [34], [35], [38]) have recently been proposed to speed up repair operations in erasure coding. In this paper, we mainly explore how to couple erasure coding with caching, so as to provide straggler tolerance.

Data layouts: Distributed storage systems divide files into logical blocks and store them across multiple nodes by following either the *contiguous layout* or the *striping layout*. For the contiguous layout, the storage system stores sequential logical blocks across nodes (one block per node). This design significantly reduces disk seeks, but limits the parallel access. For the striping layout, the storage system decomposes a logical block into smaller units and stores them across nodes. Figure 1 depicts an example of both contiguous and striping layouts, where a file is partitioned into three logical blocks of size of 2 MiB each. In this example, the contiguous layout places one logical block in a node, while the striping layout breaks a logical block into two smaller units (i.e., 1 MiB per unit) and stores the resulting six units across three nodes. Therefore, reading a file in the contiguous layout can be simply done by retrieving the logical blocks one by one, while in the striping layout it needs to assemble the smaller units into the original file. Both of these layouts are now employed in current storage systems (e.g., HDFS of Hadoop version 3 [4]).

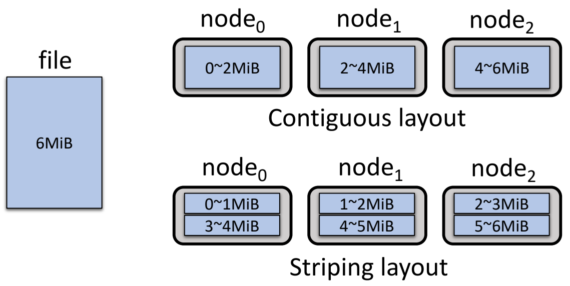


Fig. 1: Contiguous and striping layouts (only data blocks are shown here).

Stragglers: When reading a file, any straggler that stores the blocks of a file would increase the read latency, regardless of which data layout the distributed storage system adopts. For the contiguous layout, the penalty incurred by the stragglers is added to the overall read latency as the blocks are retrieved sequentially. For the striping layout, the latency of reading a file is equal to the time of reading from the slowest node.

B. Analysis

We show via a toy example how erasure coding addresses straggler tolerance. We first review the procedure of reading a file. Suppose that the k data blocks of a file are distributed in k storage nodes (see Figure 2). A client should retrieve all the k data blocks when reading the file, and the read time increases if any of the k storage nodes becomes a straggler. To bypass stragglers without altering the underlying data layout and fault tolerance, we can introduce a small group of nodes and let them cache some redundant data. Suppose that we introduce c cache nodes (where $c \leq k$) to cache c blocks (with one block per cache node). To read the file, the client can issue $k + c$ read requests to the k storage nodes and c cache nodes, and reconstruct the file once all its k data blocks are successfully received. Since the client has more choices to read the file, caching additional blocks can tolerate stragglers. Nevertheless, which kind of blocks to be cached still makes a difference.

Let p_s and p_c be the probabilities that a storage node and a cache node become a straggler, respectively. We consider two caching schemes: *data-only caching* and *parity-only caching*. Let P be the probability of hitting at least one straggler when

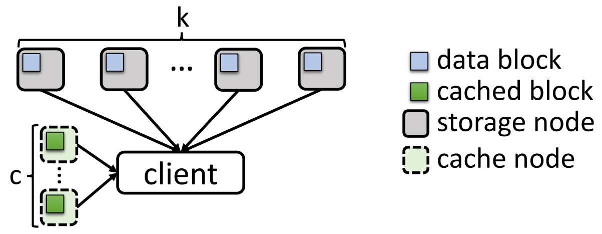


Fig. 2: Example of reading a file with caching in Section II-B.

reading a file. We calculate P under data-only caching and parity-only caching in the following analysis.

- **Data-only caching:** Data-only caching caches the replicas of a subset of data blocks in the cache nodes. Since we cannot tell certainly which nodes would become stragglers, we randomly select c out of k data blocks and cache them in the c cache nodes. Thus, in data-only caching, there are c blocks that have a replica stored in a cache node (in addition to the block copy in a storage node), and another $k-c$ blocks that have only a block copy stored in a storage node without being cached. Data-only caching can provide straggler tolerance if any i out of the c cached blocks (where $i \leq c$) have one replica residing in a straggler. We compute P under data-only caching as:

$$P = 1 - \sum_{i=0}^{c} \underbrace{\binom{c}{i} \cdot {p_s}^i \cdot (1-p_s)^{k-i}}_{i \text{ storage nodes are stragglers}} \cdot \underbrace{(1-p_c)^i}_{i \text{ cache nodes are normal}} .$$

  Note that selective replication also caches data blocks; in particular, it selects the data blocks that are most likely to reside in stragglers to cache. In our toy example, the probability that each storage node becomes a straggler is identical, so selective replication is actually identical to data-only caching that we consider here.
- **Parity-only caching:** Parity-only caching caches c parity blocks in the cache nodes, which are generated from k data blocks via an (n,k) MDS code (i.e., $n = k + c$). As any k out of the $k+c$ data and parity blocks can reconstruct the original file (i.e., the MDS property), we can retrieve any k blocks from the k storage nodes and c cache nodes. The probability P under parity-only caching is now equal to the probability that more than c blocks are stored in the straggler nodes, i.e.,

$$P = 1 - \sum_{i=0}^{c} \sum_{j=0}^{i} \underbrace{\binom{k}{j} \cdot {p_s}^j \cdot (1-p_s)^{k-j}}_{j \text{ storage nodes are stragglers}} \cdot \underbrace{\binom{c}{i-j} \cdot {p_c}^{i-j} \cdot (1-p_c)^{c-i+j}}_{(i-j) \text{ cache nodes are stragglers}} .$$

Figure 3 depicts P under no-caching (i.e., $c = 0$), data-only caching, and parity-only caching for different combinations of k, c, p_s and p_c. Figure 3(a) plots the probability versus k with $c = 1$, $p_s = 0.005$, and $p_c = 0.005$. P increases linearly with k under no-caching and data-only caching, while parity-only caching remains to have a small value of P as k increases. Figure 3(b) plots P versus c with $k = 4$, $p_s = 0.005$, and $p_c = 0.005$. For parity-only caching, caching one parity block already keeps P very low (2.48E-4), and further increasing c only reduces P slightly. Figure 3(c) plots P versus p_s with $k = 4$, $c = 1$, and $p_c = 0.005$. Parity-only caching with $c = 1$ can still keep P very low even when p_s increases to 0.01, while P under data-only caching increases linearly with p_s. Figure 3(d) plots P versus p_c with $k = 4$, $c = 1$, and $p_s = 0.005$. It shows that p_c has a negligible effect on the probability of hitting stragglers. For data-only caching and parity-only caching, P with $p_c = 0.01$ remains almost the same as that when each cache node would never be a straggler (i.e., $p_c = 0$).

