LACS: Load-Aware Cache Sharing with Isolation Guarantee

Yinghao Yu[†], Wei Wang[†], Jun Zhang[‡], Khaled Ben Letaief[†]
[†]Hong Kong University of Science and Technology
[‡]Hong Kong Polytechnic University
[†]{yyuau, weiwa, eekhaled}@ust.hk [‡]jun-eie.zhang@polyu.edu.hk

***Abstract*—Cluster caching has been increasingly deployed in front of cloud storage to improve I/O performance. In shared, multi-tenant environments such as cloud datacenters, cluster caches are constantly contended by many users. Enforcing *performance isolation* between users hence becomes imperative to cluster caching. A user's caching performance critically depends on two factors: (1) the *amount* of cache allocation and (2) the *load* of servers in which its files are cached. However, existing cache sharing policies *only* provide guarantees on the amount of cache allocation, while remaining *agnostic* to the load of cache servers. Consequently, "mice" users having files *co-located* with "elephants" contributing heavy data accesses may experience extremely long latency, hence receiving no isolation. In this paper, we propose a Load-Aware Cache Sharing scheme (LACS) to enforce isolation between users. LACS keeps track of the load contributed by each user and reins back the congestions caused by elephant users by throttling their cache usage and network bandwidth. We have implemented LACS atop Alluxio, a popular cluster caching system. EC2 deployment shows that LACS achieves performance isolation in the presence of elephants, while improving the mean read latency by up to 80.4% (25.3% on average) over the state-of-the-art load balancing technique.**

I. Introduction

Memory caches are being aggressively used to provide high throughput, low latency I/O in a multitude of data-intensive applications, ranging from big data analytics [1]–[3], to cloud storage [4]–[6], and to machine learning [7]. However, memory caches remain a constrained resource and are heavily contended by many users in shared, multi-tenant environments such as cloud datacenters. The primary challenge in this regard is to *efficiently* share caches with guaranteed *performance isolation* between users.

There are two key factors dictating the caching performance of a user: (1) the *total amount* of caches allocated and (2) the *placement* of its cached files. The former determines the *quantity* of the allocation, i.e., how much data a user can cache in memory—the more the better. The placement of cached files, on the other hand, determines the *quality* of the allocation. For example, users having files co-located with some *hot* data objects in a cache server likely suffer from long latency as those objects may result in severe network congestion.

However, existing cache sharing policies simply focus on the quantity of allocation, without concern for the quality, and hence cannot provide performance isolation between users. In particular, these policies seek to provide a *guaranteed amount of minimum cache space* for individual users by means of *fair sharing* [8]–[10], while leaving the actual file placement to the decision of the *load balancing* algorithms. Nevertheless, prevalent load balancing algorithms, such as Round-Robin [11], [12], do not differentiate data accesses from individual users but aim to balance the *aggregated load* across machines. In this regard, less popular objects are often co-located with the hot ones to achieve balanced load. Consequently, users caching less popular objects may end up with a significant delay due to the heavy congestion caused by the co-located hot objects. Meaning, there is no performance guarantee for these users.

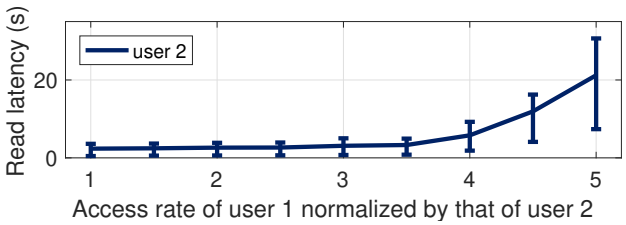


Fig. 1: An empirical study with two users sharing a 5-node caching cluster. The curve shows the average latency of user 2 as the load of user 1 increases. Errors bars depict the 95^{th} and 5^{th} percentiles.

We illustrate this problem through an empirical study. We deployed an EC2 [13] cluster of 5 cache servers using Alluxio [5], [14], a popular cluster caching system. Each cache server is an `m4.xlarge` instance. The cluster is shared by two users, using *max-min fair* cache allocation [8]. We use the default Round-Robin load balancer [12] in Alluxio. We gradually increased the access rate of one user (user 1) and examined how this may affect the performance of the other (user 2). Fig. 1 depicts the measured read latency of user 2. As the load of user 1 surges to a certain level (3 times that of user 2), the read latency of co-located user 2 starts to grow exponentially. We see from this experiment that simply maintaining a guaranteed amount of cache allocation without considering the underlying load interference between users is *incapable* of isolating their caching performance.

A naive solution to this problem is to set up a *separate* caching instance for each user with *dedicated* cache space and *reserved* link bandwidth on each machine. However, this approach can be highly inefficient for two reasons. First,

in cloud environments, it is common to have users with *shared* data objects, e.g., multiple users querying the same Parquet file using Spark SQL. Setting up separate caching instances precludes the sharing opportunity of those commonly interested files but mandates multiple copies of the same objects, leading to poor cache utilization. Second, reserving dedicated bandwidth for each user is also inefficient, as users may occasionally turn inactive, during which their idle bandwidth could have been otherwise utilized by other active users.

In this paper, our goal is to enforce performance isolation between users in shared cluster caches while still attaining high caching efficiency. To this end, we propose LACS, a Load-Aware Cache Sharing scheme, which keeps track of the *load* contributed by each user and *jointly* determines cache allocation and the file placement. To attain high efficiency, LACS strives to share both the cached files and the network bandwidth among users. In case that a user contributes too much load that would compromise performance isolation, LACS quickly intervenes by throttling its cache accesses and bandwidth usage, so as to curb its negative impact on other users. We show through theoretical analysis that LACS provides guaranteed performance isolation.

We have implemented LACS as a pluggable cache manager atop Alluxio [14] and evaluated its performance via EC2 deployments. Experimental results confirm that LACS achieves performance isolation while attaining high caching efficiency. In the presence of users contributing heavy loads, LACS improves the mean latency by up to 80.4% (25.3% on average) over the state-of-the-art load balancing technique. We have also measured the computational overhead of LACS. For 30 users sharing 5k files, LACS quickly configures the cache allocations and file placement in less than 2 seconds.

II. Background and Motivation

In this section, we provide background information on cluster caching and motivate the need for performance isolation. We show that existing cache sharing policies either suffer from poor efficiency or fail to provide isolation guarantees.

A. Background

Cluster caching Storage I/O remains a major bottleneck in today's data-intensive clusters [1], [5], [15]. In light of this problem, cluster caching systems, such as Alluxio [14], Redis [6], and Memcached [4], have been increasingly deployed in front of cloud object stores (e.g., Amazon S3 [16]) for faster I/O. By persisting a large volume of data objects in the caching layer, cluster caching can speed up I/O-intensive applications by orders of magnitude [5], [17]. In this paper, we primarily target the storage-side caching to improve the I/O performance.

Cache sharing and the need of isolation As a constrained resource, memory caches are routinely contended by many users in cloud environments. Therefore, how should caches be shared among contending users becomes critically important. This problem appears even more challenging as users typically have highly *imbalanced load* [18], [19]. For example, in a 3000-node Facebook cluster, the top 5% hot files were 7x more popular than the bottom 75% [18]. Meaning, a small number of aggressive users have contributed a large fraction of data accesses. If not handled appropriately, the presence of these aggressive users would significantly delay data access of the other users in a shared cluster, as demonstrated in Fig. 1. We therefore mandate a cache sharing scheme that provides *guaranteed performance isolation* between users.

