POCache: Toward Robust and Configurable Straggler Tolerance with Parity-Only Caching

Mi Zhang[a], Qiuping Wang[a], Zhirong Shen[b], Patrick P. C. Lee[a,*]

[a] *The Chinese University of Hong Kong, Hong Kong, China*
[b] *Xiamen University, Xiamen, China*

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. It further adopts a configurable straggler-aware cache algorithm (CSAC) that takes into account both file access popularity and straggler estimation to decide which parity blocks should be cached. CSAC enables POCache to configure various cache admission and eviction algorithms with straggler awareness and supports cache prefetching. We implement a POCache prototype atop Hadoop 3.1 HDFS, while preserving the performance and functionalities of normal HDFS operations. 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.

Keywords: stragglers, erasure coding, caching

Note: A preliminary version [75] of this paper was presented at the 35th International Conference on Massive Storage Systems and Technology (MSST 2019). In this extended version, we analyze the effect of straggler tolerance with different caching schemes in the face of correlated failures, propose a configurable straggler-aware algorithm for POCache and include additional evaluation results.

*Corresponding author
Email addresses: mzhang@cse.cuhk.edu.hk (Mi Zhang), qpwang@cse.cuhk.edu.hk (Qiuping Wang), shenzr@xmu.edu.cn (Zhirong Shen), pclee@cse.cuhk.edu.hk (Patrick P. C. Lee)

1. Introduction

Large-scale storage systems are susceptible to high performance variability or long tails for various reasons [26], such as hardware slowdown [32, 34], load imbalance [58], resource sharing [68], and workload skewness [22, 76]. Such performance variability and long tails are often caused by the presence of *stragglers* (also known as "gray failures" [38] or "fail-slow faults" [32]), 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 [32]. Some systems address straggler tolerance via *selective replication*, which caches replicas for popular objects [14, 24, 35, 64] to avoid accessing stragglers (i.e., hotspots with overloaded requests) under skewed workloads. However, the popularity of objects can sharply change in a short period of time [39], and caching all objects is infeasible due to the high redundancy overhead of replication. Thus, selective replication is arguably ineffective when the available cache space is limited [39, 58].

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 [31, 37] and fail-slow failures [42, 43]. Here, we explore how erasure coding, coupled with caching, tolerates stragglers that cause performance variability and long tails in a real-world distributed storage system.

Although previous studies have also explored erasure-coded caching (e.g., [9, 10, 33, 58]), there remain several 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 the in-memory cache (e.g., 30% in EC-Cache [58]), thereby degrading the normal I/O performance 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. The main idea of POCache is to cache only *parity blocks* (i.e., the redundant blocks encoded from file data), which we show can effectively tolerate stragglers with limited caching and bandwidth overhead. POCache targets on the *write-once-read-many workloads*, in which files cannot be modified once being written. We

summarize our contributions as follows.

- We show via mathematical analysis that caching only parity blocks provides more effective straggler tolerance than caching only data blocks (as in selective replication). In particular, caching only one parity block effectively mitigates the impact of stragglers. Our analysis provides several 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. POCache caches parity subblocks rather than the whole parity blocks.

- We design a *configurable straggler-aware cache algorithm* (CSAC) that decides which parity blocks should be cached by taking into account both file access popularity and straggler estimation. Compared to the straggler-aware cache algorithm in our conference version [75], CSAC is configurable, such that it can incorporate different cache replacement policies with straggler awareness. It takes a best-effort approach to cache the parity blocks of popularly accessed files based on the estimation of which nodes are likely to be stragglers. CSAC also enables POCache to prefetch the parity blocks of the files that are affected by the estimated stragglers into the cache space. 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 [3]. 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.

Compared to the work in [75], we make the following new contributions. We first analyze the effect of different caching schemes in the presence of correlated stragglers, by quantifying the ratio of read requests that hit stragglers. Second, we propose and implement a configurable straggler-aware cache algorithm (CSAC), which supports various cache admission and eviction algorithms and cache prefetching. Furthermore, we evaluate the performance of POCache on running MapReduce workloads and the caching efficiency of CSAC.

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

The rest of the paper proceeds as follows. Section 2 introduces the background details of erasure coding, and motivates that parity-only caching can reduce

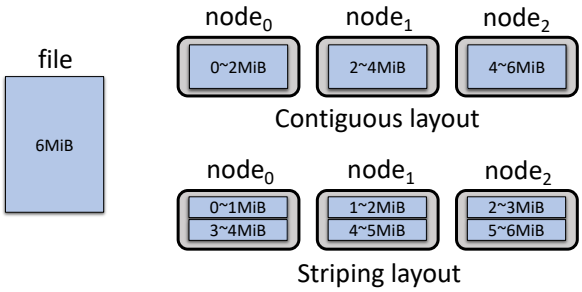


Figure 1: Contiguous and striping layouts (only data blocks are shown here).

the probability of hitting stragglers. Section 3 presents the design of POCache. Section 4 describes the implementation details of POCache on Hadoop 3.1 HDFS. Section 5 shows our evaluation results on both local and Amazon EC2 clusters. Section 6 reviews related work, and finally Section 7 concludes the paper.

2. Background and Motivation

In this section, we 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.

2.1. Basics

Erasure coding: Erasure coding provably incurs much less redundancy than replication under the same degree of fault tolerance [66]. 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 [56], 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 [59], which are extensively employed in current commodity storage systems (e.g., Ceph [67], QFS [54], and HDFS [3]). Many repair-efficient techniques (e.g., [46, 47, 52]) have recently been proposed to speed up repair operations in erasure coding. In this paper, we explore how to couple erasure coding with caching 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

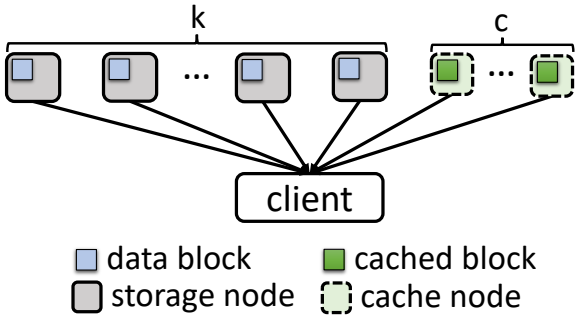


Figure 2: Example of reading a file with caching in Section 2.2.

nodes. Figure 1 depicts an example of both contiguous and striping layouts, where a file is partitioned into three logical blocks of size 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. Thus, reading a file in the contiguous layout can be 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 layouts are supported in current storage systems (e.g., HDFS of Hadoop version 3 [3]).

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.

2.2. Analysis

We show via a toy example how erasure coding addresses straggler tolerance. We first review the procedure of reading a file without considering access skewness. Suppose that the k data blocks of a file are distributed in k storage nodes (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 tolerate 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 caused by either independent failures or correlated failures.

2.2.1. Independent Stragglers Only

Let p_s and p_c be the probabilities that a storage node and a cache node become stragglers, 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 reading a file. We calculate P under both caching schemes as follows.

