FairRide: Near-Optimal, Fair Cache Sharing

Qifan Pu and Haoyuan Li, *University of California, Berkeley;* Matei Zaharia, *Massachusetts Institute of Technology;* Ali Ghodsi and Ion Stoica, *University of California, Berkeley*

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/pu

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

FairRide: Near-Optimal, Fair Cache Sharing

Qifan Pu, Haoyuan Li (UC Berkeley), Matei Zaharia (MIT), Ali Ghodsi, Ion Stoica (UC Berkeley)

Abstract –

Memory caches continue to be a critical component to many systems. In recent years, there has been larger amounts of data into main memory, especially in shared environments such as the cloud. The nature of such environments requires resource allocations to provide both performance isolation for multiple users/applications and high utilization for the systems. We study the problem of fair allocation of memory cache for multiple users with shared files. We find that, surprisingly, no memory allocation policy can provide all three desirable properties (isolation-guarantee, strategy-proofness and Pareto-efficiency) that are typically achievable by other types of resources, e.g., CPU or network. We also show that there exist policies that achieve any two of the three properties. We find that the only way to achieve both isolation-guarantee and strategy-proofness is through *blocking*, which we efficiently adapt in a new policy called FairRide. We implement FairRide in a popular memory-centric storage system using an efficient form of *blocking*, named as *expected delaying*, and demonstrate that FairRide can lead to better cache efficiency (2.6× over isolated caches) and fairness in many scenarios.

1 Introduction

Caches are a crucial component of most computer systems, characterized by two features: their impact on application performance, and their limited size compared to the total amount of data. With in-memory caches increasingly being used for large-scale data processing clusters [4, 26] in addition to databases and key-value stores [19, 37, 10, 30, 8], caches also play a key role in today's multi-tenant cloud environments. In a shared environment with multiple users, however, the problem of managing caches becomes harder: how should a provider allocate space across multiple users, each of which wants to keep their own datasets in memory?

Unfortunately, traditional caching policies do not provide a satisfactory answer to this problem. Most cache management algorithms (e.g., LRU, LFU) have focused on *global efficiency* of the cache (Figure 1a): they aim to maximize the overall hit rate. Regardless of being commonly used in today's cache systems for cloud serving (Redis [9], Memcached [7]) and big data storage (HDFS Caching [5]), this has two problems in a shared environment. First, users who read data at long intervals may gain little or no benefit from the cache, simply because their data is likely to be evicted out of the memory. Second, applications can also easily *abuse* such systems by making spurious accesses to increase their access rate.

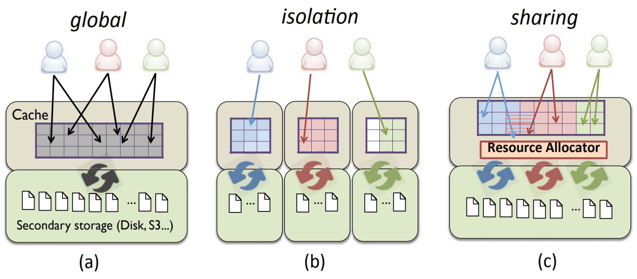


Figure 1: Different schemes. *Global*: single memory pool, agnostic of users or applications; *Isolation*: static allocations of memory among multiple users, possibly under-utilization (blank cells), no sharing; *Sharing*: allowing dynamic allocations of memory among users, and one copy of shared files (stripe cells).

There is no incentive to dissuade users from doing this in a cloud environment, and moreover, such shifts in the cache allocation can happen even with non-malicious applications. We show later that a strategic user can outperform a non-strategic user by 2.9×, simply by making spurious accesses to her files.

The other common approach is to have *isolated caches* for each user (Figure 1b). This gives each user performance guarantees and there are many examples in practice, e.g., hypervisors that set up separate buffer caches for each of its guest VMs, web hosting platforms that launch a separate `memcached` instance to each tenant. However, providing such performance guarantees comes at the cost of inefficient utilization of the cache.

This inefficiency is not only due to users not fully utilizing their allocated cache, but also because that a cached file can be accessed by multiple users at a time and isolating cache leads to multiple copies of such *shared files*. We find such *non-exclusive sharing* to be a defining aspect of cache allocation, while other resources are typically exclusively shared, e.g., a CPU time slice or a communication link can be only used by a single user at a time. In practice, there are a significant number of files shared across users in many workloads, e.g., we observe more than 30% files are shared by at least two users from a production HDFS log. Such sharing is likely to increase as more workloads move to multi-tenant environments.

In this paper, we study how to share cache space between multiple users that access shared files. To frame the problem, we begin by identifying desirable properties that we'd like an allocation policy to have. Building on common properties used in sharing of CPU and network resources [20], we identify three such properties:

	Isolation Guarantee	Strategy-Proofness	Pareto-Efficiency
global (e.g., LRU)	✗	✗	✓
max-min fairness	✓	✗	✓
FairRide	✓	✓	near-optimal
`None Exist`	✓	✓	✓

Table 1: **Summary of various memory allocation policies against three desired properties.**

- *Isolation Guarantee*: no user should receive less cache space than she would have had if the cache space were statically and equally divided between all users (i.e., assuming n users and equal shares, each one would get $1/n$ of the cache space). This also implies that the user's cache performance (e.g., cache miss ratio) should not be worse than isolation.

- *Strategy Proofness*: a user cannot improve her allocation or cache performance at the expense of other users by gaming the system, e.g., through spuriously accessing files.

- *Pareto Efficiency*: the system should be *efficient*, in that it is not possible to increase one user's cache allocation without lowering the allocation of some other user. This property captures operator's desire to achieve high utilization.

These properties are common features of allocation policies that apply to most resource sharing schemes, including CPU sharing via lottery or stride scheduling [39, 15, 35, 40], network link sharing via max-min fairness [28, 13, 17, 23, 33, 36], and even allocating multiple resources together for compute tasks [20]. Somewhat unexpectedly, there has been no equivalent policy for allocation of cache space that satisfies all three properties. As shown earlier, global sharing policies (Figure 1a) lack isolation-guarantee and strategy-proofness, while static isolation (Figure 1b) is not Pareto-efficient.

The first surprising result we find is that this deficiency is no accident: in fact, for sharing cache resources, *no policy can achieve all three properties*. Intuitively, this is because cached data can be shared across multiple users, allowing users to game the system by "free-riding" on files cached by others, or optimizing usage by caching popular files. This creates a strong trade-off between Pareto efficiency and strategy-proofness.

While no memory allocation policy can satisfy the three properties (Table 1), we show that there are policies that come close to achieving all three in practice. In particular, we propose FairRide, a policy that provides both isolation-guarantee (so it always performs no worse than isolated caches) and strategy-proofness (so users are not incentivized to cheat), and comes within 4% of global efficiency in practice. FairRide does this by aligning each user's benefit-cost ratio with her private preference, through *probabilistic blocking* (Section 3.4), i.e., probabilistically disallowing a user from accessing a cached file if the file is not cached on behalf of the user. Our proof in Section 5 shows that *blocking* is required to achieve strategy-proofness, and that FairRide achieves the property with minimal blocking possible.

