SP-Cache: Load-Balanced, Redundancy-Free Cluster Caching with Selective Partition

Yinghao Yu, Renfei Huang, Wei Wang, Jun Zhang, Khaled Ben Letaief
Hong Kong University of Science and Technology
{yyuau,weiwa,eejzhang,eekhaled}@ust.hk

Abstract—**Data-intensive clusters increasingly employ in-memory solutions to improve I/O performance. However, the routinely observed file popularity skew and load imbalance create hot spots, which significantly degrade the benefits of in-memory caching. Common approaches to tame load imbalance include copying multiple replicas of hot files and creating parity chunks using storage codes. Yet, these techniques either suffer from high memory overhead due to cache redundancy or incur non-trivial encoding/decoding complexity. In this paper, we propose an effective approach to achieve load balancing without cache redundancy or encoding/decoding overhead. Our solution, termed SP-Cache, *selectively partitions* files based on their popularity and evenly caches those partitions across the cluster. We develop an efficient algorithm to determine the optimal number of partitions for a hot file—too few partitions are incapable of mitigating hot spots, while too many are susceptible to stragglers. We implemented SP-Cache in Alluxio, a popular in-memory distributed storage for data-intensive clusters. EC2 deployment and trace-driven simulations show that, compared to the state-of-the-art solution called EC-Cache [1], SP-Cache reduces the file access latency by up to 40% in both the mean and the tail, using 40% less memory.**

I. Introduction

Today's data-parallel clusters critically rely on *in-memory* solutions for high-performance data analytics [2]–[7]. By caching data objects in memory, I/O-intensive applications can gain order-of-magnitude performance improvement over traditional on-disk solutions [2], [4], [5].

However, one key challenge faced by in-memory solutions is the severe *load imbalance* across cache servers. In production clusters, data objects typically have the *heavily skewed popularity*—meaning, a small number of hot files account for a large fraction of data accesses [1], [8], [9]. The cache servers containing hot files hence turn into *hot spots*. This problem is further aggravated by the *network load imbalance*. It is reported in a Facebook cluster that the most heavily loaded links have over $4.5\times$ higher utilization than the average for more than 50% of the time [1]. The routinely observed hot spots, along with the network load imbalance, result in a significant degradation of I/O performance that could even eliminate the performance advantage of in-memory solutions (Sec. II).

Therefore, maintaining load balance across cache servers is the key to improving the performance of cluster caches. State-of-the-art solutions in this regard include *selective replication* [8] and *erasure coding* [1], both of which resort to *redundant caching* to mitigate hot spots.

Selective replication creates multiple replicas for files based on their popularity: the more popular a file is, the more replicas it has. File access requests can then be distributed to multiple servers containing those replicas, hence mitigating the load on the hot spots. However, replication results in high memory overhead as hot files are usually of large sizes [1], [8], [9]. Given the limited memory space, selective replication does not perform well in cluster caches [1], [10] (Sec. III-A).

Erasure coding comes as an alternative solution to achieve load balancing with reduced memory overhead [1]. In particular, a (k, n) erasure code divides a file into k partitions and generates $n - k$ *parity partitions*. Any k of the n partitions are sufficient to decode the original file. This results in better load balancing as the load of read requests are spread across multiple servers, and the memory overhead is usually much smaller than that of replication. However, erasure coding incurs salient encoding/decoding overhead. In fact, even with the highly optimized implementation [11], the computational overhead can still delay the I/O requests by 30% on average [1].

In this paper, we propose a different approach that achieves load balancing in cluster caches *without* memory redundancy or encoding/decoding overhead. Our approach, which we call *selective partition*, divides files into multiple partitions based on their popularity: the more popular a file is, the more partitions it is split into. File partitions are *randomly cached* by servers across the cluster. The benefits of this approach are three-fold. First, it *evenly spreads* the load of read requests across cache servers, leading to improved load balancing. Second, it increases the *read parallelism* of hot files, which, in turn, improves the I/O performance. Third, simply splitting files into partitions adds *no storage redundancy*, *nor* does it incur the computational overhead for encoding/decoding.

However, it remains a challenge to *judiciously* determine how many partitions a file should be split into. On one hand, too few partitions are insufficient to spread the load of read requests, making it incapable of mitigating hot spots. On the other hand, reading too many partitions from across servers adds the risk of being slowed down by *stragglers*.

To address this challenge, we model selective partition as a *fork-join* queueing system [12], [13] and establish an *upper-bound analysis* to quantify the mean latency in reads. We show that the optimal number of partitions can be efficiently obtained by solving a convex optimization problem. Based on this result, we design SP-Cache, a *load-balanced*, *redundancy-free* cluster caching scheme that optimally splits files to minimize the mean latency while mitigating the impact of stragglers. We show that SP-Cache improves load balancing by a factor of $O(L_{\max})$

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

compared to the state-of-the-art solution called EC-Cache [1], where $L_{\max}$ measures the load of the *hottest* file.

We have implemented SP-Cache atop Alluxio [2], [14], a popular in-memory distributed storage that can be used as the caching layer on top of disk-based cloud object stores (e.g., Amazon S3 [15] and Azure Storage [16]) and compute-collocated cluster file systems (e.g., HDFS [17] and Gluster [18]). We evaluated SP-Cache through both EC2 [19] deployment and trace-driven simulations. Experimental results show that despite the presence of intensive stragglers, SP-Cache reduces the mean and the tail (95th percentile) read latency by up to 40% compared to EC-Cache [1]. Owing to its redundancy-free nature, SP-Cache achieves all these benefits with 40% less memory footprint than EC-Cache.

II. Background and Motivation

In this section, we briefly survey the cluster caching systems and motivate the need to achieve load balancing therein.

A. Cluster Caching

Due to the recent technological advances in datacenter fabrics [20] and the emergence of new high-speed network appliances [21]–[23], the gap between network bandwidth and storage I/O bandwidth is rapidly narrowing [24]–[27]. Consequently, the performance bottleneck of cloud systems is shifting from network to storage I/O. Prior work has shown that accessing data from the local hard disk provides no salient benefits over remote reads [28], [29]. This conclusion also extends to solid state devices (SSD). As disk locality becomes irrelevant, cloud object stores, such as Amazon S3 [15], Windows Azure Storage [16], and OpenStack Swift [30], gradually replace compute-collocated storages—notably HDFS [17]—as the primary storage solutions for data-intensive applications.

However, cloud object stores remain bottlenecked on disk I/O [1], as reading from disk is at least two orders of magnitude slower than reading from memory. In light of this problem, cluster caching systems, such as Alluxio [14], Memcached [6], and Redis [31], are increasingly deployed in front of cloud object stores to provide low-latency data access at memory speed. In this paper, we primarily target the storage-side caching to improve the I/O performance. Our solution can also be applied to compute-collocated file systems, such as HDFS [17], provided that high-speed networks are available.

B. Load Imbalance and Its Impact

A plaguing problem faced by cluster caching is the routinely observed *load imbalance* across cache servers. We show through experiments that severe load imbalance results in significant I/O latencies, marginalizing the performance benefits provided by cluster caching.

Load Imbalance Prior works [1], [8] have identified two sources of load imbalance in production clusters: the *skewed file popularity* and the *imbalanced network traffic*.

It has been widely observed in datacenters that file (data object) popularity is heavily skewed and usually follows a Zipf-like distribution [1], [2], [8], [9]. That is, a large fraction of data access requests are contributed by only a small number of hot files. Fig. 1 depicts the distribution of popularity and file size in the Yahoo! cluster trace [32]. The trace contains the collective statistics of data accesses to over 40 million files in a period of two months. We observe that the majority of files ($\sim 78\%$) have cold data that has rarely been accessed (< 10 times). Only 2% are hot with high access counts (≥ 100). These files are usually much larger (15-30$\times$) than the cold ones. Consequently, cache servers containing these files are easily overloaded given their large sizes and high popularity.

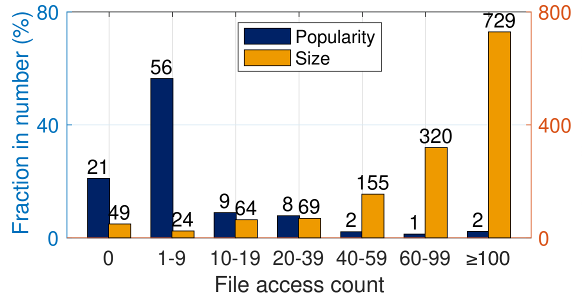


Fig. 1: Distribution of file popularity (blue) and size (orange) observed in a Yahoo! cluster [32].

This problem is further aggravated in the presence of network load imbalance, which is prevalent in production datacenters [1], [33]–[35]. For example, a recent study [1] measured the ratio of the maximum and the average utilizations across all up- and down-links in a Facebook cluster. The result shows that the ratio stays above $4.5\times$ more than half of the time, suggesting a severe imbalance.