- *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 as long as all the blocks residing in a straggler are cached. We compute P under data-only caching as:

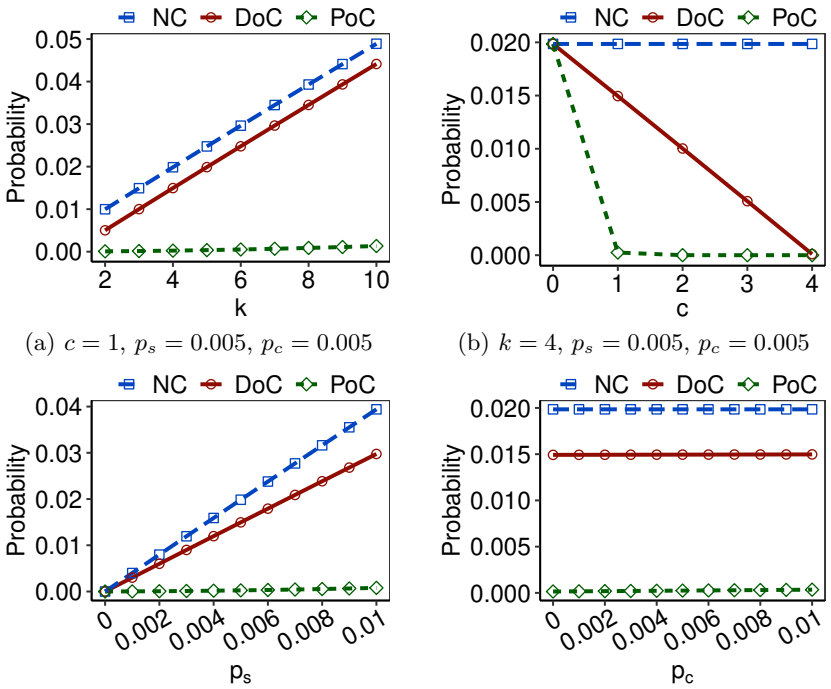
$$P = 1 - \sum_{i=0}^{c} \underbrace{\binom{c}{i} \cdot {p_s}^i \cdot (1-p_s)^{k-i}}_{i \text{ storage nodes are stragglers}} \cdot \underbrace{(1-p_c)^i}_{i \text{ cache nodes are normal}} .$$

  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



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

Figure 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). Here, we consider independent stragglers only.

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., $p_c = 0$).

2.2.2. Correlated Stragglers

Modern data centers group nodes into racks, where the nodes in the same rack are connected by a *top-of-rack (ToR) switch* and different racks are interconnected by a *network core* [13]. In such a hierarchical data center, the blocks of a file are distributed across different nodes in distinct racks (aka. *flat placement*) [31, 37], or fewer racks where each rack stores more than one block (aka. *hierarchical*

placement) [74]. Thus, reading a file needs to access data from different number of racks under different block placement policies.

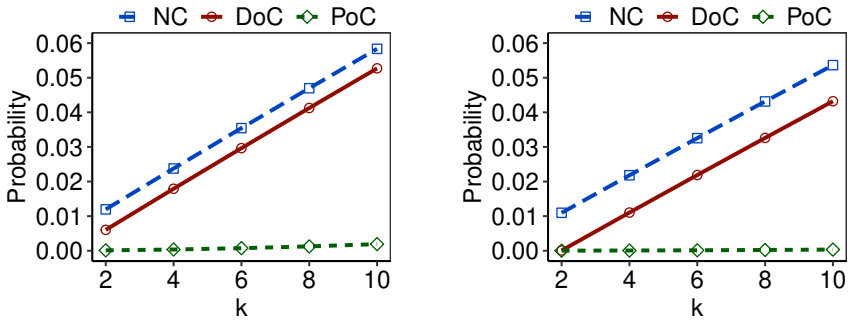
Correlated failures make all nodes in a rack become straggler nodes, introducing correlated stragglers. We denote the probability of rack slowdown and the number of blocks stored in each rack by p_r and a respectively. Here, we assume both the number of data blocks k and the blocks in cache c are multiples of a for simplicity. Thus, k data blocks are distributed across $r = \frac{k}{a}$ racks. To maximize the rack-level straggler tolerance, the blocks in cache are stored in different racks from the racks where data blocks are stored. We calculate P under both data-only caching and parity-only caching in the presence of independent and correlated stragglers as follows.

- *Data-only caching:* As we explained previously, data-only caching can only tolerate the straggler nodes which have an exact replica in cache. The same applies to the straggler tolerance caused by rack slowdown. That is, data-only caching can provide straggler tolerance when the blocks residing in the slow racks have the cached replicas. Thus, the probability P under data-only caching in the best cases (i.e., tolerating the maximum number of rack failures) is:

$$P = 1 - \sum_{i=0}^{\frac{c}{a}} \sum_{j=0}^{c-ia} \underbrace{\binom{\frac{c}{a}}{i} \cdot {p_r}^i \cdot (1-p_r)^{r-i}}_{\text{storage nodes in } i \text{ slow racks are stragglers}} \cdot \underbrace{\binom{c-ia}{j} \cdot {p_s}^j \cdot (1-p_s)^{k-ia-j}}_{j \text{ storage nodes in normal racks are stragglers}} \cdot \underbrace{(1-p_r)^{\lceil \frac{ia+j}{a} \rceil} \cdot (1-p_c)^{ia+j}}_{(ia+j) \text{ cache nodes are normal}}.$$

- *Parity-only caching:* With c parity blocks in cache, parity-only caching tolerates up to c blocks residing in straggler nodes or racks. Therefore, the probability P under parity-only caching in the best cases equals to:

$$P = 1 - \sum_{i=0}^{\frac{c}{a}} \sum_{j=0}^{c-ia} \sum_{l=0}^{c-ia-j} \sum_{v=0}^{\lfloor \frac{l}{a} \rfloor} \underbrace{\binom{r}{i} \cdot {p_r}^i \cdot (1-p_r)^{r-i}}_{\text{storage nodes in } i \text{ slow racks are stragglers}} \cdot \underbrace{\binom{k-ia}{j} \cdot {p_s}^j \cdot (1-p_s)^{k-ia-j}}_{j \text{ storage nodes in normal racks are stragglers}} \cdot \underbrace{{p_r}^v (1-p_r)^{(\frac{c}{a}-v)}}_{\text{cache nodes in } v \text{ slow racks are stragglers}} \cdot \underbrace{\binom{c-va}{l-va} \cdot {p_c}^{l-va} \cdot (1-p_c)^{c-l}}_{(l-va) \text{ cache nodes in normal racks are stragglers}}$$



(a) Flat placement where $a = 1$ (b) Hierarchical placement where $a = 2$

Figure 4: Probability P that a read hits a straggler for different block placement policies under no-caching (NC), data-only caching (DoC), and parity-only caching (PoC) where the cached blocks can tolerate one rack slowdown (i.e., $c = a$). Here, we consider both independent and correlated stragglers ($p_s = 0.005$, $p_c = 0.005$, $p_r = 0.001$).

