Balancing Fairness and Efficiency for Cache Sharing in Semi-external Memory System

Shanjiang Tang
College of Intelligence and Computing,Tianjin University
Tianjin, China
tashj@tju.edu.cn

Qifei Chai
College of Intelligence and Computing,Tianjin University
Tianjin, China
chaiqifei@tju.edu.cn

Ce Yu
College of Intelligence and Computing,Tianjin University
Tianjin, China
yuce@tju.edu.cn

Yusen Li
School of Computer Science, Nankai University
Tianjin, China
liyusen@nbjl.nankai.edu.cn

Chao Sun
College of Intelligence and Computing,Tianjin University
Tianjin, China
sch@tju.edu.cn

ABSTRACT

Data caching and sharing is an effective approach for achieving high performance to many applications in shared platforms such as the cloud. DRAM and SSD are two popular caching devices widely used by many large-scale data application systems such Hadoop and Spark. Due to the limited size of DRAM as well as the large access latency of SSD (relative to DRAM), there is a trend of integrating DRAM and SSD (called semi-external memory) together for large-scale data caching.

In this paper, we focus on the semi-external memory cache sharing for multiple users/applications. Two critical metrics, i.e., *fairness* and *efficiency*, are considered for the semi-external memory with several challenges. First, it should be sensitive to DRAM and SSD for the semi-external memory in view of their different access latencies. Second, it is crucial to have a policy that can balance fairness and efficiency *elastically* since there tends to be a tradeoff between them. Third, there is a cheating problem for efficiency cache allocation and we should have a robust allocation policy to address it.

We propose a new policy called *ElasticSEM* for the semi-external memory. It performs the fair allocation of cache resources as a whole with the awareness of different access latencies between DRAM and SSD. Moreover, it contains a knob that allows users to tune and balance fairness and performance flexibly with a guarantee of θ-relaxed fairness, where θ-relaxed fairness refers to as the maximum difference of estimated cache resource allocations between any two users in SEM. Finally, we implement ElasticSEM in an in-memory storage system called Alluxio. The testbed experimental results show that ElasticSEM can achieve high performance and fairness.

ICPP'20, Aug 17–18, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **Information systems** → **Information storage systems**; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks*.

KEYWORDS

Semi-external Memory; Fairness; Efficiency; Cache Sharing; ElasticSEM; Cheating

ACM Reference Format:
Shanjiang Tang, Qifei Chai, Ce Yu, Yusen Li, and Chao Sun. 2020. Balancing Fairness and Efficiency for Cache Sharing in Semi-external Memory System. In *ICPP'20: 49th International Conference on Parallel Processing (ICPP).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In the current era of big data, data caching is an efficient approach for most large-scale data processing frameworks (e.g, Piccolo [28], M3R [32], Spark [34, 44]) and storage systems (e.g., Redis [12], Memcached [16], Tachyon [24]) to achieve high performance for data analytics. Given the fact that the memory (DRAM) I/O is generally several orders of magnitude faster than that of hard disks (HDD), in-memory caching solutions [16, 44] have been widely exploited as the main toolchain for high-performance data caching [30].

However, the cost of DRAM is much expensive compared to disks, increasing from a few thousand dollars to tens of thousands of dollars when it exceeds 64 GB per machine [10]. It constrains the capacity of DRAM severely subject to the limited budget. Compared to DRAM, flash memory has a larger capacity and lower cost. Due to these, Solid State Disks (SSD), an instance of NAND flash memory and Non-volatile memory (NVM), has become increasingly popular in recent years. It is filling the price/performance gap between DRAM and HDD. Given these, there is a new popular memory model called *Semi-External Memory (SEM)* [10, 27] that extends DRAM by integrating SSD together to overcome the capacity limitation of DRAM. It has been widely used by many applications [10, 27, 46, 47].

In this work, we consider the data caching for multiple users in SEM under a shared environment such as cloud. Compared to the

isolated cache allocation without sharing, cache sharing across users can improve the efficiency of SEM cache significantly. First, it can maximize the SEM utilization by allowing overloaded users to use the idle SEM resources from underloaded users. Second, instead of having multiple copies of the same data for multiple users in the isolated allocation, cache sharing only needs to keep one copy of the same data in SEM by enabling users to share the same data, which can save space for caching more data [29]. Third, the cache sharing provide opportunities to globally improve cache efficiency by replacing cached data of low access frequencies with those of high access frequencies.

Besides cache efficiency, fairness is another key factor concerned by users in the shared SEM cache system. *Max-min fairness* is one of the most prevalent fair allocation policies. It achieves fairness by maximizing the minimum resource allocation for users in the system [18, 36, 37]. It has been widely used for a variety of computer resources, including CPU [11, 39], GPU [7, 22] and network link [15, 17]. When it comes to the storage system, despite a number of fairness work available on DRAM [13, 23, 29] and SSD [31, 43], all of them study the fair resource allocation for each storage device separately. For SEM, a user's caching data are stored on both DRAM and SSD simultaneously. From a user's viewpoint, it is most likely that he/she only cares about the overall allocation and performance result rather than the *separate* resource allocation from each storage device. It indicates for SEM that we should take DRAM and SSD as a whole in the cache resource allocation for users.

However, there are several challenges in this regard. First, the SSD is multiple orders of magnitude slower than DRAM in data access latency [47]. It means that we should be aware of this and cannot simply treat DRAM as the same as SSD in resource allocation. Second, there is often a tradeoff between the fairness and efficiency in resource allocation according to prior work [21, 35]. Keeping 100% fairness strictly tends to result in low efficiency. Reversely, pursuing for a high efficiency is often at the cost of compromised fairness. It indicates that it is important to have a cache allocation policy that can balance such a tradeoff. Third, as we will show in Section 2, there can be a cheating problem for efficiency cache allocation in SEM. It is thus necessary to have a *robust* allocation policy that can disincentivize users to cheat.

We propose *ElasticSEM*, an elastic knob-based fairness-efficiency allocation policy, to concern with the tradeoff between fairness and efficiency for SEM. It allows users to balance the fairness and cache efficiency flexibly in SEM via a tunable knob argument in the range of $[0, 1]$. With a user's setting of knob value, ElasticSEM can maximize the cache efficiency while offering a QoS of θ-relaxed fairness guarantee, where θ is the maximum difference of estimated cache resource allocations between any two users in SEM. We particularly show that it *does not always have a strict* tradeoff between fairness and efficiency, and then present a mathmatical formula telling users how to set the knob value under such a *non-strict* tradeoff scenario. To the best of our knowledge, ElasticSEM is the first fair policy that integrates DRAM and SSD as a whole in cache resource allocation for SEM by having different weights to DRAM and SSD based on their data access latencies. Moreover, ElasticSEM is *robust* since it can automatically detect cheats and prevent cheating users from getting benefits. We have implemented ElasticSEM in Alluxio, a popular in-memory file sytem. The testbed experimental results illustrate that ElasticSEM is highly elastic and can achieve high efficiency and fairness in SEM.

The rest of the paper is organized as follows. Section 2 gives the background and motivation of the paper. Section 3 presents the semi-external memory cache model. Section 4 makes an introduction and analysis of ElasticSEM allocation policy, followed by the experimental evaluation in Section 5. We review the related work in Section 6. Finally, we conclude the paper in Section 7.

2 BACKGROUND AND MOTIVATION

Memory Cache. Memory cache is an essential and widely used system in serving many big data applications for high performance. Big companies like Google, Baidu and Alibaba have equipped thousands of cache servers that run a number of cache systems such as memcached [16], Redis [12] and Alluxio [24] for a wide variety of interactive and batch applications. Small companies can exploit caching services (e.g., ElasticCache [1], Redis Labs [3]) provided by cloud providers such as Amazon EC2 and Redis cloud for their big data applications.

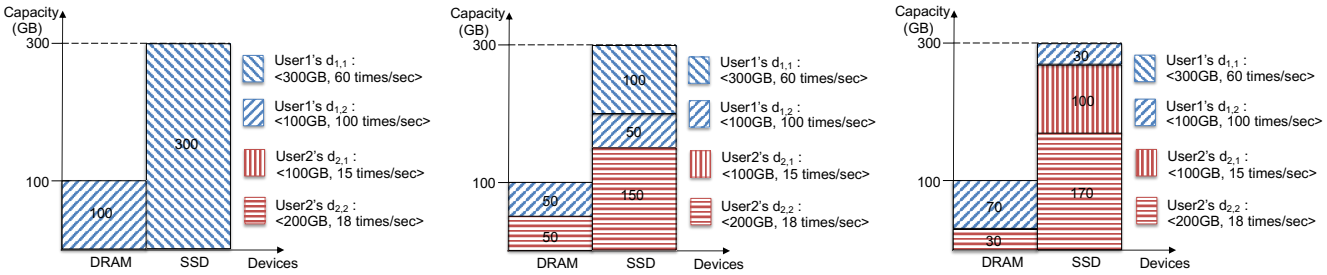
In face of 'big data', the size of DRAM is however often limited due to the high cost of DRAM as well as its high power consumption, restricting large-scale data applications from getting high in-memory hit-rates that is essential for high performance [10]. To resolve this limitation, flash memory such as solid-state drive (SSD) can be leveraged. Although SSD is multiple orders of magnitude slower in latency than DRAM, it has a larger capacity, lower cost and lower power requirement. It can be used as an extension of DRAM to form a new hybrid memory system called *Semi-external Memory*(SEM) for scaling the performance of large-scale data applications [6, 27, 46, 47].