Impact of Load Imbalance The skew in file popularity, together with the imbalanced network traffic, create hot spots among cache servers. To illustrate how these overloaded machines may impair the system's I/O performance, we stress-tested a small cluster of cache servers.

Setup: We deployed Alluxio [14]—a popular in-memory distributed storage—on a 30-node Amazon EC2 [19] cluster. The nodes we used are `m4.large` instances, each with a dual-core processor, 8 GB memory, and 0.8 Gbps network bandwidth. The cluster is used to cache 50 files (40 MB each). We launched another 20 `m4.large` instances as *clients*. Each client submits the file read requests to the Alluxio cluster as a Poisson process with a rate from 0.25 to 0.5 requests per second. Therefore, the aggregated access rates are 5-10 requests per second. We created imbalanced load with skewed file popularity following a Zipf distribution with exponent 1.1 (i.e., high skewness).

Diminishing benefits of caching: We ran two experiments. In the first experiment, all files were cached in memory; in the second experiment, we disabled cluster caching and spilled files to the local hard disk. For each experiment, we measured the mean read latency under various request rates and depict the results in Fig. 2. When the cluster is less loaded (5 requests per second), in-memory caching provides salient benefits, improving the mean read latency by $5\times$. However, as the load ramps up, the hot spots among cache servers become

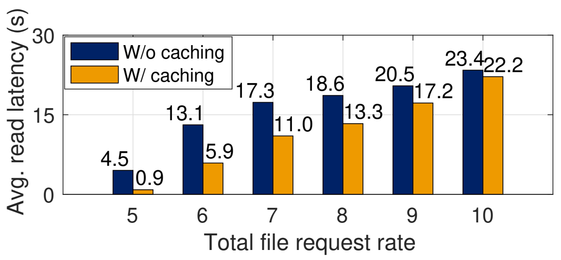


Fig. 2: The average read latencies with and without caching as the load of read requests increases.

more pronounced, and the benefits of caching quickly diminish. Notably, with request rate greater than 9, the read latency is dominated by the network congestion on hot-spot servers, and in-memory caching becomes *irrelevant*.

Therefore, there is a pressing need to achieve load balancing across servers. We next review existing techniques and show that they all fall short in minimizing latency.

III. Inefficiency of Existing Solutions

Prior art resorts to *redundant caching* to achieve load balancing, either by *copying multiple replicas* of hot files—known as *selective replication* [8], [36]—or by creating *coded partitions* of data objects, e.g., EC-Cache [1]. However, these techniques enforce an *unpleasant trade-off* between load balancing and cache efficiency. On one hand, caching more replicas (coded partitions) helps mitigate hot spots as the load of read requests can be spread to more servers. On the other hand, the overhead in memory and/or computation due to redundant caching harms efficiency.

A. Selective Replication

Selective replication replicates files based on their popularity [8], [36], i.e., the more popular a file is, the more replicas are copied across servers. A file read request is then randomly served by one of the servers containing the replica of that file. This way, the load of read requests are evenly distributed, leading to improved load balancing.

While selective replication is proven effective for disk-based storage [8], it does not perform well for cluster caching [1], [10], as replication incurs high memory overhead. To illustrate this problem, we deployed an Alluxio cluster following the settings described in Sec. II-B, where the top 10% popular files were copied to multiple replicas. The aggregated request rate is set to 6. We gradually increased the number of replicas and examined how the mean latency in reads can be improved at the expense of increased memory overhead. Fig. 3 depicts the results. We observe a *linear growth* of memory overhead in exchange for only a *sublinear improvement* in read latency. Given that in-memory caches remain a constrained resource in production clusters and the fact that popular files are usually of large sizes (Fig. 1), selective replication often results in poor cache efficiency with very low hit ratio (more in Sec. VII-F).

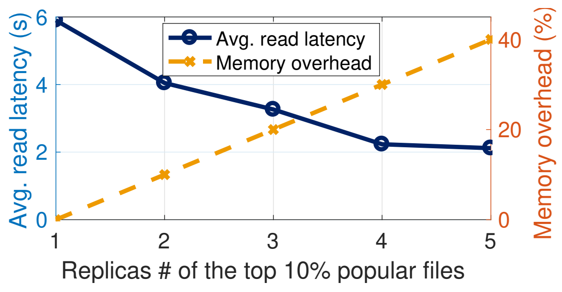


Fig. 3: Average read latency and cache cost in percentage with different replica numbers of the top 10% popular files.

B. Erasure Coding

State-of-the-art solutions employ *erasure coding* [37], [38] to load-balance cache servers without incurring high memory overhead. In particular, a (k, n) erasure coding scheme evenly splits a file into k partitions. It then computes $n - k$ *parity partitions* of the same size. The original file can be decoded using any k out of the n partitions, allowing the load of its read requests to be spread to n servers. The memory overhead is $(n - k)/k$, which is lower than that of selective replication (at least $1\times$) in practical settings. An efficient implementation of this approach goes to EC-Cache [1] which *late-binds* partitions during reads to mitigate stragglers. That is, instead of reading exactly k partitions, EC-Cache randomly fetches $k + 1$ partitions and waits for any k partitions to complete reading. EC-Cache significantly outperforms selective replication in both the median and tail read latencies [1].

However, EC-Cache requires non-trivial decoding (encoding) overhead during reads (writes). Even with a highly optimized coding scheme [39] and implementation [11], the decoding overhead may still delay the read requests by up to 30% [1]. To verify this result, we ran EC-Cache in an Amazon EC2 cluster with 30 `r3.2xlarge` memory-optimized instances, each having 61 GB memory and 8 cores. Following [1], we used a $(10, 14)$ coding scheme (i.e., memory overhead 40%) to cache files of various sizes. We launched an EC-Cache client submitting file read requests and measured the incurred decoding overhead, i.e., the decoding time *normalized* by the read latency. The results are deferred to appendix A due to the page limit. We observed more prominent decoding overhead with large files. Notably, for files greater than 100 MB which account for most of the file accesses in production clusters (Fig. 1), the decoding overhead consistently stays above 15%. We stress that this result is measured in the presence of a less advanced network, where we observed 1 Gbps bandwidth between instances. We expect the read latency dominated by the decoding overhead with high-speed networks (≥ 40 Gbps bisection bandwidth).

To sum up, existing load balancing solutions either suffer from high cache redundancy or incur non-trivial decoding/encoding overhead—either way, the I/O performance is impaired.

IV. Load Balancing with Simple Partition

In this section, we consider a simple, yet effective load-balancing technique which uniformly splits files into multiple

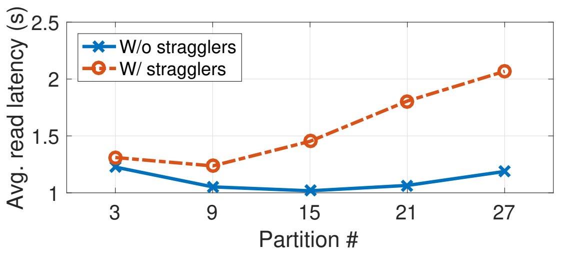


Fig. 4: Average read latency using simple partition in a 30-node cluster, with and without stragglers.

partitions so as to spread the I/O load. We explore the potential benefits as well as the problems it causes.

A. Simple Partition and Potential Benefits

We learn from EC-Cache [1] that dividing files into smaller partitions improves load balancing, yet the presence of parity partitions necessitates the decoding overhead. A simple fix is to split files *without creating coded partitions*. Intuitively, simple partition provides two benefits over EC-Cache. *First*, it requires no overhead for decoding/encoding. *Second*, it adds no storage redundancy, attaining the highest possible cache utilization. Simple partition also retains two benefits provided by EC-Cache. *First*, it mitigates hot spots under skewed popularity by spreading the load of read requests to multiple partitions. *Second*, it provides opportunities for read/write parallelism, which, in turn, speeds up the I/O of large files.

To validate these potential benefits, we configured EC-Cache in a "coding-free" mode using a (k, k) coding scheme (i.e., no parity partition). Specifically, we evenly split a file into k partitions and randomly cached them across the cluster. No two partitions of a file were placed on the same server. We re-ran the experiments in Sec. II-B using simple partition. For the purpose of stress-testing, we configured the aggregated file request rate to 10 from all clients. Note that at such a high rate, the average read latency would have stretched over 20 s *without load balancing* (Fig. 2). We study how simple partition can speed up I/O with increased read parallelism k. The results are depicted as a solid line in Fig. 4. The average read latencies drop to 1-1.3 s, suggesting 17-22$\times$ improvement over stock Alluxio without partition (Fig. 2). We stress that these improvements are achieved without any decoding overhead or cache redundancy. In contrast, the replication scheme studied in Sec. III-A is only able to attain the average latency of 2 s with 1.4$\times$ memory footprint (Fig. 3) in the presence of even lighter load (i.e., 6 requests per second).