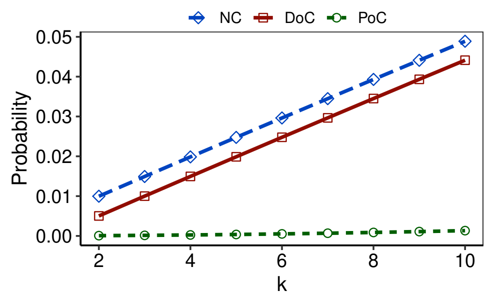
To summarize, parity-only caching can effectively bypass stragglers. Compared to caching specific data blocks, we find that caching *only one* parity block can effectively eliminate the impact of stragglers. Moreover, even though the cache nodes may also become stragglers by themselves, the overall straggler tolerance can still be maintained with parity-only caching.

C. Challenges

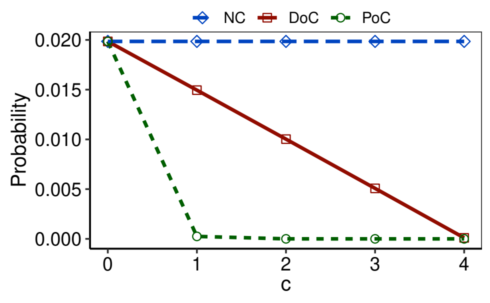
While the above analysis demonstrates the effectiveness of parity-only caching in straggler mitigation, three challenges still remain when we apply parity-only caching to real-world distributed storage systems.

First, applying erasure coding to large-size blocks easily incurs non-negligible decoding (resp. encoding) overhead to the read (resp. write) path, thereby increasing the read (resp. write) latency. To demonstrate, we measure the latencies of encoding and decoding operations of the erasure coding library ISA-L [5] using the `RawErasureCoderBenchmark` tool in Hadoop 3.1. Table I shows the time cost of network transfer, memory access, encoding, and decoding versus different block sizes for RS codes under $(n,k) = (5,4)$. We can see that the memory access as well as the encoding and decoding operations have comparable latencies, where the total latency in memory access and encoding/decoding accounts for approximately 33% of the latency in network transfer. Similar observations are also validated by EC-Cache [42], in which the decoding time takes about 30% of the read time. However, previous studies (e.g., EC-Cache [42] and Sprout [10]) do not address the encoding and decoding overhead when employing erasure-coded caching. Our measurement indicates that to sustain the performance improvement of parity-only caching, one should carefully conduct the encoding and decoding operations in the I/O path.

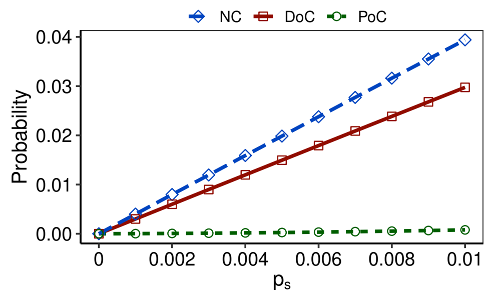
Second, how to design an efficient cache algorithm to mitigate the impact of stragglers remains a challenging issue. Many cache algorithms aim to maximize the hit ratio (i.e., the ratio that the requested data has been cached) by making caching decisions based on the file access pattern only. For example, EC-Cache [42] directly employs the least recently used (LRU) cache algorithm, which is also the default cache



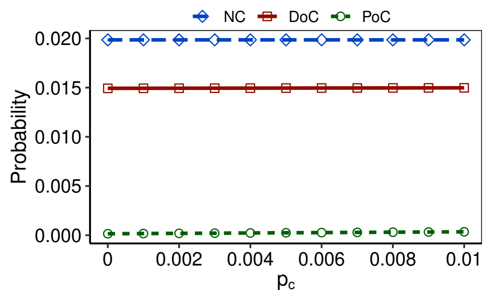
(a) $c = 1$, $p_s = 0.005$, $p_c = 0.005$



(b) $k = 4$, $p_s = 0.005$, $p_c = 0.005$



(c) $k = 4$, $c = 1$, $p_c = 0.005$



(d) $k = 4$, $c = 1$, $p_s = 0.005$

Fig. 3: Probability P that a read hits a straggler for different combinations of k, c, p_s, and p_c under no-caching (NC), data-only caching (DoC), and parity-only caching (PoC).

TABLE I: Latencies versus block size for RS codes under $(n, k) = (5, 4)$.

Block Size	Network Transfer	Memory Access	Encoding or Decoding
16 MiB	51.2 ms	9.7 ms	7.4 ms
32 MiB	102.4 ms	19.2 ms	14.8 ms
64 MiB	204.8 ms	38.4 ms	29.7 ms
128 MiB	409.6 ms	76.6 ms	56.3 ms

Note that the time of network transfer is calculated as transferring four blocks through 10 Gbps network.

algorithm in Alluxio [33]. Existing cache algorithms may have a high probability of hitting stragglers, since they do not take into account the presence of stragglers.

Last but not least, our parity-only caching design should be independent of the underlying storage systems; in other words, our design can be generalized for different storage systems and support the upper-layer applications. The dependency on the characteristics of a specific storage systems can restrict the application of the design to other systems; for example, Sprout [10] assumes the caching data resides on the client side or in a proxy-based caching tier. Furthermore, it is important that our design should have limited overhead on the I/O workflows of the underlying storage systems.

III. POCache Design

We present POCache, a parity-only caching approach that provides robust straggler tolerance for distributed storage systems. POCache aims for the following goals: (i) mitigating the encoding and decoding overhead to avoid degrading I/O performance; (ii) managing the cache space to decrease the probability of hitting stragglers for I/O requests; (iii) preserving the data layouts and access protocols of the underlying storage systems to achieve generality.

We summarize the main ideas of POCache as follows. We first mitigate coding overhead via block slicing and incremental encoding (see Section III-A) to exploit the full parallelism of encoding and decoding operations. Note that both features have been shown to significantly reduce the repair latency in erasure-coded storage [36]; here, we leverage these features to mitigate the encoding and decoding overhead and hence achieve effective straggler tolerance. We then design a straggler-aware cache algorithm that carefully manages the cache space based on the file access popularity and the estimation of the straggler existence (see Section III-C).

We assume that parity blocks are generated on a per-file basis, such that each file is of large size and spans $k > 1$ data blocks that can be encoded together. Such an assumption holds for cloud storage workloads. For example, Microsoft OneDrive is reportedly dominated by large objects, in which almost 90% of objects are over 100 MiB [17].