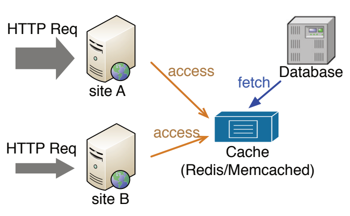
In practice, *probabilistic blocking* can be efficiently implemented using *expected delaying* (Section 4.1) in order to mitigate I/O overhead and to prevent even more sophisticated cheating models. We implemented FairRide on Tachyon [26], a memory-centric storage system, and evaluated the system using both cloud serving and big data workloads. Our evaluation shows that FairRide comes within 4% of global efficiency while preventing strategic users, meanwhile giving 2.6× more job runtime reduction over isolated caches. In a non-cooperative environment when users do cheat, FairRide outperforms max-min fairness by at least 27% in terms of efficiency. It is also worth noting that FairRide would support pluggable replacement policies as it still obeys each user's caching preferences, which allows users to choose different replacement policies (e.g., LRU, LFU) that best suit their workloads.

2 Background

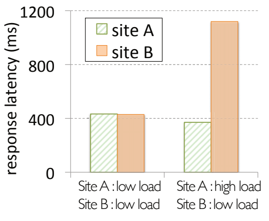
Most of today's cache systems are oblivious to the entities (users) that access data: CPU caches do not care which thread accesses data, web caches do not care which client reads a web page, and in-memory based systems such as Spark [41] do not care which user reads a file. Instead, these systems aim to maximize system efficiency (e.g., maximize hit rate) and as a result favor users that contribute more to improve efficiency (e.g., users accessing data at a higher rate) at the expense of the other users.

To illustrate the unfairness of these cache systems, consider a typical setup of a hosted service, as shown in Figure 2a. We setup multiple hosted sites, all sharing a single Memcached [7] caching system to speed up the access to a back-end database. Assume the loads of A and B are initially the same. In this case, as expected, the mean request latencies for the two sites are roughly the same (see left bars in Figure 2b). Next, assume that the load of site A increases significantly. Despite the fact that B's load remains constant, the mean latency of its requests increases significantly (2.9×) and the latency for A's requests surprisingly drops! Thus, an increase in A's load improves the performance of A, but degrades the performance of B. This is because A accesses the data more frequently, and in response the cache system starts loading more results from A while evicting B's results.

While the example is based on synthetic web workload, this problem is very real, as demonstrated by the many questions posted on technical forums [6], on how



(a)



(b)

Figure 2: (a) Typical cache setup for web servers. (b) Site B suffers high latency because unfair cache sharing.

to achieve resource isolation across multiple sites when using either Redis [9] or Memcached [7]. It turns out that none of the two popular caching systems provide any guarantee for performance isolation. This includes customized distributions from dominant cloud service providers, such as Amazon ElastiCache [1] and Microsoft Azure Redis Cache [3]. As we will show in Section 7, for such caching systems, it is easy for a strategic user to improve her performance and hurt others (with 2.9× performance gap) by making spurious access to files.

To provide performance isolation, the default answer in the context of cloud cache services today is to setup a separate caching instance per user or per application. This goes against consolidation and comes at a high cost. Moreover, cache isolation will eliminate the possibility of *sharing cached files*, which makes isolation even more expensive as there is a growing percentage of files to be shared. We studied a production HDFS log from a Internet company and observed 31.4% of files are shared by at least two users/applications. The shared files also tend to be more frequently accessed compared to non-shared files, e.g., looking at the 10% most accessed files, shared files account for as much as 53% of the accesses. The percentage of sharing can go even higher pair-wise: 22% of the users have at least 50% of their files accessed by another user. Assuming files are of equal sizes, we would need at least 31.4% more space if we assign isolated instances for each user, and even more cost on additional cache as the percentage of shared files in the working set is even larger.

Going back to the example in Figure 2, one possible strategy for *B* to reclaim some of its cache back would be to artificially increase its access rate. While this strategy may help *B* to improve its performance, it can lead to worse performance overall (e.g., lower aggregate hit rate). Worse yet, site *A* may decide to do the same: artificially increase its access rate. As we will show in this paper, this may lead to everyone losing out, i.e., everyone getting worse performance than when acting truthfully. Thus an allocation policy such as LRU is not strategy proof, as it does incentivize a site to misbehave to improve its performance. Furthermore, like with prisoner's dilemma, sites are incentivized to misbehave even if this leads to worse performance for everyone.

While in theory users might be incentivized to misbehave, a natural question is whether they are actually doing so in practice. The answer is "yes", with many real-world examples being reported in the literature. Previous works on cluster management [38] and job scheduling [20] have reported that users lie about their resource demands to game the system. Similarly, in peer-to-peer systems, "free-riding" is a well known and wide spread problem. In an effort to save bandwidth and storage, "free-riders" throttle their uplink bandwidth and remove files no longer needed, which leads to decreased overall performance [32]. We will show in Section 3 that shared files can easily lead to free-riding in cache allocation. Finally, as mentioned above, cheating in the case of caching is as easy as artificially increasing the access rate, for example, by running an infinite loop that accesses the data of interest, or just by making some random access. While some forms of cheating do incur certain cost or overhead (e.g., CPU cycles, access quota), the overhead is outweighed by the benefits obtained. On the one hand, a strategic user does not need many spurious accesses for effective cheating, as we will show in Section 7. If a caching system provides interfaces for users to specify file priorities or evict files in the system, cheating would be even simpler. On the other hand, many applications' performances are bottlenecked at I/O, and trading off some CPU cycles for better cache performance is worthwhile.

In summary, we argue that any caching allocation policy should provide the following three properties: (1) *isolation-guarantee* which subsumes performance isolation (i.e., a user will not be worse off than under static isolation), (2) *strategy-proofness* which ensures that a user cannot improve her performance and hurt others by lying or misbehaving, and (3) *Pareto efficiency* which ensures that resources are fully utilized.

3 Pareto Efficiency vs. Strategy Proofness

In this section we show that—under the assumption that the isolation-guarantee property holds—there is a strong trade-off between Pareto efficiency and strategy-proofness, that is, it is not possible to simultaneously achieve both in a caching system where files (pages) can be shared across users.

Model: To illustrate the above point, in the remainder of this section we consider a simple model where multiple users access a set of files. For generality we assume each user lets the cache system know the *priorities* in which her files can be evicted, either by explicitly specifying the priorities on the files or based on a given policy, such as LFU or LRU. For simplicity, in all examples, we assume that all files are of unit size.

```
FUNC u.access(f)    // user u accessing file f
if ($f \in Cache.fileSet$) then
    return CACHED_DATA;
else
    return CACHE_MISS;
end if

FUNC cache(u, f)    // cache file f for user u
while ($Cache.availableSize < f.size$) do
    $u1 = users.getUserWithLargestAlloc()$;
    $f1 = u1.getFileToEvict()$;
    if ($u1 == u$ and
      $u.getPriority(f1) > u.getPriority(f)$) then
        return CACHE_ABORT;
    end if
    $Cache.fileSet.remove(f1)$;
    $Cache.availableSize$ += $f1.size$;
    $u.allocSize$ -= $f1.size$;
end while
$Cache.fileSet.add(f)$
$Cache.availableSize$ -= $f.size$;
$u.allocSize$ += $f.size$;
return CACHE_SUCCEED;
```

Algorithm 1: Pseudocode for accessing and caching a file under max-min fairness.

Utility: We define each user's utility function as the *expected cache hit rate*. Given a cache allocation, it's easy to calculate a user's expected hit rate by just summing up her access frequencies of all the files cached in memory.

3.1 Max-min Fairness

One of the most popular solutions to achieve efficient resource utilization while still providing isolation is *max-min fairness*.