B. Problems of Simple Partition

However, simple partition is not without problems. First, it *uniformly* divides each file into k partitions, irrespective of its size and popularity. This is *unnecessary* and *inefficient*. For less popular files, which usually dominate in population [8], [9], spreading their load provides marginal benefits in improving load balancing. Rather, the increased read parallelism may result in salient networking overhead due to TCP connections and the incast problem [40], [41]. Referring back to Fig. 4 (solid line), with too many partitions ($k > 15$), the networking overhead outweighs the benefits of improved load balancing.

Second, simple partition is susceptible to *stragglers*, as reading from many servers in parallel is bottlenecked by the slowest machine. To illustrate this problem, we manually injected stragglers into the cluster. Specifically, for each partition read, we slept the server thread with probability 0.01 and delayed the read completion by a factor randomly drawn from the distribution profiled in the Microsoft Bing cluster trace [42]. We measured the average read latency and depict the results as the dashed line in Fig. 4. As the read parallelism k increases, the latency caused by stragglers quickly dominates, leading to even longer delay.

In light of these problems, we wonder: is it possible to achieve load-balanced, redundancy-free cluster caching using file splitting while still being resilient to stragglers? We give an affirmative answer in the following sections.

V. SP-Cache: Design and Analysis

In this section, we present SP-Cache, a load balancing scheme that selectively partitions hot files based on their sizes and popularities. We analyze its performance and seek an optimal operating point to minimize the average read latency without amplifying the impact of stragglers.

A. SP-Cache Design Overview

SP-Cache employs *selective partition* to load-balance cluster caches under skewed popularity. In a nutshell, it evenly splits a file into small partitions, where the number of partitions is *in proportion to the expected load* of that file. Specifically, for file i, let S_i be its size and P_i be its popularity. The *expected load* of file i is measured by $L_i = S_i P_i$. Let k_i be the number of partitions file i is split into. With SP-Cache, we have

$$k_i = \lceil \alpha L_i \rceil = \lceil \alpha S_i P_i \rceil, \tag{1}$$

where α is a system-wide *scale factor* applied to all files. This results in the *uniform load* across partitions, i.e., $L_i/k_i \approx \alpha^{-1}$.

SP-Cache *randomly places* k_i partitions across N servers in the cluster, where *no two partitions* are cached in the same server. Random placement improves load balancing. It ensures each server to store *approximately an equal number* of partitions. Given the uniform load across partitions, each server is expected to have the balanced load.

B. Benefits

SP-Cache is more efficient than simple partition (Sec. IV). It differentiates the *vital few* from the *trivial many*, in that a small number of hot, large files (vital few) are subject to *finer-grained* splitting than a large number of cold, small files (trivial many). As the former is the main source of congestion, spreading their load to more partitions mitigates the congestion on the hot spots more than doing so to the latter. Moreover, given the small population of hot files [8], [9], splitting them results in fewer partitions than splitting a large number of cold files. This, in turn, results in a reduced number of concurrent TCP connections, alleviating the incast problem [40], [41].

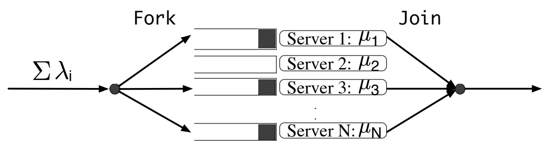


Fig. 5: The fork-join queuing model for SP-Cache.

Moreover, we show that SP-Cache achieves better load balancing than EC-Cache [1]. In particular, we denote by X^{SP} the total load on any particular server under SP-Cache, where X^{SP} is a random variable. Let X^{EC} be similarly defined for EC-Cache. We use the variance of X^{SP} (X^{EC}) to measure the degree of load imbalance—a higher variance implies the more severe load imbalance. The following theorem holds.

Theorem 1. *Consider SP-Cache with scale factor α and EC-Cache with a (k,n) erasure code. In a cluster where the number of servers is* much greater *than the number of partitions of any particular file under the two schemes, we have*

$$\frac{\mathrm{Var}(X^{EC})}{\mathrm{Var}(X^{SP})} \to \frac{\alpha}{k}\frac{\sum_i L_i^2}{\sum_i L_i}. \tag{2}$$

Theorem 1 can be established by showing that the load from each file follows Bernoulli distribution under the two caching schemes. The complete proof is deferred to appendix B.

It is easy to show that under heavily skewed popularity, the variance bound (2) approaches $\frac{\alpha}{k}L_{\max}$, where $L_{\max}$ measures the load of the *hottest* file, i.e., $L_{\max} = \max_i L_i$. As α and k are both constants, Theorem 1 states that compared to EC-Cache, SP-Cache improves load balancing by a factor of $O(L_{\max})$ in a large cluster.

C. Determining the Optimal Scale Factor

Despite the promising benefits offered by SP-Cache, it remains a challenge to judiciously determine the scale factor α. On one hand, choosing a small α results in a small number of partitions that are *insufficient* to mitigate the hot spots. On the other hand, choosing a too large α results in high I/O parallelism, adding the risk of being slowed down by stragglers.

We address this challenge with the optimal scale factor which is *large enough* to load-balance cluster caches, but also *small enough* to restrain the impact of stragglers. Specifically, we model SP-Cache as a *fork-join queue* [12], [13] and establish an *upper bound* for the mean latency as a function of scale factor α. Based on this analysis, we propose an efficient search algorithm which exponentially increases α to reduce the mean latency until the improvement becomes marginal. We settle on that α as a sweet spot, for it yields "just-enough" partitions to attain load balancing.

Model We model SP-Cache as a fork-join queue [12], [13] illustrated in Fig. 5. In particular, SP-Cache "forks" each file read to multiple reads on its partitions. Upon completion, all those partition reads "join" together to reassemble the file.

For tractable analysis, we consider Poisson arrivals of the read requests for each file. We shall verify in Sec. VII-F that this technical assumption is not critical with real-world request arrival sequences. Let λ_i be the request rate of file i. We measure the popularity of file i as

$$P_i = \frac{\lambda_i}{\sum_j \lambda_j}. \tag{3}$$

We model each cache server as an independent $M/G/1$ queue [43] with a FIFO service discipline. We derive the mean service delay on server s, which is the partition transfer delay averaged over all reads. Specifically, let C_s be the set of files having partitions cached on server s, and B_s the available network bandwidth. For a partition of file $i \in C_s$, the transfer delay depends on its size $\frac{S_i}{k_i}$ and the network bandwidth B_s. To account for the possible network jitters, we model the transfer delay as exponentially distributed with mean $\frac{S_i}{k_i B_s}$. The chance that file i's partition gets accessed is simply its request rate normalized by the aggregated rate, i.e., λ_i/Λ_s, where Λ_s is the aggregated request rate on server s and is given by

$$\Lambda_s = \textstyle\sum_{i\in C_s} \lambda_i. \tag{4}$$

The mean service delay on server s is then computed as

$$\mu_s = \textstyle\sum_{i\in C_s} \frac{\lambda_i}{\Lambda_s}\frac{S_i}{k_i B_s}. \tag{5}$$

Note that to make the analysis tractable, we assume a non-blocking network (i.e., no delay in the network) and do not model the stragglers. Our goal is to analyze the impact of scale factor α on load balancing with respect to the mean latency.

Mean Latency We denote by $Q_{i,s}$ as the read latency file i experiences on server s, which includes both the queuing delay and the service delay (transfer delay of a partition). As the file read is bottlenecked by the *slowest* partition read, the mean read latency of file i is given by

$$\bar{T}_i = \mathbb{E}[\max_{s:C_s\ni i} Q_{i,s}]. \tag{6}$$

Summing up the mean latency over files, weighted by their popularities, we obtain the mean read latency in the system:

$$\bar{T} = \textstyle\sum_i P_i \bar{T}_i. \tag{7}$$

The mean read latency critically depends on scale factor α. Intuitively, having a large α results in a large number of small partitions, which reduces both the overall queuing delay (better load balancing) and the transfer delay (small partitions).

Upper Bound Analysis Unfortunately, exactly quantifying the mean latency (7) in the fork-join system remains *intractable* due to the complex correlation between the partition placement (C_s) and the queueing dynamics [44]–[46]. Instead, we resort to establishing a tight *upper bound* to quantify the mean latency.