Caching performance Many metrics have been used in the literature to measure the caching performance, such as *cache hit ratio* and *latency*. As we are interested in the *end-to-end performance* in cluster caching, we use the *mean read latency* as the primary metric, which includes both the *I/O* and *network delay*. The former measures how long a data object can be read from a cache server, which critically depends on if the object can be directly accessed in memory; the latter measures how long it takes to transfer the object from the server to the client, which is dictated by the available network bandwidth.

B. A Naive Way to Enforce Isolation

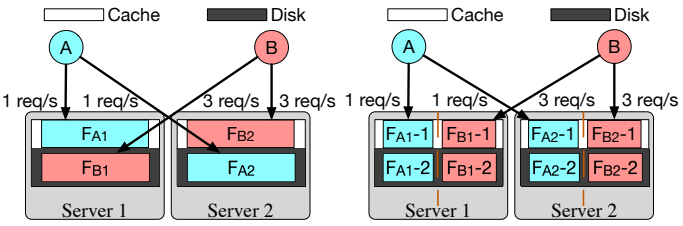
A straightforward way to enforce performance isolation is to implement *isolated caches*. That is to say, for each user, we set up an *isolated caching instance* in each machine with *dedicated*, *evenly-partitioned* cache space and network bandwidth. More precisely, each of K users is guaranteed with $1/K$ of both the total amount of caches and the link bandwidth of each cache server. User's read latency hence gets isolated, *irrespective* of the access load and file placement of the other users.

However, maintaining isolated caches can be highly inefficient. First, it is common to have multiple users requesting the same datasets in caching clusters. In fact, it is reported that over 30% of files were shared by at least two users in a production HDFS cluster [8]. With isolated caches, these files cannot be shared but are replicated to multiple copies in separate caching instances, resulting in low cache utilization. Second, reserving dedicated bandwidth on a per-user basis prevents dynamic sharing of unused network resources, which, in turn, prolongs the read latency of active users.

C. Prior Art and Its Problem

Given the inefficiency of maintaining isolated caches, existing solutions resort to *fair allocation* to provide guarantees on the *minimum amount* of caches a user can expect, without explicit control over the underlying file placement. However, we show that this may result in *arbitrarily long* read latency.

Max-min fairness [20]–[22] has been used as the primary principle for fair cache allocation [8]. In a nutshell, it seeks to maximize the minimum cache allocation across all users. Consider a toy example in Fig. 2, where two users share two cache servers. Each server has *one unit* of cache space but unlimited disk. Each user has two files of *unit size* to cache. The file access rate is 1 for user A (i.e., 1 read request for each file per second), and 3 for user B. Limited by the network bandwidth, each server can serve 5 read requests per second (i.e., transferring 5 files per second). As shown in Fig. 2a, with max-min fairness, each user is allocated one unit of cache space and uses it to cache one file. As each user has two files to cache, the other file must be spilled to disk.



(a) Max-min fair allocation (b) Isolated allocation

Fig. 2: An example showing that max-min fair allocation does not provide isolation. There are two users and two cache servers, each with one unit cache size. Each user accesses two files (of unit size), and the access rates are shown as the weights associated with the edges. (a) Cache allocation with max-min fairness and the default load balancer. (b) Isolated allocation.

The file placement determines the load of cache servers, which critically impacts the network latency and hence the caching performance. However, this is out of the control of the cache allocation policy but determined by the load balancer [12] in the cluster. As shown in Fig. 2a, the balancer co-locates the two users' files in each server, so as to achieve balanced load, i.e., 4 requests per second per machine.

Unfortunately, this placement *violates* performance isolation. Assuming Poisson arrival of read requests and exponential service time, we model each cache server as an M/M/1 queue [20], where the mean network delay of user A is $\frac{1}{5-(3+1)} = 1$ second. On the other hand, if we implement isolated caches, as shown in Fig. 2b, each user gets half of the cache space (0.5 units) and half of the network bandwidth (2.5 requests per second) in each server. The mean network delay of user A is $\frac{1}{2.5-1} = 0.67$ seconds, shorter than that in Fig. 2a. Notice that both max-min fair allocation and isolated caches result in the same cache hit ratio (50%) and hence the same I/O latency. Putting it all together, we see that user A would rather stay isolated for shorter read latency. In fact, if we further increase the access rate of user B beyond 4, max-min fairness would result in *infinitely long* network delay (i.e., M/M/1 queue *unstable*) in both servers.

While inspired by a toy example, our conclusion generally holds true. Without accounting for file placement, max-min fairness alone is incapable of isolating users. Our experiment in Fig. 1 further demonstrates this problem in real systems.

Other fair allocation schemes Existing works also use other fair allocation schemes to provide isolation guarantee for cluster caching, such as proportional fairness [10] and cooperative game [9]. However, they remain agnostic to the load interference between co-located users and hence share the same problem as max-min fair allocation [8]. To the best of our knowledge, no existing cache sharing schemes provides guaranteed performance isolation between users. We shall address this problem in the next two sections.

III. Model and Formulation

In this section, we describe our model for cluster caching that jointly accounts for the impact of both the cache allocation and file placement, based on which we formulate the cache sharing problem with isolation guarantee.

A. Model

Cache allocation and placement Without loss of generality, we consider an M-node cluster serving N files of unit size,[1] where no file is duplicated in the cluster.[2] Let $\mathbf{P} = [p_{ij}]$ be the file *placement matrix* consisting of $N \times M$ binary indicators, i.e., $p_{ij} = 1$ if file i is placed on machine j, and $p_{ij} = 0$ otherwise. Let $\mathbf{A}$ be the *cache allocation vector*, where each component a_i denotes the amount of caches allocated to file i. As a cached file can be *non-exclusively* accessed by *all* users in a shared cluster [4], [6], [14], we do not differentiate allocations between users. We allow fractional caching (i.e., $0 \le a_i \le 1$) but not cross-node. The cache allocation in each machine j should not exceed its total capacity, i.e.,

$$\sum_i a_i p_{ij} \le c, \text{ for all machine } j, \tag{1}$$

where c is the cache capacity of each machine. For simplicity, we assume *homogeneous* cache servers with equal cache capacity.[3] Yet, our solution can be easily generalized to heterogeneous settings.

Given cache allocation $\mathbf{A}$ and file placement $\mathbf{P}$, we derive the mean read latency for each user through a queueing model, which consists of two components, the *I/O* and *network delay*.

I/O delay For user k, let λ_{ki} be its access rate (load) of file i. Summing it up over all files, we obtain the total access rate contributed by user k:

$$\lambda_k = \sum_i \lambda_{ki}. \tag{2}$$

Among all those accesses, only a fraction get served in memory, while the remaining are served on disks. As reading from disk is orders-of-magnitude slower than in-memory read, the I/O latency is dominated by the former. We are hence more interested in quantifying the *disk access rate*. Recall that each file i is allocated a_i amount of caches, meaning, the remaining $1 - a_i$ of the file data must be read from disk. We compute the disk access rate of each user k as

$$\lambda_k^{\text{disk}} = \sum_i \lambda_{ki}(1 - a_i). \tag{3}$$

Therefore, the chance that the user's access request gets a cache miss and is served on disk is simply the disk access rate normalized by the total access rate, i.e., $\lambda_k^{\text{disk}}/\lambda_k$. In our model, we omit the latency of in-memory read, and let Δ be the I/O delay of reading a unit of data on disk. We therefore derive the mean I/O latency of user k as

$$T_k^{\text{IO}} = \frac{\lambda_k^{\text{disk}}}{\lambda_k}\Delta. \tag{4}$$

[1] A file of size s can be equivalently treated as s small files of unit size.