Efficiency vs Fairness. We focus on SEM caching, which enables users to cache their data on DRAM and SSD. It has been supported by many existing in-memory caching systems such as fatcache [2] and Alluxio [24]. Typically, most of the above cache systems focus on the *global* system efficiency (i.e., maximize cache hit rates) and are oblivious to the entities (users) that access data. For example, web caches do not care about which user accesses a webpage. As a consequence, users who access data at a higher rate (i.e., contributing more to system efficiency improvement) would get more cache resources than the other users, resulting in unfairness.

To illustrate these points, consider the following examples for SEM allocation.

EXAMPLE 1. *Consider a SEM consisting of 100 GB DRAM and 300 GB SSD, where the latency ratio of DRAM to SSD is 1/6. It is shared by two users 1 and 2 equally. User 1 contains two data $d_{1,1}$ (size: 300 GB, access frequency: 60 times/sec) and $d_{1,2}$ (size: 100 GB, access frequency: 100 times/sec). User 2 has two data $d_{2,1}$ (size: 100 GB, access frequency: 15 times/sec) and $d_{2,2}$ (size: 200 GB, access frequency: 18 times/sec).*

Figure 1 (a) presents the SEM allocation result under the global sharing policy (e.g., LFU), which is taken by existing cache systems for efficiency maximization. It always chooses data with larger access frequencies and tries first to cache them in DRAM than SSD (We will show in Lemma 1 of later section that putting data with higher access frequencies in DRAM first can achieve better allocation efficiency than in SSD). The final allocation turns to



(a) Global resource sharing cache allocation. The total allocation efficiency is $13000(= 100 \cdot 100/1 + 60 \cdot 300/6)$**.**

(b) Separate max-min fairness cache allocation. The total allocation efficiency is $8180(= 100 \cdot 50/1 + 18 \cdot 50/1 + 100 \cdot 50/6 + 60 \cdot 100/6 + 18 \cdot 150/6)$

(c) Global max-min fairness cache allocation. The total allocation efficiency is $8800(= 100 \cdot 70/1 + 18 \cdot 30/1 + 100 \cdot 30/6 + 18 \cdot 170/6 + 15 \cdot 100/6)$**.**

Figure 1: Cache allocations for Example 1 under different allocation policies. The capacities of DRAM and SSD are 100 **and** 300**, respectively. The latency ratio of DRAM to SSD is 1/6.**

be that all data of User 1 are cached in SEM and but there is no data cached for User 2, resulting in *unfairness* problem for User 2 although it achieves the maximum overall cache efficiency (See formal definition and estimation of cache efficiency in Section 3) of $13000(= 100 \cdot 100/1 + 60 \cdot 300/6)$.

Max-min fairness is one of the most popular fairness policy. It achieves fairness by maximizing the minimum allocation across all users. To address the unfairness problem above, one natural solution is to perform max-min fairness *separately* for DRAM and SSD among users in SEM (named as *separate max-min fairness*). Figure 1 (b) illustrates the allocation result for Example 1 under the separate max-min fairness policy. User 1 and 2 both receive the same amount of DRAM and SSD cache resources in SEM while achieving a maximum cache efficiency of $8180(= 100 \cdot 50/1 + 18 \cdot 50/1 + 100 \cdot 50/6 + 60 \cdot 100/6 + 18 \cdot 150/6)$.

Another fairness approach is to perform max-min fairness *globally* across users by taking DRAM and SSD as a whole (named as *global max-min fairness*). In contrast to the separate max-min fairness allocation, it allows users to trade some DRAM resources for more SSD resources and vice versa across users for maximizing overall efficiency. Figure 1 (c) shows the allocation result of the global max-min fairness for Example 1. It allows users to trade 1G DRAM for 6G SSD according to the latency ratio of DRAM to SSD, and vice vera. Compared to Figure 1 (b), User 2 trades 20GB DRAM with User 1 for more 120GB SSD in Figure 1 (c) so that User 1 can cache more data of $d_{1,2}$ (with the most access frequency) in DRAM in order for efficiency maximization (according to Lemma 1). After trading, it keeps fairness for User 1 and 2 while getting an overall cache efficiency of $8800(= 100 \cdot 70/1 + 18 \cdot 30/1 + 100 \cdot 30/6 + 18 \cdot 170/6 + 15 \cdot 100/6$, larger than the separate max-min fairness in Figure 1 (b). It indicates for SEM that the global max-min fairness outperforms the separate max-min fairness in efficiency.

By comparing Figure 1 (a) with Figure 1 (b) (or Figure 1 (c)), it shows that there is a tradeoff between fairness and efficiency in SEM cache allocation. Moreover, although the global resource sharing policy gains the highest cache efficiency, it can bring the *cheating* problem for users compared with the separate max-min fairness and global max-min fairness policies. Going back to Example 1 with the global resource sharing policy (e.g., LFU), User 2 can *cheat* the SEM system to reclaim some of its cache back by artificially increasing its data access rate (e.g., increasing access rate of $d_{2,2}$ from 18 to 120 times/sec). While lying can help User 2 improve its cache efficiency, it decreases the overall cache efficiency (i.e., worse performance). Worse still, if every user does the same thing of artificially increasing its data access frequencies, it will lead to worse cache efficiency for every user than when acting truthfully. Therefore, the global resource sharing policy is not robust to users' cheating.

Summary. Through the examples above, we have made the following observations. 1) There is a tradeoff between fairness and efficiency in SEM cache resource allocation for users; 2) Global resource sharing policy such as LFU has a cheating problem; 3) Global max-min fairness is superior to separate max-min fairness in SEM allocation.

Thus, in this work, we seek to design a *flexible* and *robust* fairness-efficiency I/O scheduler for SEM cache that can balance fairness and efficiency while detecting and dealing with cheating problem by itself.

3 SEMI-EXTERNAL MEMORY CACHE MODEL

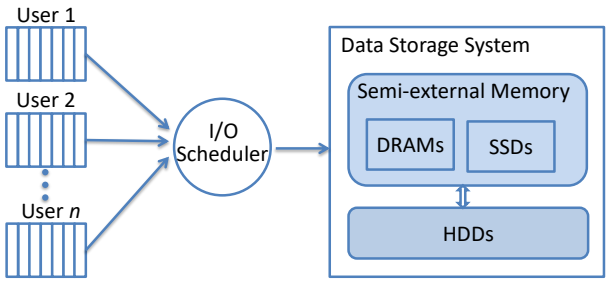


Figure 2: The semi-external memory cache model.

As illustrated in Figure 2, the storage system of the semi-external memory cache model consists of DRAMs, SSDs and HDDs arrays. The DRAM and SSD are independent cache devices without frequent data migrations among them. Let S_{DRAM}, S_{SSD} and

S_{HDD} (subject to $S_{DRAM} < S_{SSD} < S_{HDD}$) denote the storage capacities of DRAM, SSD and HDD respectively, and the data access latencies are referred to as t_{DRAM}, t_{SSD} and t_{SSD} (subject to $t_{DRAM} < t_{SSD} < t_{HDD}$) respectively.

A user makes a number of I/O requests to the SEM system. The target of each request is either the DRAM, SSD or HDD, and is known to the I/O scheduler. An access to the DRAM and SSD is referred to as a *cache hit* and access to the HDD is a *cache miss*. Let $\mathbf{d}_i = \{d_{i,1}, d_{i,2}, \cdots, d_{i,j}, \cdots\}$ denote the set of data currently cached for user i, and $l(d_{i,j})$ and $f(d_{i,j})$ represent the length (or size) and access frequency for User i's data $d_{i,j}$, respectively. Let $L(d_{i,j}) \in \{DRAM, SSD, HDD\}$ be the stored location for User's data $d_{i,j}$. The data access frequency and stored location for different applications are generally different and can be varying at runtime.