A. Mitigating Coding Overhead

Block slicing: Our observation is that the actual erasure coding functionalities under Galois Field arithmetic (see Section II) perform in small-size coding units (e.g., bytes) [41]. Specifically, the n blocks of a stripe are divided into coding units, such that the coding units at the same offset across the n blocks are independently encoded/decoded. Thus, we can slice blocks into smaller-size subblocks (e.g., 1 MiB) and perform encoding/decoding at the subblock level. Figure 4 shows the idea of block slicing, in which the subblocks at

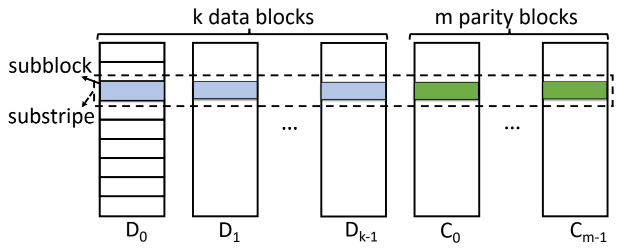


Fig. 4: Block slicing. D_i and C_j respectively denote the i-th data block and the j-th parity block, where $0 \leq i \leq k-1$ and $0 \leq j \leq m-1$.

the same offset of the n blocks in a stripe form a *substripe*. Instead of encoding k large blocks to generate parity blocks for the whole stripe, we divide the stripe into multiple substripes, in which we encode k data subblocks in each substripe to generate $m = n - k$ parity subblocks. Since the substripes are independently encoded/decoded, the encoding/decoding operations across different substripes can be parallelized, and the encoding/decoding overhead can be masked. For each parity block being cached, we now cache its corresponding parity subblocks.

Note that block slicing applies to both contiguous and striping layouts. For simplicity, for the contiguous layout, we assume that the block size is divisible by the subblock size; for the striping layout, we assume that each subblock is a multiple of the unit size.

Incremental encoding: In addition to block slicing, we employ incremental encoding to further exploit the parallelization of encoding when we generate parity subblocks to cache. Our observation is that in practical erasure codes (e.g., RS codes), a parity subblock is encoded via a linear combination of k data subblocks, and the addition operations are *associative*. Thus, we can incrementally compute a parity subblock from the data subblocks one by one, instead of waiting for all k data subblocks to be available before starting the encoding operation.

We use an example to elaborate the idea. Suppose that a parity subblock c_0 is generated from $k = 3$ data subblocks d_0, d_1, and d_2 as: $c_0 = \alpha_0 d_0 + \alpha_1 d_1 + \alpha_2 d_2$, where α_0, α_1, and α_2 are encoding coefficients, and both the addition and multiplication operations are in Galois Field arithmetic. Incremental encoding decomposes the encoding operation into three steps: (i) $c'_0 = \alpha_0 d_0$ (when d_0 is written); (ii) $c''_0 = c'_0 + \alpha_1 d_1$ (when d_1 is written); and (iii) $c_0 = c''_0 + \alpha_2 d_2$ (when d_2 is written), where c'_0 and c''_0 are intermediate results. Thus, incremental encoding can start the encoding operation as soon as a data subblock arrives while a stream of data subblocks is written, and both the write and incremental encoding operations are done in parallel.

B. Choice of (n, k) Erasure Codes

We now discuss how to select an appropriate (n, k) erasure code, so as to tolerate stragglers effectively while achieving low usage of the cache space.

Selection of k: The selection of k determines the cache space usage. With a smaller k, POCache forms more stripes for a file and hence caches more parity blocks. In practice, the value of k is jointly determined by the distribution of file sizes and the available capacity of the cache space. We evaluate the impact of different values of k in Section V.

Selection of n: Recall that caching only one parity block can reduce the probability of hitting stragglers even under different values of k (see Figure 3(a)). Thus, we set n, such that $m = n - k = 1$, in our implementation and evaluation based on the value of k.

C. Straggler-Aware Cache Algorithm

We propose a straggler-aware cache (SAC) algorithm to manage the cache space, with the objective of minimizing the *straggler hit ratio* (i.e., the ratio of read requests that hit stragglers). We design SAC by taking into account both the file access popularity and the estimation of the existing stragglers.

File access pattern: The access patterns in real-world storage workloads are usually *skewed* and obey *temporal locality*. By skewed, we mean that a small fraction of files usually occupies a large proportion of file accesses (e.g., following a Zipf distribution [42], [44]). By temporal locality, we mean that the files that are accessed recently are more likely to be accessed again in near future. For example, it is reported that 75% of the re-accesses will occur within six hours in real-world workloads [16]. Thus, SAC prefers to cache parity blocks for recently accessed files.

Straggler estimation: SAC considers the existence of stragglers when making decisions on caching. To minimize the straggler hit ratio, SAC prefers to cache parity blocks for the files that would be affected by the stragglers. Thus, SAC estimates the presence of stragglers based on the monitoring information.

SAC identifies the straggler nodes and records them in a *straggler list* (S) for making decisions later. The straggler estimation procedure proceeds as follows. SAC collects each node's *service rate* (ν) and calculates the mean value (μ) and standard deviation (σ) of all service rates. Here we define the service rate of a node as the ratio of the amount of data that it serves to the service time that it takes. Then SAC identifies the stragglers according to the *three-sigma rule* [3], in which the nodes whose $\nu < \mu - 3\sigma$ are treated as abnormal and included in the straggler list.

Putting it all together: SAC caches parity blocks for the recently accessed files that are affected by the existing stragglers and evicts the least-recently-accessed files. Specifically, SAC caches the parity blocks for a file if there exists a node that stores the data blocks of the file and is recorded in the straggler list. We use a linked list L to record the files whose parity blocks are cached. Files in L are sorted by the time of the last access in descending order (i.e., the head of L is the file that is the most recently accessed and the tail records the file that is the least recently accessed). Note that if no straggler node is detected (i.e., S is empty), SAC makes decisions on caching

Algorithm 1 Straggler-Aware Cache Algorithm

```
1: Initialize the available caching space $A$
2: Create a linked list $L$
3: Given the latest list of straggler nodes $S$
4: function QUERY($f$)
5:     if $f$ is in $L$ then
6:         return cached
7:     else if $S$ is empty OR some nodes in $S$ store data blocks of $f$ then
8:         return shouldCache
9:     else
10:        return shouldNotCache
11:    end if
12: end function
13: function UPDATE($f$, DECISION)
14:    Initialize the list of files to be evicted, $E = \{\}$
15:    if DECISION == cached then
16:        Move $f$ to the head of $L$
17:    else if DECISION == shouldCache then
18:        $n \leftarrow$ number of parity blocks to cache for $f$
19:        while $A < n$ do
20:            $e \leftarrow$ file at the tail of $L$
21:            $A \leftarrow A$ + number of parity blocks of $e$ in cache
22:            Add $e$ to $E$
23:        end while
24:        $A \leftarrow A - n$
25:        Add $f$ to the head of $L$
26:    end if
27:    return $E$
28: end function
```

based on file access popularity only, which we now use the least-recently-used (LRU) algorithm.

