OpuS: Fair and Efficient Cache Sharing for In-Memory Data Analytics

Yinghao Yu[†], Wei Wang[†], Jun Zhang[†],Qizhen Weng[†], Khaled B. Letaief[†‡]
[†]Hong Kong University of Science and Technology
[‡]Hamad bin Khalifa University, Doha, Qatar
[†]{yyuau, weiwa, eejzhang, qwengaa, eekhaled}@ust.hk [‡]kletaief@hbku.edu.qa

***Abstract*—We study the fair cache allocation problem in shared cloud environments, where many users and applications contend for the main memory to cache *shared* datasets or files. Unlike other resources such as CPUs and networks, in-memory caches can be *non-exclusively* shared across many users, e.g., a cached columnar dataset queried by many Spark SQL jobs. This results in a unique challenge of the "*free-riding*" problem, where a user lies about its caching preferences to trick other users to cache files for it, using their allocated cache space. We show that existing cache allocation policies either suffer from such manipulations or result in poor efficiency. To address this problem, we propose a new cache allocation algorithm, termed OpuS, or *Opportunistic Sharing* for high efficiency. We show that OpuS provides *performance isolation* between users and is *strategy-proof* against "free-riding" manipulations. We have implemented OpuS as a pluggable cache manager in Alluxio, a popular memory-centric filesystem. Cluster deployment and trace-driven simulations demonstrate that OpuS allocates each user a fair share of caches while achieving *near-optimal* efficiency in cache utilization.**

I. Introduction

Today's data analytics clusters are shifting towards *in-memory* computations, ranging from big data analytics [1]–[3] to distributed machine learning [4]–[6] and key-value stores [2], [7], [8]. By caching input and intermediate datasets in memory, users can gain orders of magnitude improvement for I/O-intensive jobs. With a surging volume of data moving into the main memory, users compete for the limited in-memory caches in shared, multi-tenant clouds. To ensure both performance isolation for individual users and high utilization of the system, there is a pressing need to enforce *fair cache allocation*, with the following three desirable properties as summarized in [9]:

1) ***Isolation guarantee:*** by sharing the caches, each user should get *no fewer* files in memory than it would have had with an *isolated* and *evenly partitioned* cache.
2) ***Strategy-proofness:*** a user *cannot* have more files in memory at the expense of another by *lying* about its demand of caching a file.
3) ***Pareto efficiency:*** it is *not* possible to cache more files for a user *without* evicting files of another.

Unfortunately, prevalent cache management policies, e.g., both recency- and frequency-based algorithms (notably LRU and LFU), aim to optimize the *global* cache hit ratio, without any isolation guarantee. Consequently, users accessing files at long intervals or low frequencies tend to get their files evicted from the memory.

On the other hand, simply maintaining an *isolated cache* for each user, while guaranteeing isolation, is highly inefficient. It is common to have multiple users or jobs requesting the same dataset in shared cloud environments. For example, a columnar dataset stored as Parquet files [10] is usually queried by many Spark SQL jobs from different users. In fact, it has been observed in a production HDFS cluster that over 30% of files were shared by at least two users [9]. Providing isolated caches leads to multiple copies of *shared* files, which not only wastes memory space but also results in a significant drop in the write performance due to frequent in-memory replications.

Moreover, the special characteristic that cached files can be *non-exclusively shared* across multiple users poses a unique "*free-riding*" problem [9]. That is, instead of using its own allocated cache to persist files it requests, a strategic user seeks to "free-ride" on others who have a strong need to cache the same files, and thus can save its cache space at the expense of others. The traditional *max-min fair* allocation is subject to exactly this manipulation [9]. In fact, Pu et al. [9] established a negative result that *no* allocation can achieve all the three properties listed above due to free-riding. The state-of-the-art solution goes to FairRide [9], which seeks to block free-riders, and provides service isolation with the minimum efficiency loss. However, as we shall show in Sec. III-D, FairRide fails to prevent free-riding, as strategic users can still game the system to improve their cache performance at the expense of others.

In this paper, we revisit the fundamental problem of fair and efficient cache allocation in shared cloud environments. We propose a new cache allocation algorithm, termed OpuS (**Opp**ort**u**nistic **S**haring for high efficiency). We show that OpuS is capable of 1) providing isolation guarantee, 2) eliminating the incentive of free-riding, and 3) attaining near-optimal efficiency in cache utilization. OpuS achieves these properties through a *two-stage* allocation algorithm built on the classic Vickrey-Clarke-Groves (VCG) mechanism [11]–[13]. In the first stage, OpuS strives to share caches for high efficiency by framing *proportional fairness* into the VCG mechanism. If the resultant allocation provides isolation guarantee, OpuS settles on sharing. Otherwise, it resorts to *isolated allocation* in the second stage, in which each user is allocated an isolated cache of equal size.

We have implemented OpuS as a pluggable cache manager in Alluxio [14], a popular in-memory filesystem for big data analytics. We evaluated the performance of OpuS through both cluster deployment in Amazon EC2 and trace-driven simulations at large scale. Our EC2 deployment confirms that OpuS eliminates the incentive of free-riding manipulations. In terms of efficiency measured by the cache hit ratio, OpuS outperforms FairRide by 16.6% and comes within 7% of the global optimum. We also evaluated the computational overhead of our implementation in a wide range of settings. Evaluation results show that OpuS is capable of making cache allocations in a few seconds with up to 150 users sharing the cache in multi-tenant clouds.

II. System Model and Design Objective

In this section, we describe our model for cache allocation in shared cloud environments. We then formalize the three desirable properties for cache allocation outlined in the previous section.

A. Background and Model

Background: Given the huge performance advantage of in-memory computations, cloud users aggressively cache their frequently accessed datasets or *files* (e.g., RDDs [1], HDFS and Parquet [10] files) in RAM. A file, if requested by multiple users, can be non-exclusively shared by all of them. For example, a MapReduce job performing data extraction usually caches its output for SQL queries submitted by other users. A machine learning job dynamically predicting the ad click-through rate caches the trained model in parameter servers [5], [6], which is simultaneously used by multiple business-critical jobs for ad recommendation.

Caching Request: We assume that there are N users sharing M files in a cluster. The *cache preference* of user i for file F_j is denoted by $p_{i,j}$, which can be either *explicitly* reported by the user through an API or *implicitly* inferred from its access history, e.g., the access frequency. We normalize each user's caching preference such that $\sum_{j=1}^{M} p_{i,j} = 1$. Specifically, if user i has no interest in caching a file F_j, its caching preference of that file is set to zero, i.e., $p_{i,j} = 0$. Preference normalization eliminates the discrimination against users with relatively low caching demands (e.g., low file access rate), improving fairness.

We model the file caching requests by a *weighted bipartite graph* consisting of the *user vertices* and the *file vertices*. In the graph, a user vertex i and a file vertex F_j is connected through an edge if user i requests to cache file F_j, and the caching preference $p_{i,j}$ is associated with the edge as the *weight*. Fig. 1 illustrates an example, where user A requests to cache two files F_1 and F_2, and user B requests to cache F_2 and F_3.

Cache Allocation: Without loss of generality, we assume that files are of *unit* size[1] and can be cached *in fractions* (e.g., caching only a few blocks of the file). Given the file caching requests of users, the cache allocation algorithm determines *which* files are cached and *how much* memory is allocated to each file. Formally, for file F_j, the algorithm computes its cache allocation a_j, where $0 \leq a_j \leq 1$. The constraint is that caches should *not* be overallocated, i.e., $\sum_{j=1}^{M} a_j \leq C$, where C is the total capacity of the available cluster caches.