Prior work [44] shows that in a fork-join queue, the mean latency can be upper-bounded by solving a *convex optimization problem*. Applying this result [44, Lemma 2], we bound the mean read latency for file i as follows:

$$\begin{aligned}\bar{T}_i \le \hat{T}_i = \min_{z\in\mathbb{R}}\Bigg\{ z + \sum_{s:C_s\ni i}\frac{1}{2}(\mathbb{E}[Q_{i,s}] - z) \qquad (8)\\ + \sum_{s:C_s\ni i}\frac{1}{2}\Big[\sqrt{(\mathbb{E}[Q_{i,s}]-z)^2 + \mathrm{Var}[Q_{i,s}]}\Big]\Bigg\},\end{aligned}$$

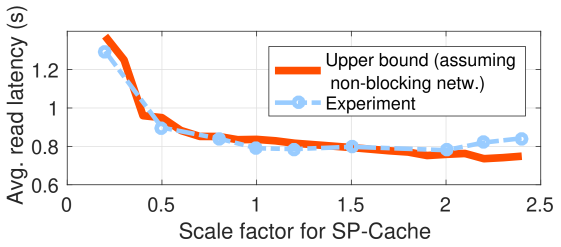


Fig. 6: Comparison of the derived upper bound and the average read latency measured in the EC2 cluster.

where z is an auxiliary variable introduced to make the upper bound as tight as possible, and $\text{Var}[\cdot]$ measures the variance.

While the latency bound (8) is not closed-form, it can be efficiently computed as (8) is a convex optimization problem given the expectation and variance of latency $Q_{i,s}$. Using the Pollaczek-Khinchin transform and the moment generating function for $M/G/1$ queue [43], we have

$$\mathbb{E}[Q_{i,s}] = \frac{S_i}{k_i B_s} + \frac{\Lambda_s \Gamma_s^2}{2(1-\rho_s)}, \tag{9}$$

and

$$\text{Var}[Q_{i,s}] = (\frac{S_i}{k_i B_s})^2 + \frac{\Lambda_s \Gamma_s^3}{3(1-\rho_s)} + \frac{\Lambda_s^2 (\Gamma_s^2)^2}{4(1-\rho_s)^2}, \tag{10}$$

where Γ_s^2 and Γ_s^3 denote the second and third moments of the service delay on server s, and ρ_s is the request intensity and is given by $\rho_s = \Lambda_s \mu_s$. Since the service delay is exponentially distributed with mean $\frac{S_i}{k_i B_s}$, we have

$$\Gamma_s^2 = \textstyle\sum_{i \in C_s} \frac{\lambda_i}{\Lambda_s} \cdot 2(\frac{S_i}{k_i B_s})^2, \tag{11}$$

and

$$\Gamma_s^3 = \textstyle\sum_{i \in C_s} \frac{\lambda_i}{\Lambda_s} \cdot 6(\frac{S_i}{k_i B_s})^3. \tag{12}$$

Summary: Putting it all together, given the scale factor α, we upper-bound the mean read latency as follows. We first compute the number of partitions $k_i = \lceil \alpha S_i P_i \rceil$ for each file i, based on which the expectation and variance of its read latency can be obtained, i.e., (9) and (10). Plugging them into (8), we solve a convex optimization problem and obtain the upper bound of the mean read latency for file i. We now upper-bound the mean latency of the system by replacing the latency of each file with its upper bound in (7).

Experimental verification: To examine how accurate the derived upper bound characterizes the mean read latency, we deployed a 31-node EC2 cluster and used it to cache 300 files (100 MB each) under skewed popularity. The detailed settings of our experiment are given in Sec. VII-A. Fig. 6 compares the derived upper bound and the mean latency measured in the cluster with various scale factors. The upper bound, though derived in a fork-join queuing model under some technical assumptions, closely tracks the average read latency measured in the EC2 cluster. Yet, as our model does not account for the networking overhead (TCP connections and incast problem) and stragglers, the measured latency occasionally goes above the theoretical upper bound.

Determining the Optimal Scale Factor We observe in Fig. 6 that as scale factor α increases, the mean latency dips quickly until an "elbow point" is reached ($\alpha = 1$), beyond which the latency plateaus for a short while and starts to rise as α turns large ($\alpha > 2$). This is by no means an accident. Intuitively, configuring a larger α results in better load balancing owing to finer-grained partitions. The price paid is the increased networking overhead and straggler impact. The gains outweigh the price before α reaches the elbow point, by which the load imbalance remains the main source of read latency. However, this is no longer the case after α passes the elbow point. The load is sufficiently balanced across servers, and the overhead of networking and stragglers becomes increasingly prominent, eventually dominating the latency.

Therefore, we should settle on the "elbow point" for the optimal scale factor. We use the derived upper bound to accurately estimate the mean latency. To locate the elbow point, we resort to an *exponential search* algorithm. Specifically, the search starts with an α such that the most heavily loaded file is split into $\frac{N}{3}$ partitions. The algorithm iteratively searches the optimal α. In each iteration, it inflates α by $1.5\times$ and examines the improvement in the derived latency bound. The search stops when the improvement drops below 1%. We shall show in Sec. VII that using the scale factor determined by this simple algorithm, SP-Cache reduces the mean (tail) latency by up to 50% (55%) as compared to EC-Cache [1].

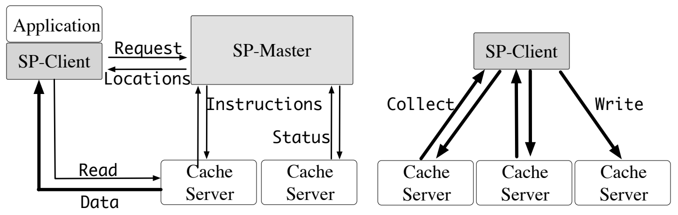


Fig. 7: Architecture overview of SP-Cache. (a) Applications interact with `SP-Client` for file access. (b) `SP-Client` periodically re-partitions files based on the instructions of `SP-Master`.

VI. Implementation

We have implemented SP-Cache atop Alluxio [14], a popular in-memory storage for data-parallel clusters. In this section, we describe the overall architecture of SP-Cache and justify the key design decisions made in the implementation.

A. Architecture Overview

Fig. 7(a) gives an overview of the system architecture. SP-Cache consists of two components: `SP-Master` and `SP-Client`. The `SP-Master` implements the main logic of selective partition described in Sec. V. It oversees many Alluxio cache servers and maintains the metadata (e.g., popularity) of files stored on those servers. The `SP-Client`, on the other hand, accepts the read/write requests from applications and interacts with cache servers for partition collecting, file reassembling, and load re-balancing based on the instructions issued by the `SP-Master`.

Reads Fig. 7(a) shows the data flow for reads. Similar to Alluxio [14], an application submits read requests to an `SP-Client` through the provided API. The client contacts the

`SP-Master`, who returns a list of cache servers containing the partitions of the requested file. The master also updates the access count for the requested file, so as to keep track of the file popularity for future *load re-balancing*. The `SP-Client` then communicates with the servers in the list and reads file partitions *in parallel*. Upon the completion of parallel reading, the client reassembles partitions to recover the original file and passes it to the application.

Writes SP-Cache directly writes a new file to a *randomly* selected cache server *without splitting*, given that cold files usually dominate in population (Sec. II). For each file stored in the cluster, SP-Cache keeps track of its popularity and periodically adjusts the number of partitions based on its load: once the file turns hot, it will get re-partitioned. We elaborate on how this can be done in the next subsection.

B. Periodic Load Balancing with Re-partitioning

As file popularities may change over time, SP-Cache *periodically load-balances* cache servers by *re-partitioning* the stored files. Following the recommendations in [8], SP-Cache re-partitions files every 12 hours based on the access count measured in the past 24 hours. To do so, the `SP-Master` instructs each cache server to report its current network bandwidth (measured through *sample reads*). Based on the bandwidth and the popularity information, the master computes the optimal scale factor α using the method described in Sec. V-C. Specifically, our implementation uses CVXPY [47] to solve Problem (8). To speed-up load re-balancing, the master launches *multiple* clients to re-partition files *in parallel*. As illustrated in Fig. 7(b), each client collects a *subset* of files from cache servers, splits each file into a number of partitions based on the new scale factor, and randomly places those partitions across cache servers.

The effectiveness of periodic load balancing is supported by the evidence that the file popularity in production clusters is *relatively stable* in a short term (e.g., days). In fact, it has been observed in a Microsoft cluster that around 40% of the files accessed on any given day were also accessed four days before and after [8]. A similar conclusion can also be drawn from the Yahoo! cluster trace [32]: nearly 27% of the files remain hot for more than a week.

C. Implementation Overhead

Metadata SP-Cache requires only a small amount of metadata maintained in the master node. For each file i, the `SP-Master` stores the partition count k_i and a list of the k_i servers containing those partitions. Compared to the file metadata maintained in Alluxio, the storage overhead is negligible.