In a nutshell, max-min fairness aims to maximize the minimum allocation across all users. Max-min fairness can be easily implemented in our model by evicting from the user with the largest cache allocation, as shown in Algorithm 1. When a user accesses a file, f, the system checks whether there is enough space available to cache it. If not, it repeatedly evicts the files of the users who have the largest cache allocation to make enough room for f. Note that the user from which we evict a file can be the same as the user who is accessing file f, and it is possible for f to not be actually cached. The latter happens when f has a lower caching *priority* than any of the other user's files that are already cached. At line 10 from Algorithm 1, the *user.getPriority*() is called to obtain *priority*. Note caching *priority* depends on the eviction policy. In the case of LFU, *priority* represents file's access frequency, while in the case of LRU it can represent the inverse of the time interval since it has been accessed. Similar to access frequency, *priority* need not to be static, but rather reflects an eviction policy's instantaneous preference.

If all users have enough demand, max-min fairness ensures that each user will get an equal amount of cache, and max-min fairness reduces to static isolation. However, if one or more users do not use their entire share, the unused capacity is distributed across the other users.

3.2 Shared Files

So far we have implicitly assumed that each user accesses different files. However, in practice multiple users may share the same files. For example, different users or applications can share the same libraries, input files and intermediate datasets, or database views.

The ability to share the same allocation across multiple users is a key difference between caching and traditional environments, such as CPU and communication bandwidth, in which max-min fairness has been successfully applied so far. With CPU and communication bandwidth, only a single user can access the resource that was allocated to her: a CPU time slice can be only used by a single process at a time, and a communication link can be used to send a single packet of a single flow at a given time.

A natural solution to account for shared files is to "charge" each user with a *fraction* of the shared file's size. In particular, if a file is accessed by k users, and that file is cached, each user will be charged with $1/k$ of the size of that file. Let $f_{i,j}$ denote file j cached on behalf of user i, and let k_j denote the number of users that have requested the caching of file j. Then, the total cache size *allocated* to user i, $alloc_i$, is computed as

$$alloc_i = \sum_j \frac{\text{size}(f_{i,j})}{k_j}. \quad (1)$$

Consider a cache that can hold 6 files, and assume three users. User 1 accesses files $A, B, C, \ldots$ user 2 accesses files $A, B, D, \ldots$, and user 3 accesses files $F, G, \ldots$. Assuming that each file is of unit size, the following set of cached files represent a valid max-min allocation: A, B, C, D, F, and G, respectively. Note that since files A and B are shared by the first two users, each of these users is only charged with half of the file size. In particular, the cache allocation of user 1 is computed as $\text{size}(A)/2 + \text{size}(B)/2 + \text{size}(C) = 1/2 + 1/2 + 1 = 2$. The allocation of user 2 is computed in a similar manner, while allocation of user 3 is simply computed as $\text{size}(F) + \text{size}(G) = 2$. The important point to note here is that while each user has been allocated the same amount of cache as computed by Eq. 1, users 1 and 2 get three files cached (as they get the benefit of sharing two of them), while user 3 gets only two.

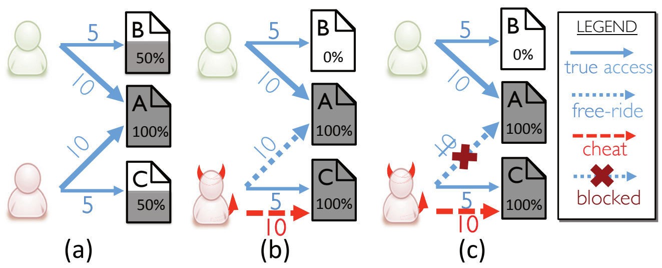


Figure 3: Example with 2 users, 3 files and total cache size of 2. Numbers represent access frequencies. (a). Allocation under *max-min fairness*; (b). Allocation under *max-min fairness* when second user makes spurious access (red line) to file C; (c). Blocking free-riding access (blue dotted line).

3.3 Cheating

While max-min fairness is strategy-proof when users access different files, this is no longer the case when files are shared. There are two types of cheating that could break strategy-proofness: (1) Intuitively, when files are shared, a user can "free ride" files that have been already cached by other users. (2) A thrifty user can choose to cache files that are shared by more users, as such files are more economic due to cost-sharing.

Free-riding To illustrate "free riding", consider two users: user 1 accesses files A and B, and user 2 accesses files A and C. Assume size of cache is 2, and that we can cache a fraction of a file. Next, assume that every user uses LFU replacement policy and that both users access A much more frequently than the other files. As a result, the system will cache file A and "charge" each user by 1/2. In addition, each user will get half of their other files in the cache, *i.e.,* half of file B for user 1, and file B for user 2, as shown in Figure 3(a). Each user gets a cache hit rate of $5 \times 0.5 + 10 = 12.5$[1] hits/sec.

Now assume user 2 cheats by spuriously accessing file C to artificially increase its access rate such that to exceed A's access rate (Figure 3(b)), effectively sets the *priority* of C higher than B. Since now C has the highest access rate for user 2, while A remains the most accessed file of user 1, the system will cache A for user 1 and C for user 2, respectively. The problem is that user 2 will still be able to benefit from accessing file A, which has already been cached by user 1. At the end, user 1 gets 10 hits/sec, and user 2 gets 15 hits/sec. In this way, user 2 free-rides on user 1's file A.

Thrifty-cheating To explain the kind of cheating where a user carefully calculates cost-benefits and then changes file priorities accordingly, we first define cost/(hit/sec) as the amount of budget cost a user pays to get 1 hit/sec access rate for a unit file. To optimize over the utility, which is defined as the total hit rate, a user's optimal strategy is not to cache the files that one has highest access frequencies, but the ones with lowest cost/(hit/sec). Compare a file of 100MB, shared by 2 users and another file of 100MB, shared by 5 users. Even though a user access the former 10 times/sec and the latter only 8 times/sec, it is overall economic to cache the second file (comparing 5MB/(hit/sec) vs. 2.5MB/(hit/sec)).

[1]When half of a file is in cache, half of the page-level accesses to the file will result in cache miss. Numerically, it is the equal to missing the entire file 50% of the time. So hit rate is calculated as access rate multiplied by percentage cached.

The consequence of "thrift-cheating", however, is more complicated. As it might appear to improve user and system performance at first glance, it doesn't lead to an equilibrium where all users are content about their allocations. This can cause users to constantly game the system which leads to a worse outcome.

In the above examples we have shown that due to another user cheating, one can experience utility loss. A natural question to ask is, how bad could it be? i.e. What is the upper bound a user can lose when being cheated? By construction, one can show that for two-user cases, a user can lose up to 50% of cache/hit rate when all her files are shared and "free ridden" by the other strategic user. As the free-rider evades charges of shared files, the honest user double pays. This can be extended to a more general case with n ($n > 2$) users, where loss can increase linearly with the number of cheating users. Suppose that cached files are shared by n users, each user pays $\frac{1}{n}$ of the file sizes. If $n-1$ strategic users decide to cache other files, the only honest user left has to pay the total cost. In turn, the honest user has to evict at most $(\frac{n-1}{n})$ of her files to maintain the same budget.

It is also worth mentioning that for many applications, moderate or even minor cache loss can result in drastic performance drop. For example, in many file systems with overall high cache hit ratio, the effective I/O latency with caching could be approximated as $T_{IO} = Ratio_{miss} Latency_{miss}$. A slight difference in the cache hit ratio, e.g. from 99.7% to 99.4%, means 2× I/O average latency drop! This indeed necessitates strategy-proofness in cache policies.