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

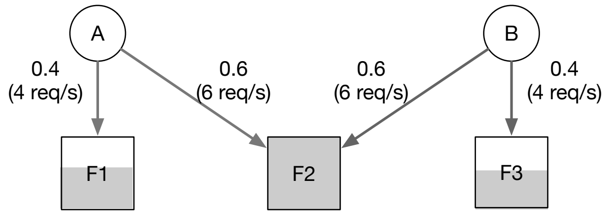


Fig. 1: An example of the caching request graph where two users A and B share three files F_1, F_2, and F_3 of unit size. Users' cache preferences are shown as the weights associated with the edges. The numbers in brackets denote the historical access frequencies. The cluster has two units of caches in total, which are allocated to cache the entire F_2 and half of F_1 and F_3 (highlighted as shaded areas).

We stress that there is *no* need to differentiate how cache allocation a_j is distributed to different users, simply because a cached file can be *non-exclusively* shared across them. Referring back to the example of Fig. 1, we assume that there are 2 units of caches, which are allocated to persist file F_2 *as a whole* and files F_1 and F_3 *in half*. Since F_2 is cached for the two users, it is accessible to both of them.

User Utility: Once cache allocation $\mathbf{a} = \langle a_1, \ldots, a_M \rangle$ has been determined, we measure the cache performance of a user by its *utility*, defined as the sum of allocations weighted by its caching preferences, i.e.,

$$U_i(\mathbf{a}) = \sum_{j=1}^{M} a_j p_{i,j}. \tag{1}$$

Specifically, if the caching preference of a file is sorted out based on its access frequency, the utility function (1) measures exactly the *expected cache hit ratio*. In the example of Fig. 1, both users have the same utility: $0.4 \times \frac{1}{2} + 0.6 = 0.8$.

B. Desirable Properties

We motivate three properties for cache allocation: isolation guarantee, strategy-proofness, and Pareto efficiency. These properties are not only desired for cache allocation [9], but also required for allocating other resources such as CPUs and networks in multi-tenant clouds [15].

*1) **Isolation Guarantee**:* In a nutshell, isolation guarantee ensures each user to cache *no fewer* files in memory than it would have had with an *isolated* cache of *equal* size.

Definition 1 (Isolation guarantee)**.** *Let $\bar{U}_i$ be the utility of user i if the entire cache space is* equally *divided among N users, each receiving an* isolated *cache of size $\frac{C}{N}$. An allocation $\mathbf{a}$ is said to provide* isolation guarantee *if each user i receives a utility no less than that of isolation, i.e., $U_i(\mathbf{a}) \geq \bar{U}_i$.*

We refer to Fig. 1 and show that the depicted allocation provides isolation guarantee. By Eq. (1), both users gain the same utility 0.8 from cache sharing in Fig. 1. We now consider the case of isolation, where the entire caches are evenly split into two isolated ones for two users, each receiving a unit size. Since both users prefer file F_2 the most, they would cache it *separately*. Each user gains a lower utility 0.6 and thus is better off sharing than isolation.

*2) **Strategy-proofness**:* Multi-tenant systems are prone to *harmful manipulations* [9], [16]. We shall show later that a manipulator may game the system to "free-ride" on other users by faking its caching preferences, e.g., making spurious accesses if the cache preferences are inferred from historical access frequency. We hence require *strategy-proofness* to eliminate the incentive of such manipulations.

Definition 2 (Strategy-proofness)**.** *No user can fake its caching preference to gain more utility* at the expense of others. *Specifically, for user i, let $\mathbf{a}$ be the cache allocation when it is truthful, and $\mathbf{a}'$ be the allocation when it is not. We have $U_i(\mathbf{a}') > U_i(\mathbf{a})$* only if *$U_k(\mathbf{a}') \geq U_k(\mathbf{a})$ for all user k. Meaning, cheating can only benefit other users.*

*3) **Pareto Efficiency**:* Cloud operators strive to achieve high utilization of in-memory caches, which promotes the property of Pareto efficiency.

Definition 3 (Pareto efficiency)**.** *It is* not *possible to improve a user's utility without decreasing that of another.*

We shall use the three properties above as our guidelines to develop a fair cache allocation algorithm. Ideally, our objective is to achieve all the three properties at the same time.

III. Analysis of Existing Solutions

In this section, we analyze existing cache management policies, including recency- and frequency-based cache replacement policies, isolated caches, max-min fair allocations, and the state-of-the-art solution FairRide [9]. We show that these policies either result in poor cache utilization or suffer from strategic manipulations.

A. Recency- and Frequency-based Cache Replacement

Recency- and frequency-based cache replacement algorithms, notably LRU (least-recently-used) and LFU (least-frequently-used), serve as the *de facto* cache management policies in today's data analytics clouds, e.g., [1]–[3], [7], [8]. These algorithms aim to maximize the global cache hit ratio but are *incapable* of providing isolation among users. Users accessing files at long intervals or low frequencies may gain little performance improvement in shared caches, as their requested files are likely to be evicted out of the memory. Worse still, recency- and frequency-based algorithms promote cheating. By *spuriously* accessing their preferred files, strategic users can force the system to evict files requested by others, grabbing more cache space from the well-behaved users [9].

B. Isolated Caches

Maintaining an isolated cache for each user is another common approach in shared cloud environments. By *evenly* dividing in-memory caches into *dedicated* partitions for individual users, we can trivially achieve both isolation guarantee and strategy-proofness. However, the price we paid is a significant utilization loss in two aspects. First, multiple copies of a shared file will be cached *separately* for different users, wasting not only the memory space, but also the write bandwidth. Second, users with low memory footprint may not fully utilize their allocated caches, resulting in even more utilization loss.

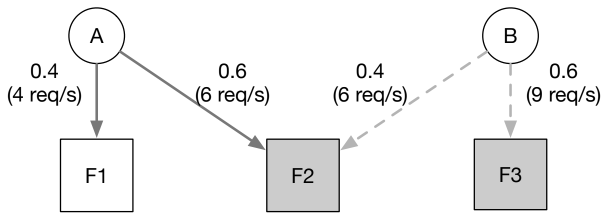


Fig. 2: Illustration of the free-riding manipulation under max-min fair allocation in Fig. 1. User B lies about its caching preferences (shown as the dotted lines) by spuriously accessing file F_3. This allows user B to cache the entire F_3 while free-riding on file F_2 cached solely by user A (highlighted as shaded area).

C. Max-min Fair Allocations

The popular max-min fair allocation seeks to maximize the minimum allocation across users. It is essentially a *market-based* allocation where users are given budget to purchase or trade resources in a *perfectly competitive* market. Max-min fairness and its variants have achieved a wide success for allocating other resources, such as CPU cycles and link bandwidth, in shared, multi-tenant environments [17]–[19]. These resources, once allocated to a user, *cannot* be accessed by another. In contrast, memory caches distinguish themselves as being *non-exclusively sharable* across multiple users at the same time. This gives rise to a new problem of "free-riding," due to which directly applying max-min fairness to cache allocation is *not* strategy-proof.

Max-min Fair Cache Allocation: To converge to a max-min fair allocation, each of the N users is given an equal budget of $\frac{C}{N}$ cache units and uses it to cache its desired files. If multiple users want to cache the same file, they *evenly* share the cost of caching. Meaning, each user is charged $\frac{1}{n}$ units, where n is the number of users paying to cache.