Algorithm 1 elaborates the workflow of SAC. We first initialize the caching space A (in unit of blocks) and create a linked list L (lines 1-2). SAC makes decisions on caching according to the latest straggler list S, which is updated periodically (line 3). For every read request to a file f, SAC calls a function QUERY. In the function of QUERY, SAC returns `cached` if the parity blocks of f have been cached (lines 5-6). When f is not cached, SAC decides to cache its parity blocks by returning `shouldCache` if the straggler list is empty or the estimated stragglers store data blocks of f (lines 7-8); otherwise, it refuses to bring f into cache by returning `shouldNotCache` (lines 9-11). Then, another function UPDATE is called with the result of QUERY which is named DECISION. In the function UPDATE, SAC takes different steps given the inputs (i.e., f and DECISION). If DECISION is `cached` (i.e., the parity blocks of f have been cached before), then SAC only updates L by moving f to the head of L (lines 15-16), as f is the most recently accessed file now. If DECISION is `shouldCache` (i.e., the parity blocks of f should be cached but have not been cached before), SAC evicts the least-recently-accessed files until there is enough cache space (lines 17-23), caches the parity blocks of f (line 24), and adds f to the head of L (line 25). For `shouldNotCache`, SAC does not take any operations. Note that UPDATE returns the list of files to be evicted for managing the cache space (line 27).

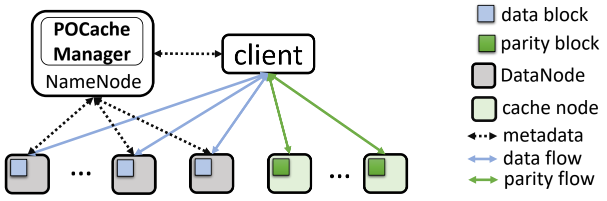


Fig. 5: Integration of POCache into HDFS.

IV. Implementation

We implement POCache on Hadoop 3.1 HDFS [4] and describe the implementation details as below.

A. Reads in HDFS

HDFS is a distributed file system that stores data across multiple servers in a fault-tolerant manner [45]. It comprises a single *NameNode* and multiple *DataNodes*. The NameNode is in charge of maintaining the system metadata and managing the file system namespace, while DataNodes are responsible for storing the file data in units of fixed-size blocks. To protect against data loss, the earlier versions of HDFS employ replication as the only redundancy mechanism and store multiple copies (i.e., replicas) for each block in different DataNodes. Since Hadoop 3, HDFS supports both replication and erasure coding as the redundancy techniques.

Hadoop 3.1 HDFS adopts different data layouts for replication and erasure coding. For replication, it adopts the contiguous layout, where each logical block is stored alone in each node; while for erasure coding, it employs the striping layout, where each logical file block is divided into smaller units called *cells* that are distributed across multiple DataNodes (see Section II-A). POCache works for both contiguous and striping layouts.

Since Hadoop 2.4, HDFS provides straggler tolerance via *hedged reads* (for replication-based storage only). Specifically, if a client issues a read request and receives no response after some time (called the *waiting threshold*), it issues another read request to a block replica stored in a different DataNode. It uses the earliest response of the two issued read requests as the result. Obviously, the performance of hedged reads is related to the waiting threshold, where a small threshold may falsely generate duplicate requests while a large threshold prolongs the latency.

B. Integration

Figure 5 shows how POCache is implemented based on HDFS. We highlight the implementation details below.

Manager: Inside the NameNode, we add a module called *Manager*, which tracks the cached parity blocks and makes decisions on caching according to the cache algorithms.

When the client writes a file to HDFS, the NameNode is first contacted and it asks the Manager whether to cache its parity blocks or not. As the newly written files are likely to be accessed again later [11], the Manager allows to cache their parity blocks if there remains enough cache space. Note that the

NameNode does not know the file size until the ending of the write process. Thus, we prepare to reserve enough available space for a file to cache its parity blocks. The maximum number of parity blocks of a file depends on several parameters, including the largest file size that POCache can cache for during writes, block size, and the number of data blocks per stripe (k), all of which are specified in the configuration.

When the client reads a file, the NameNode first identifies the block locations of the file. It then checks with the Manager whether there is any cached parity block for the file. If so, the locations of both data blocks and the parity blocks are returned to the client. We adopt *proactive reads*, in which the client issues reads to the k data blocks (i.e., the original file) and the cached parity blocks, such that the client can immediately decode the file once receiving any k blocks (either data or parity blocks). Proactive reads eliminate the impact of the waiting thresholds as in hedged reads, and do not need to know in advance which DataNodes are the stragglers. Since the redundancy overhead of erasure coding (i.e., $\frac{m}{k}$) is limited and modern data centers now typically have sufficient network bandwidth (e.g., 10 Gbps or even larger), the additional network traffic due to the extra reads has limited impact on read performance.

To support various cache algorithms, the Manager adopts a generic interface consisting of two primitive APIs:

- `Query`, which returns the admission decision made by the cache algorithm for a file. This function is called for every file request. For a file that is not cached, it returns `shouldCache` if the cache algorithm decides to cache its parity blocks; otherwise, it returns `shouldNotCache`. For the files that has been cached, it returns `cached`.
- `Update`, which updates related information and returns the eviction decision made by the cache algorithm. This function is called with the result of Query. The information to be updated depends on the cache algorithm. For example, SAC updates its linked list to record the file access pattern. The returned value is a list of files to be evicted. If there is no need to perform eviction, the returned value is an empty list.

The workflow of SAC provides an example of implementing this interface, where the Manager maintains a separate thread to update the straggler list periodically. Moreover, we implement least recently used (LRU), least frequently used (LFU), and adaptive replacement cache (ARC) following the above APIs. We evaluate the performance of different cache algorithms in Section V.

Client: We modify the HDFS client to augment its read/write operations to support POCache. Specifically, for writes, we modify the classes `DFSOutputStream` and `DFSStripedOutputStream` to generate parity blocks for any newly written files. For reads, we modify the classes `DFSInputStream` and `DFSStripedInputStream`, such that the metadata of the cached parity blocks of the file being read is transferred to the client together with the block locations. The HDFS client implements both block slicing and incremental encoding, and interacts with the Manager on the caching operation. Note that we do not modify applications that run atop HDFS to ensure that our integration is transparent to the upper-layer applications. That is, they can still issue normal reads/writes through the HDFS client interface.

DataNode: Each DataNode monitors a `sendBlock` function (i.e., the function that sends data from the storage node to the client) by recording the amount of data sent and elapsed time. Then, every DataNode reports its service rate to the Manager through periodical heartbeats for the maintenance of the straggler list.

Cache: We implement the cache as a key-value store using Redis [7]. We use the Java client interface, Jedis [6], to connect to the Redis cache. Note that the cache can be deployed alongside the DataNodes in the HDFS cluster. Our microbenchmarks (see Section V-B) show that the read performance of the Redis cache is faster and more stable than that of a DataNode. Here, we adopt strong consistency to ensure that the parity blocks in cache are correct and the latest.

V. Evaluation

We now show via evaluation that POCache effectively provides straggler tolerance for Hadoop 3.1 HDFS under both contiguous and striping layouts. Our evaluation aims to answer the following questions:

- What is the read performance of POCache with and without stragglers compared to other read mechanisms?
- What are the performance breakdowns of read/write operations in POCache? Can block slicing and incremental encoding effectively mitigate the overhead of coding operations?
- Can the SAC algorithm effectively manage the cache space?

We evaluate POCache on a local cluster (see Sections V-A-V-C) and Amazon EC2 (see Section V-D). The local cluster provides a controlled environment for us to configure the existence of stragglers, while Amazon EC2 enables us to evaluate the natural existence of stragglers in open environments.