3.4 Blocking Access to Avoid Cheating

At the heart of providing strategy-proofness is this question of how free-riding can be prevented. In the previous example, user 2 was incentivized to cheat because she was able to access the cached shared files regardless her access patterns. Intuitively, if user 2 is blocked from accessing files that she tries to free-ride, she will be dis-incentivized to cheat.

Applying blocking to our previous example, user 2 will not be allowed to access A, despite the fact that user 1 has already cached A (Figure 3(c)). The system blocks user 2 but not user 1 because user 1 is the sole person who pays the cache. As a result, user 2 gets only 1 cache size with a less important file C.

As we will show in Section 5 this simple scheme is strategy-proof. On the other hand, this scheme is unfortunately not Pareto efficient by definition, as the performance (utility) of user 2 can be improved without hurting user 1 by simply letting user 2 access file A.

Furthermore, note that it is not necessary to have a user cheating to arrive at the allocation in Figure 3. Indeed, user 2 can legitimately access file C at a much higher rate than A, In this case, we get the same allocation—file A is cached on behalf of user 1 and file C is cached on behalf of user 2—with no user cheating. Blocking in this case will reduce the system utilization by punishing a well-behaved user.

Unfortunately, the cache system cannot differentiate between a cheating and a well-behaved user, so it is not possible to avoid the decrease in the utilization and thus the violation of Pareto efficiency, even when every user in the system is well-behaved.

Thus, in the presence of shared files, with max-min fairness allocation we can achieve either Pareto efficiency or strategy-proofness, but not both. In addition, we can trade between strategy-proofness and Pareto efficiency by blocking a user from accessing a shared file if that file is not in the user's cached set of files, even though that file might have been cached by other users.

In Section 5, we will show that this trade-off is more general. In particular, we show that in the presence of file sharing there is no caching allocation policy that can achieve more than two out of the three desirable properties: isolation-guarantee, strategy-proofness, and Pareto efficiency.

4 FairRide

In this section, we describe FairRide, a caching policy that extends max-min fairness with *probabilistic blocking*. Different from max-min fairness, FairRide provides isolation-guarantee and strategy-proofness at the expense of Pareto-efficiency. We use *expected delaying* to implement the conceptual model of *probabilistic blocking*, due to several system considerations.

Figure 4 shows the control logic for a user i accessing file j under FairRide. We will compare it with the pseudo-code of max-min fairness, Algorithm 1. In max-min, a user i can directly access a cached file j, as long as j is cached in memory. While with FairRide, there is an chance that the user might get blocked for accessing the cached copy. This is key to making FairRide strategy-proof and the *only* difference with max-min fairness, which we prove in Section 5. The chance of blocking is not an arbitrary probability, but is set at $\frac{1}{n_j+1}$, where n_j is the number of other users caching the file. We will

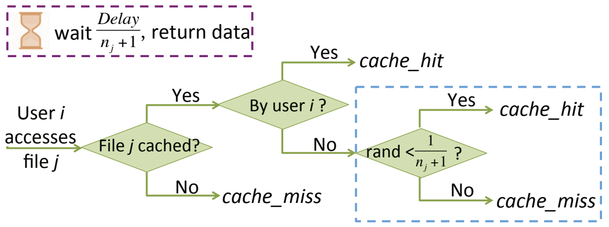

wait $\frac{Delay}{n_j+1}$, return data
User i accesses file j
File j cached?
Yes
No
cache_miss
By user i ?
Yes
cache_hit
No
rand < $\frac{1}{n_j+1}$?
Yes
cache_hit
No
cache_miss

Figure 4: With FairRide, a user might be blocked to access a cached copy of file if the user does not pay the storage cost. The blue box shows how this can be achieved with *probabilistic blocking*. In system implementation, we replace the blue box with the purple box, where we instead delay the data response.

prove in Section 5 that this is the only and minimal blocking probability setting that will make a FairRide strategy-proof.

Consider again the example in Figure 3. If user 2 cheats and makes spurious access to file C, file A will be cached on behalf of user 1. In that case, FairRide recognizes user 2 as a non-owner of the file, and user 2 has $\frac{1}{2}$ chance to access directly from the cache. So user 2's total expected hit rate becomes $5+10\times\frac{1}{2}=10$, which is worse than 12.5 before without cheating. In this way, FairRide discourages cheating and makes the policy strategy-proof.

4.1 Expected Delaying

In real systems, *probabilistic blocking* could not thoroughly solve the problem of cheating, as now a strategic user can make even more accesses in hope that one of the accesses is not blocked. For example, if a user is blocked with a probability of $\frac{1}{2}$, he can make three accesses so to reduce the likelihood of being blocked to $\frac{1}{8}$. In addition, *blocking* itself is not an ideal way to implement in a system as it further incurs unnecessary I/O operations (disk, network) for blocked users. To address this problem, we introduce *expected delaying* to approximate the expected effect of *probabilistic blocking*. When a user tries to access an in-memory file that is cached by other users, the system delays the data response with certain wait duration. The wait time should be set as the expected delay a user would experience if she's probabilistically blocked by the system. In this way, it is impossible to get around the delaying effect, and the system does not have to issue additional I/O operations. The theoretically equivalent wait time could be calculated as $t_{wait} = Delay_{mem} \times (1-p_{block}) + Delay_{disk,network} \times p_{block}$, where p_{block} is the blocking probability as described above, and $Delay_x$ being the access latency of medium x. As memory access latency is already incurred during data read time, we simply set the wait time to be $Delay_{disk,newtwork} \times p_{block}$. We will detail how we measure the secondary storage delay in Section 6.

5 Analysis

In this section, we prove that the general trade-off between the three properties is fundamental with existence of file sharing. Next in Section 5.2, we also show by proof that FairRide indeed achieves strategy-proof and isolation-guarantee, and that FairRide uses most efficient blocking probability to achieve strategy-proofness.

5.1 The SIP theorem

We state the following **SIP theorem**: *With file sharing, no cache allocation policy can satisfy all three following properties: strategy-proofness (S), isolation-guarantee (I) and Pareto-efficiency (P).*

Proof of the SIP theorem
The three properties are defined as in Section 1, and we use total hit rate as the performance metric. Reusing the example setup in Figure 3(a), we now examine a *general* policy P. The only assumption of P is that P satisfies isolation-guarantee and Pareto-efficiency, and we shall prove that such policy P must not be strategy-proof, i.e. a user can cheat to improve under P. We start with the case when no user cheats for Figure 3(a). Let y_1, y_2 be user 1 and 2's total hit rate:

$$y_1 = 10x_A + 5x_B \tag{2}$$

$$y_2 = 10x_A + 5x_C \tag{3}$$

Where x_A, x_B, x_C are fractions of the each file A, B, C cached in memory.[2] Because $x_A + x_B + x_C = 2$, and $y_1 + y_2 = 15x_A + 5(x_A + x_B + 5x_c)$, it's impossible for $y_1 + y_2 > 25$, or, for both y_1 and y_2 to be greater than 12.5. As the two users have symmetric access patterns, we assume $y_2 < 12.5$ without loss of generality.