We show that the allocation in Fig. 1 is max-min fair. Initially, each user is given an even budget of one unit for caching. Since file F_2 is the most desired for the two users, it is cached for both users each contributing 0.5 units of budget. Each user has a remaining budget of 0.5 units and uses it to cache the next desired file (file F_1 for user A and F_3 for user B). But this time, it shares the cost with nobody as it is the only one who wants to cache that file. Therefore, each user uses up its remaining budget and caches *half of* its next desired file, as shown in Fig. 1.

Free-riding: Unfortunately, the max-min fair allocation in the previous example is *not* strategy-proof. To see this, we assume that user B is strategic and claims that it prefers file F_3 to F_2. As shown in Fig. 2, assuming the cache preference is inferred from the access frequency, user B can make spurious accesses to file F_3, so as to manipulate its preference over F_2. Such a manipulation allows user B to use all its budget to persist file F_3 as a whole, forcing user A to cache its most desired file

F_2 *at its own expense*. User B can now "free-ride" on user A by sharing its cached file F_2 without paying for it. In the end, user B gains a higher utility by having more files in memory at the expense of user A.[2]

D. FairRide

FairRide Algorithm: In order to eliminate the incentive of cheating under max-min fair allocations, Pu et al. [9] proposed FairRide to penalize free riders. Specifically, FairRide identifies a user as a *free rider* of a cached file if the user wants to access that file without sharing the cost of caching. FairRide *randomly blocks* a free rider from accessing a cached file *in memory*, with probability $\frac{1}{n+1}$, where n is the number of users who pay to cache the file. A free rider, once blocked from in-memory access, has to access the file from a stable storage, *as if* it were a cache miss.[3]

We refer back to the previous example of Fig. 2. Since user B has contributed nothing to cache file F_2, it acts as a free rider, and its in-memory access to file F_2 will get blocked by FairRide with probability $\frac{1}{1+1} = \frac{1}{2}$, which decreases its utility (hit rate) to $0.6 \times \frac{1}{2} + 0.4 \times 1 = 0.7$. This is *lower* than what user B would have gained by staying truthful in Fig. 1, where it pays to cache both F_2 and F_3 and can freely access the two files without getting blocked. User B hence has no incentive to lie about its caching preference.

According to [9], by blocking free-riding with probability $\frac{1}{n+1}$, FairRide *aligns* a user's *benefit-cost ratio* with its caching preference, which is the key to inducing truthful behaviors. In general, suppose that user i is interested in caching file F_j together with the other n users, which would cost user i a budget of $\frac{1}{n+1}$ units. In return, user i is entitled to a full access to the cached file without getting blocked with probability $\frac{1}{n+1}$. Therefore, the utility gained through caching F_j is $\frac{1}{n+1}p_{i,j}$, and the benefit-cost ratio is $p_{i,j}$, which is exactly the caching preference. Note that with a given a budget, the optimal strategy of a user is to cache files in a *descending* order of the benefit-cost ratio. This implies that files are cached following their preferences, and users have no incentive to cheat.

Cheating in FairRide: However, to our surprise, the argument above does *not* always hold. We show through a counterexample that FairRide's probabilistic blocking of free-riding is *incapable* of eliminating strategic manipulations. In particular, as in Fig. 3, we assume that four users request to cache three files in a shared cluster with two units of in-memory caches. With FairRide, each user is given an equal budget of 0.5 units to cache its desired files. Fig. 3a illustrates their caching preferences.

We start with the case where each user behaves *truthfully*. We focus on user B, who seeks to cache its most desired file F_2 at the beginning. Since F_2 is also the top choice of users C and D, the three equally share its cost of caching, each contributing $\frac{1}{3}$ units. User B has $\frac{1}{6}$ units spared and uses it to cache its next desired file F_1, together with user A who has not spent its budget so far. The two users cache $\frac{1}{2} + \frac{1}{6} = \frac{2}{3}$ units of F_1, of which the first $\frac{1}{3}$ units are *shared* by both users, each contributing $\frac{1}{6}$. The second $\frac{1}{3}$ units are cached *solely* for user A, as user B has run out of budget. Therefore, user B has a full access to the first $\frac{1}{3}$ share it owns, but gets blocked with probability $\frac{1}{1+1} = \frac{1}{2}$ when accessing the second $\frac{1}{3}$ share. Adding to the fully cached file F_2, user B gains utility $0.45 \times (\frac{1}{3} + \frac{1}{3} \times \frac{1}{2}) + 0.55 = 0.78$.

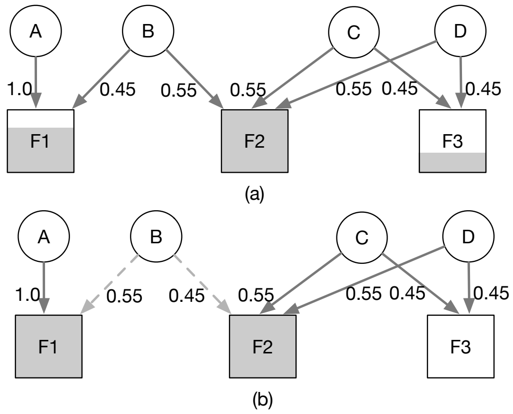


Fig. 3: An example showing that FairRide is not strategy-proof. There are two units of caches in total, and each user is allocated a budget of 0.5 units. (a) All users are truthful. (b) User B has more files cached by misreporting its caching preference.

We next show that user B can lie about its caching preference to gain a higher utility at the expense of users C and D. As shown in Fig. 3b, suppose that user B claims that it prefers file F_1 to file F_2 and uses all its budget to cache the *entire* F_1, together with user A. This forces users C and D to go *all-in* to persist file F_2. In this case, user B has a full access to file F_1, but is treated as a free rider of file F_2 subject to random blocking with probability $\frac{1}{2+1} = \frac{1}{3}$. User B gains utility $0.45 + 0.55 \times (1 - \frac{1}{3}) = 0.82$, which is higher than it would have gained by telling the truth. User B will hence report its cache preferences as in Fig. 3b instead of behaving truthfully to game for higher utility.

We stress that FairRide is not strategy-proof, regardless of whether the cache preferences are specified by users or inferred from the underlying file access frequencies. In fact, as we shown in Sec. III-C, if the preferences are determined by the actual file request rates, users can easily manipulate their caching demands by making spurious file accesses.

E. Negative Results for Cache Allocation

The violation of strategy-proofness illustrated by the counterexample above raises a natural question: can we come up with some new blocking probabilities to eliminate the incentive of cheating in FairRide? Unfortunately, we show through the following theorem that this is not possible.

Theorem 1 (Impossibility)**.** *No blocking probability $f(n)$ can achieve strategy-proofness for FairRide, where n is the number*

[2]Note that user B has many cheating options to game the system: as long as it claims that it prefers file F_3 to F_2, the outcome would remain unchanged.

[3]The implementation of FairRide does not enforce on-disk reading but artificially delays in-memory access to emulate the blocking effects.

TABLE I: Properties of existing cache allocation policies and our solution OpuS: isolation guarantee (IG), strategy-proofness (SP), and Pareto efficiency (PE).

	IG	SP	PE
Recency/Frequency			✓
Isolated Cache	✓	✓	
Max-min Fairness	✓		✓
FairRide	✓		Near-optimal
OpuS	✓	✓	Near-optimal

of users who pay to cache a file, and $f(\cdot)$ is an arbitrary function whose value falls into the range between 0 and 1.