[2] Our model can also be generalized to duplicated files by treating each replica as a separate file. File requests are evenly split across the replicas.

[3] In fact, many cluster caching systems, such as Alluxio [14], only configure uniform cache size across servers.

Network delay measures the time to transfer a file from the cache server to the client, which critically depends on the load of the machine and the available network bandwidth (service rate). Given file placement $\mathbf{P}$, the load of machine j is the sum of the access rate of those users having files placed on it:

$$\Lambda_j = \sum_k \sum_i \lambda_{ki} p_{ij}. \tag{5}$$

For tractable analysis, we model each cache server as an M/M/1 queue [23] with Poisson request arrivals and exponential service time. We shall show in Sec. VI-D that our solution also works well for real-world request arrivals. Let μ_j be the request service rate. As each request reads a file of unit size, the request service rate is exactly the machine's bandwidth. By the property of M/M/1 queue, the mean network delay on machine j, including both queuing and serving, is given by

$$D_j = \frac{1}{\mu_j - \Lambda_j}. \tag{6}$$

To maintain a stable queue with a finite delay, we require the load not exceeding the network bandwidth, i.e., $\Lambda_j < \mu_j$.

We now derive the mean network delay for user k as the weighted sum of each machine's network delay over the user's load distribution, i.e.,

$$T_k^{\text{Net}} = \sum_j \frac{\sum_i \lambda_{ki} p_{ij}}{\lambda_k} D_j, \tag{7}$$

where the inner fraction measures the user's load in machine j normalized by its total load contribution.

Read latency Adding up the I/O and network delay, we obtain the mean read latency of user k, i.e.,

$$T_k = T_k^{\text{IO}} + T_k^{\text{Net}}. \tag{8}$$

B. Problem Formulation

Our goals are two-fold: (1) achieving *efficient* cache sharing and (2) providing *isolation guarantee*. By efficient sharing, we aim to minimize the mean latency *per request*, which is the cumulative latency summed up over all users' read requests normalized by their total load contributions, i.e.,

$$\text{minimize}_{\mathbf{A},\mathbf{P}} \quad \bar{T} = \sum_k \lambda_k T_k / \sum_k \lambda_k. \tag{9}$$

To provide guaranteed performance isolation, each user should be *better off* sharing the cluster than having an *isolated cache* (definition given in Sec. II-B). In particular, let $T_k^\star$ be the minimum mean read latency user k can expect in an isolated cache. We require $T_k \le T_k^\star$ for all user k.

Putting it all together, we formulate a cache sharing problem:

$$\begin{aligned} \mathcal{P}_1 : \underset{\mathbf{A},\mathbf{P}}{\text{minimize}} \quad & \bar{T} = \textstyle\sum_k \lambda_k T_k / \sum_k \lambda_k, \\ \text{subject to} \quad & T_k \le T_k^\star \text{ for all user } k; \\ & \textstyle\sum_i a_i p_{ij} \le c \text{ for all machine } j. \end{aligned} \tag{10}$$

Unfortunately, problem $\mathcal{P}_1$ is NP-hard as it reduces to the *supermodular* minimization problem with Knapsack constraints [24]. We therefore turn to an efficient heuristic solution.

IV. Load-Aware Cache Sharing

In this section, we present our solution, called *L*oad-*A*ware *C*ache *S*haring (LACS), for achieving high efficiency without compromising isolation.

A. Algorithm Overview

In a nutshell, the design of LACS consists of two phases, a *sharing* phase and an *adjusting* phase. LACS starts with the sharing phase, in which it strives to attain the optimal efficiency by *sharing* cache servers among users, while leaving isolation as a secondary concern. More precisely, LACS solves problem $\mathcal{P}_1$ *without* the isolation constraints:

$$\begin{aligned} \mathcal{P}_2 : \underset{\mathbf{A},\mathbf{P}}{\text{minimize}} \quad & \bar{T}, \\ \text{subject to} \quad & \textstyle\sum_i a_i p_{ij} \le c \text{ for all machine } j. \end{aligned} \tag{11}$$

LACS then checks whether the sharing outcome meets the isolation requirement, i.e., no user is worse off sharing than isolation. If it does, LACS settles on sharing. Otherwise, it transits to the adjusting phase. Specifically, LACS identifies "elephant" users contributing too much load to the cluster. As these users are the main sources of congestion, LACS throttles their cache usage and network bandwidth, so as to isolate their negative impacts from others. The reclaimed caches and bandwidth can then be offered to the other users for performance improvement. LACS repeats this procedure until all users have guaranteed performance isolation. Intuitively, this ensures isolation with the *minimum* efficiency loss.

We next elaborate the two phases in detail.

B. Sharing Phase

As an initial attempt, LACS seeks to maximize efficiency by solving cache sharing problem $\mathcal{P}_2$ with optimized cache allocation $\mathbf{A}$ and file placement $\mathbf{P}$. However, jointly optimizing allocation and placement is NP-hard due to the Knapsack constraints [24] in problem $\mathcal{P}_2$. In this regard, we *sequentially* optimize cache allocation and file placement in *two stages*.

Cache allocation In the first stage, LACS determines, for each file i, its cache allocation a_i without considering placement. In particular, we model the caching system as a pool of caches aggregated from all M machines, with total capacity Mc. In this all-in-one resource model, minimizing read latency is equivalent to maximizing the system's cache access rate, i.e.,

$$\text{maximize}_{\mathbf{A}} \quad \textstyle\sum_k \lambda_k^{\text{cache}}, \tag{12}$$

where $\lambda_k^{\text{cache}} = \sum_i \lambda_{ki} a_i$ is the cache access rate of user k.

In the meantime, we also require each user to be *at least no worse off* than having an isolated allocation that is $1/K$ of the total cache capacity. Adding this constraint would significantly increase the chance of meeting the isolation requirement at the end of the sharing phase. Specifically, with isolated cache allocation Mc/K, the best strategy for user k is to use it to cache the top Mc/K frequently accessed files, and the optimal cache access rate the user can expect is

$$\lambda_k^{\text{iso}} = \textstyle\sum_{i=1}^{Mc/K} \lambda_{k[i]}, \tag{13}$$

where $\lambda_{k[i]}$ is the access rate of the i-th popular file of user k. We require $\lambda_k^{\text{cache}} \geq \lambda_k^{\text{iso}}$ for all user k.