Figure 4 shows the probability P of no-caching, data-only caching, and parity-only caching under flat and hierarchical block placement respectively, in the face of independent and correlated stragglers. Figure 4 (a) depicts the probability P versus k with $a = 1$, $c = 1$, $p_r = 0.001$, $p_s = 0.005$, and $p_c = 0.005$. Here, k data blocks are distributed across k racks. The probability is higher than that without rack slowdown in Figure 3 (a). Parity-only caching still achieves a small probability P (1.91E-3) as k increases. Figure 4 (b) shows the probability P under hierarchical placement where $a = 2$. The probability of no-caching here is slightly lower than that under flat placement because the blocks are distributed across less racks. We cache two blocks to tolerate one rack slowdown, i.e., $c = 2$. Parity-only caching achieves a probability of 3.33E-4 or lower while data-only caching has a probability of up to 4.32E-2 when k is less than or equal to 10.

Block Size	Network Transfer	Memory Access	Encoding /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.

Table 1: Latencies versus block size for RS codes under $(n, k) = (5, 4)$.

From the above analysis, parity-only caching provides robust straggler tolerance against independent and correlated failures.

2.3. 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 it, 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 1 shows the latencies of network transfer, memory access, encoding, and decoding versus different block sizes for RS codes under $(n, k) = (5, 4)$. 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 [58], in which the decoding time takes about 30% of the read time. However, previous studies (e.g., EC-Cache [58] 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, we should reduce the encoding/decoding overhead 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 [58] directly employs the least recently used (LRU) cache algorithm, which is also the default cache algorithm in Alluxio [45]. Existing cache algorithms study the tradeoff between the hit ratio and the latency when different objects have different access costs [21, 44, 62, 71], but how to manage the parity-only cache space (i.e., which parity blocks to cache) to minimize the probability of hitting stragglers still remains an open issue.

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.

3. 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 the coding overhead via block slicing and incremental encoding (Section 3.1) 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 [46, 47, 52]; here, we leverage these features to mitigate the encoding and decoding overhead in the context of caching and hence achieve effective straggler tolerance. We then design a configurable straggler-aware cache algorithm that carefully manages the cache space based on the file access popularity and the estimation of the straggler existence (Section 3.3). Currently, POCache supports write-once-read-many workloads (Section 1) and does not support updates, appends, or partial file writes. Such a scenario is reasonable for HDFS, which is designed for data analytics (e.g., MapReduce) [3].

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 [23]. POCache is designed to reduce the access latencies of a file in the presence of stragglers. Note that the coding parameters (n, k) in POCache are independent of those of the underlying storage systems.

3.1. Mitigating Coding Overhead

Block slicing: Our observation is that the actual erasure coding functionalities under Galois Field arithmetic (Section 2) perform in small-size coding units (e.g., bytes) [56]. 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 5 shows the idea of block slicing, in which the subblocks at the same

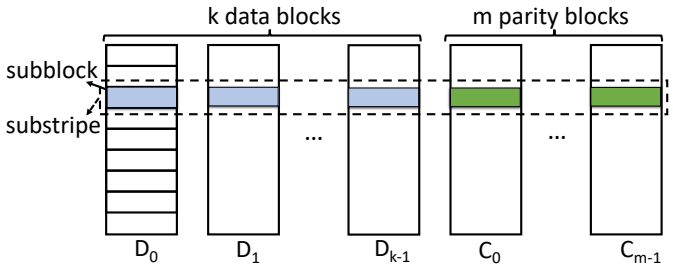


Figure 5: Block slicing. D_i and C_j respectively denote the i-th data block and the j-th parity block, where $0 \leq i \leq k-1$ and $0 \leq j \leq m-1$.

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.

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. Note that incremental encoding does not increase the computing cost of encoding. This can be regarded as pipelining the arrival of blocks with encoding. If the blocks never arrive, incremental encoding will receive the ending mark of the data stream and complete the encoding process; otherwise, incremental encoding waits for next blocks to arrive.

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.

3.2. 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 5.

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

3.3. Configurable Straggler-Aware Cache Algorithm

We propose a configurable straggler-aware cache (CSAC) 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 CSAC by taking into account both the estimation of existing stragglers and file access popularity. In particular, CSAC is configurable, in the sense that it is extensible for different cache management algorithms depending on the workload characteristics.

Straggler estimation: CSAC considers the existence of stragglers when making decisions on caching. To minimize the straggler hit ratio, CSAC prefers to cache parity blocks for the files that would be affected by stragglers. Thus, CSAC estimates the presence of stragglers based on the monitoring information. Specifically, CSAC identifies the straggler nodes and records them in a *straggler list* as follows. It periodically collects each node's *service rate* (denoted by ν), defined as the ratio of the amount of data being served to the service time being taken. It calculates the mean value (denoted by μ) and standard deviation (denoted by σ) of all service rates. Finally, it identifies the stragglers according to the *three-sigma rule* [57], in which the nodes whose $\nu < \mu - 3\sigma$ are treated as abnormal and included in the straggler list.

Generalized cache management: To capture the file access popularity, CSAC supports generalized cache management by allowing both the cache admission and cache eviction algorithms to be configurable.

- *Cache admission:* The cache admission algorithm decides whether to cache the parity blocks of an accessed file that has no cached parity blocks. A simple cache admission algorithm is *cache-upon-access* (i.e., always caching the parity blocks of the accessed file). Other algorithms can take into account the frequency [27] and object size [20] information.

- *Cache eviction:* The cache eviction algorithm decides which parity blocks of a file to evict from cache when the available cache space is insufficient. The decision often depends on the recency and frequency of file accesses. Examples of cache eviction algorithms include least recently used (LRU) (i.e., evicting the file that is the least recently accessed), least frequently used (LFU) (i.e., evicting the file that the lowest accessed frequency), and adaptive replacement cache (ARC) [51] (i.e., evicting a file determined jointly by its recency and frequency).

Our current CSAC implementation realizes cache-upon-access for cache admission, and LRU, LFU, and ARC for cache eviction.

Cache prefetching: CSAC supports cache prefetching to proactively tolerate stragglers. Based on the straggler list estimated as above, we can determine whether reading a file would hit a straggler, and hence prefetch the parity blocks of the file into the cache space. A question here is whether a file would be accessed in near future. Given the limited cache space, it is ineffective to simply cache all the files which have data blocks residing in straggler nodes.

CSAC uses a file access prediction algorithm to decide which files should have their parity blocks prefetched into the cache space. The algorithm aims to predict how likely a file is to be accessed based on the historical access pattern. One example of the prediction algorithm is to leverage the file access recency pattern, such that the more recently accessed file is more likely to be accessed soon. In this case, the prediction algorithm can return the list of files sorted by the latest access times of all files. The prediction algorithm is also configurable with other prediction approaches, such as predicting the file popularity based on the number of concurrent accesses and the file size [14].

CSAC triggers cache prefetching when new stragglers are detected. If a file is predicted to be accessed soon and has some blocks residing in the straggler nodes, CSAC prefetches its parity blocks into the cache space. To avoid prefetching too much data that leads to high I/O overhead, CSAC ensures that the amount of parity blocks prefetched each time is no more than a configurable ratio (denoted by θ_p) of the total cache size. If there is insufficient space for caching parity blocks, CSAC evicts some cached parity blocks according to the eviction algorithm.