Computational Overhead Finding the optimal scale factor α appears the main computational overhead in our implementation. Nevertheless, our evaluations show that even with 10k files, the optimal scale factor can be configured within 90 seconds (details in Sec. VII-B). As the computation is only needed every 12 hours, its overhead can be amortized and is less of a concern.

VII. Evaluations

In this section, we provide comprehensive evaluations on SP-Cache through EC2 deployment and trace-driven simulations. The highlights of our evaluations are summarized as follows:

1) The upper-bound analysis in Sec. V provides a reliable guidance to search for the optimal scale factor with low computational overhead (Sec. VII-B).
2) With 40% less memory overhead than EC-Cache [1], SP-Cache reduces the average read latency by 29-50% and the tail latency by 22-55% (Sec. VII-C).
3) SP-Cache is resilient to stragglers, improving the read latencies by up to 40% over EC-Cache in both the mean and the tail (Sec. VII-D).
4) With limited cache budget, SP-Cache achieves a higher cache hit ratio than EC-Cache (Sec. VII-E).

A. Methodology

Cluster Setup We have deployed SP-Cache in an Amazon EC2 cluster with 51 `r3.2xlarge` instances. Each node has 8 CPU cores, 61 GB memory. We measured 1 Gbps network bandwidth between instances using `iPerf`. We used 30 nodes as the cache servers, each with 10 GB cache space, one node as the master, and the remaining 20 nodes as clients continuously submitting read requests as independent Poisson processes.

Skewed Popularity We configured the skewed file popularity to follow a Zipf distribution [1], [48]–[50]. Unless otherwise specified, the exponent parameter of the Zipf distribution is set to 1.05 (i.e., high skewness).

Metrics We use the mean and the tail (95th percentile) read latencies as the primary performance metrics. We calculate the *improvement of latencies* as

$$\text{Latency improvement} = \tfrac{D-D_{\text{SP}}}{D} \times 100\%, \qquad (13)$$

where D_{SP} and D denote the latencies measured under SP-Cache and the compared scheme, respectively.

In addition, we measure the degree of load imbalance by the *imbalance factor*, defined as

$$\eta = \tfrac{L_{\max}-L_{\text{avg}}}{L_{\text{avg}}}, \qquad (14)$$

where $L_{\max}$ and L_{avg} are the maximum and average load across servers. Lower values of η imply better load balancing.

Baselines We benchmark SP-Cache against three baselines.

EC-Cache: We used a $(10, 14)$ erasure coding scheme in EC-Cache, which is shown to achieve the best performance [1]. The cache redundancy is 40%. The EC-Cache implementation we used in evaluations is provided by the authors of [1].

Selective replication: For a fair comparison, we copied the top 10% popular files to 4 replicas. Therefore, assuming equal-sized files, the overall cache redundancy incurred by selective replication is $10\% \times 4 = 40\%$—the same as that of EC-Cache.

Fixed-size chunking: Fixed-size chunking is a common practice for many distributed storage/caching systems, e.g., HDFS [17], Windows Azure Storage [16], and Alluxio [14]. With fixed-size chunking, files are split into multiple chunks of a constant size, distributed randomly across servers.

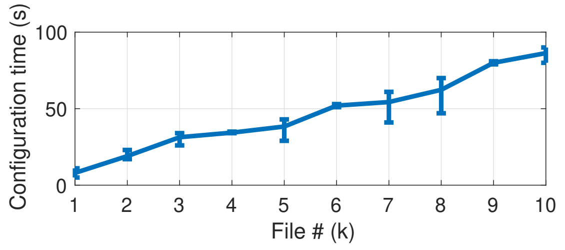


Fig. 8: The configuration time required to compute the optimal scale factor. The error bars show the minimum and the maximum overhead in 5 trials.

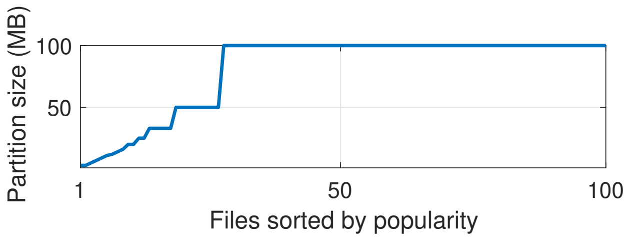


Fig. 9: The partition sizes configured by SP-Cache for files ordered by popularity (from the most popular to the least). Only the top 30% of hot files get partitioned.

B. Configuration of the Scale Factor and Partition Size

We first show that SP-Cache is able to configure the optimal scale factor α based on the derived upper bound. We ran the experiment with 300 files (100 MB each), and set the total access rate to 8 requests per second. Fig. 6 compares the derived upper bound and the average read latency measured in the experiments. Notice that in Fig. 6, we explore a larger range of α than what SP-Cache would search (Sec. V-C) to demonstrate the tightness of the bound. We observe that the "elbow point" of the upper bound well aligns with that of the mean latency, suggesting that the upper-bound analysis can be used to accurately locate the optimal scale factor α.

Overhead The computational overhead of configuring the optimal α depends on the number of files, as it requires the latency upper bound (8) to be computed for *each* file. To quantify this overhead, we measured the runtime required to configure the optimal α in the master node with 1-10k files. Fig. 8 shows the average configuration time in 5 trials, where the error bars depict the maximum and the minimum. With more files, the configuration time linearly increases. Nevertheless, even with 10k files, it takes SP-Cache no more than 90 seconds to finish configuration. Since the configuration is only needed every 12 hours, its overhead is negligible.

Partition size Fig. 9 shows the optimal partition sizes SP-Cache chooses for files ordered by popularity (using the configured scale factor) in an experiment with 100 files (100 MB each). SP-Cache only partitions the top 30% of hot files but leaves the others untouched (no splitting). The variance in the optimal partition numbers also indicates that configuring a uniform partition number regardless of the file popularity would be highly inefficient, even with a small number of files.

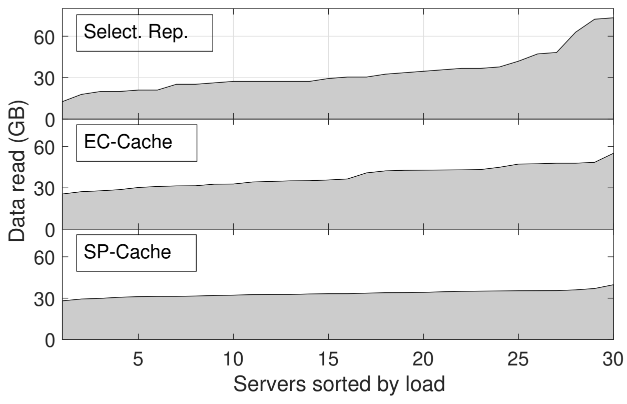


Fig. 10: Load distribution under the three load-balancing schemes. The load of a server is measured by the total amount of data reads. The client request rate is 18.

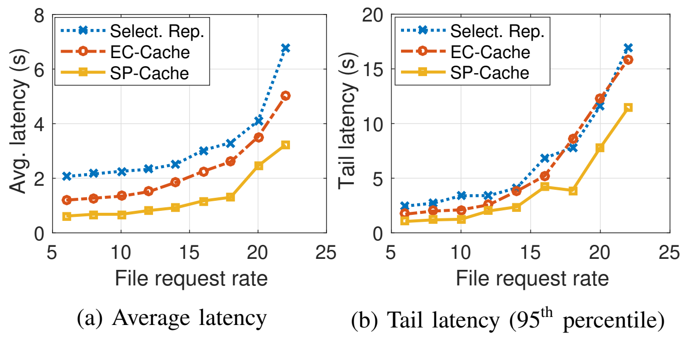


Fig. 11: Mean and tail (95th) latencies under skewed file popularity.

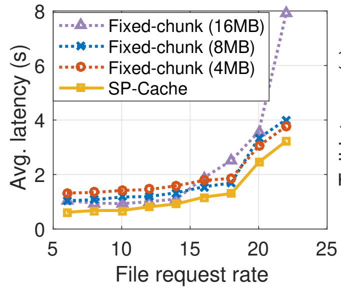
C. Skew Resilience

We evaluated SP-Cache against the two baselines under skewed popularity in the EC2 cluster with *naturally occurred* stragglers. We cached 500 files each of size 100 MB. Note that the total cache space (300 GB) is *sufficient* to hold all 500 files and their replicas (parity partitions). For the purpose of stress-testing, we configured the aggregated request rate from all clients to 18.