To summarize, LACS determines cache allocation by solving the following problem:

$$\begin{aligned} \mathcal{P}_3: \underset{\mathbf{A}}{\text{maximize}} \quad & \textstyle\sum_k \sum_i \lambda_{ki} a_i \\ \text{subject to} \quad & \textstyle\sum_i a_i \leq Mc; \qquad (14) \\ & \textstyle\sum_i \lambda_{ki} a_i \geq \lambda_k^{\text{iso}}, \text{ for all } k. \qquad (15) \end{aligned}$$

As problem $\mathcal{P}_3$ is in essence a *linear programming* problem, it can be efficiently solved. Let $\mathbf{A}^\star$ be the resultant cache allocation obtained from solving problem $\mathcal{P}_3$. Notice that the sharing phase results in higher cache efficiency than the max-min fair allocation. Solving the problem $\mathcal{P}_3$ *maximizes* the overall cache hit ratio under the constraint of isolation guarantee in terms of cache allocation, while max-min fair allocation is only one of the solutions satisfying this constraint.

File placement With cache allocation $\mathbf{A}^\star$, LACS determines, in the second stage, how files should be placed to minimize the average read latency. However, directly fitting file placement $\mathbf{P}$ to the given cache allocation formulates a 0-1 integer programming problem, which is NP-hard. Therefore, we take a slight detour in two steps. In the first step, given allocation $\mathbf{A}^\star$, we optimally distribute the *file access load* to M machines. In the second step, we use this load distribution to guide file placement.

In particular, for each machine j, let Λ_j^{cache} and Λ_j respectively measure its cache load and the total load including both in-memory and on-disk accesses. We rewrite per-request mean latency $\bar{T}$ in terms of Λ_j^{cache} and Λ_j as

$$\bar{T} = \textstyle\sum_j \frac{\Lambda_j}{\sum_l \Lambda_l} \left(\frac{1}{\mu_j - \Lambda_j} + (1 - \frac{\Lambda_j^{\text{cache}}}{\Lambda})\Delta \right).$$

That is, $\bar{T}$ is an average of per-request delay in each machine weighted by the fraction of load it contributes. Here, the two components inside the outer bracket respectively calculate machine j's mean network and I/O delay.

Step-1 We find the optimal load distribution (Λ_j and Λ_j^{cache}) for each machine by solving the following problem:

$$\begin{aligned} \mathcal{P}_4: \underset{\{\Lambda_j, \Lambda_j^{\text{cache}}\}}{\text{minimize}} \quad & \bar{T}, \\ \text{subject to} \quad & 0 \leq \Lambda_j^{\text{cache}} \leq \Lambda_j \leq \mu_j; \qquad (16) \\ & \textstyle\sum_j \Lambda_j = \sum_k \lambda_k; \qquad (17) \\ & \textstyle\sum_j \Lambda_j^{\text{cache}} = \sum_{k,i} \lambda_{ki} a_i^\star. \qquad (18) \end{aligned}$$

Here, constraint (16) is required to prevent overloading (machine load exceeding its bandwidth); constraint (17) requires the load from all users to be fully distributed to machines; constraint (18) requires the total memory access rate matches the caching decisions $\mathbf{A}^\star$ given in the previous step.

Problem $\mathcal{P}_4$ is *convex* as the objective $\bar{T}$ is a convex function of Λ_j and Λ_j^{cache} and all constraints (16)-(18) are linear. Problem $\mathcal{P}_4$ can hence be solved using standard techniques [25].

Step-2 Now that we obtained the optimal load distribution that results in the minimum latency, we place files onto machines in a way such that the actual load approximately follows that optimal distribution. We employ a *best-fit* placement strategy. Each machine j maintains two quotas, one for the total load and the other for the cache load. Their initial values are respectively set to Λ_j and Λ_j^{cache}. We sequentially place files in *descending* order of their popularity (total access rate). For each file, it is first placed onto the machine having the maximum quota to accommodate its cache load. If no machine has sufficient cache load quota, we place the file onto the machine with the maximum quota of the total load. Our intuitions are two-fold. First, popular files deserve a higher caching priority, as they impact the caching performance in a more significant way. Second, such a best-fit placement strategy closely approximates the load distribution given in the previous step. Algorithm 1 details this placement strategy.

Algorithm 1 File Placement in the Sharing Phase

```
procedure GET_PLACEMENT(K, A*)                          ▷ K is the user set
    {Λ_j, Λ_j^cache} ← P_4(K)
    N ← {1, 2, ..., N}                                  ▷ File set
    P ← 0                                               ▷ Initialization
    while N ≠ ∅ do                                      ▷ Best-fit placement
        i ← arg max_i Σ_{k∈K} λ_ki   ▷ The file with the next largest load
        load ← Σ_{k∈K} λ_ki
        memory_load ← load · a_i^⋆
        if max_j Λ_j^cache ≥ memory_load then
            j ← arg max_j Λ_j^cache
            p_ij ← 1
            Λ_j ← Λ_j − load
            Λ_j^cache ← Λ_j^cache − memory_load
        else
            j ← arg max_j Λ_j
            p_ij ← 1
            Λ_j ← Λ_j − load
        N ← N − {i}
    return P
```

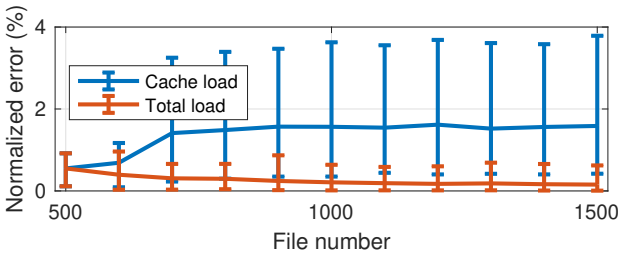


Fig. 3: Average normalized error of the best-fit placement strategy. Error bars depict the 95$^{\text{th}}$ and 5$^{\text{th}}$ percentiles over 50 trials.

To examine the accuracy of Algorithm 1, we have run experiments in a 30-node EC2 cluster (the detailed experiment setup is the same as that described in Sec. VI-C) and measured the error of the algorithm output normalized by the theoretic values. Fig. 3 depicts the results. As the number of files increases from 500 to 1500, the average normalized errors of per-machine total load (Λ_j) and cache load (Λ_j^{cache}) converge to steady values of around 0.2% and 1.6%, respectively. The 95$^{\text{th}}$ percentile errors are less than 1% and 4%, respectively. Therefore, the best-fit placement algorithm produces quite accurate results satisfying the integer constraints.

Checking At the end of the sharing phase, LACS checks if the outcome provides isolation guarantee for all users. If it does, LACS settles on the sharing outcome. Otherwise, it transits to the adjusting phase to enforce isolation.

C. Adjusting Phase

The root cause that isolation guarantee cannot be provided through sharing is the presence of "elephant" users who make aggressive data accesses to the caching cluster. In the adjusting phase, LACS identifies those elephants and throttles their cache and bandwidth usage to prevent them from harming the others.

Hunting elephants We label a user k as an *elephant* if its total file request rate λ_k exceeds what it deserves in an isolated cache, i.e., $1/K$ of the total bandwidth.

Definition 1 (Elephant user)**.** *A user k is said to be an* elephant *if its aggregated access rate is greater than $\frac{1}{K}$ of the total bandwidth, i.e., $\lambda_k > \frac{\sum_j \mu_j}{K}$.*

As elephant users ask for more bandwidth than they deserve in isolated caches, they may cause long network delay, harming the performance of the co-located non-elephant users. To address this problem, we start by throttling the cache usage of elephants and re-allocating the recycled caches to the non-elephants, in the hope that such additional cache allocation would compensate for their prolonged network delay. If this is still not enough, we further throttle the bandwidth allocation for elephant users to enforce isolation.