Suppose there are n users with the shared weights of $\mathbf{w} = \{w_1, w_2, \cdots, w_n\}$. The I/O requests of each user are stored in a user-specific queue from which they are dispatched to internal queues of the storage array by the I/O scheduler. The I/O scheduler is aware of the target device (DRAM, SSD or HDD) of a request, and dynamically determines where to cache the requested data (e.g., in DRAM or in SSD).

The cache model takes DRAM and SSD of SEM as a whole based on their different access latencies. Typically, we define and estimate the cache efficiency $\varphi(\mathbf{d}_i)$ for User i in SEM according to data access frequency and data size as follows,

$$\varphi(\mathbf{d}_i) = \sum_{d \in \mathbf{d}_i^{DRAM}} \frac{f(d) \cdot l(d)}{t_{DRAM}} + \sum_{d \in \mathbf{d}_i^{SSD}} \frac{f(d) \cdot l(d)}{t_{SSD}}, \quad (1)$$

where $\mathbf{d}_i^{DRAM} = \{d | d \in \mathbf{d}_i \wedge L(d) = DRAM\}$ and $\mathbf{d}_i^{SSD} = \{d | d \in \mathbf{d}_i \wedge L(d) = SSD\}$.

4 ELASTIC SEMI-EXTERNAL MEMORY ALLOCATION

This section describes an elastic fairness-efficiency resource allocation model that can balance the tradeoff between fairness and allocation efficiency flexibly as needed.

4.1 Allocation Model

We first define some terms used in the allocation model. The *fair share* of a user is referred to as the cache resources it obtains when each of the resources is split among all users equally. Let s_i represent the weighted fair share of User i. The total amount of DRAM and SSD cache resources for User i after equal partition are $S_{DRAM} \cdot \frac{w_i}{\sum_{1 \le j \le n} w_j}$ and $S_{SSD} \cdot \frac{w_i}{\sum_{1 \le j \le n} w_j}$, respectively. Due to the significant gap of data access latency between DRAM and SSD, we cannot simply treat DRAM and SSD the same. One fairness approach is to perform the max-min fairness for DRAM and SSD across users *separately*. However, we have shown in Section 2 that the *separate max-min fairness* is not efficient in performance. Alternatively, we can improve its performance via the *global max-min fairness* that combines DRAM and SSD as a whole in resource allocation for users (See Section 2). That is, we make $1/t_{SSD}$ DRAM resources trade for $1/t_{DRAM}$ SSD resources and vice versa according to their different access latencies. Then, we can compute the fair share s_i as

$$s_i = \frac{w_i}{\sum_{1 \le j \le n} w_j} \cdot \frac{S_{DRAM}}{t_{DRAM}} + \frac{w_i}{\sum_{1 \le j \le n} w_j} \cdot \frac{S_{SSD}}{t_{SSD}}. \quad (2)$$

Let H_i^{DRAM} and H_i^{SSD} denote the total amount of estimated allocations for user i from DRAM and SSD devices of the shared SEM, respectively. An allocation is called *fair* when the total estimated resources H_i (i.e., $H_i = H_i^{DRAM} + H_i^{SSD}$) obtained by every user $i \in [i, n]$ in the shared SEM system is proportional to its own fair share. That is, the fairness is achieved for the global max-min fairness when the following holds,

$$\frac{H_i}{s_i} = \frac{H_j}{s_j}, \forall i, j \in [1, n]. \quad (3)$$

However, the global max-min fairness only targets at 100% fairness, which is at the expense of the global cache efficiency significantly. As we have illustrated in Figure 1 (c), the allocation efficiency for the global max-min fairness is 8800, which is only $8800/13000 = 67.7\%$ of the global resource sharing policy as shown in Figure 1 (a). Reversely, seeking for the maximum allocation efficiency can result in *poor* fairness. In Figure 1 (a), it achieves the maximum allocation efficiency by making user 1 possess all SEM resources and no resources for User 2, which is however quite *unfair* for User 2. It indicates that there tends to have a tradeoff between fairness and allocation efficiency.

Moreover, for any cache allocation policy in SEM, we have a general guideline for its data caching as follows,

LEMMA 1. *It is more efficient to cache data of higher access frequency in DRAM than in SSD for any allocation policy.*

PROOF. Assume by the contradiction that the best cache efficiency occurs when there exists a data d_1 in SSD with a higher access frequency than a data d_2 in DRAM, i.e., $L(d_1) = SSD$, $L(d_2) = DRAM$ and $f(d_1) > f(d_2)$. Consider two data $d_1^{'}$ and $d_2^{'}$ of the same size, where $d_1^{'}$ and $d_2^{'}$ are sub-data of d_1 and d_2 (i.e., $d_1^{'} \subseteq d_1$ and $d_2^{'} \subseteq d_2$), respectively. It then holds $f(d_1^{'}) = f(d_1)$, $f(d_2^{'}) = f(d_2)$, and $l(d_1^{'}) = l(d_2^{'})$. The cache efficiency contributed by $d_1^{'}$ and $d_2^{'}$ can be estimated as

$$\varphi(d_1^{'} \bigcup d_2^{'} | L(d_1^{'}) = SSD \wedge L(d_2^{'}) = DRAM) = f(d_1^{'}) \cdot \frac{l(d_1^{'})}{t_{SSD}} + f(d_2^{'}) \cdot \frac{l(d_2^{'})}{t_{DRAM}}. \quad (4)$$

If we switch the locations of data $d_1^{'}$ and $d_2^{'}$ so that $d_1^{'}$ is located in DRAM and $d_2^{'}$ is in SSD while other data keep unchanged, then the cache efficiency contributed by $d_1^{'}$ and $d_2^{'}$ is

$$\varphi(d_1^{'} \bigcup d_2^{'} | L(d_1^{'}) = DRAM \wedge L(d_2^{'}) = SSD) = f(d_1^{'}) \cdot \frac{l(d_1^{'})}{t_{DRAM}} + f(d_2^{'}) \cdot \frac{l(d_2^{'})}{t_{SSD}}. \quad (5)$$

We can get that the value of Formula (5) is larger than Formula (4), violating the assumption and our proof completes. □

4.2 ElasticSEM Allocation Policy

We propose an elastic fairness-efficiency policy named *ElasticSEM* that enables users to balance the fairness and efficiency flexibly. Instead of pursuing for 100% fairness strictly as the global max-min fairness policy does in SEM cache, we compromise fairness for increased allocation efficiency by tolerating some degree of fairness loss. Typically, we categorize the fairness into two types, namely, *strict fairness* and *relaxed fairness*. The strict fairness means that the normalized allocation shares of all users should be equal (i.e., Formula (3) should be guaranteed). In contrast, the relaxed fairness

tolerates some degree (marked by θ) of unfairness between users. Formally, we define θ-relaxed fairness by changing Formula (3) as

$$\max_{1\le i,j\le n}\{\frac{H_i}{s_i}-\frac{H_j}{s_j}\}\le\theta. \tag{6}$$

The global max-min fairness focuses on the strict fairness across users, which is at the expense of the allocation efficiency dramatically. In contrast, ElasticSEM, as a fairness-efficiency tradeoff allocation policy, is interested in the relaxed fairness, which can leave some space for efficiency improvement.

4.2.1 ElasticSEM Design. The fairness-efficiency tradeoff allocation can be achieved with a mix of two phases allocations: *fairness-stage allocation* (i.e., purely for fairness optimization) and *efficiency-stage allocation* (i.e., purely for efficiency optimization). ElasticSEM first guarantees the relaxed fairness by performing the fairness-stage allocation with the global max-min fairness policy. Next, it performs the efficiency-stage allocation for efficiency maximization using the global resource sharing policy. To allow users to control and tune the tradeoff flexibly, ElasticSEM provides users with a knob $\sigma\in[0,1]$ to balance the two phases allocations flexibly. Let $\bar{H_i}$ and $H_i^{'}$ be the resulting allocations for ElasticSEM in the fairness-stage allocation and efficiency-stage allocation, respectively. Then, we have

$$H_i=\bar{H_i}+H_i^{'}. \tag{7}$$

In the phase of fairness-stage allocation, instead of guaranteeing the strict fairness of s_i for each user, ElasticSEM focuses on the relaxed fairness of $s_i\cdot\sigma$ (i.e., $\bar{H_i}=s_i\cdot\sigma$). Rewriting Formula (7), it holds

$$H_i=s_i\cdot\sigma+H_i^{'}. \tag{8}$$

According to Formula (2) and (8), the system then can leave $(\frac{S_{DRAM}}{t_{DRAM}}+\frac{S_{SSD}}{t_{SSD}})(1-\sigma)$ resources for efficiency-stage allocation. The small value of σ favors the efficiency optimization. In contrast, the large value of σ benefits for the fairness-stage allocation. Particularly, ElasticSEM reduces to the global max-min fairness when $\sigma=1$, and to the global resource sharing policy when $\sigma=0$.