Algorithm details: CSAC caches parity blocks for the files that are admitted to cache or affected by existing stragglers. Specifically, CSAC 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 (denoted by S). If no straggler node is detected (i.e., S is empty), CSAC decides whether to cache the parity blocks of a file based on the cache admission algorithm (denoted by G_a). It also chooses which parity blocks of a file to evict from cache based on the cache eviction algorithm (denoted by G_e). If prefetching is enabled, it predicts the files that are likely accessed soon via the file access prediction algorithm (denoted by G_p) and caches the parity blocks of the files affected by the stragglers.

Algorithm 1 elaborates the workflow of CSAC. It first initializes the total amount of cache space T and the available cache space A where $A = T$ (in unit of blocks) (line 1). It also initializes the cache admission algorithm G_a, the cache eviction algorithm G_e, the file access prediction algorithm G_p, and the prefetching ratio θ_p (line 2). It manages the cache space according to the latest straggler list S, which is updated periodically (line 3). For every read request to a file f, CSAC calls a function QUERY, which determines if the file f should be cached. If the parity blocks of f have already been cached, the QUERY function returns `cached` (lines 5-6). When f is not cached, CSAC returns `shouldCache` if (i) the cache admission algorithm G_a decides to cache f while the straggler list is empty or (ii) the estimated stragglers store data blocks of f (lines 7-8); otherwise,

it declines 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 UPDATE function, CSAC first updates the information on file f (e.g., access recency and frequency) (line 14). If DECISION is `shouldCache` (i.e., the parity blocks of f should be cached but have not been cached before), CSAC evicts the files chosen by cache eviction policy until there is enough cache space (lines 15-23), caches the parity blocks of f (line 24). Note that UPDATE returns the files to be evicted for caching the new parity blocks (line 26).

If cache prefetching is enabled and new straggler nodes are detected, CSAC calls the PREFETCH function to obtain a list of files for cache prefetching. When a file f is predicted to be read in near future (line 31), CSAC further checks if the data blocks of f are stored in any existing straggler node and f is not cached and if the amount of parity blocks to prefetch does not exceed the prefetch limit (i.e., $T \times \theta_p$); if so, CSAC includes f into the list of files for cache prefetching (lines 32-38). Finally, the PREFETCH function returns the list of files to be prefetched (line 40).

Remarks: Compared to SAC proposed in [75], CSAC generalizes the cache management with different admission/eviction policies and incorporates cache prefetching to tolerate stragglers proactively. Actually, SAC is a special case of CSAC which uses cache-upon-access as admission algorithm and LRU as eviction algorithm without prefetching. CSAC further reduces the straggler hit ratio by incorporating different eviction policies. And with prefetching enabled, CSAC reduces the 95th-percentile latency, especially under limited cache space. Our evaluation results in Section 5.3 demonstrate the effectiveness of CSAC with different configurations under different cache sizes.

4. Implementation

We implement POCache on Hadoop 3.1 HDFS [3] and describe the implementation details as below.

4.1. Reads in HDFS

HDFS is a distributed file system that stores data across multiple servers in a fault-tolerant manner [61]. 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

Algorithm 1 Configurable Straggler-Aware Cache Algorithm

```
Initialize the total amount of cache space T and the amount of available cache space A = T
Initialize the cache admission algorithm G_a, the cache eviction algorithm G_e, the file access prediction algorithm G_p, and the prefetching ratio θ_p
Given the latest list of straggler nodes S
function QUERY(f)
    if parity blocks of f are cached then
        return cached
    else if (S is empty AND G_a decides to cache f) OR (some nodes in S store the data blocks of f) then
        return shouldCache
    else
        return shouldNotCache
    end if
end function
function UPDATE(f, DECISION)
    Update the information on f
    Initialize the set of files to be evicted, E = {}
    if DECISION == shouldCache then
        n_f ← number of parity blocks to cache for f
        while A < n_f do
            e ← file decided by G_e to be evicted
            A ← A + number of parity blocks of e in cache
            Add e to E
        end while
        A ← A − n_f
        Cache the parity blocks of f
    end if
    return E
end function
function PREFETCH()
    Initialize the set of files to be prefetched, P = {}
    Set the amount of cache space for prefetching, N = 0
    for file f predicted to read by G_p do
        if some nodes in S store data blocks of f AND file f is not cached then
            n_f ← number of parity blocks to cache for f
            if N + n_f ≤ T * θ_p then
                Add f to P
                N ← N + n_f
            end if
        end if
    end for
    return P
end function
```

are distributed across multiple DataNodes (Section 2.1). POCache works for both contiguous and striping layouts.

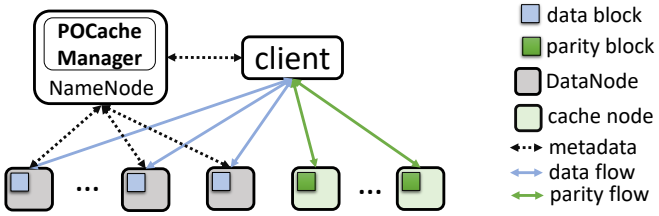


Figure 6: Integration of POCache into HDFS.

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 and uses the earliest response as the result. Thus, 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.

4.2. Integration

Figure 6 shows how POCache is implemented based on HDFS. We highlight the implementation details below.

Manager: Inside the NameNode, we add a module called the *Manager*, which tracks the cached parity blocks, performs cache admission and eviction decisions according to the specified algorithms, and manages cache prefetching operations.

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 [14], 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.

The Manager manages the cache space according to the cache admission and eviction algorithms specified in CSAC (Section 3.3). For straggler estimation, the Manager maintains a separate thread to update the straggler list periodically. For every file request, the Manager calls the QUERY function to obtain the admission decision made by the cache admission algorithm. Also, the Manager calls the UPDATE function based on the returned result of the QUERY function to update file access information and obtain the eviction decision.

For cache prefetching, the Manager starts a separate thread to prefetch the parity blocks into the cache space. Specifically, when new straggler nodes are detected and cache prefetching is enabled, the Manager first calls the PREFETCH function to obtain the list of files for caching. The separate thread then retrieves the data blocks from the files, generates the parity blocks, and adds the parity blocks into the cache space.

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 and cache 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 (i.e., the client performs encoding and caches the parity blocks if the Manager admits a file into the cache). 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 (Section 5.2) show that the read performance of the Redis cache is faster and more stable than that of a DataNode.

5. 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 CSAC algorithm effectively manage the cache space?

We evaluate POCache on a local cluster (Sections 5.1-5.3) and Amazon EC2 (Section 5.4). 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. Compared with our conference version [75], we add new evaluation results (i.e., Experiments 6, 10, and 13-15) and update Experiments 11-12, by evaluating the performance of POCache on running MapReduce workloads and the caching efficiency of CSAC.

5.1. Read Performance

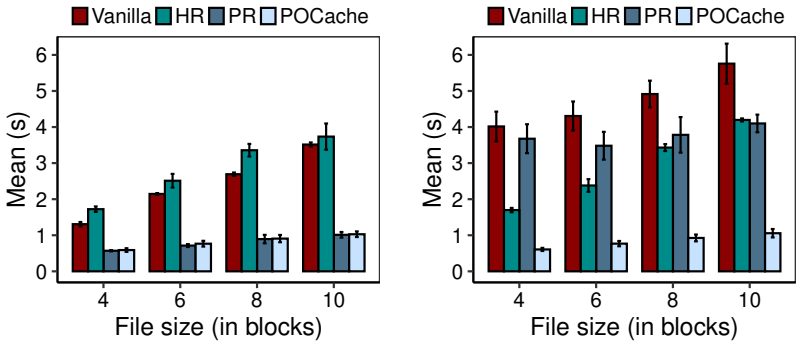
We first evaluate the read performance of POCache with and without stragglers on a local cluster (Experiments 1-5). We then evaluate the performance of running MapReduce workloads in Experiment 6.