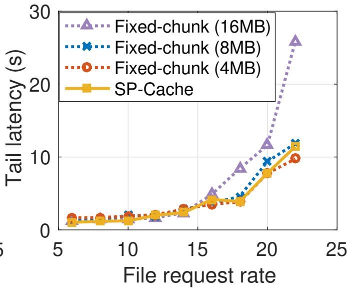
Load Balancing To study how well SP-Cache results in better load balancing than the two baselines, we measured the load of each server (i.e., the amount of data reads) under each scheme. Fig. 10 compares the load distributions under the three schemes. SP-Cache achieves the best load balancing, with imbalance factor $\eta = 0.18$. This is 2.4× and 6.6× better than EC-Cache ($\eta = 0.44$) and selective replication ($\eta = 1.18$), respectively.

Read Latency Fig. 11 compares the mean and tail read latencies of the three schemes under various request rates. Owing to the improved load balancing, SP-Cache consistently outperforms the two baselines. The benefits of SP-Cache become more prominent as the request rate surges. In particular, compared to EC-Cache (selective replication), SP-Cache significantly improves the mean and tail latencies by 29-50% (40-70%) and 22-55% (33-63%), respectively.

Fixed-size chunking Fig. 12 compares SP-Cache against

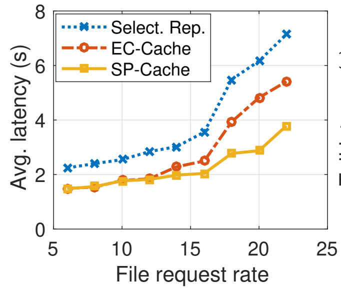


(a) Average latency

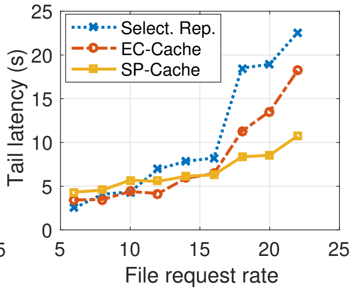


(b) Tail latency (95th percentile)

Fig. 12: Mean and tail (95th) latencies compared with fixed-size chunking.



(a) Average latency.



(b) Tail latency (95th percentile).

Fig. 13: Mean and tail (95th) latencies with injected stragglers.

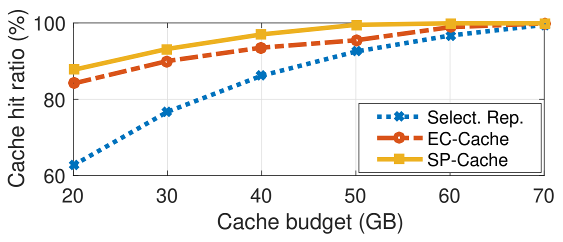


Fig. 14: Cache hit ratio with throttled cache budget.

fixed-size chunking with chunk size of 4, 8, and 16 MB. We observe similar problems as simple partition (Sec. IV-B). On one hand, configuring small chunks results in heavy network overhead due to the increased read parallelism. We see in Fig. 12a that at low request rates (< 15), the average read latency increases as the chunk size gets smaller, e.g., up to 46% (32%) slower than SP-Cache with 4 MB (8 MB) chunks—an evidence that network overhead dominates. On the other hand, configuring large chunks, though saving the network overhead, fails to mitigate hot spots under heavy loads (request rate > 15). In fact, the mean latency with 16 MB chunks is over $2\times$ that of SP-Cache when the access rate rises to 22.

In terms of the tail latency, fixed-size chunking achieves comparable performance to SP-Cache with small chunk sizes (e.g., 4 and 8 MB), as it effectively reduces the hot spots, which are the main source of congestions in our experiments.

D. Resilience to Stragglers

Redundant caching is proven resilient to stragglers [1], [8]. To evaluate how SP-Cache, which is *redundancy-free*, performs in this regard, we turn to controlled experiments with more intensive stragglers than that has been observed in the EC2 cluster. Specifically, we manually injected stragglers following the pattern profiled from a Microsoft Bing cluster trace (Sec. IV-B). We turned each cluster node to stragglers with probability 0.05 (i.e., intensive stragglers [42]).

Fig. 13 shows the results. Despite intensive stragglers, SP-Cache reduces the mean latency by up to 40% (53%) compared to EC-Cache (selective replication). Yet, the presence of stragglers results in prolonged *tail* latencies. In fact, SP-Cache exhibits slightly longer tails than the two redundant caching baselines at low request rate, as reading files from many locations adds the chance of encountering an injected straggler. As the request rate increases, most of the read requests get congested on the hot spots, and the load imbalance becomes the main source of the tail latency. Consequently, the tail latencies under the two baselines quickly ramp up. In contrast, SP-Cache effectively tames load imbalance across servers, reducing the tail by up to 41% (55%) over EC-Cache (selective replication).

E. Hit Ratio with Throttled Cache Budget

We stress that the benefits of SP-Cache evaluated so far were realized with 40% less memory than the two baselines. Should the same cache budget be enforced, SP-Cache would have attained even more significant benefits. To this end, we *throttled* the cluster caches and measured the *cache hit ratio* under the three load-balancing schemes. Specifically, we refer to the cluster settings in Sec. VII-C and used the LRU (least-recently-used) policy for cache replacement. Fig. 14 compares the cache hit ratio of the three schemes with various cache budget. Owing to the redundancy-freeness, SP-Cache keeps the most files in memory and achieves the highest cache hit ratio. In comparison, selective replication falls short, as caching multiple replicas of hot files requires to evict the same number of otherfiles out of the memory.

F. Trace-driven Simulation

Previous evaluations have assumed the uniform file size and Poisson arrivals of the read requests. We next remove these assumptions through trace-driven simulations with the real-world size distribution and request arrivals.

Workload We synthesized the workload based on the file size distribution and the request arrivals from two public traces. Specifically, our simulation generated 3k files. The file sizes follow the distribution in the Yahoo! traces [32] (Fig. 1); the file popularity follows a Zipf distribution with exponent 1.1. We assume that a larger file is more popular than a smaller one. As the Yahoo! trace [32] provides no request arrival information, we refer to the Google cluster trace [51] which contains the submission sequence of over 660k Google cluster jobs (e.g., MapReduce and Machine Learning). Since cluster jobs usually read input at the beginning, we simply use the job submission sequence as the read request arrivals.

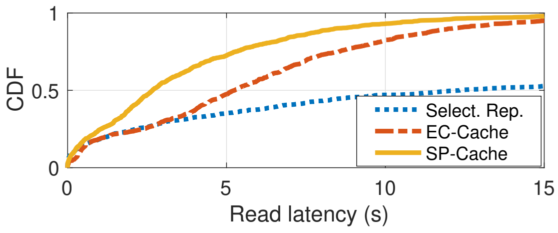


Fig. 15: Distributions of the read latencies under three load-balancing schemes in trace-driven simulations.

Settings We simulated a cluster of 30 cache servers, each with 10 GB memory and 1 Gbps network bandwidth. We manually injected stragglers into the simulated cluster as described in Sec. VII-D. We assume that a cache miss causes $3\times$ longer read latency than a cache hit. We used a $(10, 14)$ coding scheme in EC-Cache and set the decoding overhead as 20% (Sec. III-B).

Results Fig. 15 shows the distributions of the read latencies under the three load-balancing schemes. SP-Cache keeps in the lead with the mean latency of 3.8 s. In comparison, the mean latencies measured for EC-Cache and selective replication are 6.0 s and 44.1 s, respectively. As hot files have large sizes in production clusters, redundant caching results in even lower cache utilization, inevitably harming its I/O performance.

VIII. Limitation and Discussion

While SP-Cache significantly outperforms the existing solutions, our current implementations have several limitations. We discuss those limitations and leave them for future explorations.

Short-term Popularity Variation With periodic load balancing, SP-Cache is unable to timely handle the short-term popularity shifts, e.g., bursts of access to certain files. To address this problem, we can enable *online and dynamic adjustment of partition granularity* in case that some files may turn hot (or cold) in a short period of time. We expect SP-Cache to respond to popularity variations much faster than EC-Cache and selective replication. To quickly adjust the partition granularity in an online fashion, SP-Cache can split and combine the existing partitions. This can be done in a *distributed manner* and incurs only a small amount of data transfer. In contrast, EC-Cache needs to collect *all* the partitions at the master node for re-encoding; selective replication incurs $1\times$ bandwidth and storage overhead for every additional replica.

Fault Tolerance While SP-Cache manages to minimize the impact of stragglers, it does not provide fault tolerance for *non-transient* stragglers which can be *arbitrarily* slow to the extent of a complete failure. We stress that such fault tolerance *cannot* be achieved *without cache redundancy* [1], [8]. Nevertheless, since the underlying storage system readily handles storage faults (e.g., the cross-rack replication of HDFS [17] and S3 [15]), SP-Cache can always recover the lost data from stable storages relying on the checkpointing and recomputing mechanism of Alluxio.