A. Read Performance

We first evaluate the read performance of POCache with and without stragglers on a local cluster.

Local cluster setup: Our local cluster consists of 15 machines, each of which is installed with Ubuntu 16.04 and has a quad-core Intel Core i5-3570 3.40GHz CPU, 16 GiB RAM, and a Seagate ST1000DM003 7200RPM 1 TiB SATA disk. All machines are connected via a 10 Gbps Ethernet switch. We use one dedicated machine for data caching by running Redis, and deploy Hadoop 3.1 HDFS on the remaining 14 machines. In the deployment of HDFS, we run the NameNode on one machine, and execute a variable number of DataNodes and HDFS clients on the remaining 13 machines (see details below). We employ the benchmarking tool DFS-Perf [1] to generate read workloads and collect the elapsed time results of all I/O requests. To inject a straggler on the local cluster, we choose a DataNode and run the Linux tool `stress` [8], which issues an extensive amount of I/O requests to exhaust the local I/O resources.

By default, we set the block size and the subblock size as 64 MiB and 1 MiB, respectively, and allocate the cache space to store 100 blocks (i.e., $A = 100$ in Algorithm 1). We set $k = 4$, such that a file is composed of four data blocks. We focus on single-client performance, while we also evaluate multi-client performance.

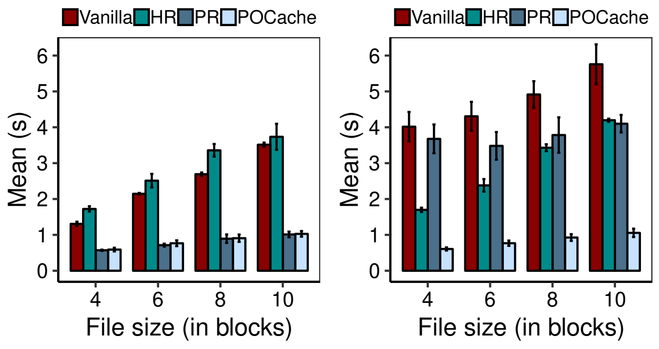
Experiment 1 (Single-client reads under the contiguous layout): We first consider the contiguous layout, in which one client issues a read request to a file of size equal to k blocks. We compare POCache with the following: (i) the default reads in HDFS (named Vanilla), (ii) hedged reads in HDFS (HR) (see Section IV-A), and (iii) parallel reads (PR), in which a client issues reads to multiple data blocks in parallel without deploying POCache.

For both Vanilla and PR, we deploy k DataNodes and store each of the k blocks in one DataNode. We disable replication, so any straggler DataNode is expected to slow down the file read. For HR, we use $k + 1$ DataNodes and 2-way replication (i.e., two replicas for each block), so that if one of the DataNodes becomes a straggler, HR can read the other replica from the remaining normal k DataNodes. Also, we set the waiting threshold of HR as zero (i.e., HR always issues duplicate requests to read a block), since our evaluation shows that in this setting HR can achieve the best performance in the presence of stragglers. For POCache, we deploy k DataNodes to store the k data blocks and cache one parity block in a separate cache node. We repeat each experiment for ten runs, and show the average result with the standard deviation represented by the error bars.

Figures 6(a) and 6(b) depict the average single-client read latencies without any straggler and with one straggler, respectively. Without any straggler, PR and POCache have the smallest read latency as they retrieve the blocks in parallel; HR has a higher latency than Vanilla because HR introduces additional overhead by doubling all I/O requests. In the presence of a straggler, the read latencies of Vanilla and PR increase with high variance. POCache reduces the average read latency by 81.7-85.2% and 40.4-77.5% compared to Vanilla and HR, respectively. This shows that straggler tolerance is attributed to parity-only caching rather than issuing parallel reads.

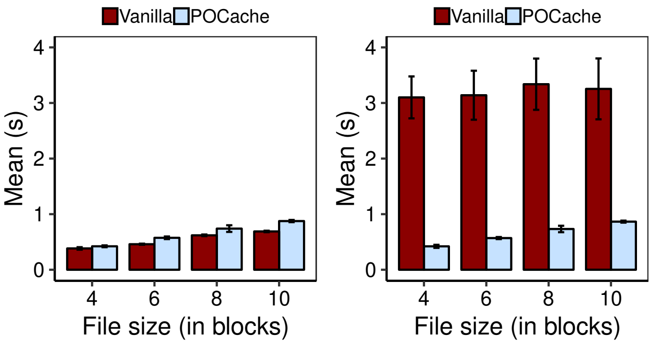
Experiment 2 (Single-client reads under the striping layout): We consider single-client reads under the striping layout. Note that HR is not supported under the striping layout, and Vanilla retrieves data blocks in parallel like PR. Thus, we consider Vanilla and POCache only. We use the same deployment as in the contiguous layout.

Figures 7(a) and 7(b) depict the average single-client read latencies without any straggler and with one straggler, respectively. When there is no straggler, Vanilla and POCache have similar read latencies, although POCache has a slightly higher latency for the same file size. The reason is that



(a) Without any straggler (b) With one straggler

Fig. 6: Experiment 1 (Single-client reads under the contiguous layout).



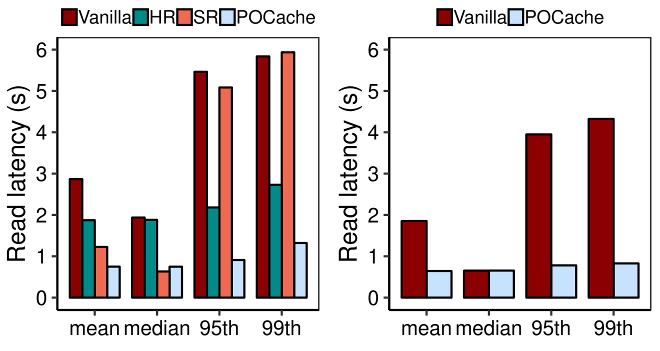
(a) Without any straggler (b) With one straggler

Fig. 7: Experiment 2 (Single-client reads under the striping layout).

POCache reads one more parity block for each request than Vanilla and performs decoding operation if the first k received blocks include the parity block. However, when there is a straggler, the latencies of Vanilla increase to almost five times of those without straggler for all file sizes. POCache keeps the latencies as in the case without any straggler, and reduces the latencies of Vanilla by 73.0-87.9%.