After the minimum allocation of $s_i\cdot\sigma$ is guaranteed for each user i, the system moves to the phase of efficiency-stage allocation. In this phase, the global resource sharing policy (e.g., LRU, LFU) can be taken for efficiency optimization.

THEOREM 1. *ElasticSEM is a θ-relaxed fairness policy where* $\theta=\max_{1\le i\le n}\frac{(1-\sigma)\cdot\sum_{j=1}^{n}w_j}{w_i}$.

PROOF. According to the relaxed fairness definition, our proof is equivalent to finding a θ such that $\max_{1\le i\le n}\{\frac{H_i}{s_i}-\frac{H_j}{s_j}\}\le\theta$. For any two users $\forall i,j\in[1,n]$,

$$\begin{aligned}\max_{1\le i\le n}\{\frac{H_i}{s_i}-\frac{H_j}{s_j}\}&=\max_{1\le i,j\le n}\{\frac{s_i\cdot\sigma+H_i^{'}}{s_i}-\frac{s_j\cdot\sigma+H_j^{'}}{s_j}\}\\&=\max_{1\le i,j\le n}\{\frac{H_i^{'}}{s_i}-\frac{H_j^{'}}{s_j}\}\le\max_{1\le i\le n}\frac{H_i^{'}}{s_i}.\end{aligned} \tag{9}$$

Moreover, it holds

$$\sum_{i=1}^{n}s_i=\frac{S_{DRAM}}{t_{DRAM}}+\frac{S_{SSD}}{t_{SSD}}.$$

and

$$\begin{aligned}&0\le\sum_{i=1}^{n}H_i\le\frac{S_{DRAM}}{t_{DRAM}}+\frac{S_{SSD}}{t_{SSD}}\Rightarrow 0\le\sum_{i=1}^{n}\{s_i\cdot\sigma+H_i^{'}\}\le\frac{S_{DRAM}}{t_{DRAM}}+\frac{S_{SSD}}{t_{SSD}}\\&\Rightarrow 0\le\sum_{i=1}^{n}H_i^{'}\le(1-\sigma)\cdot\sum_{j=1}^{n}s_j\Rightarrow 0\le\max_{1\le i\le n}\frac{H_i^{'}}{s_i}\le\max_{1\le i\le n}\frac{(1-\sigma)\cdot\sum_{j=1}^{n}s_j}{s_i}\\&\Rightarrow 0\le\max_{1\le i\le n}\frac{H_i^{'}}{s_i}\le\frac{(1-\sigma)\cdot\sum_{j=1}^{n}w_j}{w_i}\\&\Rightarrow\max_{1\le i\le n}\{\frac{H_i}{s_i}-\frac{H_j}{s_j}\}\le\max_{1\le i\le n}\frac{(1-\sigma)\cdot\sum_{j=1}^{n}w_j}{w_i}.\end{aligned}$$

Therefore, ElasticSEM is a θ-relaxed fairness policy by letting $\theta=\max_{1\le i\le n}\frac{(1-\sigma)\cdot\sum_{j=1}^{n}w_j}{w_i}$. □

In summary, ElasticSEM is a knob-based hybrid of the global max-min fairness and global resource sharing policies, aiming at a θ-relaxed fairness guarantee determined by the configured knob σ. In the following, we describe the cache allocation procedure for ElasticSEM policy in detail.

ElasticSEM Allocation. Algorithm 1 shows the implementation of ElasticSEM. It maintains two user lists called *FairnessGuaranteedUserSet* and *FairnessNOTGuaranteedUserSet* (*Line 2-3*). The relaxed fairness of each user (determined by the knob σ) in the *FairnessGuaranteedUserSet* is guaranteed, whereas not for users in the *FairnessNOTGuaranteedUserSet*. When a user u accesses a data d, the system checks whether there is sufficient space to cache it. If not, it repeatedly evicts the data of users from SEM so that there is enough room for data d (*Line 4-14*). Every time, it chooses a user $u^{'}$ with the cached data $d^{'}$ of the lowest priority in SEM (*Line 5*) as an eviction candidate. There are two cases for not caching data d. The first case occurs when the relaxed fairness of user u has been satisfied and the priority of its data d is not larger than the cached data $d^{'}$ (*Line 6-7*). The second case can be that the candidate user $u^{'}$ is just the user u and it is possible for data d to not be actually cached when the priority of d is lower than that of candidate data $d^{'}$ (*Line 8-9*). The caching *priority* depends on the eviction policy. For example, the *priority* in LFU represents the access frequency of data, whereas priority in LRU denotes the inverse of the time interval since it has been accessed. Similarly, in the max-min fairness policy, the *priority* refers to as the decreasing order of their resource allocation. The eviction process actually starts when the user u's relaxed fairness is not guaranteed yet (for fairness-stage allocation) or the data d has a higher priority than that of the candidate data $d^{'}$ (for efficiency-stage allocation) (*Line 10-14*). Finally, the cache allocation of Algorithm 2 works when there are enough idle resources in SEM (*Line 15*).

Analysis of ElasticSEM. ElasticSEM allows users to balance the tradeoff between fairness and efficiency flexibly by tuning the knob value. In practice, different applications may have different tradeoff degrees. Due to this, we show in the following that there can be a value range of knob under which the allocation results keep the same, which we called *Knob Ineffective Range (KIR)* (denoted by $\rho\in[0,1]$).

The KIR ρ can be retrieved by analyzing the allocations results of pure efficiency-stage allocation for which a certain degree of relaxed fairness has already been guaranteed. Let H_i^0 be the resulting allocation for User i when knob $\sigma=0$ (i.e., pure efficiency-stage allocation). The value of H_i^0 depends on many factors including the distribution of data access frequencies and data sizes. We can see

Algorithm 1 Elastic Semi-external Memory Allocation (ElasticSEM).

```
function ElasticSEM(u, d)
    FairnessNOTGuaranteedUserSet = {i ∈ [1, n]|H_i < s_i · σ}.
    FairnessGuaranteedUserSet = {i ∈ [1, n]|H_i ≥ s_i · σ}.
    while DRAM.availableSize + SSD.availableSize < d.size do
        Choose User u' from FairnessGuaranteedUserSet containing a cached data d' of the lowest priority in SEM.
        if u ∈ FairnessGuaranteedUserSet AND d.priority ≤ d'.priority then
            return CACHE_ABORT.
        else if u = u' and d'.priority > d.priority then
            return CACHE_ABORT.
        else if u ∈ FairnessNOTGuaranteedUserSet OR d.priority > d'.priority then
            if d'.location = DRAM then
                DRAM.availableSize+= d'.size, d_i^DRAM-= d'.
            else if d'.location = SSD then
                SSD.availableSize+= d'.size, d_i^SSD-= d'.
    CacheAllocation(u, d).                                  ▷ Cache data d for user u.
```

Algorithm 2 Cache allocation function.

```
function CacheAllocation(u, d)
    A = DRAM.availableSize, D_i = d_i^DRAM.
    A' = SSD.availableSize, D'_i = d_i^SSD.
    if A ≥ d.size then                                      ▷ Cache data d in DRAM of SEM.
        A− = d.size, D_i+ = d.size.
    else if A < d.size then                                 ▷ Cache data d in both DRAM and SSD of SEM.
        Split d into two parts d = {d_1, d_2} satisfying that d_1.size = d.size − A, d_2 = d − d_1.
        A− = d_1.size, A'− = d_2.size, D_i+ = d_1, D'_i+ = d_2.
```

that every user can get at least a normalized share of $min_{1\le i\le n}\frac{H_i^0}{s_i}$ resources, implying that the pure efficiency-stage allocation can guarantee the relaxed fairness when $0 \le \sigma \le min_{1\le i\le n}\frac{H_i^0}{s_i}$. According to KIS definition, we have

$$\rho = min_{1\le i\le n}\frac{H_i^0}{s_i}. \tag{10}$$

The allocation result of KIR is the same as that of pure efficiency-stage allocation, i.e.,

$$H_i = H_i^0, \qquad (0 \le \sigma \le \rho). \tag{11}$$

If $\rho = 0$, there is no KIR, meaning that it has a strict 100% tradeoff between fairness and efficiency. In contrast, there is no tradeoff between fairness and efficiency if $\rho = 1$. Typically, when there are two users, we have:

THEOREM 2. *Let H_1^0 and H_2^0 be the resulting allocation for User 1 and User 2 when knob $\sigma = 0$ under ElasticSEM, respectively. Then it holds.*

$$\begin{cases} H_1 = H_1^0, H_2 = H_2^0, & (0 \le \sigma \le \min\{H_1^0/s_1, H_2^0/s_2\}) \\ H_1 = s_1\cdot\sigma, H_2 = S - s_1\cdot\sigma, & (H_1^0/s_1 \le H_2^0/s_2 \&\& H_1^0/s_1 < \sigma \le 1) \\ H_1 = S - s_1\cdot\sigma, H_2 = s_2\cdot\sigma, & (H_1^0/s_1 > H_2^0/s_2 \&\& H_2^0/s_2 < \sigma \le 1) \end{cases} \tag{12}$$

4.2.2 *Cheating Problem for ElasticSEM.*

So far, we have implicitly assumed for ElasticSEM that users are honest toward their data access frequency in SEM cache. However, in practice, users might game/cheat the system by spuriously increasing their data access frequency for caching more data in the phase of efficiency-stage allocation. In Section 2, we have shown that such a cheating behavior can make users get benefits under the global resource sharing policy. Similarly, for the ElasticSEM policy, we show in the following that the *cheating* problem does also exist.