Now if user 2 cheats and increases her access rate of file C to 30, we can prove that she can get a total rate of 13.3, or $y_2 > 13.3$. This is partly because the system has to satisfy a new isolation guarantee:

$$y'_2 = 10x_A + 30x_C > 30 \tag{4}$$

It must hold that $x_C > \frac{2}{3}$, because $x_A \leq 1$. Also, because $x_C \leq 1$ and $x_A + x_B + x_C = 2$, we have $x_A + x_B \geq 1$ to achieve Pareto-efficiency. For the same reason, $x_A = 1$ is also necessary as it's strictly better to cache file A over file B for both users. Plugging $x_A = 1, x_C > \frac{2}{3}$ back to user 2's actual hit rate calculation (Equation 3), we get $y_2 > 13.3$.

So far, we have found a cheating strategy for a user 2 to improve her cache performance and hurt the other user. This is done under a general policy P that assumes only isolation-guarantee and Pareto-efficiency but nothing else. Therefore, we can conclude that any policy P that satisfies the two properties cannot achieve strategy-proofness. In other words, no policy can achieve all three properties simultaneously. This ends the proof for the SIP theorem.

[2] We use fractions only for simplifying the proof. The theorem holds when we can only cache a file/block in its entirety.

5.2 FairRide Properties

We now examine FairRide (as described in Section 4) against three properties.

Theorem FairRide *achieves isolation-guarantee.*

Proof Even if FairRide does complete blocking, in which each user gets strictly less memory cache, the amount of cache a user accesses is: $Cache_{total} = \sum_j size(file_j)$, j for all the files the user caches. Because FairRide splits the charges of shared files across all users, a user's allocation budget is spent up by: $Alloc = \sum_j \frac{size(file_j)}{n_j}$, with n_j being the number of users sharing $file_j$. Combining the two equations we can easily derive that $Cache_{total} > Alloc$. As $Alloc$ is also what a user can get in *isolation*, we can conclude that the amount of memory a user can access is always bigger than *isolation*. Likewise, we can prove the total hit rate user gets with FairRide is greater than *isolation*.

Theorem FairRide *is strategy-proof.*

Proof We will sketch the proof using cost-benefit analysis, following the line of reasoning in Section 3.3. With *probabilistic blocking*, a user i can access a file j without caching it with a probability of $\frac{n_j}{n_j+1}$. This means that the benefit resulted from caching is the *increased_rate*, equal to $freq_{ij}\frac{1}{n_j+1}$. The cost is $\frac{1}{n_j+1}$ for the joining user, with n_j other users already caching it. Dividing the two, the benefit-cost ratio is equal to $freq_{ij}$, user i's access frequency of file j. As a user is incentivized to cache files based on the descending order of benefit-cost ratio, this results in caching files based on actual access frequencies, rather than cheating. In other words, FairRide is incentive-compatible and allows users to perform truth-telling.

Theorem FairRide*'s uses lower-bound blocking probabilities for achieving strategy-proofness.*

Proof Suppose a user has 2 files: f_j,f_k with access frequencies of $freq_j$ and $freq_k$. We use p_j and p_k to denote the corresponding blocking probabilities if the user chooses not to cache the files. Then the benefit-cost ratios for the two files are $freq_j p_j(n_j + 1)$ and $freq_k p_k(n_k + 1)$, n_j and n_k being the numbers of other users already caching the files. For the user to be truth-telling for whatever $freq_j$, $freq_k$, n_j or n_k, we must have $\frac{p_j}{p_k} = \frac{n_k+1}{n_j+1}$. Now p_j and p_k can still be arbitrarily small or big, but note $p_j(p_k)$ must be 1 when

$n_j(n_k)$ is 0, as no user is caching file $f_j(f_k)$. Putting $p_j = 1, n_j = 0$ into the equation we will have $p_k = \frac{1}{n_k+1}$. Similarly, $p_j = \frac{1}{n_j+1}$. Thus we show that FairRide's blocking probabilities are the only probabilities that can provide strategy-proofness in the general case (assuming any access frequencies and sharing situations). The only probabilities are also the lower-bound probabilities.

6 Implementation

FairRide is evaluated through both system implementation and large-scale trace simulations. We have implemented FairRide allocation policy on top of Tachyon [26], a memory-centric distributed storage system. Tachyon can be used as a caching system and it supports in-memory data sharing across different cluster computation frameworks or applications, e.g. multiple Hadoop Mapreduce [2] or Spark [41] applications.

Users and Shares Each application running on top of Tachyon with FairRide allocation has a FairRide client ID. Shares for each user can be configured. When shares are changed during system uptime, cache allocation is re-allocated over time, piece by piece, by evicting files from the user who uses most atop of her share, i.e., $argmax_i(Alloc_i - Capacity * Share_i)$, thus converging to the configured shares eventually.

Pluggable Policy Because FairRide obeys each user's individual caching preferences, it can apply a two-level cache replacement mechanism. It first picks the user who occupies the most cache in the system, and then finds the least preferred file from that user to evict. This naturally enables "pluggable policy", allowing each user to pick a replacement policy best fit for her workload. Note this would not be possible for some global policies such as global LRU. A user's more frequently accessed file could be evicted by a less frequently accessed file just because the first file's aggregate frequency across all users is lower than the second file. We've implemented "pluggable policy" in the system and expose a simple API for applications to pick best replacement policy.

```
Client.setCachePolicy(Policy.LRU)
Client.setCachePolicy(Policy.LFU)
Client.pinFile(fileId)
```

Currently, our implementation of FairRide supports LRU (Least-Recently-Used) and LFU (Least-Frequently-Used), as well as policies that are more suited for data-parallel analytics workloads, e.g. LIFE or LFU-F that preserves all-or-nothing properties for cached files [12]. Another feature FairRide supports is "pinned files". Through a `pinfile(fileId)` API, a user can override the replacement policy and prioritize specified files.

Delaying The key to strategy-proofness in implementing FairRide is to emulate *probabilistic blocking* by *delaying* the read of a file which a user didn't cache before. Thus the amount of wait time has to approximate the wait time as if the file is not cached, for any type of read. We implement *delaying* by simply sleeping the thread before giving a data buffer to the Tachyon client. The delay time is calculated by $\frac{size(buffer)}{BW_{disk}}$, with BW_{disk} being the pre-measured disk bandwidth on the node, and $size(buffer)$ being the size of the data buffer sent to the client. The measured bandwidth is likely an over-estimate of run-time disk bandwidth due to I/O contention when system is in operation. This causes shorter delay, higher efficiency, and less strategy-proofness, though a strategic user should gain very little from this over-estimate.

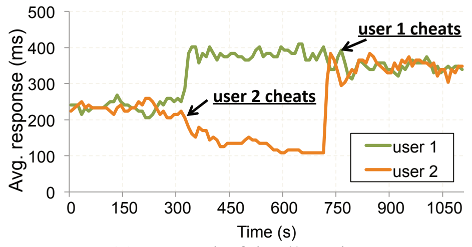
Node-based Policy Enforcement

Tachyon is a distributed system comprised of multiple worker nodes. We enforce allocation policies independently at each node. This means that data is always cached locally at the node when being read, and that when the node is full, we evict from the user who uses up most memory on that node. This scheme allows a node to select an evicting user and perform cache replacement without any global coordination.