Finer-Grained Partition For structured data with clear semantics, e.g., Parquet files [52], it is unnecessary to partition or replicate the entire file uniformly if there are discrepant popularities within the file. In this case, SP-Cache can be extended to support finer-grained partition within a file by examining the popularities of different parts of the file.

IX. Related Work

Cluster caching has been broadly employed in data-intensive clusters as disk I/O remains the primary performance bottleneck for data analytics [4], [6], [9]. To achieve load-balanced caching, various techniques have been proposed, including data placement optimizations and replication/partition schemes.

Data placement One common approach for load balancing is to optimize the data placement scheme by designing the mapping function from files to servers. For instance, consistent hashing [53], [54] is a popular choice to implement such mappings. Adaptively adjusting the hash space boundaries [36], [55] can further improve the mapping efficiency. Unlike these works, SP-Cache *obviates* the need for placement optimizations by eliminating the skew in the per-partition load (Sec. V-B). The server load can then be balanced with *random placement*.

Replication Replication has been the *de facto* load balancing technique used in the disk-based object stores, including Amazon S3 [15], OpenStack Swift [30], and Windows Azure Storage [16]. Given the skewed popularity, replicating all files *uniformly* wastes the storage capacity. Selective replication [8], [56] comes as a solution. However, as popular files often have large sizes, selective replication incurs high memory overhead, and is ruled out as a practical solution for cluster caching.

File Partition EC-Cache [1] is the work most related to SP-Cache, which also takes advantage of file partition to load-balance cache servers. SP-Cache is by no means a “coding-free” version of EC-Cache. Instead, it judiciously determines the partition number of a file based on its load contribution, whereas EC-Cache simply settles on a uniform partition scheme. To our knowledge, SP-Cache is the first work that systematically explores the benefits of selective partition.

X. Conclusions

In this paper, we have designed, analyzed, and developed SP-Cache, a load-balanced, redundancy-free cluster caching scheme for data-parallel clusters. SP-Cache selectively splits hot files into multiple partitions based on their sizes and popularities, so as to evenly spread the load of their read requests across multiple servers. We have established an upper-bound analysis to quantify the mean latency, and used it to guide the search of the optimal partition number for each file. SP-Cache effectively eliminates the hot spots while keeping the impact of stragglers to the minimum. We have implemented SP-Cache atop Alluxio. EC2 deployment and trace-driven simulations showed that SP-Cache significantly outperforms existing solutions with better load balancing in a broad range of settings. Notably, with 40% less memory footprint than EC-Cache, SP-Cache improves both the mean and the tail latencies by up to 40%, even in the presence of intensive stragglers.

Acknowledgment

We would like to thank K. V. Rashmi and the other authors of [1] for generously sharing the source code of EC-Cache for our evaluation. We also thank Bin Fan for his detailed feedback that helped improve the paper, as well as the anonymous reviewers of SC18 for their valuable comments. This research is supported in part by the Hong Kong ITF Award ITS/391/15FX and RGC GRF Grant 16200214. Yinghao was supported in part by the Hong Kong PhD Fellowship Scheme.

Appendix

A. Decoding overhead of EC-Cache

To evaluate the decoding overhead of EC-Cache, we ran EC-Cache in an Amazon EC2 cluster with 30 `r3.2xlarge` memory-optimized instances. Following [1], we used a $(10, 14)$ coding scheme. We launched an EC-Cache client submitting file read requests and measured the incurred decoding overhead, i.e., the decoding time normalized by the read latency. We depict the evaluation results in Fig. 16.

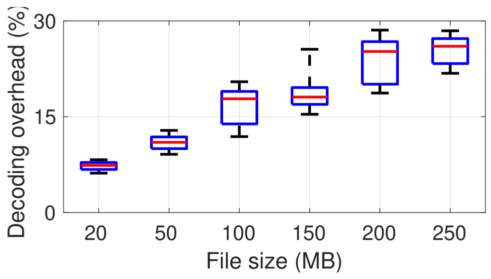


Fig. 16: Decoding overhead in EC-Cache. Boxes depict the 25th, 50th, and 75th percentiles. Whiskers depict the 5th and 95th percentiles.

B. Proof of Theorem 1

Consider any particular server. Denote X_i as the load contributed by file i to this server. We have

$$X = \sum_i X_i,$$

Assuming independent partition placement across files, we have

$$\text{Var}(X) = \sum_i \text{Var}(X_i). \tag{15}$$

To facilitate the derivation of Var(X_i), we define a binary random variable a_i indicating whether the request for file i is served by this server. The load X_i can be expressed as

$$X_i = a_i \frac{L_i}{k_i},$$

where k_i denotes the (non-parity) partition number of file i and $\frac{L_i}{k_i}$ calculates the partition-wise load of file i.

Under SP-Cache, each server has a probability of $\frac{k_i^{SP}}{N}$ to be selected to cache the partitions of file i. Therefore, a_i^{SP} follows Bernoulli distribution with parameter $\frac{k_i^{SP}}{N}$. We have

$$\text{Var}(X_i^{SP}) = (\tfrac{L_i}{k_i^{SP}})^2 \text{Var}(a_i^{SP}) = (\tfrac{L_i}{k_i^{SP}})^2 \tfrac{k_i^{SP}}{N}(1 - \tfrac{k_i^{SP}}{N}).$$

Under EC-Cache, each server has a probability of $\frac{n_i^{EC}}{N}$ to cache the partitions of file i; each server caching the partitions has a probability of $\frac{k_i^{EC}+1}{n_i^{EC}}$ to serve the request. Therefore, a_i^{EC} follows Bernoulli distribution with parameter $\frac{n_i^{EC}}{N} \cdot \frac{k_i^{EC}+1}{n_i^{EC}} = \frac{k_i^{EC}+1}{N}$. Similarly, we have

$$\text{Var}(X_i^{EC}) = (\tfrac{L_i}{k_i^{EC}})^2 \text{Var}(a_i^{EC}) = (\tfrac{L_i}{k_i^{EC}})^2 \tfrac{k_i^{EC}+1}{N}(1 - \tfrac{k_i^{EC}+1}{N}).$$

Suppose that the server number N is much larger than the partition number k_i. With (15), we have

$$\frac{\text{Var}(X^{EC})}{\text{Var}(X^{SP})} \approx \frac{\sum_i (\frac{L_i}{k_i^{EC}})^2 \frac{k_i^{EC}+1}{N}}{\sum_i (\frac{L_i}{k_i^{SP}})^2 \frac{k_i^{SP}}{N}} \approx \frac{\sum_i \frac{L_i^2}{k_i^{EC} N}}{\sum_i \frac{L_i}{\alpha N}} = \frac{\alpha}{k^{EC}} \frac{\sum_i L_i^2}{\sum_i L_i},$$

which completes the proof. □

References

[1] 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. USENIX OSDI*, 2016.
[2] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. ACM SoCC*, 2014.
[3] Presto, "https://prestodb.io/."
[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX NSDI*, 2012.
[5] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables." in *Proc. USENIX OSDI*, 2010.
[6] Memcached, "https://memcached.org."
[7] MemSQL, "http://www.memsql.com."
[8] 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. ACM Eurosys*, 2011.
[9] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: coordinated memory caching for parallel jobs," in *Proc. USENIX OSDI*, 2012.
[10] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *ACM HotNets*, 2014.
[11] Intel Storage Acceleration Library, "https://github.com/01org/isa-l."
[12] R. Nelson and A. N. Tantawi, "Approximate analysis of fork/join synchronization in parallel queues," *IEEE Trans. Computers*, vol. 37, no. 6, pp. 739–743, 1988.
[13] C. Kim and A. K. Agrawala, "Analysis of the fork-join queue," *IEEE Trans. Computers*, vol. 38, no. 2, pp. 250–255, 1989.
[14] Alluxio, "http://www.alluxio.org/."
[15] Amazon S3, "https://aws.amazon.com/s3."
[16] Windows Azure Storage, "https://goo.gl/RqVNmB."
[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE Symp. Mass Storage Syst. and Technologies, 2010*, 2010.
[18] Gluster File System, "https://www.gluster.org."
[19] Amazon Elastic Compute Cloud, "https://aws.amazon.com/ec2/," 2016.
[20] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," in *Proc. ACM SIGCOMM*, 2015.
[21] Huawei. NUWA. https://www.youtube.com/watch?v=OsmZBRB_OSw.
[22] K. Asanovic and D. Patterson, "FireBox: A hardware building block for 2020 warehouse-scale computers," in *USENIX FAST*, 2014.
[23] D. Alistarh, H. Ballani, P. Costa, A. Funnell, J. Benjamin, P. Watts, and B. Thomsen, "A high-radix, low-latency optical switch for data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 367–368, 2015.
[24] C. Scott. Latency trends. http://colin-scott.github.io/blog/2012/12/24/latency-trends/.
[25] IEEE P802.3ba 40 Gbps and 100 Gbps Ethernet Task Force. http://www.ieee802.org/3/ba/.