Let revisit Example 1 to consider the cache allocation under the ElasticSEM policy, where the knob is $\sigma = 0.5$. Figure 3 presents the allocation results for ElasticSEM in two cases, namely, *no cheating* and *cheating*. When all users are honest, it achieves a global cache efficiency of $100\cdot 100/1 + 60\cdot 75/6 + 15\cdot 25/6 + 18\cdot 200/6 = 11412.5$ while guaranteeing a relaxed fairness of $|\frac{100/1+75/6}{50/1+150/6} - \frac{25/6+200/6}{50/1+150/6}| = 1$ as illustrated in Figure 3 (a). In contrast, if for example User 2 cheats by spuriously increasing its access rate of data $d_{2,2}$ to 101, the allocation result then turns to be Figure 3 (b). The global cache efficiency in this case reduces to $18\cdot 100/1 + 60\cdot 125/6 + 100\cdot 100/6 + 18\cdot 75/6 = 4941.67$ while guaranteeing a relaxed fairness of $|\frac{100/1+75/6}{50/1+150/6} - \frac{125/6+100/6}{50/1+150/6}| = 1$ compared with Figure 3 (a). Moreover, through cheating, User 2 gets more resources in Figure 3 (b)(e.g., 100/1+75/6=112.5) than that in Figure 3 (a) (e.g., 25/6+200/6=37.5). Thus, ElasticSEM is not robust and cheating can harm its global cache efficiency.

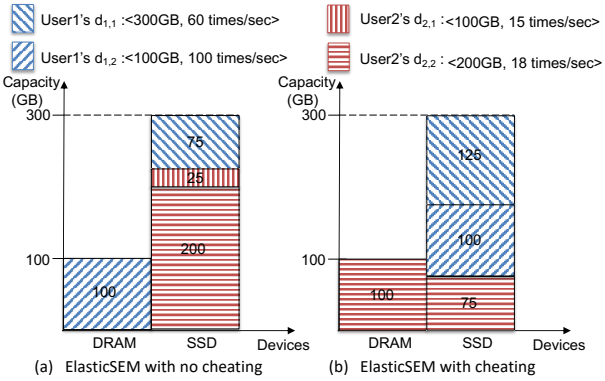


Figure 3: ElasticSEM allocation for Example 1 with and without cheating, where the knob $\sigma = 0.5$. In (b), user 2 makes spurious access to $d_{2,2}$ such that its access frequency exceeds $d_{1,2}$, which makes it obtain more resources in Figure 3 (b)(e.g., 100/1+75/6=112.5) than that in Figure 3 (a) (e.g., 25/6+200/6=37.5).

4.2.3 *Cheating Detection and Punishment.*

In Figure 3 (b), lying can help User 2 cache more data *without* any penalty. Intuitively, if there is a mechanism that can detect the cheating behavior of User 2 and penalize him at runtime, he will be disincentivized to cheat.

Typically, we consider two kinds of cheats for data caching. One cheat occurs when the data is located at the HDD (named *HDD-side Cheating*). A user might increase its access frequency spuriously so as to cache it on the SEM. An example of HDD-side Cheating is illustrated in Figure 3 (b). In contrast, another cheat can take place for *cached* data in SEM (named *SEM-side Cheating*). In order to prevent other users from replacing its cached data, a user can cheat by artificially increasing its access frequency. Both of cheats are harmful for cache efficiency and should be avoided.

Cheating Detection. We first need a mechanism that can dynamically differentiate a cheating and a well-behaved user. Due to the fact that the access frequencies of users' data in practice are often *varying* over time, it makes cheating detection become a challenging task. This is because apart from cheating users, the data access frequency of well-behaved users might also increase at runtime, making it hard for us to judge whether it is a cheating or well-behaved user.

To address it, we propose a *delay-based* cheating detector by assuming that users have no knowledge or information (e.g., the

owner of the data, data access frequency) about cached data in the SEM system. This assumption is reasonable in practice since caches in real-world systems are often transparent to end users/applications, and automatically maintained and managed by the (e.g., OS) systems. It monitors the access frequency of each data and then finds out cheating behavior based on recorded historical data access frequencies dynamically. Two cheating cases should be considered as follows.

The first case can be that in order to have its *non-cached* data to be cached into SEM as soon as possible, a cheating user might artificially increase its access frequency quickly. Then, there will be a big *jump* for its data access frequency increment. In this case, we can make cheating detection by searching such a 'jump'.

The second cheating case is more complex and harder to detect. In order to escape from being detected, a cheating user might try to *mimic* a honest user by increasing its data access frequency *smoothly* until its data is being cached. In this case, there is no 'jump' point for its data access frequency. To tackle it, we propose a *delay-based* detection approach based on previous assumption. Since users have no knowledge on the status of the SEM caching system, it is most likely that a cheating user would *continuously* increase its data access frequency in order for caching its data. However, it is not sufficient for us to judge whether such a user is cheating or not, since such an increasingly data access frequency does also exist for a well-behaved user in a *short* period of time (but not for a long time). In order to further check it, we delay to cache the data for a while (e.g., *delay_interval*). During this interval, if we observe that its data access frequency continues to become increasingly larger, it is assumed to be abnormal and treated as a cheating user. Through such a relative long period checking, we believe it can be effective to distinguish cheating and honest users (See experimental validation in Section 5).

Comparatively, the HDD-side cheating can be available in both cheating cases, whereas the SEM-side cheating is most likely to occur in the second cheating case.

Cheating Punishment. Penalization is an effective way to prevent users from cheating. However, penalization itself is often at the cost of the performance efficiency. Minimizing penalization cost is thus non-trivial in cheating prevention.

We change ElasticSEM policy by adding cheating detection and punishment, as shown in Figure 4. When a user i makes a data access request on $d_{i,j}$ (Arrow 1), we first check whether it has been cached or not. A cache hit returns if it has been cached (Arrow 2). Otherwise, it takes ElasticSEM policy depicted in Algorithm 1 to see whether $d_{i,j}$ should be cached or not (Arrow 3,4). If No, a cache miss will be given (Arrow 5). Otherwise, we move further to see whether its data caching occurs in the fairness-stage allocation or not (Arrow 6). If Yes, a cache hit returns directly and the data $d_{i,j}$ is cached with no cheating detection (Arrow 7). It is because in the fairness-stage allocation, lying does not bring any benefit for a cheating user, i.e., there is no incentiveness for a user to cheat in the fairness-stage allocation. Otherwise, it belongs to the efficiency-stage allocation (Arrow 8). Notably, if the replacing data $d^{'}$ and data $d_{i,j}$ are from the same user, there is no benefit for the cheating user. Worse yet, lying can even harm the cheating user himself when the access frequency of $d^{'}$ is larger than the *true* access frequency of data $d_{i,j}$. That is,

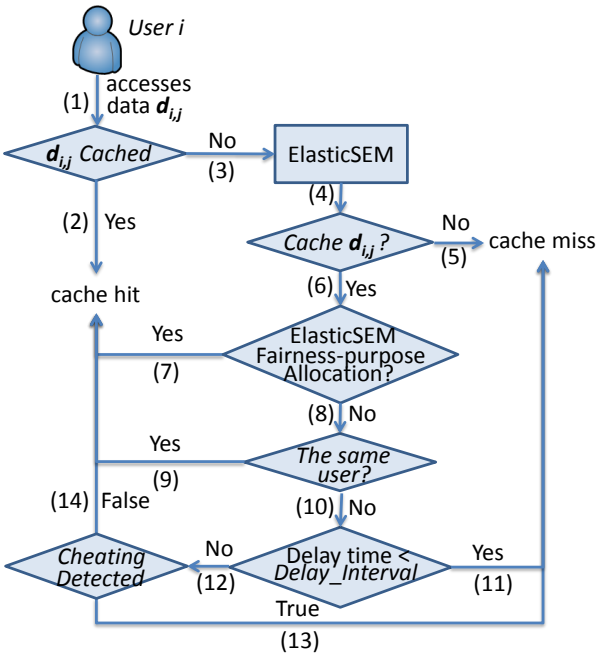


Figure 4: ElasticSEM policy with cheating detection and punishment mechanism.

it is disincentive for a user to cheat in this case. Thus, we make a cache hit by caching the data $d_{i,j}$ with no need of cheating detection (Arrow 9). Otherwise, lying can make a user get benefit and thus a cheating detection and punishment mechanism is needed in this case (Arrow 10).