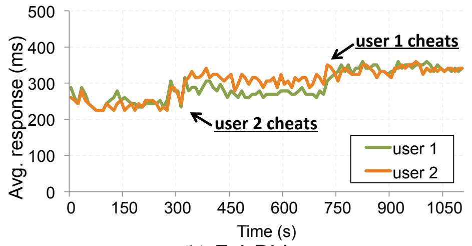
The lack of global coordination can incur some efficiency penalty, as a user is only guaranteed to get at least $1/n$-th of memory on each node, but not necessarily $1/n$-th of total memory across the cluster. This happens when users have access skew across nodes. To give an example, suppose a cluster of two nodes, each with 40GB memory. One user has 30GB frequently accessed data on node 1 and 10GB on node 2, and another user has 10GB frequently accessed data on node 1 and 30GB on node 2. Allocating 30GB on node 1 and 10GB on node 2 to the first user will outperform a 20 to 20 even allocation on each node, in terms of hit ratio for both users. Note that such allocation is still fair globally – each user gets 40GB memory in total. Our evaluation results in Section 7.6 will show that node-based scheme is within 3%∼4% compared to global fairness, because of the self-balance nature of big data workloads on Tachyon.

7 Experimental Results

We evaluated FairRide using both micro- and macro-benchmarks, by running EC2 experiments on Tachyon, as well as large-scale simulations replaying production workloads. The number of users in the workloads varies from 2 to 20. We show that while non-strategy-proof policies can cause everybody worse-off by a large margin (1.9×), FairRide can prevent user starvation within



(a) Max-min fair allocation



(b) FairRide

Figure 5: Miss ratio for two users. At $t = 300s$, user 2 started cheating. At $t = 700s$, user 1 joined cheating.

4% of global efficiency. It is 2.6× better than isolated caches in terms of job time reduction, and gives 27% higer utilization compared to max-min fairness.

We start by showing how FairRide can dis-incentivize cheating users by blocking them from accessing files that they don't cache, in Section 7.1. In Section 7.2, we compare FairRide against a number of schemes, including max-min fairness using experiments on multiple workloads: TPC-H, YCSB and a HDFS production log. Section 7.3 and Section 7.4 demonstrate FairRide's benefits with multiple users and pluggable policies. Finally, in Section 7.5, we use Facebook traces that are collected from a 2000-node Hadoop cluster to evaluate the performance of FairRide in large-scale clusters.

7.1 Cheating and Blocking

In this experiment, we illustrate how FairRide can prevent a user from cheating. We ran two applications on a 5-node Amazon EC2 cluster. The cluster contains one master node and four worker nodes, each configured with 32GB memory. Each application accessed 1000 data blocks (128MB each), among which 500 were shared. File access complied with Zipf distribution. We assumed users knew a priori which files are shared, and could cheat by making excessive accesses to non-shared files. We used LRU as cache replacement policy for this experiment.

We ran the experiment under two different schemes, max-min fair allocation (Figure 5a) and FairRide (Figure 5b). Under both allocations, the two users got similar average block response time (226ms under max-min, 235ms under FairRide) at the beginning ($t < 300s$). For max-min fair allocation, when user 2 started to cheat at $t = 300s$, she managed to lower her miss ratio over time (∼130ms), while user 1 got degraded performance with 380ms. At $t = 750s$, user 1 also started to cheat and both users stayed at high miss ratio (315ms). In this particular case, there was strong incentive for both the users to cheat at any point of time because cheating could always decrease the cheater's miss ratio (226ms→130ms,380ms→315ms). Unfortunately, both users get worse performance compared to not cheat all. Such a prisoner's dilemna would not happen with FairRide (Figure 5b). When user 2 cheated at $t = 300s$, her response time instead increases to 305ms . Because of this, both users would rather not cheat under FairRide and behave truthfully.

7.2 Benchmarks with Multiple Workloads

Now we evaluate FairRide by running three workloads on a EC2 cluster.

- **TPC-H** The TPC-H benchmark [11] is a set of decision support queries based on those used by retailers such as Amazon. The queries can be separated into two main groups: a sales-oriented group and a supply-oriented group. These two groups of queries have some separate tables, but also share common tables such as those maintaining inventory records. We treated two query groups as from two independent users.
- **YCSB** The Yahoo! Cloud Serving Benchmark provides a framework and common set of workloads for evaluating the performance of key-value serving stores. We implemented a YCSB client and ran multiple YCSB workloads to evaluate FairRide. We let half of files to be shared across users.
- **Production Cluster HDFS Log** The HDFS log is collected from a production Hadoop cluster at a large Internet company. It contains detailed information such as access timestamps and access user/-group information. We found that more than 30% of files are shared by at least two users.

We ran each workload under the following allocation schemes: 1) *isolation*: statically partition the memory space across users; 2) *best-case*: i.e. max-min fair allocation and assume no user cheats; 3) FairRide: our solution which uses *delaying* to prevent cheating; 4) *max-min* : max-min fair allocation with half users trying to game the system. We used LRU as the default cache replacement algorithm for all users and assumed cheating users know what files are shared.

We focus on three questions: 1) does sharing the cache improve performance significantly? (comparing performance gain over *isolation*) 2) can FairRide prevent cheating with small efficiency loss? (comparing FairRide with *best-case*) 3) does cheating degrade system performance significantly? (comparing FairRide with *max-min*).

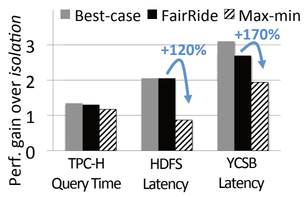


Figure 6: Summary of performance results for three workloads, showing the gain compared to isolated caches.

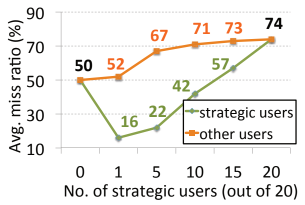


Figure 7: Average miss ratios of cheating users and non-cheating users, when there are multiple cheaters.

To answer these questions, we plot the relative gain of three schemes compared to *isolation*, as shown in Figure 6. In general, we find sharing the cache can improve performance by 1.3∼3.1×, with *best-case*. If users cheat, 15%∼220% of the gain will be lost. For the HDFS workload, we also observe that cheating causes a performance drop below *isolation*. While FairRide is very close to *best-case* with 3%∼13% overhead, it prevents the undesirable performance drop.

There are other interesting observations to note. First of all, the overhead of FairRide, is more noticeable in the YCSB benchmark and TPC-H than in the HDFS trace. We find that this is because the most shared files in the HDFS prodcution trace are among the top accessed files for both users. Therefore, both users would cache the shared files, resulting in less *blocking/delaying*. Secondly, cheating user benefits less in the HDFS trace, this is due to fact that the access distribution across files are highly long tailed in that trace, so that even cheating help user gain more memory, it doesn't show up significantly in terms of miss ratio. Finally, there is a varied degree of connection between miss ratio and application performance (read latency, query time), e.g., YCSB's read latency is directly linked to miss ratio change, while TPC-H's query response time is relatively stable. This is because, for the latter, a query typically consists of multiple stages of parallel tasks. As the completion time of a stage is decided by the slowest task, caching could only help when all tasks speed up. Therefore, a caching algorithm that can provide all-or-nothing caching for parallel tasks is needed to speed up query response time. We evaluated the Facebook trace with such a caching algorithm in Seciton 7.5.

7.3 Many Users

We want to understand how the effect of cheating relates to the number of active users in the system. In this experiment, we replay YCSB workloads with 20 users, where each pair of users have a set of shared files that they access commonly. Users can cheat by making excessive access to their private files. We increase the number of strategic users in different runs and plot the average miss ratio for both the strategic user group and the truthful user group in Figure 7. As expected, the miss ratio of the truthful group increases when there is a growing number of strategic users. What's interesting is that for the strategic group, the benefit they can exploit decreases as more and more users joining the group. With 12 strategic users, even the strategic group has worse performance compared to the no-cheater case. Eventually both groups converge at a miss ratio of 74%.