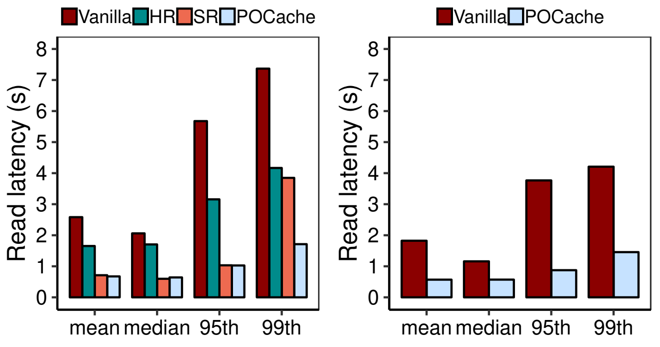
Experiment 3 (Impact of skewed workload): We study the latency characteristics under skewed read workload. We first generate a skewed workload as follows. We write 100 four-block files to HDFS and then issue 2,000 reads to the files following a Zipf distribution with a Zipfian constant of 0.9. Note that for the contiguous layout, we replace PR with selective replication (SR) in the comparison. SR not only inherits the read parallelism from PR, but also resists against stragglers by caching data blocks of the popular files. Thus, comparing POCache with SR allows to evaluate whether caching parity blocks can achieve more performance gains than caching data blocks.

Figure 8 illustrates the read latencies under the contiguous and striping layouts. For the contiguous layout, by caching a number of popular blocks, SR has a lower median latency than HR. However, the tail latencies of SR sharply increase and are higher than those of HR, as some blocks in the straggler are not cached. POCache always achieves the lowest latency and it reduces the 95th-percentile latency by 83.3% compared to



(a) Contiguous layout (b) Striping layout

Fig. 8: Experiment 3 (Impact of skewed workload).



(a) Contiguous layout (b) Striping layout

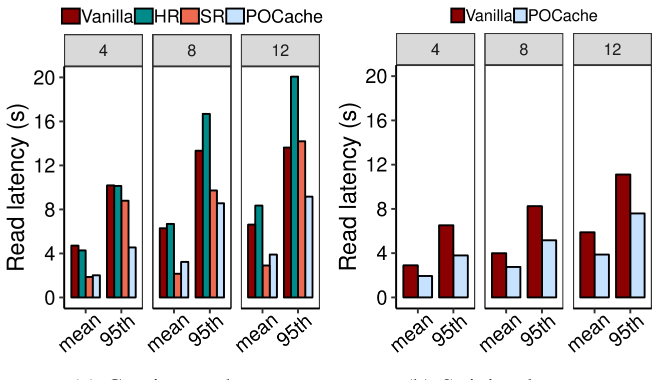
Fig. 9: Experiment 4 (Impact of multi-file workload).

Vanilla. For the striping layout, Vanilla is seriously affected by the straggler, where its 99th-percentile latency is about seven times of the median latency. In contrast, POCache keeps stable latencies in the presence of stragglers.

Experiment 4 (Impact of multi-file workload): We write 100 files of different sizes into HDFS, in which there are 15 512-MiB files, 30 256-MiB files, 17 128-MiB files, and 38 64-MiB files, by following the characteristics of data stored in the Facebook cluster [42]. We set $k = 4$ and and $m = 1$. Figure 9 depicts the read latencies. We can observe that POCache can still achieve the lowest read latency. Compared to Vanilla, POCache reduces the read latency by 68.9-81.9% for the contiguous layout, and 50.8-76.8% for the striping layout, respectively.

Experiment 5 (Impact of multi-client reads): We study the read performance with multiple clients. We deploy 4, 8, and 12 clients in the 15-machine cluster; note that some clients may be co-located with a DataNode in the same machine. We first write 100 four-block files into HDFS, and then generate a workload with the same skewness for each client. In this experiment, we consider the average and 95th-percentile latencies as the latter reflects the tail latency.

Figures 10(a) and 10(b) show the read latencies under the contiguous and striping layouts, respectively. In general, the read latencies of all mechanisms increase when more clients issue read requests. Compared to SR, POCache has a slightly



(a) Contiguous layout (b) Striping layout

Fig. 10: Experiment 5 (Impact of multi-client reads).

larger mean latency, but a much lower 95th-percentile latency. Compared to Vanilla, POCache reduces the average read latency by 41.1-57.3% and 31.1-34.2%, and the 95th-percentile read latency by 32.6-55.4% and 31.7-41.8%, under contiguous and striping layouts, respectively.

B. Microbenchmarks

We conduct microbenchmarks and provide performance breakdowns for read and write operations.

Experiment 6 (Microbenchmarks on writes): We first study the efficiency of writing a data stream to the Redis cache through block slicing and incremental encoding. The data stream is composed of four 64-MiB blocks, each of which is further partitioned into a number of subblocks. We measure the time for the following two phases when the subblock size varies: (i) *encoding*, which refers to the encoding step that produces a new intermediate parity subblock by incorporating the newly-arrived data subblock with the intermediate parity subblock that has been calculated in the last step, and (ii) *caching*, which refers to the procedure of transferring the generated parity subblocks from the HDFS client to the cache node. POCache performs encoding and caching in parallel, and the write latency denotes the elapsed time of encoding and caching the data stream. We ignore the preparation that creates the data blocks before encoding, as the preparation time is very little. We also calculate the *overhead* of introducing encoding and caching in the write path. Suppose that the write latencies of POCache and Vanilla are l and l^* respectively, then the overhead is calculated as $\frac{l-l^*}{l^*}$.

Table II shows the average results under different subblock sizes, indicating that POCache introduces no more than 23% of overhead in writes when compared to Vanilla. Also, POCache achieves the lowest overhead (i.e., 8.18%) when the subblock is 1-MiB.

Experiment 7 (Microbenchmarks on reads): We investigate the efficiency of reading data from the cache. Table III shows the average latencies and their standard deviations of reading a 1-MiB subblock from a normal DataNode, a straggler, and the Redis cache, respectively. The time of reading data from the straggler is more than seven times of that from the normal

TABLE II: Experiment 6 (Microbenchmarks on writes).

Subblock size	Encoding	Caching	Overhead
0.25 MiB	0.05ms	0.60ms	14.11%
0.5 MiB	0.09ms	1.00ms	10.57%
1 MiB	0.20ms	1.87ms	8.18%
2 MiB	0.44ms	3.28ms	12.15%
4 MiB	0.96ms	6.55ms	16.37%
8 MiB	1.89ms	12.94ms	22.33%

TABLE III: Experiment 7 (Microbenchmarks on reads).

	Mean	Stdev
Normal DataNode	7.38ms	9.22ms
Straggler DataNode	51.61ms	53.67ms
Cache node	4.92ms	2.16ms

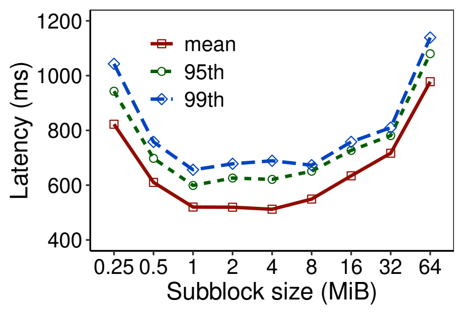


Fig. 11: Experiment 8 (Read latencies versus different subblock sizes).

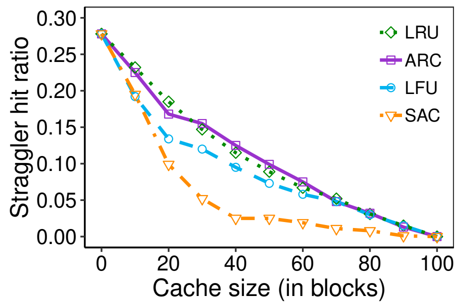


Fig. 12: Experiment 9 (Straggler hit ratio under different cache sizes).