[26] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *ACM HotNets*, 2013.
[27] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *Proc. USENIX OSDI*, 2016.
[28] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in datacenter computing considered irrelevant," in *ACM HotOS*, 2011.
[29] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. ACM SoCC*, 2017.
[30] OpenStack Swift, "https://www.swiftstack.com."
[31] Redis, "http://redis.io."
[32] Yahoo! Webscope Dataset, "https://goo.gl/6CZZCF."
[33] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. ACM IMC*, 2009.
[34] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. ACM SIGCOMM*, 2013.
[35] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009.
[36] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proc. ACM SoCC*, 2013.
[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. USENIX ATC*, 2012.
[38] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *Proc. VLDB Endowment*, vol. 6, no. 5, 2013, pp. 325–336.
[39] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.
[40] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM*, 2010.
[41] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for tcp in data-center networks," *IEEE/ACM Trans. Netw.*, vol. 21, no. 2, pp. 345–358, 2013.
[42] 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. USENIX OSDI*, 2010.
[43] B. Gnedenko and I. Kovalenko, *Introduction to Queuing Theory. Mathematical Modeling*. Birkhaeuser Boston, Boston, 1989.
[44] Y. Xiang, T. Lan, V. Aggarwal, and Y.-F. R. Chen, "Joint latency and cost optimization for erasure-coded data center storage," *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 2443–2457, 2016.
[45] G. Joshi, Y. Liu, and E. Soljanin, "On the delay-storage trade-off in content download from coded distributed storage systems," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 989–997, 2014.
[46] M. Fidler and Y. Jiang, "Non-asymptotic delay bounds for (k, l) fork-join systems and multi-stage fork-join networks," in *Proc. IEEE INFOCOM*, 2016.
[47] CVXPY, "http://www.cvxpy.org/."
[48] U. Niesen and M. A. Maddah-Ali, "Coded caching with nonuniform demands," *IEEE Trans. Inform. Theory*, vol. 63, no. 2, pp. 1146–1158, 2017.
[49] C. Roadknight, I. Marshall, and D. Vearer, "File popularity characterisation," in *Proc. ACM Sigmetrics*, 2000.
[50] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.
[51] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM SoCC*, 2012, p. 7.
[52] Apache Parquet. https://parquet.apache.org.
[53] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," *Computer Networks*, vol. 31, no. 11-16, pp. 1203–1213, 1999.
[54] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," in *Proc. ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, 2010, pp. 35–40.
[55] J. Hwang and T. Wood, "Adaptive performance-aware distributed memory caching." in *Proc. USENIX ICAC*, 2013.
[56] 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. USENIX NSDI*, 2013.
[57] SP-Cache Source Code, "https://goo.gl/exX4NE."
[58] Spark-EC2, "https://github.com/amplab/spark-ec2."

Artifact Description

We have open-sourced the implementations of SP-Cache for public access [57]. Note that we do not provide the implementation for EC-Cache [1], as the release of the source code must be granted by the authors in [1]. We present the Artifact Description to illustrate how to reproduce the evaluation results of SP-Cache.

A. Description

1) Check-list (artifact meta information):

- **Algorithm:** SP-Cache.
- **Program:** Java project and python programs.
- **Compilation:** GCC 4.6.3; Java 1.7.0.
- **Run-time environment:** Linux 3.4.37-40.44.amzn1. x86_64.
- **Hardware:** Intel® Xeon® CPU E5-2670 v2.
- **Output:** the log files generated by SP-Cache.
- **Experiment workflow:** install the dependent python packages; compile SP-Cache; set up the cluster; generate test datasets and run experiments.
- **Publicly available:** yes.

2) How software can be obtained (if available): SP-Cache can be cloned from GitHub using the following URL: https://github.com/sp-cache-for-sc/SP-Cache.git.

3) Hardware dependencies: We run experiments in a 51-node EC2 cluster with instance type of `r3.2xlarge`. To set up the EC2 cluster, we can use the scripts provided by the Spark community [58].

4) Software dependencies: SP-Cache requires GCC 4.6.3 or above, python 2.7 or above and Java 1.7 or above. To solve the convex programming for the derivation of the upper bound (8), SP-Cache relies on the python package CVXPY. To compile the entire project, maven 3.2.1 or above is required.

5) Datasets: The design target of SP-Cache is general-purpose data read/write in data-parallel clusters. Therefore, we generate files with dummy data in the evaluations.

B. Installation

On the master node of the cluster, clone SP-Cache from GitHub:

```
$ git clone https://github.com/sp-cache-for-sc/
    SP-Cache.git
```

Next, install maven and add the path as an environmental variable:

```
$ wget http://mirror.olnevhost.net/pub/apache/
    maven/binaries/apache-maven-3.2.1-bin.tar.gz
$ tar xvf apache-maven-3.2.1-bin.tar.gz
$ echo 'export M2_HOME=/path/to/maven' >> .
    bash_profile
$ echo 'export M2=$M2_HOME/bin' >> .bash_profile
$ echo 'export PATH=$M2:$PATH' >> .bash_profile
$ source .bash_profile
```

Install the dependent python packages:

```
$ pip install numpy
$ pip install cvxpy
```

Next, compile SP-Cache with maven:

```
$ cd SP-Cache
$ mvn clean install -DskipTests=true -Dlicense.
    skip=true -Dcheckstyle.skip -Dmaven.javadoc.
    skip=true
```

Finally, synchronize the SP-Cache project to all nodes in the cluster with the `spark-ec2` script:

```
$ spark-ec2/copy-dir SP-Cache
```

Now, we are ready to set up SP-Cache and run the experiments.

C. Experiment workflow

In the SP-Cache project, we have included the bash scripts for cluster setup, launching SP-Cache, running experiments and collecting the experimental results.

Before launching SP-Cache, we first need to set the memory capacity on each worker (cache server), which could be configured in the file `alluxio-env.sh` inside the `conf` folder:

```
$ ALLUXIO_WORKER_MEMORY_SIZE=${
    ALLUXIO_WORKER_MEMORY_SIZE:-"10GB"}
```

Next, synchronize the `conf` folder and set up the cluster for evaluations:

```
$ ./setup.sh
```

Now, the SP-Cache cluster has been set up. We generate the test files with

```
$ ./pre_benchmark.sh $1 $2
```

where the two parameters are file size in MB and the exponent parameter for the Zipf-distributed popularity, respectively.

To run the experiment, we use the following command:

```
$ ./run_benchmark.sh $1 $2 $3
```

The three input parameters are the client number, the total request number each client shall submit and the request arrival rate.

Notice that each client keeps the experiment logs locally. We can collect and merge them with

```
$ ./collect_results.sh $1
```

where the parameter represents the client number.

D. Evaluation and expected result

The collected results will be placed in the `results` folder. Specifically, the read latency from all clients will be written to the file `all_latency.txt`. The cache hits and server loads will be merged into files `all_fileHit.txt` and `all_workerLoads.txt`, respectively.

E. Experiment customization

Scale factor: SP-Cache will record the access counts of all files and update the partition numbers every 12 hours to re-balance the load. To efficiently evaluate the performance of SP-Cache after load re-balancing, we provide a shortcut to re-partition files without having to wait for 12 hours. We will manually trigger the re-partition of SP-Cache assuming that the popularities has been well estimated. Specifically, we run a python script `python-sp-server/scale_factor_configuration.py` to configure the scale factor, which is exactly what SP-Cache would do every 12 hours. The only difference is that we directly use the exponent parameter to calculate the popularities instead of relying on the historic access counts within the learning window. Next, we can specify the configured scale factor in the `/pre_benchmark.sh` to re-partition the files and then run the experiments.

Straggler probability: the straggler probability for SP-Cache is configured by reading the file `strag_prob.txt` in the `/root/test_files` folder. For experiments with naturally-happening stragglers (Sec. VII-C), write 0.0 in this file. For experiments with intensified straggler patterns from the Microsoft Bing cluster (Sec. VII-D), write 0.05 in this file.

Cache volume: to run the experiments with throttled cache budget (Sec. VII-E), set the corresponding cache capacity of each server in the configuration file `conf/alluxio-env.sh` and repeat the procedures in C.

F. Notes

For the performance study of EC-Cache and selective replication, we have deployed the source codes shared by the authors of EC-Cache [1]. However, we are currently not able to provide the information of their codes since the EC-Cache project has not been open-sourced yet. Nonetheless, we have reported the experimental configurations of EC-Cache and selective replication in the paper, ensuring a fair comparison of their performances with SP-Cache.