Proof: It was proved in [9] that to achieve strategy-proofness through probabilistic blocking, each user's benefit-cost ratio must align with its caching preference. It was also proved that $\frac{1}{n+1}$ is the *only* candidate probability that *may* serve this purpose [9, Sec. 5.2]. We now refer to the previous counterexample in Fig. 3 and show that even with this blocking probability, user B can attain a higher benefit-cost ratio of caching file F_1 than its preference. Indeed, if user B chooses not to cache file F_1, half of F_1 will be cached by user A, to which the access of user B gets blocked with probability $\frac{1}{1+1} = \frac{1}{2}$. User B hence gains utility $0.45 \times \frac{1}{4} = 0.11$ from the half cached F_1. Now suppose that user B changes its mind and invests all its budget on F_1 together with user A. This time, the file is *fully* cached for both users, from which user B gains utility 0.45. Therefore, the benefit-cost ratio is $\frac{0.45-0.11}{0.5} = 0.68$, higher than user B's caching preference 0.45. ■

The negative result established by Theorem 1 mandates a ground-up redesign of fair cache allocation, which we shall discuss in the next section. Table I summarizes the properties of existing cache allocation policies and our solution. We note that none of them retains all the three desirable properties. This is no accident, but a direct consequence of the following SIP theorem proved by Pu et al. [9].[4]

Theorem 2 (SIP [9, Sec. 5.1])**.** *No cache allocation algorithm can satisfy strategy-proofness (S), isolation guarantee (I), and Pareto efficiency (P) at the same time.*

The incompatibility of the three properties is *uniquely* found for cache allocation in that caches are non-exclusively shared across multiple users. Following [9], we require our solution to satisfy both isolation guarantee and strategy-proofness, with minimum efficiency loss.

IV. Opportunistic Sharing for High Efficiency

In this section, we present our solution for fair cache allocation, which we call OpuS, or **Op**port**u**nistic **S**haring for high efficiency. We show that OpuS provides isolation guarantee and is immune to harmful manipulations. The price paid is a slight efficiency loss.

[4]While the strategy-proofness claim of FairRide made in [9] is incorrect (Sec. III-D), the SIP theorem stated in the same paper [9] does hold. The proof of SIP theorem is independent of the (erroneous) strategy-proofness analysis.

A. Overview of OpuS

Among all allocation alternatives, having isolated caches of equal size for individual users serves as a *baseline*, under which users are guaranteed with isolation and have no incentive to lie about their caching preferences. However, this baseline falls short of achieving high utilization, as users are forced to maintain separate copies of a shared file. In fact, any allocation with isolation guarantee achieves *no lower* utilization. This motivates us to seek opportunities to share cached files among users for higher efficiency atop the isolation baseline.

Our solution OpuS follows exactly this intuition. OpuS is a *two-stage* algorithm. In the first stage, it strives to share caches for high efficiency without suffering harmful manipulations. It then checks if the sharing outcome provides isolation for individual users. If *no* user prefers isolation, the algorithm settles on sharing. Otherwise, the attempt of sharing caches has not succeeded, and the algorithm reduces to the baseline isolation in the second stage.

In order to achieve high cache utilization, we should prevent the algorithm from reducing to the baseline isolation *as much as possible*. This requires us to design an efficient and strategy-proof cache sharing algorithm in the first stage that *likely* provides isolation guarantee.

B. Opportunistic Cache Sharing using VCG Mechanism

Our first attempt for opportunistic cache sharing builds on the classic VCG mechanism [11]. The VCG mechanism offers two desirable properties. First, it is strategy-proof. Second, it is efficient in that it allocates resources in a *socially optimal* manner. Specifically, the VCG mechanism seeks to maximize the *social welfare*, defined as the sum of utilities of all users, i.e., to maximize $\sum_i U_i(\mathbf{a})$. To enforce truth-telling, the mechanism taxes each user the *externalities* it causes, i.e., the loss of social welfare of the others due to the presence of the user. In the context of cache sharing, the VCG tax can be charged by probabilistically blocking a user's access to a cached file.

However, as we shall show in Sec. VI-B, using the VCG mechanism to opportunistically share caches ends up with little efficiency improvement. Because it targets to optimize the global social welfare, users making smaller contributions are forced to give way to others who can contribute more. More often than not, the former users would prefer isolated caches to sharing. The result is a frequent violation of isolation guarantee under a VCG allocation, which forces the algorithm to reduce to isolated caches as the last resort.

C. OpuS: Opportunistic Sharing for High Efficiency

The lesson we learnt from the failure of the VCG mechanism is that cache sharing in the first stage must factor in the requirement of isolation guarantee. This inspires us to turn to *proportional fairness* (PF) [20] for cache allocation, under which isolation guarantee is provided while high efficiency can also be attained [21].

In particular, we say allocation $\mathbf{a}$ is *proportionally fair* if, for any other allocation $\mathbf{a}'$, the sum of proportional changes of utilities is *non-positive*, i.e.,

$$\sum_i \frac{U_i(\mathbf{a}') - U_i(\mathbf{a})}{U_i(\mathbf{a})} \le 0.$$

In other words, the PF allocation achieves the highest aggregated percentage gain among all the other allocations, hence striking a good balance between fairness and efficiency [22]. Equivalently, a PF allocation maximizes the social welfare assuming *logarithm utility* [21], [23], i.e., it maximizes $\sum_i \log U_i(\mathbf{a})$.

Unfortunately, proportional fairness is not strategy-proof and cannot be directly applied to cache allocation. To address this problem, we frame PF allocation into the VCG mechanism (referred to as VCG-PF) assuming logarithm utility, which we call *virtual utility*.

Definition 4 (Virtual utility)**.** *Given allocation* $\mathbf{a}$*, we define* $V_i(\mathbf{a}) = \log U_i(\mathbf{a})$ *as the* virtual utility *of user* i*. Here,* $U_i(\mathbf{a})$ *measures the utility user* i *gains by having a* full access *to the cached files in allocation* $\mathbf{a}$*, i.e.,* $U_i(\mathbf{a}) = \sum_j a_j p_{i,j}$*.*

With this virtual utility, we apply the VCG mechanism. This time, the mechanism is tricked to seek PF allocation, in that it maximizes social welfare in terms of the virtual (logarithm) utility.

$$\begin{aligned} \text{maximize}_{\mathbf{a}} \quad & \textstyle\sum_i V_i(\mathbf{a}) = \sum_i \log U_i(\mathbf{a}), \\ \text{s.t.} \quad & \textstyle\sum_j a_j \le C, \\ & 0 \le a_j \le 1, \ \text{for all file } F_j. \end{aligned} \tag{2}$$

We denote by $\mathbf{a}^*$ the PF allocation obtained by solving problem (2). Once the allocation has been determined, the VCG mechanism charges each user a tax payment following the Clarke pivot rule [24, Chapt. 9.3.4]. Specifically, to determine the tax payment of user i, we exclude the user from the participants and recompute PF allocation $\mathbf{a}^*_{-i}$ that solves (2) *as if* user i were *absent*. The VCG tax charged to user i is

$$T_i = \textstyle\sum_{k \ne i} V_k(\mathbf{a}^*_{-i}) - \sum_{k \ne i} V_k(\mathbf{a}^*). \tag{3}$$

Intuitively, user i pays an amount equal to the total *loss* that it causes to other users—the difference between the virtual social welfare of others with and without its participation.