Local cluster setup: Our local cluster consists of 15 machines, each of which is installed with Ubuntu 16.04 (with the Linux ext4 file system) 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. The disk access (with a sequential read bandwidth of 156 MiB/s) is the bottleneck during file accesses. 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 write/read workloads (i.e., `dfs-perf SimpleWrite/SimpleRead`) as configured (which is specified in each experiment), and collect the elapsed time results of all I/O requests (i.e., `dfs-perf-collect SimpleWrite/SimpleRead`). We run the Linux tool `stress` [8] (i.e., `stress -i 1`) on a DataNode, which issues `sync` commands to exhaust the local I/O resources, to inject a straggler on the local cluster.

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., $T = 100$ in Algorithm 1). We disable cache prefetching and later specifically evaluate the effect of cache prefetching. We realize cache-upon-access for cache admission and LRU for cache eviction. 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 ($k = 4, 6, 8, 10$ respectively). We compare POCache with the following: (i) the default reads in HDFS (named Vanilla), (ii) hedged



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

Figure 7: Experiment 1 (Single-client reads under the contiguous layout).

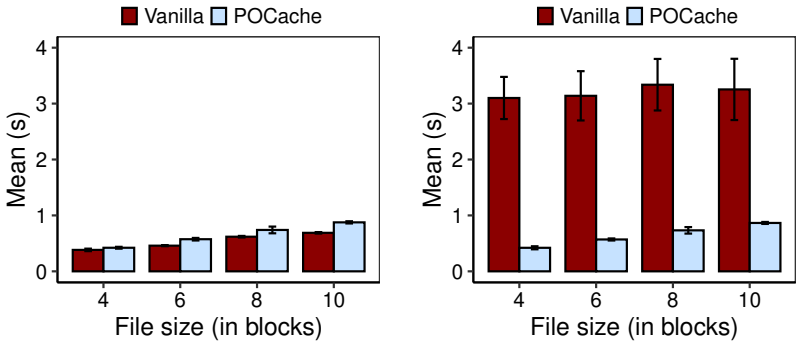
reads in HDFS (HR) (Section 4.1), and (iii) parallel reads (PR), in which a client issues reads to multiple data blocks in parallel without deploying POCache.

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 7(a) and 7(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, and read a file of size equal to k blocks ($k = 4, 6, 8, 10$ respectively).

Figures 8(a) and 8(b) depict the average single-client read latencies without



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

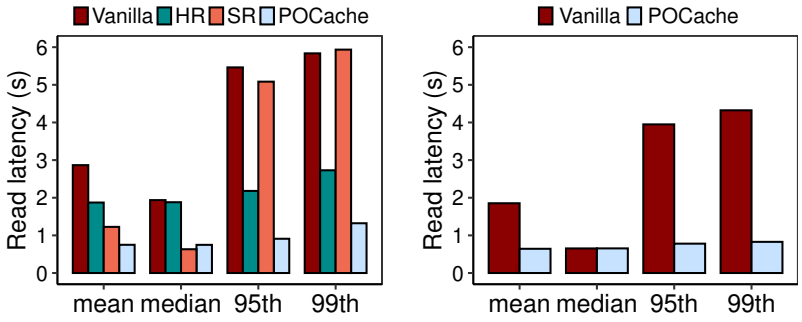
Figure 8: Experiment 2 (Single-client reads under the striping layout).

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 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, after evaluating single-file read performance in the previous experiments. 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 (Zipf-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.

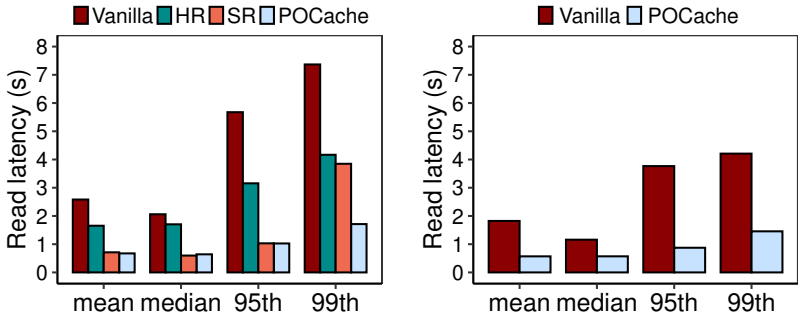
Figure 9 illustrates the read latencies under the contiguous and striping layouts. The mean latency here shows the average read time of the system (same for Experiments 4-5, 12, and 14-16). For the contiguous layout, by caching a number of popular blocks, SR has a lower median latency than HR. However, 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 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-size workload): We evaluate the read performance of files of different sizes under the same read access pattern as in



(a) Contiguous layout (b) Striping layout

Figure 9: Experiment 3 (Impact of skewed workload).



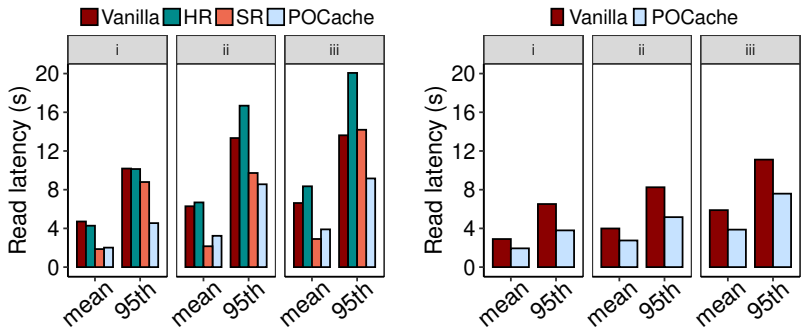
(a) Contiguous layout (b) Striping layout

Figure 10: Experiment 4 (Impact of multi-size workload).

Experiment 3 (i.e., Zipf-0.9). 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 [58]. We set $k = 4$ and $m = 1$. For a file smaller than four blocks, we pad the file with zeros to a full stripe (i.e., four blocks), stripe it in either contiguous layout or striping layout, and generate a parity block for caching. Figure 10 depicts the read latencies. We 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 read workload with the same skewness (i.e., Zipf-0.9) for each client. In this experiment, we consider the average and 95th-percentile latencies as the latter reflects the tail latency.

Figures 11(a) and 11(b) show the read latencies under the contiguous and



(a) Contiguous layout (b) Striping layout

Figure 11: Experiment 5 (Impact of multi-client reads): (i) 4 clients; (ii) 8 clients; and (iii) 12 clients.