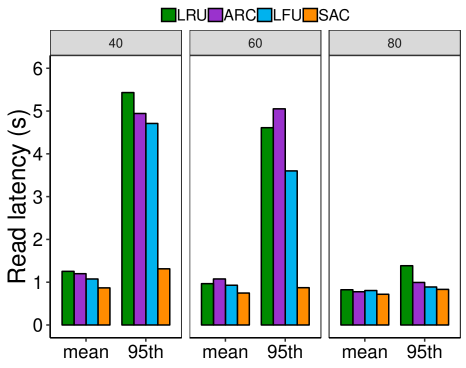


Fig. 13: Experiment 10 (Read latencies under different cache sizes).

DataNode, which shows the necessity of mitigating stragglers in data reads. Reading data from the Redis cache takes the least time, justifying that caching can be effective to mitigate stragglers.

Experiment 8 (Read latencies under differet subblock sizes): We verify the effectiveness of block slicing. We set the block size as 64 MiB and vary the subblock size from 0.25 MiB to 64 MiB. Figure 11 depicts the read latencies under different subblock sizes. We can observe that the read latency is influenced by the subblock size, where the read latency increases if the subblock size is too small (e.g., 0.25 MiB) or too large (e.g., 64 MiB). The lowest mean and tail latencies can be achieved when the subblock size is 1 MiB.

C. *Caching Efficiency*

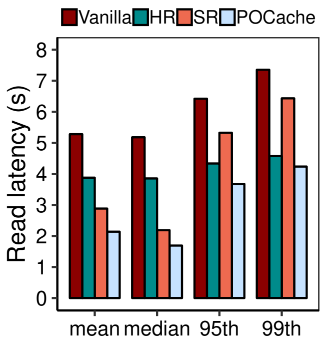
Experiment 9 (Straggler hit ratio under different cache sizes): We first measure the straggler hit ratios of LRU, ARC, LFU, and SAC under different cache sizes. Note that in the experiment, POCache uses the four cache algorithms to manage the cached parity blocks, with a difference that SAC is straggler-aware while the remaining three algorithms are not. As in previous experiments, we write 100 four-block files to HDFS and run `stress` to inject a straggler. We then vary the cache size from 0 to 100 (in unit of blocks).

Figure 12 shows the results. The straggler hit ratios generally decrease when the cache size becomes larger and SAC always achieves the smallest straggler hit ratio among all the four cache algorithms. As a straggler-aware cache algorithm, SAC is more effective in reducing the straggler hit ratio when the cache size is small. For example, when the cache size is 40, SAC reduces the straggler hit ratio to only 2.5%, while other three cache algorithms still incur around 10% of straggler hit ratios. When the cache size is 100, the Redis cache is large enough to keep all parity blocks associated with the data blocks that have been accessed, therefore the straggler hit ratios of all the cache algorithms reach zero.

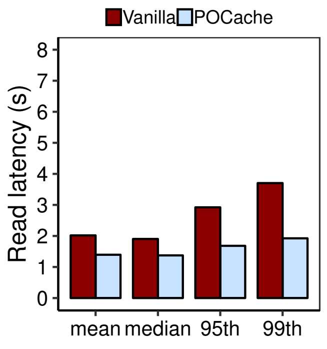
Experiment 10 (Read latencies under different cache sizes): We then study the latencies of LRU, ARC, LFU, and SAC under different cache sizes. We vary the cache size from 40 to 80, and measure the mean and the 95th-percentile latencies of the four cache algorithms. Figure 13 presents the results. We can see that the read latencies reduce with the increase of the cache size. SAC achieves the lowest latencies due to its lowest straggler hit ratio. When the cache size is 40 (i.e., 10% of the number of data blocks written), SAC can significantly reduce the long latencies caused by the stragglers. When the cache size is 80, all the four algorithms can achieve low read latencies because the parity blocks of most of the files are cached.

D. *Amazon EC2 Experiments*

Experiment 11 (Read performance on Amazon EC2): We evaluate the performance of POCache on a Amazon EC2 cluster. We deploy Hadoop atop 30 `m5.large` instances, where we use 28 instances as DataNodes and let the remaining two instances be the client and the NameNode, respectively. We also start one `m5.2xlarge` instance to serve as the cache node by running Redis. The underlying hardwares of these machines are shared by multiple tenants. The network bandwidth is around



(a) Contiguous layout



(b) Striping layout

Fig. 14: Experiment 11 (Read performance on Amazon EC2).

5 Gbps and all machines are equipped with magnetic storage. We set $k = 4$ and $m = 1$, and write 300 four-block files (i.e., 1,200 data blocks in total) into HDFS. We then generate 1,000 read requests by following the Zipf distribution with a Zipfian constant of 0.9. We set the cache size as 120 (in unit of blocks), which is 10% of the number of data blocks stored in HDFS.

Figure 14 shows the read latencies under the contiguous and striping layouts. We make two observations. First, stragglers naturally appear in the cloud environment, as the I/O and computational resources are shared and competed among different cloud users. For example, the 99th-percentile latency of Vanilla is 68.3% larger than its median latency. Second, POCache tolerates stragglers robustly by caching parity blocks even though the straggler estimation is inaccurate in such an environment where the stragglers fluctuate. POCache achieves the lowest latency among all the four policies. Specifically, POCache reduces the mean and 95th-percentile latencies of Vanilla by 64.3% and 63.9% under the contiguous layout, and by 30.9% and 42.4% under the striping layout.

VI. Related Work

Straggler tolerance: A large body of studies have addressed the straggler problem in different aspects, especially data-parallel processing frameworks [13], [46], [51], [53]. We review these studies because the scenario they target is similar to ours where reading a file requires to retrieve all of its blocks. LATE [53] handles the straggler by estimating the remaining time of each task and executing speculatively those which would degrade the job. Mantri [13] monitors the MapReduce jobs and culls the slow tasks given the causes identified. Dolly [12] clones the small jobs in order to skip the waiting phase in the speculative execution. Wrangler [51] monitors the resource utilization of the cluster and schedules the tasks carefully to avoid the potential stragglers. PBSE [46] is a path-based speculative execution to tolerate the network throughput degradation. In the context of storage, POCache tolerates stragglers with additional redundancy.