Now that user i gains $V_i(\mathbf{a}^*)$ but pays T_i, its net income is simply $V_i(\mathbf{a}^*) - T_i$. Recall that all the computations so far are based on the virtual utility taken logarithm. We exponentiate net income $V_i(\mathbf{a}^*) - T_i$ to recover the user's *net utility*:

$$U_i^{\text{net}}(\mathbf{a}^*) = \exp\Big(V_i(\mathbf{a}^*) - T_i\Big) = \exp(-T_i) \cdot U_i(\mathbf{a}^*). \tag{4}$$

Because the VCG mechanism charges non-negative tax, we have $T_i \ge 0$ and $0 < \exp(-T_i) \le 1$. In essence, user i gains $\exp(-T_i)$ share of the utility under PF allocation. From a user's perspective, this is equivalent to randomly blocking its access to the cached files with probability

$$f_i = 1 - \exp(-T_i). \tag{5}$$

Algorithm 1 OpuS: **Opp**ort**u**nistic **S**haring for high efficiency

```
 1: procedure OpuS($\{p_{i,j}\}$)                    ▷ $\{p_{i,j}\}$: caching preference
 2:     $(\mathbf{a}^*, \{T_i\}) \leftarrow$ VCG_PF($\{p_{i,j}\}$)      ▷ Seek to share cache
 3:     if Provides_IG($\mathbf{a}^*, \{T_i\}$) then
 4:         return $(\mathbf{a}^*, \{T_i\})$                ▷ Settle on cache sharing
 5:     else
 6:         return isolated allocation $\bar{\mathbf{a}}$         ▷ Reduce to isolation
 7: procedure Provides_IG($\mathbf{a}^*, \{T_i\}$)
 8:     for all user $i$ do
 9:         if $T_i > \bar{T}_i$ then          ▷ $\bar{T}_i$: break-even tax following (6)
10:             return False
11:     return True
12: procedure VCG_PF($\{p_{i,j}\}$)                     ▷ VCG-PF mechanism
13:     $\mathbf{a}^* \leftarrow$ PF allocation that solves (2)
14:     for all $i$ do
15:         $\mathbf{a}^*_{-i} \leftarrow$ PF allocation that solves (2) w/o user $i$'s presence
16:         $T_i \leftarrow \sum_{k \ne i} V_k(\mathbf{a}^*_{-i}) - \sum_{k \ne i} V_k(\mathbf{a}^*)$
17:     return $(\mathbf{a}^*, \{T_i\})$
```

We stress that the VCG tax charged to users, though resulting in a utility loss, is needed to enforce strategy-proofness. However, depending on the amount of tax, isolation guarantee can be violated. We quantify the break-even amount for each user through the following theorem. The proof is straightforward and is omitted.

Theorem 3. *For user* i*, its* break-even *tax is*

$$\bar{T}_i = \log \frac{U_i(\mathbf{a}^*)}{\bar{U}_i}, \tag{6}$$

where $\bar{U}_i$ *measures the utility it gains from isolation. User i would prefer isolation* if and only if *it is charged more than the break-even, i.e.,* $U_i^{\text{net}}(\mathbf{a}^*) < \bar{U}_i$ *iff* $T_i > \bar{T}_i$*.*

In cases when a user is charged more than the break-even tax, isolation guarantee is violated, and the attempt of seeking high efficiency through PF allocation fails. As the last resort, the algorithm turns to isolation by evenly dividing in-memory caches into dedicated partitions for individual users. We shall show in Sec. VI-B that this is less likely to happen. Because PF allocation is efficient with strong isolation guarantee (i.e., $U_i(\mathbf{a}^*) > \bar{U}_i$ for all i), the break-even tax $\bar{T}_i$ is usually high. This gives users a deep pocket to afford a high tax.

Algorithm 1 summarizes the entire process described above, which we call OpuS. To better illustrate OpuS, we give a running example as follows.

Example: We refer back to the example in Fig. 1 and explain how OpuS works. OpuS starts to seek PF allocation for high efficiency, which caches the entire file F_2 but half of F_1 and F_3 (depicted in Fig. 1). The mechanism then charges each user an amount equal to the utility loss it causes to other users. We take user B as an example for tax calculation. Should it be absent, user A would have monopolized the cache with two files F_1 and F_2 in memory. Therefore, the presence of user B results in A losing half of file F_1. User B pays the damage it causes, and is charged $T_B = \log 1 - \log 0.8 = \log 1.25$. By (4), user B gains net utility 0.64, which is higher than it would have gained from isolation (i.e., $\bar{U}_B = 0.6$). By symmetry, the same calculation also applies to user A. Since no user prefers isolation, OpuS settles on the PF allocation.

We next turn to Fig. 2, where we assume that user B lies and claims that it prefers file F_3 to F_2. We show that user B gains less net utility by cheating. OpuS starts with PF allocation. Given the claimed caching preferences, both files F_2 and F_3 are cached. Compared with Fig. 1, user B gains higher utility with more files in memory. However, such a utility gain is not justified: it pays an even greater amount of tax $T_B = \log 1 - \log 0.6 = \log 1.67$. Overall, user B gains net utility 0.6, lower than it would have gained by telling the truth.

D. Analysis

We now analyze the properties of OpuS. It is straightforward to show that OpuS provides isolation guarantee: in the worst case, OpuS reduces to isolated caches to retain this property.

Theorem 4. *OpuS provides isolation guarantee.*

We next show that OpuS is immune to *harmful* manipulations, in that no user can gain a higher utility *at the expense of others* by lying about its caching preference. We emphasize that this desirable property cannot be directly inferred from the well-known fact that telling the truth is the dominant strategy in the VCG mechanism. In fact, it is possible for a user to improve its utility by cheating OpuS. However, we will show that this can only *benefit* other users but not harm them, and hence such behavior should be allowed.

Theorem 5. *OpuS is strategy-proof.*

Proof: We consider user i. Let $\mathbf{a}$ be the allocation given by OpuS when user i truthfully reports its caching preferences. Now assume that user i misreports, and the resulting allocation changes to $\mathbf{a}'$. We show that user i can gain a higher utility by lying only if it benefits other users. We consider the following four cases, depending on if allocations $\mathbf{a}$ and $\mathbf{a}'$ are PF allocations given by the VCG mechanism of OpuS.

Case-I: Both $\mathbf{a}$ and $\mathbf{a}'$ are PF allocations given by the VCG mechanism. Based on (3) and (4), we derive the net utility that user i gains in allocation $\mathbf{a}$ by telling the truth as:

$$U_i^{\text{net}}(\mathbf{a}) = \exp\left(\sum_k \log U_k(\mathbf{a}) - \sum_{k\neq i} \log U_k(\mathbf{a}^*_{-i})\right). \quad (7)$$

Here, $\mathbf{a}^*_{-i}$ denotes the PF allocation without user i's participation, i.e., allocation $\mathbf{a}^*_{-i}$ is obtained by solving (2) as if user i were absent. Similarly, the net utility user i gains by lying is

$$U_i^{\text{net}}(\mathbf{a}') = \exp\left(\sum_k \log U_k(\mathbf{a}') - \sum_{k\neq i} \log U_k(\mathbf{a}^*_{-i})\right). \quad (8)$$

We now show that user i gains a higher net utility by telling the truth, i.e., $U_i^{\text{net}}(\mathbf{a}) \geq U_i^{\text{net}}(\mathbf{a}')$. This is equivalent to proving $\sum_k \log U_k(\mathbf{a}) \geq \sum_k \log U_k(\mathbf{a}')$, which is indeed the case as allocation $\mathbf{a}$ is PF allocation for truthful users, and it maximizes the social welfare assuming logarithm utilities.

Case-II: Both $\mathbf{a}$ and $\mathbf{a}'$ reduce to isolated caches. In this case, no user's (net) utility is changed if user i cheats.