Throttling cache usage As an initial adjustment, LACS revokes the cache allocation of elephant users and re-distributes those caches to the affected non-elephants as a compensation. Specifically, we rerun the sharing phase, but in the cache allocation stage, we simply exclude those elephant users *as if they were absent*. Consequently, elephant users are not allocated any caches but are only included in the file placement stage. After the sharing phase, LACS checks if such an adjustment is sufficient to provide isolation guarantee. If not, it turns to an even harsher adjustment as follows.

Throttling bandwidth allocation As throttling cache usage alone is incapable of retaining isolation, it suggests that some elephant users must have contributed too much load. As a last resort, we throttle their network bandwidth to what they deserve in isolated caches. To minimize efficiency loss due to isolation (Sec. II-B), LACS incrementally performs isolation, one user at a time. Specifically, it starts to isolate the "biggest" elephant contributing the heaviest load, throttling its bandwidth to $1/K$ of the total bandwidth in each machine. LACS then checks if this is sufficient to provide isolation guarantee to the other users. If not, it moves on to throttle the second "biggest" elephant contributing the second heaviest load. This process repeats until isolation guarantee can be provided.

Algorithm 2 details the entire procedure described above.

D. Analysis

We show in the following theorem that LACS provides guaranteed performance isolation.

Algorithm 2 LACS: Load Aware Cache Sharing

```
1: procedure LACS
2:     K ← {1, 2, ..., K}                                   ▷ User set
3:     A* ← P3(K)                                           ▷ Sharing phase
4:     P* ← Get_Placement(K, A*)
5:     if Provide_Isolation(A*, P*) then
6:         return (A*, P*, ∅)
7:     elephants ← Find_Elephants()                         ▷ Adjusting phase
8:     A* ← P3(K − elephants)        ▷ Revoke cache usage of elephants
9:     P* ← Get_Placement(K, A*)
10:    if Provide_Isolation(A*, P*) then
11:        return (A*, P*, ∅)
12:    isolate_list ← ∅
13:    μ̄_j ← μ_j/K
14:    while not Provide_Isolation(A*, P*) do               ▷ Throttle bandwidth
15:        k ← arg max_k λ_k                  ▷ The next "biggest" elephant
16:        isolate_list ← isolate_list ∪ {k}
17:        μ_j ← μ_j − μ̄_j                                  ▷ Take away its allocation
18:        P* ← Get_Placement(K − isolate_list, A*)
       return (A*, P*, isolate_list)

19: procedure FIND_ELEPHANTS
20:    elephants ← ∅
21:    μ̄ ← Σ_j μ_j/K                                        ▷ Evenly-split bandwidth
22:    for all user k do
23:        rate ← λ_k
24:        if rate > μ̄ then
25:            elephants ← elephants ∪ {k}
26:    return elephants

27: procedure PROVIDE_ISOLATION(A, P)
28:    for all user k do
29:        if T_k(A, P}) > T̄_k then
30:            return False
31:    return True
```

Theorem 1. *With LACS, no user has longer average read latency than it would have in isolated caches.*

Proof: For *elephant* users, they would experience *infinitely* long delays in isolation due to unstable queues. With a total load beyond the allocated bandwidth, each elephant user, if isolated, will have *at least one* cache server where the processing queue cannot remain stable. In expectation, elephant users have infinitely long delays in isolation. LACS *cannot* provide worse performance for them.

For *non-elephant* users, both their I/O and network delays with LACS are no longer than in isolated caches. Even if isolated by LACS, in the worse case, they still have guaranteed minimum cache allocation (as enforced by constraints of $\mathcal{P}_3$) and network resource (reserved bandwidth in isolation). ■

V. Implementation

We have implemented LACS as a pluggable cache manager atop Alluxio [5], [14] in around 6000 lines of codes, which have been open-source for public access [26]. In this section, we describe the key designs of our implementation.

A. Implementation

Alluxio Our implementation is based on Alluxio [5], [14]. There are three key components in Alluxio: master, worker, and client. The Alluxio master manages metadata (e.g., file names and locations) and oversees the status of workers in the cluster. Each worker stores the data objects and periodically

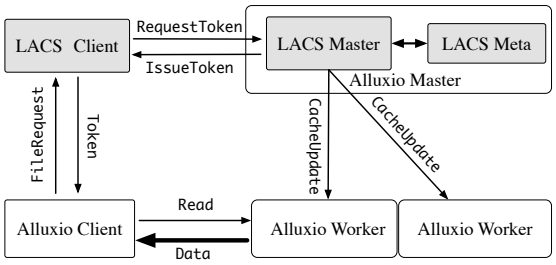


Fig. 4: Architecture overview of LACS in Alluxio. The shaded boxes highlight our implementations.

reports its status to the master. The client exposes common interfaces for users to interact with the master and workers for data accesses, e.g., metadata query and read/write.

LACS overview Fig. 4 shows the architecture overview of LACS atop Alluxio, where our implementations are highlighted in shaded boxes. LACS primarily consists of two components, i.e., `LACS-Master` and `LACS-Client`. The `LACS-Master` implements the main logic of LACS for cache sharing as described in Sec. IV. It also maintains a global view of file access history in the `LACS-Meta`, which is used to track the file access load from users. The `LACS-Client` works as the proxy of the `LACS-Master` on the client side. It forwards file access requests from an Alluxio client to `LACS-Master` who issues a read permission (*token*). The `LACS-Client` then passes that token to the Alluxio client to grant its file access. By controlling the token issuing rate, `LACS-Master` is able to throttle the bandwidth usage of elephant users.

Workflow In our implementation, an Alluxio client is bound with a `LACS-Client`, each registered at `LACS-Master` with a unique LACS ID. This ID is used by `LACS-Master` to differentiate file requests from different users.

A user initiates a file access request through an Alluxio client. The request is sent to the binding `LACS-Client`, who associates the request with its own LACS ID and forwards it to the `LACS-Master`. Upon receiving the request, the `LACS-Master` updates the file access history in `LACS-Meta` and issues a token with which the user can read the requested file from the corresponding Alluxio workers. The `LACS-Master` tallies up the file access frequency within a *learning window* and runs Algorithm 2 *periodically* to enforce performance isolation. Based on the outcome of Algorithm 2, LACS re-arranges the cached objects and file placement across workers. It also updates the list of isolated elephants identified by the algorithm.

Isolation enforcement In case that some elephant users cause too much network congestion, LACS needs to revoke their cache allocation and throttle their bandwidth on each worker. Yet, in case that an isolated user requests a file that has already been cached by others, LACS allows it to access that file directly in memory, as it causes no harm to others.

In contrast, the isolation of bandwidth must be strictly enforced. To this end, LACS employs the *token-bucket algorithm* [27], [28] that controls the rate of issuing file access tokens. In particular, the `LACS-Master` maintains a *token bucket* for each bandwidth-isolated user. The `LACS-Master` deposits tokens to a bucket at a fixed rate equal to the specified bandwidth. A file request consumes the same number of tokens as the file size. If there are enough tokens in the user's bucket, the request returns immediately; otherwise, the request is put on hold until a sufficient number of tokens have been deposited.