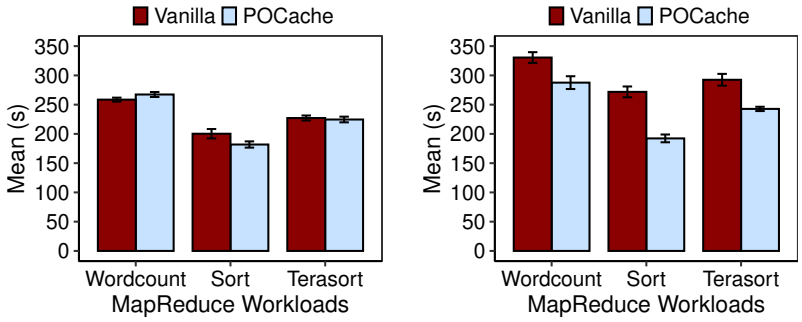
striping layouts, respectively. In general, the read latencies of all mechanisms increase when more clients issue read requests. Compared to SR, POCache has a slightly larger mean latency, but a much lower 95th-percentile latency. Compared to Vanilla, POCache reduces the average read latency by 41.1-57.3% 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.

Experiment 6 (Performance of MapReduce workloads): We evaluate the performance of running MapReduce jobs with Vanilla and POCache. We use HiBench [4] to run three MapReduce workloads (i.e., wordcount, sort, and terasort), with 128 256 MiB files as input. We set the split size, the data unit of each mapper/reducer to process, as 256 MiB. For POCache, we set the cache space to 200 (i.e., $T = 200$). To inject a straggler, we run `stress` (i.e., `stress -i 32 -d 8 --hdd-bytes 16MiB`) on a DataNode to limit the sequential read bandwidth (about 3.1 MiB/s), so as to degrade the performance of executing a map/reduce task. We plot the average execution time of each job over ten runs, including the error bars with the 95% confidence intervals. Figure 12 shows the latencies of POCache and Vanilla on executing MapReduce jobs in normal cases and with one straggler. In normal cases, POCache achieves similar performance to Vanilla. When there is a straggler, POCache reduces the latency of three workloads by 13.0-29.3%, because POCache bypasses the slowest node with cached parity blocks when reading input data from servers.

5.2. Microbenchmarks

We conduct microbenchmarks for read/write operations and MapReduce workloads.

Experiment 7 (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



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

Figure 12: Experiment 6 (Performance of MapReduce workloads).

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 2: Experiment 7 (Microbenchmarks on writes).

following two phases when the subblock size varies: (i) *encoding*, which refers to the encoding step at the client 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 2 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 8 (Microbenchmarks on reads): We investigate the efficiency of reading data from the cache. Table 3 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 DataNode, which

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

Table 3: Experiment 8 (Microbenchmarks on reads).

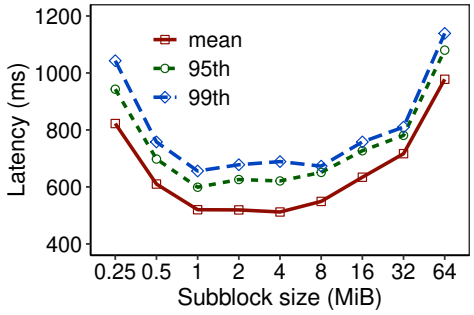


Figure 13: Experiment 9 (Read latencies versus different subblock sizes).

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 9 (Read latencies under different 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. We write a four-block file and read it for 200 times under each subblock size. Figure 13 depicts the average 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.

Experiment 10 (Microbenchmarks on MapReduce workloads): We study the breakdown performance of running wordcount job in the presence of stragglers. We use the same setting as in Experiment 6. Table 4 shows the mean latency of map tasks, reduce tasks, and reads at runtime. POCache achieves lower map/reduce latencies than Vanilla because POCache spends less time in reading the input. The read latency here is longer than reading a four-block file directly because the read latency includes the processing time of a MapReduce task as it reads data from the input and feeds it into the Mapper/Reducer.

5.3. Caching Efficiency

We compare CSAC with different cache algorithms. We write 100 four-block files to HDFS and issue 2,000 read requests following a Zipf distribution with a Zipfian constant of 0.9, to collect the straggler hit ratios and read latencies under different cache sizes (Experiments 11-15).

	Map	Reduce	Read
Vanilla	30.36 s	52.72 s	26.94 s
POCache	26.63 s	41.45 s	23.10 s

Table 4: Experiment 10 (Microbenchmarks on MapReduce workloads).

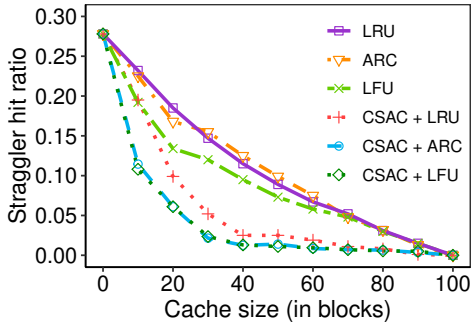


Figure 14: Experiment 11 (Straggler hit ratio under different cache sizes).

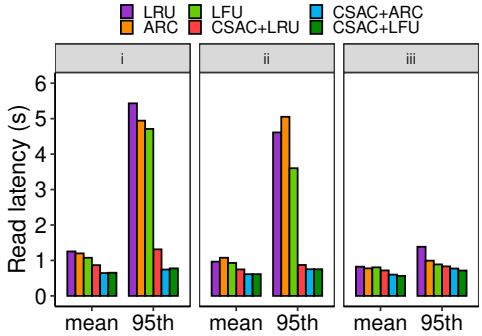


Figure 15: Experiment 12 (Read latencies under different cache sizes): (i) cache size = 40; (ii) cache size = 60; (iii) cache size = 80.

Experiment 11 (Straggler hit ratio under different cache sizes): We first measure the straggler hit ratio of LRU, ARC, LFU, and CSAC 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 CSAC is straggler-aware while the remaining three algorithms are not. Here, we set the admission policy of CSAC as cache-upon-access and the eviction policy as LRU, ARC, LFU respectively, denoted by CSAC+LRU, CSAC+ARC, CSAC+LFU. We then vary the cache size from 0 to 100 (in unit of blocks).

Figure 14 shows the straggler hit ratio under different cache sizes. The straggler hit ratios generally decrease when the cache size becomes larger and CSAC achieves the smallest straggler hit ratio among all the four cache algorithms. As a straggler-aware cache algorithm, CSAC is more effective in reducing the

straggler hit ratio when the cache size is small. For example, when the cache size is 40, CSAC reduces the straggler hit ratio below 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.

Recall that CSAC with LRU is equivalent to SAC proposed in our earlier conference version [75] (Section 3.3). Although the straggler hit ratio of CSAC+LRU is lower than that of the cache algorithms without straggler awareness (i.e., LRU, ARC, and LFU), it is higher than those of CSAC+LFU and CSAC+ARC (both of which have very similar results). For example, when the cache size is 10, the straggler hit ratio of CSAC+LRU (19.5%) almost doubles those of CSAC+LFU (10.8%) and CSAC+ARC (11.5%). Only when the cache size is larger than 60, all three configurations for CSAC achieve similar straggler hit ratios. This shows the significance of allowing configurability in CSAC, as it can support different cache management algorithms to mitigate the straggler hit ratio.