Case-III: Allocation $\mathbf{a}'$ reduces to isolated caches, but allocation $\mathbf{a}$ does not. In this case, OpuS settles on PF allocation $\mathbf{a}$ given by the VCG mechanism with isolation guarantee when user i tells the truth. Therefore, user i is better off being honest than cheating.

Case-IV: Allocation $\mathbf{a}$ reduces to isolated cache, but $\mathbf{a}'$ does not. In this case, the cheating of user i benefits all users. It promotes isolated allocation $\mathbf{a}$ to PF allocation $\mathbf{a}'$ given by the VCG mechanism with isolation guarantee. No user becomes worse off due to cheating.

In summary, we show in all the four cases that no user can gain more utility at the expense of other users by cheating. ■

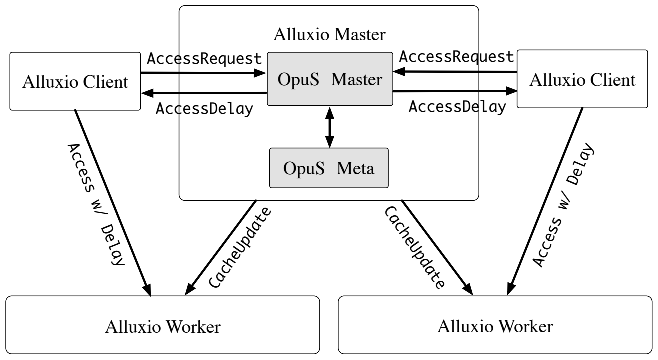


Fig. 4: Architecture of OpuS in Alluxio. The shaded boxes highlight our implementations.

V. Implementation

We have implemented OpuS as a pluggable cache manager in Alluxio [3], [14]. Our implementation is open-sourced for public access.[5] In this section, we briefly describe our implementation and its overhead.

A. Implementation

Alluxio: Alluxio [3], [14] is a memory-centric distributed filesystem enabling in-memory data sharing between different parallel frameworks (e.g., Spark and Hadoop) and their applications. Similar to HDFS, Alluxio stores files in `Workers` and uses a single `Master` to manage metadata (e.g., file permissions). The `Master` maintains a global view of cached files in the cluster and communicates with `Workers` when necessary.

OpuS Architecture: Fig. 4 gives an overview of our OpuS implementations atop Alluxio. The `OpuSMaster` implements the main cache allocation logic of Algorithm 1 and is launched along with the Alluxio `Master`.

Workflow: Upon the registration of an application, `OpuSMaster` assigns it an OpuS client ID. Instead of asking the application to report its caching preferences, `OpuSMaster` maintains a file access history for the application in the `OpuSMeta` library, from which it tallies up the file access frequency and uses it as the caching preference of that file. To cope with the changing file popularities, `OpuSMaster` collects the file access frequency within a *learning window* and runs Algorithm 1 *periodically* to enforce fair cache allocation

[5]Open-source in GitHub: https://github.com/yhust/Alluixo-for-OpuS.git

between clients. The updating rate and the learning window can be tuned at runtime in accordance with the access patterns.

When Algorithm 1 settles on cache sharing through PF allocation, `OpuSMaster` notifies `Workers` to cache the corresponding files through `CacheUpdate` messages. Later when it receives a file access request from a client, `OpuSMaster` *artificially* injects an *access delay* to emulate the effect of the VCG tax charging by means of probabilistic blocking. The injected delay should be set as the *expected* latency a user would experience when the system employs probabilistic blocking. Therefore, the expected delay is calculated as $f_i T_d$, where T_d is the pre-measured latency for reading a file from disk and f_i is the blocking probability of user i as specified in Eq. (5).

In case that the allocation reduces to isolation, `OpuSMaster` *emulates* isolated caches for individual clients. Specifically, to avoid frequent in-memory replication, the system maintains a *single* copy of a cached file, even if the file should have been kept by multiple clients in isolated caches. For each file, `OpuSMaster` tracks which clients should have kept it in their isolated caches: access to the file from the other clients are fully blocked.

B. Discussions

Overhead: We attribute the main source of overhead in our implementation to the calculation of the VCG tax: with N users, the system needs to solve $N + 1$ convex optimization problems in the form of (2). Our implementation employs CVXPY [25], a python package for convex optimization. We have not encountered any performance issue: the computation of cache allocation completes within three seconds in a wide range of parameter settings (cf. Sec. VI-C). In fact, the runtime of Algorithm 1 grows very slowly with the number of users.

Non-stationary file popularity: OpuS employs a sliding window to adjust to the non-stationary file popularities in real clusters. The length of the time window should be dynamically tuned to capture variations in access patterns. We leave this dynamic learning for future explorations and fix the time window as 20 minutes in the current implementation. Cache contents are updated every 20 minutes based on the access history in the past time window. A relatively long update interval keeps the implementation overhead at a low level, but might sacrifice cache efficiency (and even fairness) if the data popularity changes dramatically during the intervals. Fortunately, observations in production data analytics clusters [26] show that data popularity exhibits gradual ascent and decline on an *hourly* basis. Therefore, with an update rate of three times per hour, OpuS can well adjust to the changing file popularities.

Expected delay with varying file size: Recall that we enforce expected delay to emulate the effect of probabilistic blocking, where the access latency of disk I/O, denoted as T_d, should be pre-measured. For files of unit size, T_d can be easily estimated by a couple of sampling reads. In the scenario with varying file sizes, we instead benchmark the disk I/O bandwidth BW and enforce the expected delay to be $T_d = f_{size}/BW$, with f_{size} denoting the file size. In this way, we can conveniently calculate the equivalent delay of disk I/O for all files.

VI. Evaluation

We evaluate the performance of OpuS using experiments on EC2 clusters through micro- and macro-benchmarks. For larger scale evaluation, we use trace-driven simulations by feeding synthetic workloads. Our evaluation highlights are:

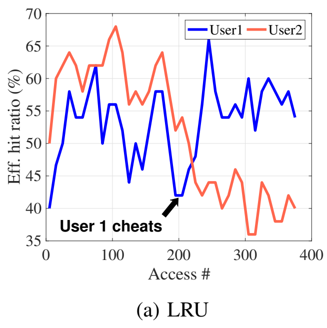
- OpuS eliminates the incentive of cheating, while LRU and FairRide fall short of this property (Sec. VI-A).
- OpuS outperforms isolated caches by $2.45\times$ and is 16.6% better than FairRide [9] in terms of cache hit ratio. In fact, OpuS comes close to the global optimum with a narrow gap less than 7% (Sec. VI-B).
- OpuS determines cache allocations in three seconds on average for up to 150 users and hence is scalable to large clusters with many users sharing the caches (Sec. VI-C).

Setup: Our implementations are based on Alluxio 1.5.0. We have respectively deployed a 5-node and a 10-node EC2 cluster for micro- and macro-benchmark experiments. Our experiments use `m4.large` instances [27], each with a dual-core 2.4 GHz processor and 8 GB memory.

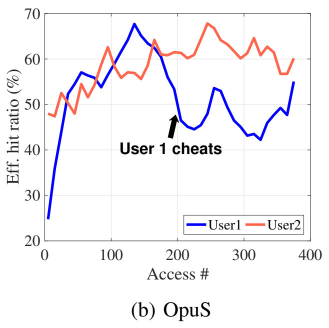
Workload: We use TPC-H queries [28] as the experiment workload for data analytics. This benchmark is a set of decision support queries, including a suite of business-oriented ad-hoc queries with broad industry-wide relevance. We have generated over 200 TPC-H datasets, each of 100 MB. Each dataset contains 8 tables storing contents such as customer information and inventory records. The size of a TPC-H table varies from 2 KB to 70 MB.