B. Discussions

Non-stationary file popularity As users' file access rate may dynamically change over time, LACS periodically updates the cache allocation and file placement to adapt to the changing file popularity. In an update, LACS collects users' file access frequencies within the previous learning window. In our implementation, the update period and window size are respectively set to 1 and 2 hours. The choice of this configuration is based on the observation that in production analytics clusters, file popularities exhibit gradual ascent and decline on an *hourly* basis [19].

Dynamic user arrival and departure In real-life clusters, users may arrive and depart over time. As we expect the frequency of user arrivals/departures to be slower than the changes in users' caching preferences, LACS can readily address it with periodic updating. Specifically, LACS keeps the record of active users and re-configures the cache allocation/placement for users remaining active in each update.

Overhead The main source of overhead in our implementations comes from the computation of the cache allocation and file placement. We employ a python package CVXPY [29] to solve these two problems. Evaluation results in Sec. VI-E confirm that our implementations can configure the cache allocation and file placement within 2 seconds for 30 users accessing 5k files. As LACS only triggers the update once per hour, the overhead is less of a concern.

VI. Evaluations

In this section, we evaluate the performance of LACS via EC2 deployments. The highlights of our evaluations are summarized as follows.

- LACS ensures performance isolation for individual users while achieving high efficiency. In the presence of elephant users, LACS speeds up the per-user average latency by up to 80.4% (25.3% on average) over the state-of-the-art load balancing technique. (Sec. VI-C)
- The performance of LACS is robust against real-world file access patterns. In experiments feeding the Google trace [30], LACS speeds up the per-user average latency by up to 70.9% (31.8% on average) over the state-of-the-art load balancing technique. (Sec. VI-D)
- LACS is efficient in computation. It takes less than two seconds on average to configure cache allocation and file placement for up to 30 users and 5k files. (Sec. VI-E)

A. Experimental Setup

Cluster We deployed LACS in a 31-node Amazon EC2 [13] cluster with 30 cache servers and 1 master node. Each node is

an m5.2xlarge instance with an 8-core 3.1 GHz Intel Xeon processor and 32 GB RAM. We set up another 20 client nodes (also m5.2xlarge instances) which submit read requests to the caching cluster. We measured 4.7 Gbps network bandwidth between the client nodes and cache servers.

Workload We configured each client/user to submit file requests following a Poisson process with exponential submission intervals. To examine the robustness of LACS against real-world request patterns, we also fed the submission traces profiled from production clusters [30] in the experiment (Sec. VI-D). Each request randomly selects a file to read. Unless otherwise specified, we configured the skewed file popularity following a Zipf distribution [18], [31] with an exponent parameter of 1.05 (i.e., high skewness).

Baselines We benchmark LACS against three baseline caching policies.

Max-min allocation: the cache allocation follows the max-min fair scheme; files are placed in a Round-Robin manner, which is the default placement policy of the prevalent caching systems [4], [14].

Selective replication: the top 10% most popular files are replicated into 3 copies to balance the loads [19], [32]. Cache servers employ the *least-frequently-used* (LFU) policy to keep the most popular files in memory.

Isolation: users have evenly-divided and dedicated cache space and network bandwidth on each cache server. This serves as the performance baseline of isolation guarantee.

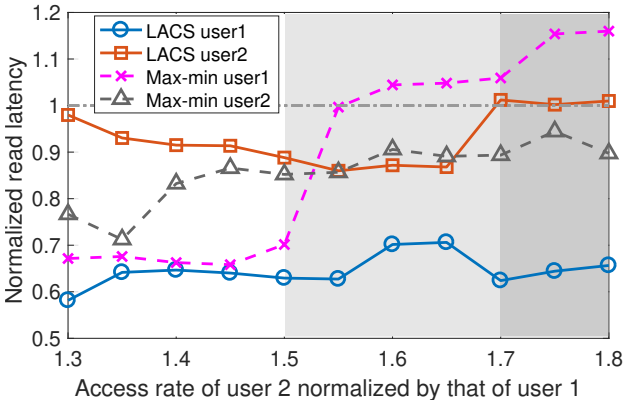


Fig. 5: [Micro-benchmark] Mean read latencies under LACS and max-min allocation. The results are normalized by the user's average delay in isolated allocation.

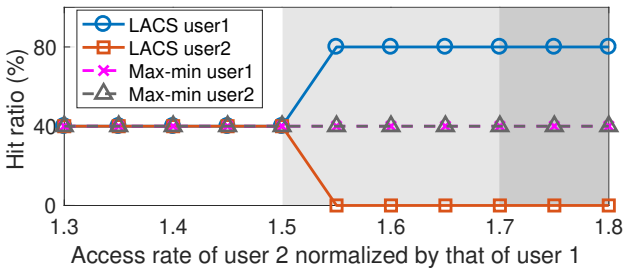


Fig. 6: [Micro-benchmark] Cache hit ratio under LACS and max-min allocation.

B. Micro-Benchmarks

We first examine how LACS enforces isolation guarantee and protects users against elephants with a set of micro-benchmark experiments. We launched two users, user 1 with low file access demands (compared with the cluster capacity) and user 2 with aggressive demands. We set up a LACS cluster with 10 cache servers storing 100 files (100 MB each). The total cache capacity of the cluster is configured as 4 GB. Each user accesses 50 files with *equal* probability, and the accessed files of two users are completely different with *no* overlap.

We gradually increased the access rate of user 2 and measured the average read latency, normalized by that in isolated caches, and cache hit ratio of the two users. Fig. 5 and Fig. 6 show the results under LACS (solid lines) and max-min allocation (dashed lines), respectively. We observed three phases (highlighted with different background colors in the two figures). LACS adjusted its strategies to assure isolation guarantee accordingly.

- When the access rate of user 2 is lower than 1.5x that of user 1, LACS settles on the sharing phase without any throttling. Both LACS and max-min allocation provide fair cache performance around (40% cache hit ratio) for the two users and achieve isolation guarantee (normalized latency < 1).
- If the normalized rate of user 2 is greater than 1.5x, LACS starts to revoke its cache allocation to cache more files for user 1. We observe in Fig. 6 that, under LACS, the hit ratio of user 2 drops to 0 while user 1's cache hit ratio increases to 80%. On the contrary, the max-min allocation is agnostic to the increased load of user 2 but retains the max-min fair cache allocation (equal allocation in this case). Fig. 5 shows that, under max-min allocation, the performance of user 1 is worse than that in isolation when user 2 has a normalized access rate beyond 1.5x. In comparison, LACS enforces isolation guarantee for both users by allocating user 1 more cache space. Notice that, although user 2 has no cache allocation, it is still better off than in isolation because it has much lower network delay in the shared cluster.
- If the load of user 2 continues to grow, after a certain point (> 1.7x normalized rate), the congestion it causes to user 1 could not be compensated by giving up its cache allocation. In this case, LACS has to isolate user 2 to protect user 1. As a result, user 1 still enjoys around 40% performance improvement over isolated caches, while with max-min allocation its read latency ramps up quickly and becomes longer than in isolation.