Experiment 12 (Read latencies under different cache sizes): We study the latencies of LRU, ARC, LFU, and CSAC under different cache sizes. We vary the cache size from 40 to 80, and measure the mean and the 95th-percentile latencies of different cache algorithms. Figure 15 presents the read latencies under different cache sizes. We see that the read latencies decrease with the cache size. CSAC achieves the lowest latencies as it mitigates the straggler hit ratio. When the cache size is 40 (i.e., 10% of the number of data blocks written), CSAC+LRU, CSAC+ARC, and CSAC+LFU reduce the 95th-percentile latency by 72.1-75.8%, 84.2-86.3%, and 83.5-85.7% compared to other cache algorithms without straggler awareness, respectively. When the cache size is 80, all cache algorithms achieve low read latencies because 80% of files have parity blocks in cache.

Experiment 13 (Impact of different cache strategies): To evaluate the efficacy of caching parity and data, we compare POCache (CSAC+LFU) with caching data of a file according to different cache management algorithms: (i) SR, which only considers access pattern using LFU algorithm (as LFU achieves the lowest straggler hit ratio among LRU and ARC); (ii) DC1, which only caches the data block on the estimated straggler node; and (iii) DC2, which manages the data caching with CSAC+LFU, i.e., considering both access pattern and straggler estimation. Figure 16 shows the straggler hit ratio of different cache strategies under different cache sizes. SR has the highest straggler hit ratio across different cache sizes because it only considers the file access pattern. DC1 achieves a slightly lower straggler hit ratio than DC2 as it specifically caches the data blocks on the straggler node, but has a little higher straggler hit ratio than POCache. POCache achieves the lowest straggler hit ratio because caching parity blocks tolerates the straggler even when the estimation is inaccurate.

Experiment 14 (Read latencies of different cache strategies): Figure 17 shows the read latencies of different cache strategies under different cache sizes. We compare POCache with the data cache strategies used in Experiment 13

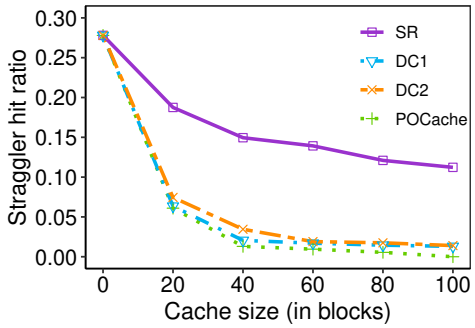


Figure 16: Experiment 13 (Impact of different cache strategies) Straggler hit ratio under different cache sizes.

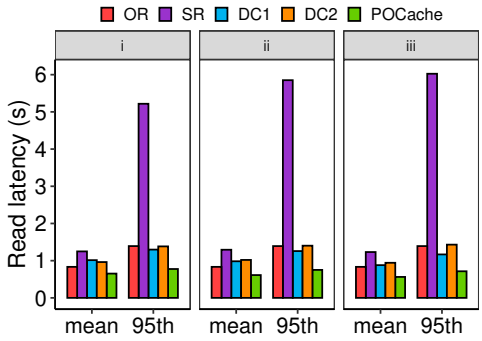


Figure 17: Experiment 14 (Read latencies of different cache strategies): (i) cache size = 40; (ii) cache size = 60; (iii) cache size = 80.

(i.e., SR, DC1, DC2) and OR, which only reads from the healthy node under three-way replication. SR has the longest read latencies because it has the highest straggler hit ratio. OR, DC1, and DC2 achieve similar performance by only requesting data blocks from healthy nodes or caching specific data blocks, which shows their efficacy in mitigating the impact of straggler nodes. POCache achieves the lowest read latencies as the cached parity can always tolerate the slowest node during reads.

Experiment 15 (Read latencies with cache prefetching): We now enable cache prefetching in CSAC and study its efficacy. Here, the file access prediction uses the file access recency to decide the list of files for cache prefetching. We use LFU as the eviction policy of CSAC (i.e., CSAC+LFU), which outperforms CSAC+LRU and achieves similar results to that of CSAC+ARC. We set the prefetching ratio θ_p as 40%. Figure 18 depicts the read latencies of LRU, ARC, LFU, CSAC when the cache size is 20. When cache prefetching is enabled, CSAC reduces the 95th-percentile read latency by 48.8% compared to without cache prefetching. Note that cache prefetching incurs a slightly higher 99th-percentile latency since the cache prefetching process needs to retrieve data blocks for

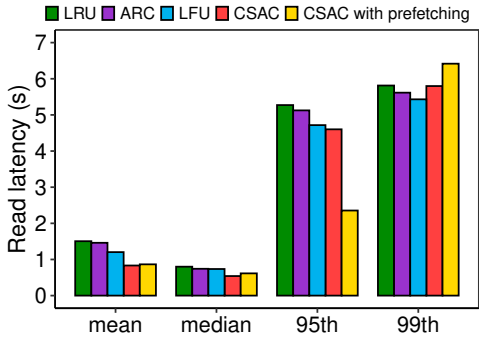


Figure 18: Experiment 15 (Read latencies with cache prefetching).

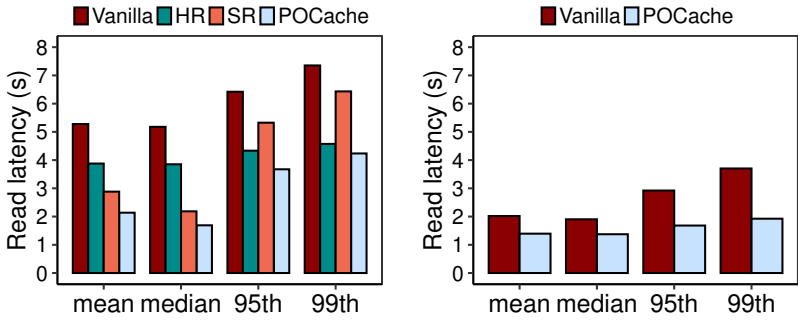
parity block generation, thereby incurring extra I/O overhead.

Summary: Our proposed CSAC achieves the smallest straggler hit ratios and the lowest read latencies among all the measured cache algorithms. For example, when the cache size is 40, CSAC reduces the straggler hit ratio below 2.5% and reduces the 95th-percentile latency by up to 85.7%. When cache prefetching is enabled, CSAC can further reduce the read latencies; for example, it reduces the 95th-percentile read latency by 48.8% when the cache size is 20.

5.4. Amazon EC2 Experiments

Experiment 16 (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 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 19 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.



(a) Contiguous layout (b) Striping layout

Figure 19: Experiment 16 (Read performance on Amazon EC2).

6. Related Work

Straggler tolerance: A large body of studies have addressed the straggler problem in different aspects, especially data-parallel processing frameworks [16, 63, 69, 72]. 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 [72] handles the straggler by estimating the remaining time of each task and executing speculatively those which would degrade the job. Mantri [16] monitors the MapReduce jobs and culls the slow tasks given the causes identified. Dolly [15] clones the small jobs in order to skip the waiting phase in the speculative execution. Wrangler [69] monitors the resource utilization of the cluster and schedules the tasks carefully to avoid the potential stragglers. PBSE [63] is a path-based speculative execution to tolerate the network throughput degradation. IASO [55] detects stragglers based on timeout signals and isolates them to mitigate their impact. Some recent studies [17, 60] mitigate the impact of straggler tasks with coding to reduce the job completion time in distributed computing systems. In the context of storage, POCache tolerates stragglers with additional redundancy. Some studies [11, 41, 65] present theoretical analysis of redundancy for straggler mitigation, while our analysis mainly studies the straggler hit ratios of different caching mechanisms.