File popularity: Unless otherwise specified, we assume that user's file preferences follow the Zipf distribution, which is in line with the skewed data access patterns observed in production clusters [29], [30]. Due to the lack of traces in production clusters, we consider a short period while the file popularity remains stationary. Nonetheless, this assumption is supported by two facts. First, the data popularity does not change dramatically within a short time in production clusters [26]. Second, our evaluations in Sec. VI-C show that OpuS can update the cache allocation within three seconds in a wide range of parameter settings. Therefore, we focus on the performance study in a snapshot with stable file popularities.

Metric: We use the *effective hit ratio* as our main metric. Recall that our implementation emulates probabilistic blocking through delayed access. Therefore, the actual access latency is in proportion to the blocking probability. In our experiments, we count a delayed access as a *fractional* cache miss, where the fraction equals the *blocking probability*. We then tally up the total cache misses to compute the effective hit ratio for each user, with which we can easily compare the efficacy of OpuS with other schemes.

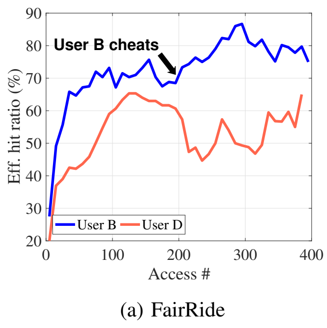


(a) LRU

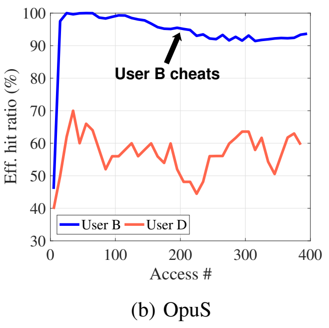


(b) OpuS

Fig. 5: [Cluster] Effective cache hit ratios measured for two users. User 1 started cheating from the 200th access.



(a) FairRide



(b) OpuS

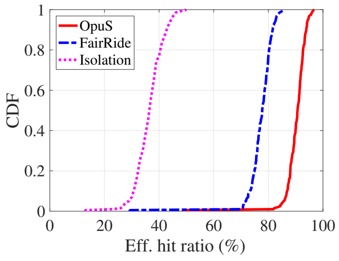
Fig. 6: [Cluster] Effective cache hit ratios measured for two users. User B started cheating from the 200th access.

A. *Strategy-proofness*

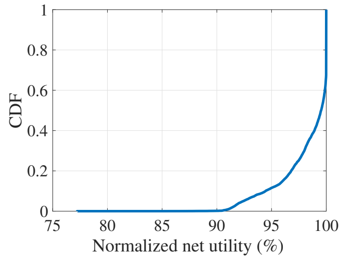
We start with micro-benchmark experiments to demonstrate how OpuS prevents users from cheating, whereas LRU and FairRide fail to provide such a guarantee.

LRU: By default, Alluxio uses the LRU policy to evict cached files, which allows strategic users to get more of their preferred files cached through spurious access. To demonstrate this problem, we consider two users accessing six TPC-H datasets. User 1 starts to triple its access rate after the 200th access. The total cache volume is configured to 300 MB. We run experiments using LRU and OpuS, respectively, and measure the effective cache hit ratio for both users. As shown in Fig. 5a, with LRU, user 1 manages to increase its hit ratio through spurious file access, which forces user 2 to give up caches with a dramatic drop in the hit ratio. This is not possible with OpuS. As depicted in Fig. 5b, user 1 can only hurt itself by cheating, while user 2 gets isolated with a stable hit ratio.

FairRide: We next illustrate that FairRide suffers from a similar problem in that users can carefully calculate the benefit-cost-ratio and misreport their cache preferences to free-ride on others. We run the experiment based on the previous example in Fig. 3, with four users caching three TPC-H datasets. User B starts to cheat after the 200th access, accessing file F_1 more frequently than F_2. We compare FairRide against OpuS. As shown in Fig. 6a, user B successfully gamed FairRide to increase its hit ratio by free-riding user D, who has witnessed a dramatic drop in performance. In contrast, we show in Fig. 6b that with OpuS, user B can only get worse off when cheating.



(a) CDF of effective hit ratio



(b) CDF of normalized net utility

Fig. 7: [Cluster] Macro-benchmark evaluations. (a) The distribution of effective hit ratio. (b) The distribution of normalized net utility (after-tax normalized by before-tax).

B. *Cache Efficiency*

We next evaluate the cache efficiency of OpuS using both macro-benchmark experiments and trace-driven simulations.

Macro-benchmark Experiment: We configure 20 users to randomly query 60 TPC-H datasets stored in a 10-node EC2 cluster. The file caching preferences of each user are generated following the Zipf distribution, with an exponent parameter of 1.1. The system cache volume is set to 5 GB.

We compare the cache performance of OpuS against FairRide and isolation. For each policy, we measure the effective hit ratio for each user over 20K data accesses and depict the CDF in Fig. 7a. OpuS outperforms the other two alternatives by a significant margin, with the highest average hit ratio as 90.3%. This is 2.45× of isolation (36.8%) and 16.6% higher than FairRide (77.4%). We attribute OpuS's superior performance to two determining factors: the high efficiency of PF allocations and the extremely low VCG tax charged to each user. Notice that the VCG tax, enforced through probabilistic blocking, is the key to guarantee strategy-proofness in OpuS. In our experiment, we observe that the VCG tax enforced by OpuS is in general quite light, i.e., OpuS delays the file access very slightly. Fig. 7b shows the CDF of the net utility (after-tax) normalized by the original utility the user gains from PF allocations without tax payment. From the CDF curve, more than 90% of the original utility is guaranteed at almost all times, and over half of the time, the net utility is more than 97%, i.e., the artificially enforced delay of OpuS is less than 3%.

Simulations: We resort to simulations for larger scale evaluations. We synthesize the data access patterns based on the traces in macro-benchmark experiments. We generate more than 5K different preference distributions for each user to repeat the simulations and plot the averaged results.

We first study the performance of OpuS with a varying number of users, ranging from 50 to 150. We generate 100 TPC-H datasets and configure the system cache capacity to 6 GB. Fig. 8 compares the average effective hit ratio measured across users using OpuS, FairRide, isolation, and the optimal LFU policy which optimizes the global hit ratio without any fairness

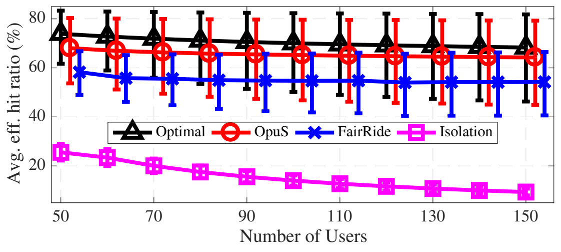


Fig. 8: Trace-driven simulations: average effective cache hit ratio of users. The error bars measure the 5th and 95th percentiles.