We propose an adaptive delay-based punishment approach. It defines a term called *User Cheating Degree* (denoted as ψ_i) for each user i based on the number of data cheating it has made (denoted as m_i) over a monitoring time window T^w configured by users, i.e., $\psi_i = m_i$. Typically, a user i is honest when $\psi_i = 0$ and the larger value of ψ_i indicates the cheating user i is more likely to make cheat for its data caching request. Using it, we can make different punishments among cheating users by giving more penalties to those users with larger *User Cheating Degree*, which can be effective to stop users from cheating.

Specifically, there is an argument *Delay_Interval* (denoted as t_i^{del}) for each user i, which is proportional to *User Cheating Degree*, i.e.,

$$t_i^{del} = \Delta t^{del} \cdot (\psi_i + 1), \tag{13}$$

where Δt^{del} is a minimum delay time configured by users. In our experiment below, we initialize Δt^{del} to be the disk bandwidth, which is estimated by running a file of unit size. When a user's data satisfies the caching condition in the efficiency-stage allocation, we will delay it for *Delay_Interval* for two purposes (Arrow 11). One is to act as a punishment for a user based on his cheating history by giving it a cache miss without caching the data before its delay time exceeds the *Delay_Interval* (Arrow 11). The other is for cheating detection by delaying it using previous delay-based cheating detector. It analyzes whether it is a cheating data or not after *Delay_Interval* (Arrow 12). If cheating detection is true, it returns

a cache miss without caching the data (Arrow 13). Moreover, the system maintains a data cheating blocklist by putting all cheating data detected during a time window T^w into it. As a punishment, any cheating data in the data cheating blocklist will not be considered in the later data caching. Otherwise, the data will be cached by returning a cache hit (Arrow 14).

5 EXPERIMENTAL EVALUATION

We have implemented ElasticSEM in Alluxio-1.4.0, and evaluated ElasticSEM policy using both micro- and macro- benchmarks in an Alluxio cluster.

5.1 Experimental Setup

Alluxio Cluster. We deploy the Alluxio framework in a cluster consisting of 11 computing nodes each with 8 CPU cores and 16 GB memory. For each node, we configure 4GB memory as DRAM cache and use 8GB memory to emulate SSD cache. We set one machine as the master and the remaining 10 machines as slaves.

Macro-benchmark. We evaluate ElasticSEM by running three workloads:

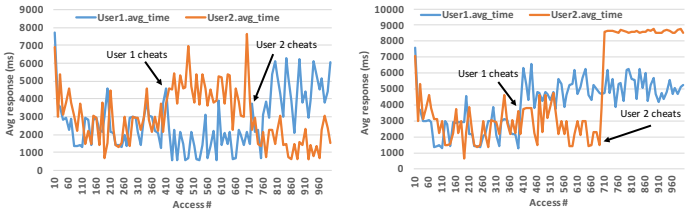
- *Synthetic Facebook Workload*. We synthsize Facebook workload according to the distribution of job submission time, input data bytes and input data access frequency derived from SWIM's Facebook workload traces (e.g., FB-2010_samples_24_times_1hr_withInputPaths_0.tsv) [5]. We found that the data access of Facebook workload trace complies with Zipf distribution. The jobs are from Hive benchmark [4], consisting of four applications, i.e., uservisits aggregation, grep search (selection), rankings-uservisits join and rankings selection.
- *Purdue Workload*. We have generated over 30 datasets, each of 1 GB based on Wikipedia data. Five benchmarks (e.g., WordCount, Grep, Inverted-index, Term-Vector and Multi-wordcount) are randomly chosen from Purdue benchmarks suite [8] to access these data for computation.
- *TPC-H Workload*. The TPC-H benchmark contains a set of analytic queries for users' decision support. We have generated over 300 TPC-H datasets, each of 200 MB. Each dataset consists of eight separate and individual tables, ranging from 10KB to 80MB.

Micro-benchmark. We assume there are two users with equal share of SEM cache resources. Each user accesses 40 files in the system. We assume that users knew a priori which files are currently cached in the system, and could game the sytem by making excessive accesses those files they want to cache.

In the following, we use the macro-benchmarks to evaluate the performance of ElasticSEM policy (Section 5.2.3), while evaluating the cheating problem as well as fairness (Section 5.2.1) and efficiency tradeoff with micro-benchmarks (Section 5.2.2).

5.2 Experimental Results

5.2.1 Cheating and Punishment. In this section, we start with micro-benchmarks to illustrate that ElasticSEM can dis-incentivize users to cheat, whereas the global resource allocation policy cannot. The LFU is adopted as the cache replacement policy for this experiment. We configure the knob of ElasticSEM to be zero for pure efficiency resource allocation.



(a) Global resource sharing allocation with LRU. **(b) ElasticSEM allocation with Knob $\rho = 0$.**

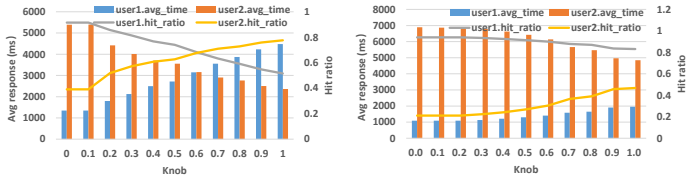
Figure 5: The average response time measured for two users under different allocation policies. User 1 starts cheating at the 400^{th} acess. User 2 started cheating at the 700^{th} access.

Figure 5 illustrates the experimental results for global resource allocation and ElasticSEM. The two allocation policies achieve a close average response time (2790ms under the global resource allocation, 2860ms under the ElasticSEM) before the 400^{th} data access. However, for global resource allocation as shown in Figure 5 (a), User 1 can manange to reduce its average response time by about 1100ms when he (or she) cheats at the 400^{th} access, degrading the performance of User 2 about 1500ms. Likewise, User 2 can also improve its cache hit (by about 1000ms) at the expense of User 1's performance by cheating the system at the 700^{th} access. It indicates that cheating users can get *benefits* under the global resource allocation at the expense of honest users. The reason is that there is a lack of cheating detection and punishment mechanism for global resource allocation, making cheating users be able to cache more data (i.e., more *benefits*) than they should have.

In comparison, ElasticSEM can prevent cheating users from getting benefits. Figure 5 (b) presents the allocation results for ElasticSEM. When User 1 starts to cheat at the 400^{th} access, it gets worse performance than no cheating period (i.e., $< 400^{th}$ access). Similarly, User 2 gets degraded performance at the 700^{th} access when it cheats. This is because ElasticSEM is equipped with a cheating detection and punishment mechanism, under which cheating behavior can be detected and will be punished by delaying its data access (See Section 4.2.3).

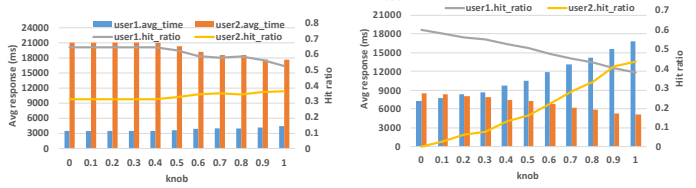
5.2.2 Fairness vs Efficiency under Different Knobs. Recall in Section 4.2.1 that ElasticSEM is an elastic knob-based cache allocation policy that can flexibly balance fairness and efficiency across users. In this section, we show the impact of knob configuration on the system efficiency and fairness. Particularly, as we will show below, the tradeoff balance of knob configuration is sensitive to the *file size* distribution and *access pattern* distribution of users' data.

We consider two users with two kinds of distributions on the file size as well as file access pattern, i.e., skewed and non-skewed distributions. Given that many data in practice follows Zipf distribution in production cluster [9, 30], we assume for skewed distribution that it follows Zipf distribution with an exponent parameter of 0.9. For non-skewed distribution, we assume that it follows the uniform distribution. Assume that there are 40 files for each user, and one user has a larger number of data accesses than another user. The cache sizes for DRAM and SSD are set to be 10 GB and 30GB, respectively. We consider four possible workloads with different file access patterns



(a) User 1 has 40 files of 1 GB each. It has 5000 data accesses in total and the data access complies with uniform distribution. User 2 has 40 files of 1 GB each. It has 1000 data accesses in total and the data access complies with Zipf distribution.