7.4 Pluggable Policies

Next, we evaluated the benefit of allowing pluggable policies. We ran three YCSB clients concurrently with each client running a different workload. The characteristics of the three workloads are summarized below:

User ID	Workload	Distribution	Replacement
1	YCSB(a)	zipfian	LFU
2	YCSB(d)	latest-most	LRU
3	YCSB(e)	scan	priority [3]

In the experiment, each YCSB client sets up the best replacement specified in the above table with the system. We compared our system with traditional caching systems that support only configuration of one uniform replacement policy, applied to all users. We ran the system with uniform configuration three times, each time with a different policy (LRU, LFU and priority). As shown in Figure 8, by allowing the users to specify a best replacement policy on their own, our system is able to provide gain of the best case for each of the user among all uniform configurations.

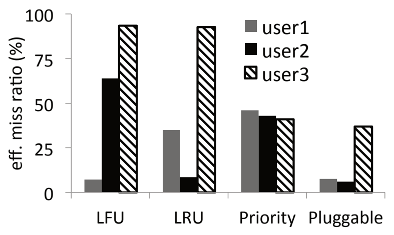
7.5 Facebook workload

Our trace-driven simulator performed a detailed and faithful replay of a task-level trace of Hadoop jobs collected from a 2000-node cluster from Facebook during the week of October 2010. Our replay preserved read and write sizes of tasks, locations of input data as well as job characteristics of failures, stragglers.

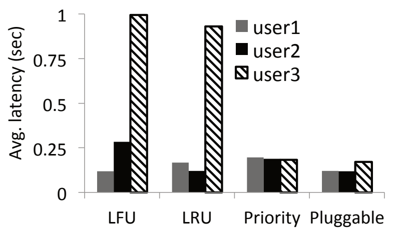
To make the effect of caching relevant to job completion time, we also use LIFE and LFU-F from PACMan [12] as cache replacement policies. These policies performed *all-or-nothing* cache replacement for files and can improve job completion time better than LRU or LFU, as it speeds all concurrent tasks in one stage [12]. In a nutshell, LIFE evicts files based on largest-incomplete-file-first eviction, and LFU-F is based on least-accessed-incomplete-file-first. We also set each

[3]Priority replacement means keeping a fixed set of files in cache. Not the best policy here, but still better than LFU and LRU for the scan workload.

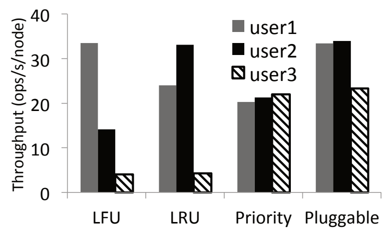
[4]Effective miss ratio. For FairRide, we count a delayed access as a "fractional" miss, with the fraction equal to the blocking probability, so we can effectively compare miss ratio between FairRide and other schemes.



(a) Effective Miss Ratio



(b) Average Latency

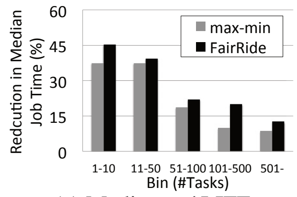


(c) Throughput

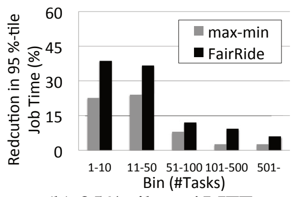
Figure 8: Pluggable policies.

	job time		cluster eff.		eff. miss%[4]	
	u1	u2	u1	u2	u1	u2
isolation	17%	15%	23%	22%	68%	72%
global	54%	29%	55%	35%	42%	60%
best-case	42%	41%	47%	43%	48%	52%
max-min	30%	43%	35%	47%	63%	46%
FairRide	39%	40%	45%	43%	50%	55%

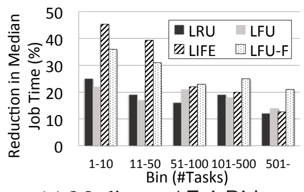
Table 2: Summary of simulation results on reduction in job completion time, cluster efficiency improvement and hit ratio under different scheme, with no caching as baseline.



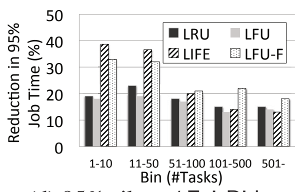
(a) Median, w/ LIFE



(b) 95%-tile, w/ LIFE



(c) Median, w/ FairRide



(d) 95%-tile, w/ FairRide

Figure 9: Overall reduction in job completion time for Facebook trace.

node in the cluster with 20Gb of memory so miss ratio was around 50%. The conclusion would hold for a wider range of memory size.

We adopted a more advanced model of cheating in this simulation. Instead of assuming users know what files are shared a priori, a user cheats based on the cached files she observes in the cluster. For example, for a non-blocking scheme such as max-min fairness, a user can figure out what shared files are cached by other users by continuously probing the system. She would avoid sharing the cost of those files and only cache files for her own interest.

Caching improves overall performance of the system. Table 2 provides a summary of reduction in job completion time and improvement in cluster efficiency (total task run-time reduction) compared to a baseline with no caching, as well as miss ratio numbers. Similar to previous experiments, *isolation* gave lowest gains for both users and *global* improved users unevenly (compared to *best-case*). FairRide suffered minimal overhead of blocking (2% and 3% in terms of miss ratio compared to *best-case*, 4% of cluster efficiency) but could prevent cheating of user 2 that can potentially hurt user 1 by 15% in terms of miss ratio. Similar comparisons were observed in terms of job completion and cluster efficiency, FairRide can outperform *max-min* by 27% in terms of efficiency and has 2.6× more improvement over *isolation*.

Figure 9 also shows the reduction in job completion time across all users, plotted in median completion time (a) and 95 percentile completion time (b) respectively. FairRide preserved better overall reduction compared to *max-min*. This was due to the fact that marginal improvement of the cheating user was smaller than the performance drop of the cheated. FairRide also prevented cheating from greatly increasing the tail of job completion time (95 percentile) as the metric was more dominated by the slower user. We also show the improvement of FairRide under different cache policies in (c) and (d).

7.6 Comparing Global FairRide

How much performance penalty does node-based FairRide suffer compared to global FairRide, if any? To answer this question, we ran another simulation with the Facebook trace to compare against two global FairRide schemes. The two global schemes both select a evicting user based on users' global usage, but differ in how they pick evicting blocks: a "naive" global scheme chooses from only blocks on that node, similar to the node-based approach, and an "optimized" global scheme chooses from any user blocks in the cluster. We use LIFE as the replacement policy for both users.

Cluster size	200	500	1000
Node-based FairRide	51%	44%	41%
Global FairRide, Naive	25%	21%	17%
Global FairRide, Optimized	54%	47%	44%

Table 3: Comparing against global schemes. Keep total memory size as constant while varying the number of nodes in the cluster. Showing improvement over no cache as in the reduction in median job completion time.