To wrap up, in the presence of elephant users, LACS consistently enforces isolation guarantee. LACS achieves this by first throttling the cache usage of elephants. If getting cache allocation of elephants fails to compensate for the loss of others, LACS will isolate the elephants as the last resort. On the contrary, although max-min allocation promises fair cache hit ratio for each user, it provides no isolation guarantee in

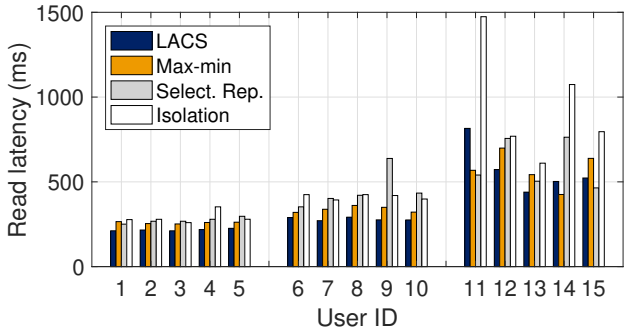


Fig. 7: [Macro-benchmark] Mean read latencies under LACS, max-min allocation, selective replication, and isolated caches with *light* loads.

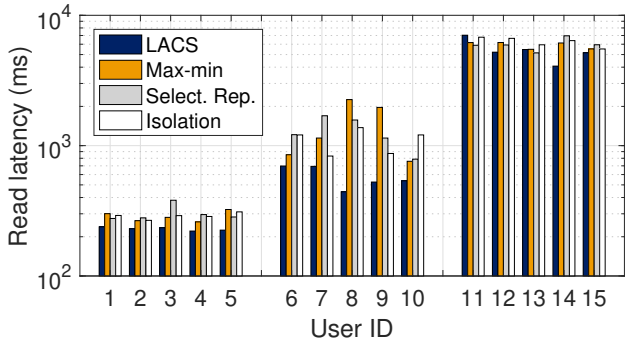


Fig. 8: [Macro-benchmark] Mean read latencies under LACS, max-min allocation, selective replication, and isolated caches in the presence of elephant users (users 11-15).

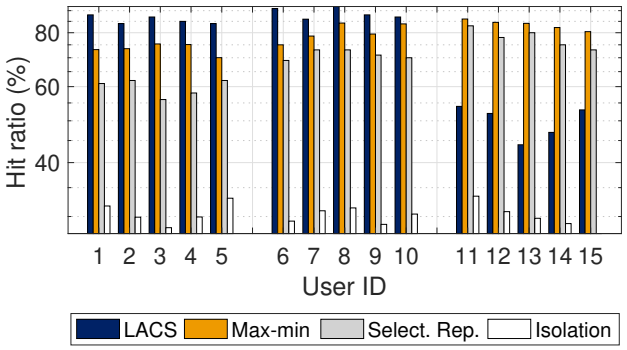


Fig. 9: [Macro-benchmark] Cache hit ratio under LACS, max-min allocation, selective replication, and isolated caches in the presence of elephant users (users 11-15).

terms of the I/O performance (read latencies).

C. Macro-Benchmarks

Next, we benchmark the performance of LACS in a broader range of settings. We scaled up the caching cluster with 30 servers and configured three groups of users with *low*, *medium*, and *high* access rates, respectively, to simulate diverse application I/O demands. Each group consists of 5 users. There are 500 files (100 MB each) stored in the cluster. Each cache server is configured to have 1 GB memory (30 GB in total) and 100 GB disk (3 TB in total). We consider two cases, *light-load* and *heavy-load*, in which we evaluated LACS against the three baselines solution, i.e., max-min allocation, selective replication, and isolated caches.

Light load We start with a light-load case where users in the three groups respectively make 4 (users 1-5), 6 (users 6-10), and 9 (users 11-15) requests per second. As there is no elephant user, LACS settles on the *sharing phase* without further adjustment. We measured the mean read latencies of requests for each user. Fig. 7 shows the results. In this scenario, we observe that LACS and max-min allocation achieve performance isolation (i.e., shorter average latency than in isolation for all users). LACS slightly outperforms max-min allocation, as it strives to optimize the overall latency in the sharing phase. We also note that selective replication favors the users with high demands, e.g., users 11 and 15, by caching more of their files and replicas in memory. This leads to less cache allocation for other users and hence the failure of achieving isolation guarantee for some of them, e.g., users 9 and 10 in this experiment.

Heavy load Next, we increased the access rate of users in the three groups to 4, 10, and 23, respectively. High-demand users (users 11-15) now become elephants, as their heavy load starts to cause congestions. We expect that load-agnostic schemes like max-min allocation and selective replication are *unable* to restrain the harmful impacts of elephants on others. This is confirmed in Fig. 8. With max-min allocation and selective replication, some users (e.g., users 3, 7, 8, and 9) suffer from much longer delays than they would have in isolation.

In comparison, LACS guarantees isolation. It first revokes the cache allocation of elephants and offers it to the other users, as illustrated by the improved cache hit ratio of low-demand users in Fig. 9. Unfortunately, simply having compensation on cache allocation failed to retain isolation for them. LACS hence further throttles the bandwidth of an elephant (user 11) to mitigate network congestion. We observe in Fig. 8 that this leads to performance isolation for others. We note that user 11 has a slightly longer delay than in isolation as LACS also penalizes its cache performance. While this seems to violate performance isolation, in the long run, the impact of cache performance will become even more marginal as user 11 is bottlenecked on the network delay. Except for the throttled user 11, LACS provides the shortest average delays for every user[4], achieving up to 80.4% (25.3% on average) and 71.9% (27.7 on average) improvement over max-min allocation and selective replication, respectively.

Fig. 9 compares the cache hit ratios under different policies. All the sharing policies, e.g., LACS, max-min allocation, and selective replication, greatly improve the cache utilization over the isolated caches. Low- and medium- demand users have worse cache performance under selective replication than with LACS or max-min allocation, as selective replication favors high-demand users by caching and replicating their files in memory. On the contrary, high-demand users are penalized by LACS. Their cache allocations are revoked to compensate for the congestion they cause. Nonetheless, their cache hit ratios

[4]Notice that other elephant users also benefit from throttling user 11 due to less network congestion.

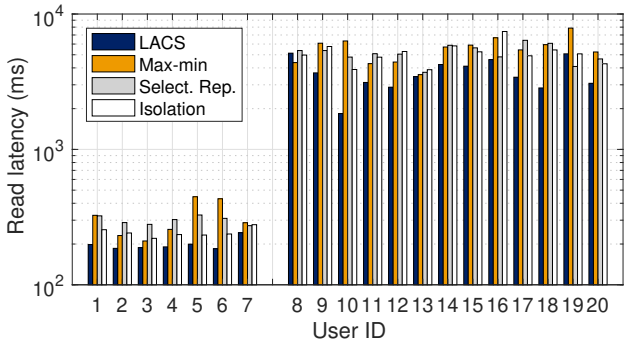


Fig. 10: [Google Trace] Mean read latencies under LACS, max-min allocation, selective replication, and isolated caches with access patterns profiled in the Google trace [30].

are not zero as they may access the files cached by other users.

D. Robustness against Real-World File Access Patterns

So far, we have assumed Poisson arrivals of file requests. Next, we measure how LACS performs against real-world request arrivals. To this end, we profiled the submission intervals from a Google trace [30] containing over 2000 jobs in a 12k-node cluster. We extracted two typical workloads [30], [33] in the trace, i.e., *online* and *offline* jobs, and emulated their file request patterns in our experiments with LACS. Online jobs include business-critical interactive queries which may have burst I/O demands (submission intervals in microseconds), while offline jobs (e.g., collecting and analyzing logs) only run periodically at much lower data access frequency.