(b) User 1 has 40 files of 1 GB each. It has 5000 data accesses in total and the data access complies with Zipf distribution. User 2 has 40 files of 1 GB each. It has 1000 data accesses in total and the data access complies with uniform distribution.



(c) User 1 has 40 files of 1 GB each. It has 5000 data accesses in total and the data access complies with uniform distribution. User 2 has 40 files of different sizes. It has 1000 data accesses in total and the data access complies with Zipf distribution, where we assume that its hot data are of large data blocks.

(d) User 1 has 40 files of different Sizes. It has 5000 data accesses in total and the data access complies with Zipf distribution, where we assume that its hot data are of large data blocks. User 2 has 40 files of 1 GB each. It has 1000 data accesses in total and the data access complies with uniform distribution.

Figure 6: The system efficiency for User 1 and User 2 under different knobs configurations. The cache volume of SEM system is set to 10GB for DRAM and 30GB for SSD, respectively. We particularly show that the sensitivity of knob configuration on the tradeoff between fairness and efficiency is related to the cached data distribution and their sizes.

and file sizes distributions. Both average response time and cache hit ratio are presented for different knob configurations as illustrated in Figure 6.

First, depending on file sizes and data access distributions, there are different KIRs ρ (e.g., $\rho = 0.1, 0.3, 0.4$ and 0 for Figure 6 (a)~6 (d), respectively) for four workloads. In Figure 6 (a), its $\rho = 0.1$ since although User 2's total data accesses is smaller than User 1, the Zipf data access distribution of User 2 makes it still have some popular files whose access frequencies are larger than User 1 of uniform data access distribution. In contrast, by exchanging the file access distribution of User 1 and User 2 as shown in Figure 6 (b), its ρ enlarges to be 0.3. It is because under the Zipf distribution, User 1 has some non-popular files whose file access frequencies are smaller than User 2 of uniform distribution. Figure 6 (c) and 6 (d) are contrast workloads to Figure 6 (a) and 6 (b) with respect to skewed data sizes, respectively. It shows us that besides the file access frequency, the data size distribution does also have an impact on the ρ (i.e., tradeoff degree).

Second, ElasticSEM can balance the tradeoff between fairness and efficiency flexibly. For example, in Figure 6 (a), when $0 \leq \sigma \leq 0.1$, there is no tradeoff between (relaxed) fairness and efficiency so varying the knob σ has no impact on average response time and cache hit ratio. However, when $0.1 < \sigma \leq 1$, there is a tradeoff between (relaxed) fairness and efficiency and our knob-based ElasticSEM can balance such a tradeoff by looking at the trend of hit ratio or average response time under different knob configurations. e.g., the hit ratio of User 1 with uniform distribution decreases to 0.512 when we increase its knob up to one, which is close to the ideal hit ratio of $0.5 = 20/40$ for User 1 in the non-sharing scenario (i.e., 20 GB cache volume, 40 GB data).

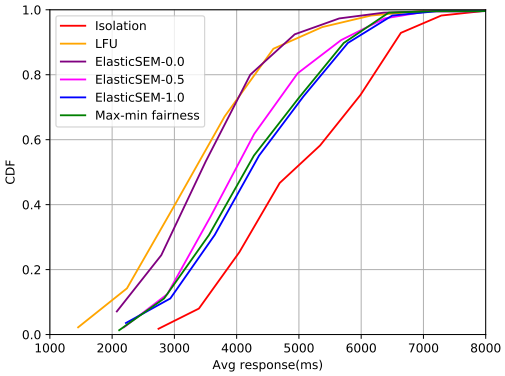


Figure 7: The CDF of average response time for various cache allocation policies.

5.2.3 Performance Comparison. This section comes to evaluate the performance of ElasticSEM with the macro-benchmarks. Figure 7 gives the CDF of average response time for Isolation (non-cache sharing), LFU (global resource sharing), max-min fairness and ElasticSEM under different knob configurations.

First, cache sharing (e.g., LFU, ElasticSEM, max-min fairness) can have a better performance than isolation. For max-min fairness policy, more than 70% of data accesses whose response time is within 5000ms, whereas only 51% for isolation. In comparison, ElasticSEM can further improve its CDF up to about 90% when we decrease its knob from 1.0 to 0.0. The performance improvement for cache sharing mainly attributes to the resource preemption of unused resources by over-demanded users from under-demanded users. In this example, Purdue workload would yield its unused 10GB data share to other overloaded users in the sharing scenario, improving the cache resource utilization and hereby the overall performance compared to the non-sharing case (Isolation).

Second, for ElasticSEM, we can improve the performance efficiency by decreasing the knob value. When the knob $\sigma = 1.0$, its CDF curve of response time is much close to that of max-min fairness. Moreover, when the knob $\sigma = 0$, its CDF curve becomes close to that of LFU. All of the two indicate that 1) ElasticSEM can balance the fairness and performance via the knob tuning and 2) ElasticSEM is highly efficient in performance.

5.2.4 Overhead Evaluation. Compared to traditional heursitic cache allocation policies such as LRU and LFU, ElasticSEM is much more complex since it integrates DRAMs and SSDs as a whole in SEM cache allocation and involves fairness-stage allocation and efficiency-stage allocation for a given knob. Moreover, ElasticSEM

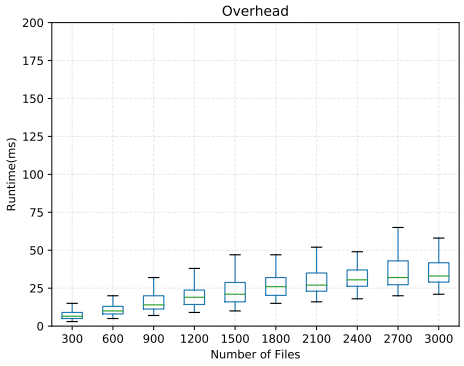


Figure 8: The overhead of ElasticSEM in Alluxio.

is equipped with a mechansim that can prevent users from cheating. This section evaluates the overhead of ElasticSEM with different number of files, where the *overhead* refers to the time for ElasticSEM making a decision of whether or how to cache the data since a I/O request is submitted.

We consider 20 users with a total number of files from 300 to 3000 with each of 256MB. Figure 8 presents the overhead (time) of ElasticSEM to make decision on data caching in SEM. First, it illustrates that the overhead time of ElasticSEM increases *slightly* with a more number of files. Second, compared to the data r/w time that generally takes seconds as shown in the aforementioned experiments, the overhead of ElasticSEM is minor and negligible. In summary, ElasticSEM is a *lightweight* fairness-efficiency I/O scheduler.

6 RELATED WORK

Fairness-efficiency Resource Allocation. In the literature, there are a large body of studies on the tradeoff between fairness and efffficiency in multi-resource allocation. Joe-Wong et al. [20] captured the tradeoff between fairness and efficiency by proposing a unifying *mathematical* framework for multi-resource allocation, which is however purely theoretical and cannot be directly applied for real systems. In comparison, our proposed ElasticSEM is practical and has implemented in Alluxio. QKnober [38] and Tetris [19] are both knob-based fairness-efficiency schedulers for big data processing systems such as Hadoop by considering CPUs and memory resources. Danna et al. [14] and Wang et al. [41, 42] studied the fairness-efficiency for packet processing in multi-resource allocation, where CPU and link bandwidth are considered. All of the above work focused on the multi-resources of different types (e.g., CPU, memory, bandwidth), whereas we consider the multi-resources of the same type (i.e., storage resource).

There are also some work focused on the multi-resources of the same types. Our prior work [35] studied the fairness and efficiency in Coupled CPU-GPU architecture [45] by proposing a knob-based scheduler called EMRF, where the alloctions of computing resources such as CPU and GPU are considered. In contrast, this work focused on the storage resources of DRAM and SSD. Wang et al. [40] proposed a bottleneck-aware allocation policy for multi-tiered storage consisting of SSD and HDD to balance fairness and efficiency for users, where SSD plays a cache role. However, they only focused on the I/O allocation without considering the amount of cache resources allocated. Also, their approach are not flexible, where users cannot change the tradeoff as needed. In contrast, we focused on SEM, whose DRAM and SSD both act as caches. Our proposed ElasticSEM is a knob-based fairness-efficiency I/O scheduler that provides users with a knob parameter to flexibly balance the tradeoff between fairness and efficiency.