Erasure coding: Recent studies explore the use of erasure coding to improve read performance. Cocytus [54] and MemEC [52] store entire erasure-coded stripes in memory for low-latency access. EC-Cache [42] addresses load imbalance by splitting and encoding individual objects into stripes that will be stored in the memory entirely. Some studies address stragglers in erasure-coded storage systems, in which all data is persistently stored in erasure-coded form. Hu *et al.* [28] propose proactive degraded reads to reduce the tail latency due to degraded reads (which trigger more I/Os than normal reads). Li *et al.* [36] perform erasure coding across sectors on hard disks, so that local file systems can recover from transient sector-read failures without triggering read retries. EC-Store [9] avoids retrieving chunks from the heavy-loaded servers by placing and moving chunks dynamically in erasure-coded storage systems. Agar [25] caches the fragments in the remote site for geo-distributed store with erasure coding to minimize the read latency. The closest related work to ours is Sprout [10], which shows that caching erasure-coded data can reduce access latency. The main differences between Sprout and our work include: (i) Sprout keeps erasure-coded data either on the client side or in a proxy-based caching tier, while our caching tier can be deployed alongside the storage nodes (see Figure 2); (ii) Sprout does not address how to mitigate the non-negligible encoding/decoding overhead for large files as in our work; and (iii) Sprout targets erasure-coded storage (like [28], [36]), while our work can be applied to any form of storage (either replicated or erasure-coded).

Caching: To enable low-latency storage services, modern storage systems often deploy in-memory caches (e.g., Memcached [22], [39], Redis [7], ElastiStore [2]) to buffer frequently accessed objects in memory. Recent studies [19], [21], [37] further improve the internal performance or the hit ratios of existing in-memory caches. Alluxio (formerly called Tachyon) [33] provides in-memory fault tolerance via lineage for data-intensive applications. NetCache [32] realizes caching in programmable network switches. RobinHood [14] proposes tail-latency-aware caching, which identifies the cache-poor backends (i.e., the servers that contribute to the 99th-percentile request latency) and shifts cache space from other backends to the cache-poor backends. POCache targets straggler tolerance with robustness and space efficiency in mind.

VII. Conclusion

We present POCache, a robust approach of caching parity blocks with a dedicated cache algorithm to mitigate the performance degradations due to stragglers. We first analyze the effectiveness of parity-only caching, which achieves a low probability of hitting stragglers with limited cache space. To apply it in real-world storage systems, we propose block slicing and incremental encoding to reduce the encoding and decoding penalties. We further design a straggler-aware cache algorithm that takes into account the file popularity and straggler appearance when managing the cache space. Our evaluation results on both local and Amazon EC2 clusters demonstrate the effectiveness of POCache in achieving robust straggler tolerance.

Acknowledgments: This work was supported in part by the Research Grants Council of Hong Kong (GRF 14216316 and CRF C7036-15G).

References

[1] DFS-Perf. http://pasa-bigdata.nju.edu.cn/dfs-perf.

[2] ElastiCache. https://aws.amazon.com/elasticache/.

[3] Empirical Rule. https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule.

[4] HDFS 3.1. https://hadoop.apache.org/release/3.1.1.html.

[5] ISA-L. https://software.intel.com/en-us/storage/ISA-L.

[6] Jedis. https://github.com/xetorthio/jedis.

[7] Redis. https://redis.io/.

[8] Stress. https://linux.die.net/man/1/stress.

[9] M. Abebe, K. Daudjee, B. Glasbergen, and Y. Tian. EC-Store: Bridging the Gap between Storage and Latency in Distributed Erasure Coded Systems. In *Proc. of IEEE ICDCS*, 2018.

[10] V. Aggarwal, Y.-F. R. Chen, T. Lan, and Y. Xiang. Sprout: A Functional Caching Approach to Minimize Service Latency in Erasure-Coded Storage. *IEEE/ACM Trans. on Networking*, 25:3683–3694, Dec. 2017.

[11] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proc. of ACM EuroSys*, 2011.

[12] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proc. of USENIX NSDI*, 2013.

[13] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. of USENIX OSDI*, 2010.

[14] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. Robinhood: Tail Latency Aware Caching - Dynamic Reallocation from Cache-Rich to Cache-Poor. In *Proc. of USENIX OSDI*, 2018.

[15] Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, and E. Zadok. On the Performance Variation in Modern Storage Stacks. In *Proc. of USENIX FAST*, 2017.

[16] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[17] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *Proc. of USENIX ATC*, 2017.

[18] Y. Cheng, A. Gupta, and A. R. Butt. An In-Memory Object Caching Framework with Adaptive Load Balancing. In *Proc. of EuroSys*, 2015.

[19] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: A Dynamic Multi-tenant Key-value Cache. In *Proc. of USENIX ATC*, 2017.

[20] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. of the ACM*, 56(2):74–80, Feb. 2013.

[21] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*, 2013.

[22] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004.

[23] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, 2010.

[24] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, et al. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proc. of USENIX FAST*, 2018.

[25] R. Halalai, P. Felber, A.-M. Kermarrec, and F. Taïani. Agar: A Caching System for Erasure-Coded Data. In *Proc. of IEEE ICDCS*, 2017.

[26] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proc. of USENIX FAST*, 2016.

[27] Y.-J. Hong and M. Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proc. of ACM SoCC*, 2013.

[28] Y. Hu, Y. Wang, B. Liu, D. Niu, and C. Huang. Latency Reduction and Load Balancing in Coded Storage Systems. In *Proc. of ACM SoCC*, 2017.

[29] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.

[30] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proc. of ACM HotOS*, 2017.

[31] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, and K. B. R. van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proc. of ACM HotNets*, 2014.

[32] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proc. of ACM SOSP*, 2017.

[33] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proc. of ACM SoCC*, 2014.

[34] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair Pipelining for Erasure-Coded Storage. In *Proc. of USENIX ATC*, 2017.

[35] X. Li, R. Li, P. P. C. Lee, and Y. Hu. OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems. In *Proc. of USENIX FAST*, 2019.

[36] Y. Li, H. Wang, X. Zhang, N. Zheng, S. Dahandeh, and T. Zhang. Facilitating Magnetic Recording Technology Scaling for Data Center Hard Disk Drives through Filesystem-Level Transparent Local Erasure Coding. In *Proc. of USENIX FAST*, 2017.

[37] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. of USENIX NSDI*, 2014.

[38] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proc. of ACM EuroSys*, 2016.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*, 2013.

[40] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proc. of VLDB Endowment*, 2013.

[41] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, 2009.

[42] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proc. of USENIX OSDI*, 2016.

[43] I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[44] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proc. of ACM SoCC*, 2012.

[45] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, 2010.

[46] R. O. Suminto, C. A. Stuardo, A. Clark, H. Ke, T. Leesatapornwongsa, B. Fu, D. H. Kurniawan, V. Martin, M. R. G. Uma, and H. S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proc. of ACM SoCC*, 2017.

[47] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proc. of USENIX NSDI*, 2015.

[48] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.

[49] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of USENIX OSDI*, 2006.

[50] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proc. of USENIX NSDI*, 2013.

[51] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and Faster Jobs using Fewer Resources. In *Proc. of ACM SoCC*, 2014.

[52] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee. Erasure Coding for Small Objects in In-Memory KV Storage. In *Proc. of ACM SYSTOR*, 2017.

[53] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of USENIX OSDI*, 2008.

[54] H. Zhang, M. Dong, and H. Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proc. of USENIX FAST*, 2016.

[55] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proc. of ACM SoCC*, 2014.