For we configured 7 and 13 users emulating offline and online jobs in production clusters, respectively. Users submit file requests following the corresponding distribution profiled in the trace. Fig. 10 compares the read latencies under LACS, max-min allocation, selective replication, and isolated caches. With the configured cluster capacity, the 13 online jobs are recognized as elephants by LACS, of which users 8 and 19 are isolated to provide performance isolation for offline jobs. Fig. 10 shows that only LACS achieves isolation guarantee. Max-min allocation and selective replication fail to protect offline jobs as they are agnostic to the heavy loads contributed by online jobs. LACS results in the shortest latency for all users other than the isolated users. The maximum (average) improvements over max-min allocation and selective replication are 70.9% (31.8%) and 61.7% (29.5%), respectively.

E. Overhead

Finally, we quantify the computational overhead of LACS. We measured how long it takes for LACS to compute the cache allocation and file placement (Algorithm 2) with 30 users. Fig. 11 shows the boxplot of the results in 100 trials. The average runtime is less than 2 seconds for up to 5k files. As LACS re-configures the cache allocations once an hour, its overhead can be amortized and is less of a concern.

VII. Related Works

Fair cache sharing As cache sharing becomes prevalent in public datacenters, many recent works [8]–[10] focus on achieving fair cache allocation among multiple users. However, to our knowledge, all these works assume an *all-in-one* cache model and remain *agnostic* to the data placement across cache servers. We have shown in this paper that sharing policies built on such a simplified model provide no isolation.

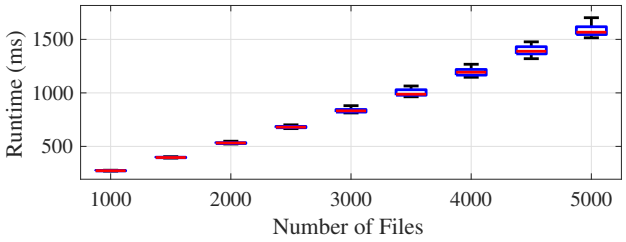


Fig. 11: [Overhead] The time for LACS to compute the caching vector $\mathbf{A}$ and file placement matrix $\mathbf{P}$. Boxes depict the 25^{th}, 50^{th}, and 75^{th} percentiles. Whiskers depict the 5^{th} and 95^{th} percentiles.

Load balancing Load balancing algorithms [18], [19], [34], [35] for cluster caching have been developed to mitigate hot spots for improved caching performance. A common approach in this regard is to employ consistent hashing [34], [35] to optimize the mapping function from files to servers (file placement). However, these solutions aim to balance the *aggregated* load of cache servers, without concerning the performance of individual users. File replication [19], [32] is another popular strategy for load balanced caching. We have shown in the evaluation section that replication schemes cannot provide isolation guarantee, either, as they remain agnostic to the impact of loads contributed by heavy-demand users.

Multi-resource allocation Multi-resource fair allocations, notably Dominant Resource Fairness (DRF) [22], strive to provide desirable properties of fairness when allocating multiple types of resources. In particular, DRF bundles resources of different types, including CPU cycles, link bandwidth, and JVM memory, with a fixed ratio specified by users' demands. However, DRF and its variants cannot be applied to cache sharing as memory caches are *non-exclusively* sharable among multiple tenants, while DRF-like policies assume resources can only be *exclusively* allocated.

VIII. Conclusion

In this paper, we have motivated the need of achieving performance isolation in cluster caches. We have demonstrated, through empirical studies, that existing cache sharing policies either violate isolation or result in low cache utilization. We have proposed LACS, a *load-aware* cache sharing scheme, to enforce isolation between users while attaining high cache efficiency. LACS identifies users with aggressive demands and throttles their cache and bandwidth usage, so as to refrain them from harming other users. We have implemented LACS in Alluxio. EC2 deployments showed that LACS provides guaranteed performance isolation with substantially shorter read latency than state-of-the-art load balancing techniques.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedback. This research was partly supported by the Hong Kong Research Grants Council under Grant No. 26213818.

REFERENCES

[1] 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.
[2] Druid, "http://druid.io."
[3] Presto, "https://prestodb.io/."
[4] Memcached, "https://memcached.org."
[5] 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.
[6] Redis, "http://redis.io."
[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *Proc. USENIX OSDI*, 2016.
[8] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, "FairRide: near-optimal, fair cache sharing," in *Proc. USENIX NSDI*, 2016.
[9] M. Kunjir, B. Fain, K. Munagala, and S. Babu, "Robus: Fair cache allocation for data-parallel workloads," in *Proc. ACM SIGMOD*, 2017.
[10] Y. Yu, W. Wang, J. Zhang, Q. Weng, and K. B. Letaief, "OpuS: Fair and efficient cache sharing for in-memory data analytics," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2018.
[11] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit Round-Robin," *IEEE Trans. Netw.*, vol. 4, no. 3, pp. 375–385, 1996.
[12] Alluxio Configuration, "https://goo.gl/ZgUGe4."
[13] Amazon Elastic Compute Cloud, "https://aws.amazon.com/ec2/," 2016.
[14] Alluxio, "http://www.alluxio.org/."
[15] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proc. USENIX NSDI*, 2015.
[16] Amazon S3, "https://aws.amazon.com/s3."
[17] Big Data, In-Memory, and Object Storage, "https://goo.gl/z5EJ1C."
[18] 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.
[19] 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.
[20] J. Jaffe, "Bottleneck flow control," *IEEE Trans. Commun.*, vol. 29, no. 7, pp. 954–962, 1981.
[21] L. Massoulié and J. Roberts, "Bandwidth sharing: objectives and algorithms," in *Proc. IEEE INFOCOM*, 1999.
[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX NSDI*, 2011.
[23] L. Kleinrock, *Queueing systems, volume 2: Computer applications*. wiley New York, 1976, vol. 66.
[24] J. Lee, V. S. Mirrokni, V. Nagarajan, and M. Sviridenko, "Non-monotone submodular maximization under matroid and knapsack constraints," in *Proc. ACM Symp. Theory of Comput.*, 2009, pp. 323–332.
[25] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.
[26] LACS Source Code, "https://github.com/yhust/lacs.git."
[27] J. Wroclawski, "Specification of the controlled-load network element service," Tech. Rep., 1997.
[28] S. Sahu, P. Nain, C. Diot, V. Firoiu, and D. Towsley, "On achievable service differentiation with token bucket marking for TCP," in *Proc. ACM SIGMETRICS*, vol. 28, no. 1, 2000, pp. 23–33.
[29] CVXPY, "http://www.cvxpy.org/."
[30] 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.
[31] 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.
[32] J. D. Cook, R. Primmer, and A. de Kwant, "Compare cost and performance of replication and erasure coding," *hitachi Review*, vol. 63, p. 304, 2014.
[33] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones." in *Proc. NSDI*, vol. 13, 2013, pp. 185–198.
[34] 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.
[35] J. Hwang and T. Wood, "Adaptive performance-aware distributed memory caching." in *Proc. USENIX ICAC*, 2013.