Semi-external Memory. To overcome the capacity limitation of DRAM for big data applications, many existing studies [6, 10, 27, 46, 47] instead take semi-external memory as an alternative and show good performance results. Badam et al. [10] provided a hybrid SSD/RAM memory management system named SSDAlloc that extends the DRAM with SSDs for new and existing applications in a system. Abello et al. [6] proposed an semi-external computing model for graph data applications by fitting the vertex set in memory while putting edge set of a graph in SSD. Pearce et al. [27] experimentally demonstrated the benefits of using semi-external memory with its proposed asynchronous graph traversal approach compared to a serial in-memory alternative. FlashGraph [46] is a semi-external memory-based graph-processing engine adopting the concept of putting vertex state in memory and edge lists on SSDs. It outperforms its in-memory implementation by up to 80% as well as PowerGraph, which is a well-known distributed in-memory graph engine. To ease the graph programming and I/O optimization in SEM, an extensible parallel SEM graph library called Graphyti [26] is built on top of FlashGraph. Graphmp [33] is an efficient semi-external-memory big graph processing system for a single machine. Zheng et al. [47] studied sparse matrix multiplication in semi-external memory by putting the sparse matrix on SSDs and dense matrices in memory. Mhembere et al. [25] developed a NUMA-optimized in memory, distributed and semi-external memory library called knor for k-means algorithm atop of FlashGraph. In constrast to previous studies that focused on the performance optimization for SEM applications, we consider the tradeoff balancing between fairness and efficiency for cache sharing in SEM by integrating DRAM and SSD as a whole in its cache resource allocation across multiple users/applications. Moreover, we find that there is a cheating problem for efficiency cache allocation and address it in our proposed ElasticSEM policy.

7 CONCLUSION

Semi-external memory has been widely used as a cache for many big data applications, given that it can overcome the capacity limitation of DRAM by extending it with SSD. We particularly show that it is crucial to take DRAM and SSD as a whole for fairness allocation rather than separately for each cache device as previous studies did. Fairness and efficiency are two critical metrics for users in resource allocation, which however has a tradeoff between each other. We propose a knob-based fairness-efficiency allocation policy called ElasticSEM, consisting of fairness-stage allocation and efficiency-stage allocation, to allow users to balance such a tradeoff flexibly for semi-external memory while guaranteeing the θ-relaxed fairness under a given knob (See Theorem 1). It integrates DRAM and SSD as a whole wih the awareness of different latencies between DRAM and SSD by having different weights to them. We identify the cheating problem in the efficiency-stage allocation and propose

a cheating detection and punishment mechanism to address it. We implement ElasticSEM in Alluxio and our experiments demonstrate the effectivness of our approach.

Finally, we want to claim that although our approach in this paper focuses on the semi-external memory, its idea is general and can be directly applied to other heterogenous cache devices such as DRAM/NVM and cache systems including Memcached and Redis.

8 ACKNOWLEDGMENTS

This work was funded by National Key Research and Development Program of China (2018YFB0204305).

REFERENCES

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

[2] Memcache on ssd. In *https://github.com/twitter/fatcache*.

[3] Redis labs. In *https://redislabs.com/*.

[4] Apache hive performance benchmarks. In *https://issues.apache.org/jira/browse/HIVE-396*, 2009.

[5] Swim. In *https://github.com/SWIMProjectUCB/SWIM/tree/master/workloadSuite*, 2010.

[6] James Abello, Adam L Buchsbaum, and Jeffery R Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[7] P. Aguilera, K. Morrow, and N. S. Kim. Fair share: Allocation of gpu resources for both performance and fairness. In *ICCD'14*, pages 440–447, Oct 2014.

[8] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and T. N. Vijaykumar. Puma: Purdue mapreduce benchmarks suite. In *ECE Technical Reports*, 2012.

[9] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *NSDI'12*, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[10] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *NSDI'11*, pages 211–224, Berkeley, CA, USA, 2011. USENIX Association.

[11] Bogdan Caprita, Jason Nieh, and Clifford Stein. Grouped distributed queues: Distributed queue, proportional share multiprocessor scheduling. In *PODC '06*, pages 72–81, New York, NY, USA, 2006. ACM.

[12] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[13] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant memory key-value cache. *CoRR*, abs/1610.08129, 2016.

[14] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *INFOCOM'12*, pages 846–854. IEEE, 2012.

[15] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89*, pages 1–12, New York, NY, USA, 1989. ACM.

[16] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[17] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *SIGCOMM '12*, pages 1–12, New York, NY, USA, 2012. ACM.

[18] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI'11*, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[19] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[20] Carlee Joe-Wong, Soumya Sen, Tian Lan, and Mung Chiang. Multi-resource allocation: Fairness–efficiency tradeoffs in a unifying framework. *IEEE/ACM TON'13*, 21(6):1785–1798, 2013.

[21] Carlee Joe-Wong, Soumya Sen, Tian Lan, and Mung Chiang. Multiresource allocation: Fairness-efficiency tradeoffs in a unifying framework. *IEEE/ACM Trans. Netw.*, 21(6):1785–1798, December 2013.

[22] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *GPGPU-7*, pages 1:1–1:8, New York, NY, USA, 2014. ACM.

[23] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. ROBUS: fair cache allocation for multi-tenant data-parallel workloads. *CoRR*, abs/1504.06736, 2015.

[24] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC '14*, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[25] Disa Mhembere, Da Zheng, Carey E Priebe, Joshua T Vogelstein, and Randal Burns. knor: A numa-optimized in-memory, distributed and semi-external-memory k-means library. In *HPDC'17*, pages 67–78, 2017.

[26] Disa Mhembere, Da Zheng, Carey E Priebe, Joshua T Vogelstein, and Randal Burns. Graphyti: A semi-external memory graph library for flashgraph. *arXiv preprint arXiv:1907.03335*, 2019.

[27] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC '10*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[28] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI'10*, pages 293–306, Berkeley, CA, USA, 2010. USENIX Association.

[29] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. Fairride: Near-optimal, fair cache sharing. In *NSDI'16*, pages 393–406, Berkeley, CA, USA, 2016. USENIX Association.

[30] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *OSDI'16*, pages 401–417, Berkeley, CA, USA, 2016. USENIX Association.

[31] Kai Shen and Stan Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *USENIX ATC'13*, pages 67–78, Berkeley, CA, USA, 2013. USENIX Association.

[32] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012.

[33] Peng Sun, Yonggang Wen, Ta Nguyen Binh Duong, and Xiaokui Xiao. Graphmp: An efficient semi-external-memory big graph processing system on a single machine. In *ICPADS'17*, pages 276–283. IEEE, 2017.

[34] Shanjiang Tang, Bingsheng He, Ce Yu, Yusen Li, and Kun Li. A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

[35] Shanjiang Tang, BingSheng He, Shuhao Zhang, and Zhaojie Niu. Elastic multi-resource fairness: balancing fairness and efficiency in coupled cpu-gpu architectures. In *SC'16*, pages 875–886. IEEE, 2016.

[36] Shanjiang Tang, Bu-Sung Lee, and Bingsheng He. Fair resource allocation for data-intensive computing in the cloud. *IEEE Transactions on Services Computing*, 11(1):20–33, 2018.

[37] Shanjiang Tang, Zhaojie Niu, Bingsheng He, Bu-Sung Lee, and Ce Yu. Long-term multi-resource fairness for pay-as-you use computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):1147–1160, 2018.

[38] Shanjiang Tang, Ce Yu, Chao Sun, Jian Xiao, and Yinglong Li. Qknober: a knob-based fairness-efficiency scheduler for cloud computing with qos guarantees. In *ICSOC'18*, pages 837–853. Springer, 2018.

[39] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI '94*, Berkeley, CA, USA, 1994. USENIX Association.

[40] Hui Wang and Peter Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *FAST'14*, pages 229–242, 2014.

[41] Wei Wang, Chen Feng, Baochun Li, and Ben Liang. On the fairness-efficiency tradeoff for packet processing with multiple resources. In *ACM CoNext14*, pages 235–248, 2014.

[42] Wei Wang, Shiyao Ma, Bo Li, and Baochun Li. Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling. In *INFOCOM'17*, pages 1–9. IEEE, 2017.

[43] Minhoon Yi, Minho Lee, and Young Ik Eom. Cffq: I/o scheduler for providing fairness and high performance in ssd devices. In *IMCOM '17*, pages 87:1–87:6, New York, NY, USA, 2017. ACM.

[44] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[45] Feng Zhang, Jidong Zhai, Bo Wu, Bingsheng He, Wenguang Chen, and Xiaoyong Du. Automatic irregularity-aware fine-grained workload partitioning on integrated architectures. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2019.

[46] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *FAST'15*, pages 45–58, Berkeley, CA, USA, 2015. USENIX Association.

[47] Da Zheng, Disa Mhembere, Vince Lyzinski, Joshua T. Vogelstein, Carey E. Priebe, Randal Burns, undefined, undefined, undefined, and undefined. Semi-external memory sparse matrix multiplication for billion-node graphs. *IEEE TPDS'17*, 28(5):1470–1483, 2017.