As we find out, the naive global scheme has a great performance drop (23%∼25% improvement difference compared to node-based FairRide), noticeably in Table 3. This is due to the fact that the naive scheme is unable to allocate in favor of frequently accessing user per node. With the naive global scheme, memory allocations on each node quickly stabilizes based on initial user accesses. A user can get an unnecessarily large portion of memory on a node because she accesses data earlier than the other, although her access frequency on that node is low in general. The optimized global scheme fixes this issue by allowing a user to evict least preferred data in the whole cluster and it makes sure the $1/n$-th of memory allocated must store her most preferred data. We observe an increase of average hit ratio by 24% with the optimized scheme, which reflects the access skew for the underlying data. What's interesting is that the optimized global scheme is only 3%∼4% better than node-based scheme in terms of job complete time improvement. In addition to the fact data skew is not huge (considering 24% increase for hit ratio), the all-or-nothing property of data-parallel caching again comes into play. Global scheme on average increases the number of completely cached files by only 7%, and because now memory allocation is skewed across the cluster, there is an increased chance that tasks cannot be scheduled to co-locate with cached data, due to CPU slot limitation. Finally, we also observe that as the number of nodes increases (while keeping the total CPU slots and memory constant), there is a decrease in improvement in all three schemes, due to less tasks can be scheduled with cache locality.

8 Related Works

Management of shared resources has always been been an important subject. Over the past decades, researchers and practitioners have considered the sharing of CPU [39, 15, 35, 40] and network bandwidth [28, 13, 17, 23, 33, 36], and developed a plethora of solutions to allocate and schedule these resources. The problem of cache allocation for better isolation, quality-of-service [24] or attack resilience [29] has also been studied under various contexts, including CPU cache [25], disk cache [31] and caching in storage systems [34].

One of the most popular allocation policies is *fair sharing* [16] or *max-min fairness* [27, 14]. Due to the nice properties, it has been implemented using various methods, such as round-robin, proportional resource sharing [39] and fair queuing [18], and has been extended to support multiple resource types [20] and resource constraints [21]. The key differentiator for our work from the ones mentioned above, is that we consider shared data. None of the works above identifies the impossibility of three important properties with shared files.

There are other techniques that have been studied to provide fairness and efficiency of shared cache. Prefetching of data into the cache before access, either through hints from applications [31] or predication [22], can improve the overall system efficiency. Profiling applications [25] is useful for provding application-sepcific information. We view these techniques as orthogonal to our work. Other techniques such as throttling access rate requires the system to identify good thresholds.

9 Conclusions

In this paper, we study the problem of cache allocation in a multi-user environment. We show that with data sharing, it is not possible to find an allocation policy that achieves isolation-guarantee, strategy-proofness and Pareto-efficiency simultaneously. We propose a new policy called FairRide. Unlike previous policies, FairRide provides both isolation-guarantee (so a user gets better performance than on isolated cache) and strategy-proofness (so users are not incentivized to cheat), by blocking access from cheating users. We provide an efficient implementation of the FairRide system and show that in many realistic workloads, FairRide can outperform previous policies when users cheat. The two nice properties of FairRide come at the cost of Pareto-efficiency. We also show that FairRide's cost is within 4% of total efficiency in some of the production workloads, when we conservatively assume users don't cheat. Based of the appealing properties and relatively small overhead, we believe that FairRide can be a practical policy for real-world cloud environments.

Acknowledgement

We thank Radhika Mittal, Shivaram Venkataraman, Rachit Agarwal and other members of Berkeley CS and AMPLab for providing many feedbacks on this work; Scott Shenker and Eric J. Friedman for providing guidance during early stage of the project; Srikanth Kandula for helpful feedbacks on the draft; and the anonymous reviewers for valuable comments; and finally, our shepherd, Thomas Moscibroda, for helping to shape the final version of the paper. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Intel, Microsoft, Pivotal, Samsung, Schlumberger, Splunk, State Farm, Virdata and VMware.

References

[1] Amazon elasticache. https://aws.amazon.com/elasticache/.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Azure cache - redis cache cloud service. http://azure.microsoft.com/en-us/services/cache/.

[4] Distributed Memory: Supporting Memory Stor-

age in HDFS. http://hortonworks.com/blog/ddm/.

[5] Hdfs caching. http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching/.

[6] Isolation in Memcached or Redis. http://goo.gl/FYfrOK;http://goo.gl/iocFrt;http://goo.gl/VeJHvs.

[7] Memcached, a distributed memory object caching system. http://memcached.org/.

[8] MemSQL In-Memory Database. http://http://www.memsql.com/.

[9] Redis. http://redis.io/.

[10] The column-store pioneer: MonetDB. https://www.monetdb.org.

[11] TPC-H. http://www.tpc.org/tpch.

[12] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. Pacman: coordinated memory caching for parallel jobs. In *NSDI'12*.

[13] Bennett, J. C., and Zhang, H. Wf2q: worst-case fair weighted fair queueing. In *INFOCOM'96*.

[14] Cao, Z., and Zegura, E. W. Utility max-min: An application-oriented bandwidth allocation scheme. In *INFOCOM'99*.

[15] Caprita, B., Chan, W. C., Nieh, J., Stein, C., and Zheng, H. Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems. In *ATC'05*.

[16] Crowcroft, J., and Oechslin, P. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *SIGCOMM CCR, 1998*.

[17] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM CCR, 1989*.

[18] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM'89*.

[19] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. Sap hana database: Data management for modern business applications. *SIGMOD Rec., 2012*.

[20] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. NSDI'11.

[21] Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Choosy: Max-min fair sharing for datacenter jobs with constraints. EuroSys'13.

[22] Gill, B. S., and Bathen, L. A. D. Amp: Adaptive multi-stream prefetching in a shared cache. In *FAST'07*.

[23] Goyal, P., Vin, H. M., and Chen, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *SIGCOMM CCR, 1996*.

[24] Iyer, R., Zhao, L., Guo, F., Illikkal, R., Makineni, S., Newell, D., Solihin, Y., Hsu, L., and Reinhardt, S. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS'07*.

[25] Kim, S., Chandra, D., and Solihin, Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. PACT'04.

[26] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC'14*.

[27] Ma, Q., Steenkiste, P., and Zhang, H. Routing high-bandwidth traffic in max-min fair share networks. In *SIGCOMM CCR, 1996*.

[28] Massoulié, L., and Roberts, J. Bandwidth sharing: objectives and algorithms. In *INFOCOM'99*.

[29] Moscibroda, T., and Mutlu, O. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security'07*.

[30] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS OSR, 2010*.

[31] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. *SIGOPS'95*.

[32] Piatek, M., Isdal, T., Anderson, T., Krishnamurthy, A., and Venkataramani, A. Do incentives build robustness in bit torrent. In *NSDI'07*.

[33] Shreedhar, M., and Varghese, G. Efficient fair queuing using deficit round-robin. *TON'96*.

[34] Shue, D., Freedman, M. J., and Shaikh, A. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI'12*.

[35] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S. K., Gehrke, J. E., and Plaxton, C. G. A proportional share resource allocation algorithm for real-time, time-shared systems. In *RTSS'96*.

[36] Stoica, I., Shenker, S., and Zhang, H. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *SIGCOMM'98*.

[37] Stonebraker, M., and Weisberg, A. The

voltdb main memory dbms. http://voltdb.com/.

[38] Verma, A., Pedrosa, L. D., Korupolu, M., Oppenheimer, D., and Wilkes, J. Large scale cluster management at google with borg. Eurosys'15.

[39] Waldspurger, C. A., and Weihl, W. E. Lottery scheduling: Flexible proportional-share resource management. In *OSDI'94*.

[40] Waldspurger, C. A., and Weihl, W. E. Stride scheduling: deterministic proportional-share resource management. In *MIT Tech Report, 1995*.

[41] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*.