Erasure coding: Recent studies explore the use of erasure coding to improve read performance. Cocytus [73] and MemEC [70] store entire erasure-coded stripes in memory for low-latency access. EC-Cache [58] addresses load imbalance by splitting and encoding individual objects into stripes that are stored in the memory entirely. In contrast, POCache only caches parity blocks to tolerate the presence of stragglers. Specifically, POCache caches only one parity block for each file, which is shown to be effective in straggler tolerance based on our analysis. Some studies address stragglers in erasure-coded storage systems, in which all data is persistently stored in erasure-coded form. Hu *et al.* [36] propose proactive degraded reads to reduce the tail latency due to degraded reads (which trigger more I/Os than normal reads). Li *et al.* [48] 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 [33] caches the fragments in the remote site for geo-distributed store with erasure coding to minimize the read latency. TTLCache [12] is a novel caching policy for jointly optimizing mean and tail latency in erasure-coded storage. Liu *et al.* [50] propose an offline caching scheme according to future data popularity and network latency information to achieve low latency in distributed coded storage systems. 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) We show via mathematical analysis that caching only one parity block is effective to mitigate the impact of stragglers, and thus POCache only caches one parity block instead of multiple parity blocks in Sprout; (ii) POCache proposes the straggler hit ratio to measure the probability of hitting stragglers, and a configurable straggler-aware cache algorithm to manage the cache space to reduce the straggler hit ratio, while Sprout is not specifically designed to bypass stragglers; (iii) 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 (Figure 2); (iv) Sprout does not address how to mitigate the non-negligible encoding/decoding overhead for large files as in our work; and (v) Sprout targets erasure-coded storage (like [36, 48]), while our work can be applied to any form of storage (either replicated or erasure-coded). Note that the latter three differences are mainly related to the implementation.

Caching: To enable low-latency storage services, modern storage systems deploy in-memory caches (e.g., Memcached [30, 53], Redis [7], ElastiCache [2]) to buffer frequently accessed objects. Recent studies [18, 25, 28, 29, 49] further improve the internal performance or hit ratios of in-memory caches. Alluxio (formerly called Tachyon) [45] provides in-memory fault tolerance via lineage for data-intensive applications. NetCache [40] realizes caching in programmable network switches. RobinHood [19] proposes tail-latency-aware caching, which identifies the cache-poor backends and shifts cache space from other backends to the cache-poor backends. POCache targets straggler tolerance with emphasis on robustness and space efficiency.

7. 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 configurable straggler-aware cache algorithm (CSAC) that takes into account the file popularity and straggler appearance when managing the cache space. CSAC

allows users to configure different cache management algorithms and support cache prefetching. Our evaluation results on both local and Amazon EC2 clusters demonstrate the effectiveness of POCache in achieving robust straggler tolerance. Our future work is to enable data updates, appends, and partial file writes in POCache. To support these functions, we need to update the cached parity blocks efficiently and make the cached parity blocks consistent with the data blocks belonging to the same stripe.

Acknowledgments: This work was supported in part by Research Grants Council of Hong Kong (AoE/P-404/18), the National Key R&D Program of China (2021YFF0704001), Natural Science Foundation of China (No. 62072381), and CCF-Huawei Innovation Research Plan (CCF-HuaweiST2021003).

References

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

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

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

[4] HiBench. `https://github.com/Intel-bigdata/HiBench`.

[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] M. F. Aktaş and E. Soljanin. Straggler Mitigation at Scale. *IEEE/ACM Transactions on Networking*, 27(6):2266–2279, 2019.

[12] A. O. Al-Abbasi and V. Aggarwal. TTLCache: Taming Latency in Erasure-Coded Storage Through TTL Caching. *IEEE Transactions on Network and Service Management*, 17(3):1582–1596, 2020.

[13] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of ACM SIGCOMM*, 2008.

[14] 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.

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

[16] 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.

[17] A. Badita, P. Parag, and V. Aggarwal. Single-Forking of Coded Subtasks for Straggler Mitigation. *IEEE/ACM Transactions on Networking*, 29(6):2413–2424, 2021.

[18] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *Proc. of USENIX NSDI*, 2018.

[19] 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.

[20] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *Proc. of USENIX NSDI*, 2017.

[21] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *Proc. of USENIX ATC*, 2017.

[22] 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.

[23] 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.

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

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

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

[27] G. Einziger, R. Friedman, and B. Manes. Tinylfu: A Highly Efficient Cache Admission Policy. *ACM Trans. on Storage*, 13(4):1–31, 2017.

[28] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *Proc. of USENIX NSDI*, 2019.

[29] 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.

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] G. Joshi, Y. Liu, and E. Soljanin. On the Delay-Storage Trade-Off in Content Download from Coded Distributed Storage Systems. *IEEE Journal on Selected Areas in Communications*, 32(5):989–997, 2014.

[42] S. Kiani, N. Ferdinand, and S. C. Draper. Exploitation of Stragglers in Coded Computation. In *Proc. of IEEE ISIT*, 2018.

[43] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran. Speeding Up Distributed Machine Learning Using Codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, 2017.

[44] C. Li and A. L. Cox. GD-Wheel: A Cost-Aware Replacement Policy for Key-Value Stores. In *Proc. of EuroSys*, 2015.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] K. Liu, J. Peng, J. Wang, and J. Pan. Optimal Caching for Low Latency in Distributed Coded Storage Systems. *IEEE/ACM Transactions on Networking*, 2021.

[51] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of USENIX FAST*, 2003.

[52] 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.

[53] 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.

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

[55] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *Proc. of USENIX ATC*, 2019.

[56] 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.

[57] F. Pukelsheim. The Three Sigma Rule. *The American Statistician*, 48(2):88–91, 1994.

[58] 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.

[59] 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.

[60] S. Sasi, V. Lalitha, V. Aggarwal, and B. S. Rajan. Straggler Mitigation with Tiered Gradient Codes. *IEEE Transactions on Communications*, 68(8):4632–4647, 2020.

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

[62] H. Sigurbjarnarson, P. O. Ragnarsson, J. Yang, Y. Vigfusson, and M. Balakrishnan. Enabling Space Elasticity in Storage Systems. In *Proc. of ACM SYSTOR*, 2016.

[63] 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.

[64] 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.

[65] D. Wang, G. Joshi, and G. Wornell. Using Straggler Replication to Re-

duce Latency in Large-scale Parallel Computing. *ACM SIGMETRICS Performance Evaluation Review*, 43(3):7–11, 2015.

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

[67] 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.

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

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

[70] 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.

[71] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

[72] 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.

[73] 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.

[74] M. Zhang, S. Han, and P. P. C. Lee. A Simulation Analysis of Reliability in Erasure-Coded Data Centers. In *Proc. of IEEE SRDS*, 2017.

[75] M. Zhang, Q. Wang, Z. Shen, and P. P. C. Lee. Parity-Only Caching for Robust Straggler Tolerance. In *Proc. of IEEE MSST*, 2019.

[76] 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.