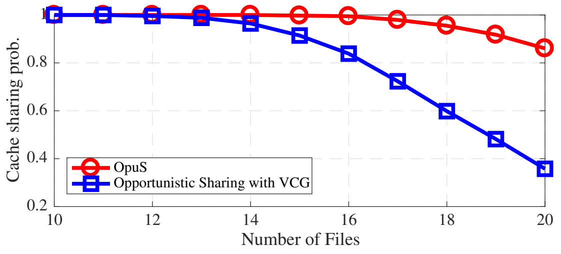


Fig. 9: Trace-driven simulations: the chance of settling on cache sharing. The error bars measure the 5th and 95th percentiles.

concern, considering stationary file popularity. Throughout our experiments, OpuS consistently outperforms FairRide and isolation, and comes close to the global optimum within 7%. Except for isolation, all the other three policies enable cache sharing. The result is stable hit ratios for individual users irrespective of the number of competitors sharing caches.

We next evaluate an alternative way of performing opportunistic cache sharing using the classic VCG mechanism (cf. Sec. IV-B). We generate 30 users querying TPC-H datasets of varying sizes from 10 GB to 20 GB. Fig. 9 compares the chances of settling on sharing using the classic VCG mechanism and OpuS. In most cases, OpuS has over 90% of chances to allow users to share caches. In contrast, with the classic VCG mechanism, the chance drops quickly to less than 40% with a large input data size. Thus, using the classic VCG mechanism to perform opportunistic cache sharing has a high risk to reduce to isolation, with low cache utilization.

C. Overhead

We evaluate the overhead of our prototype implementation in Alluxio in a 5-node EC2 cluster. We measure the time OpuS takes to compute a cache allocation (Algorithm 1) with a varying number of users. We generate 6 GB of TPC-H datasets and configure the cache capacity to 3 GB. We measure the computation time in the master node with up to 150 users. Fig. 10 shows the boxplot of the results in 100 trials. With more users sharing the cache, the configuration time for the cache allocation linearly increases. Nevertheless, even with 150 users, it takes OpuS approximately 3 seconds to configure the allocation. Recall that the cache allocation is updated every 20 minutes. OpuS can timely react to the changing file popularities.

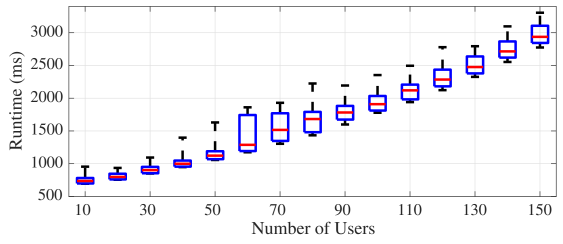


Fig. 10: [Cluster] The time to compute an allocation with up to 150 users. Boxes depict the 25th, 50th, and 75th percentiles. Whiskers depict the 5th and 95th percentiles.

VII. Related Work

Fair Resource Allocation: Policies for fair resource sharing have been extensively studied in multi-tenant systems for allocating CPU cycles and network bandwidth [15], [19], [20], [31]. However, unlike these resources, memory caches have a defining characteristic of being *non-exclusively* sharable across multiple users. This poses a unique challenge of free-riding manipulation that cannot be effectively prevented by traditional fair sharing policies.

Fair Cache Sharing: Among the few available policies designed for allocating memory caches, ROBUS [21] resorts to cooperative game theory and searches for cache allocations lying in the *core*. However, computing the core allocations not only incurs a heavy overhead but also invites strategic behaviors. To our knowledge, FairRide [9] is the only cache allocation policy that settles for eliminating free-riding manipulations. However, as shown in Fig. 3, a user can still game the system to improve its cache performance at the expense of others.

Sharing Public Goods: In the context of economics, caches can be viewed as a *public good* that is non-exclusively shared by many users. While the economic literature has developed many allocation mechanisms to enforce truthful sharing for public goods [12], [23], [32], these mechanisms seek to optimize the global social welfare but fail to provide the isolation guarantee. The design of OpuS learns from these mechanisms, but goes beyond and ensures isolation with a minimum efficiency loss.

VIII. Conclusions

In this paper, we have studied the problem of fair cache allocation for in-memory data analytics. We have shown that existing cache allocation policies either suffer from free-riding manipulations or result in poor cache utilization. We have proposed a new fair cache allocation scheme called OpuS, and demonstrated that OpuS guarantees isolation and is immune to manipulations. These two desirable properties come at a slight loss in the global cache efficiency. We have implemented OpuS in Alluxio as a pluggable cache manager. Evaluation results show that OpuS effectively eliminates the incentives of cheating and significantly outperforms the state-of-the-art solutions, achieving a performance within 7% of the global optimum.

Acknowledgements

We thank our shepherd, Mosharaf Chowdhury, and the anonymous reviewers for their valuable feedback. This research is supported in part by ITF Award ITS/391/15FX and RGC GRF Grant 16200214. Yinghao and Qizhen were supported in part by the Hong Kong PhD Fellowship Scheme.

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] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: Data management for modern business applications," *ACM SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, 2012.

[3] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. ACM Symp. Cloud Computing*, 2014.

[4] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in Apache Spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, 2016.

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *Proc. NIPS LearningSys*, 2015.

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *Proc. OSDI*, vol. 14, 2014, pp. 583–598.

[7] Memcached, "https://memcached.org."

[8] Redis, "http://redis.io."

[9] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, "FairRide: near-optimal, fair cache sharing," in *Proc. USENIX NSDI*, 2016.

[10] Apache Parquet. https://parquet.apache.org.

[11] W. Vickrey, "Counterspeculation, auctions, and competitive sealed tenders," *The Journal of finance*, vol. 16, no. 1, pp. 8–37, 1961.

[12] E. H. Clarke, "Multipart pricing of public goods," *Public choice*, vol. 11, no. 1, pp. 17–33, 1971.

[13] T. Groves, "Incentives in teams," *Econometrica: Journal of the Econometric Society*, pp. 617–631, 1973.

[14] Alluxio, "http://www.alluxio.org/."

[15] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands." in *Proc. USENIX NSDI*, 2016, pp. 407–424.

[16] 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.

[17] J. Jaffe, "Bottleneck flow control," *IEEE Trans. Commun.*, vol. 29, no. 7, pp. 954–962, 1981.

[18] L. Massoulié and J. Roberts, "Bandwidth sharing: objectives and algorithms," in *Proc. IEEE INFOCOM*, 1999.

[19] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks," in *Proc. ACM SIGCOMM*, 1998.

[20] F. Kelly, "Charging and rate control for elastic traffic," *Trans. Emerging Telecommunications Technologies*, vol. 8, no. 1, pp. 33–37, 1997.

[21] M. Kunjir, B. Fain, K. Munagala, and S. Babu, "ROBUS: Fair cache allocation for multi-tenant data-parallel workloads," *preprint arXiv:1504.06736*, 2015.

[22] R. Cole, V. Gkatzelis, and G. Goel, "Positive results for mechanism design without money," in *International conference on Autonomous agents and multi-agent systems*, 2013, pp. 1165–1166.

[23] ——, "Mechanism design for fair division: allocating divisible items without payments," in *ACM EC*, 2013.

[24] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic game theory*. Cambridge University Press Cambridge, 2007.

[25] CVXPY, "http://www.cvxpy.org/."

[26] 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 Comput. syst.*, 2011, pp. 287–300.

[27] Amazon Elastic Compute Cloud, "https://aws.amazon.com/ec2/," 2016.

[28] TPC-H, "http://www.tpc.org/tpch/."

[29] 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.

[30] 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. OSDI*, 2016, pp. 401–417.

[31] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *IEEE RTSS*, 1996.

[32] T. Groves and J. Ledyard, "Optimal allocation of public goods: A solution to the" free rider" problem," *Econometrica: Journal of the Econometric Society*, pp. 783–809, 1977.