Stateful Dominant Resource Fairness: Considering the Past in a Multi-Resource Allocation

Hugo Sadok, Miguel Elias M. Campista and Luís Henrique M. K. Costa
GTA/PEE/COPPE – Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil
Email: {sadok, miguel, luish}@gta.ufrj.br

***Abstract*—The multi-resource allocation problem arises in different scenarios, from cloud computing systems to shared clusters. Users often have heterogeneous demands and dynamic workloads. Different mechanisms were proposed to fairly divide multiple resources, most notably, Dominant Resource Fairness (DRF). Even though DRF satisfies several desirable properties, it considers fairness only in the static setting. We propose Stateful DRF (SDRF), an extension of DRF that looks at past allocations and enforces fairness in the long run while keeping the fundamental properties of DRF. We prove that SDRF is strategyproof, since users cannot manipulate the system by misreporting their demands; incentivizes sharing, because no user is better off if resources are equally partitioned; and is efficient, as no allocation can be improved without decreasing another. In SDRF, user priorities change over time. To avoid recalculating priorities at every task scheduling decision, we also propose Live Tree, a general-purpose data structure that keeps elements with predictable time-varying priorities ordered. We conduct large-scale simulations based on Google cluster traces of 30 million tasks over one month. Results show that SDRF reduces users' waiting time on average. This improves fairness, by increasing the number of completed tasks for users with lower demands, with negligible impact on high-demand users.**

I. Introduction

Resource allocation is a fundamental problem in any shared computing system. Cloud computing centers and modern clusters are usually shared by users with different resource constraints [1]–[3]. The amount of resources given to each user directly impacts the system performance from both fairness and efficiency standpoints [4]. In single-resource systems, max-min fairness is the most widely used and studied allocation policy [5], [6]. The main idea is to maximize the minimum allocation a user receives. It was originally proposed to ensure an equal share of link capacity for every flow in a network [7]. Since then, max-min has been applied to a variety of individual resource types, including CPU, memory and I/O [6]. Nevertheless, when it comes to *multi-resource* allocation, max-min is unable to ensure fairness [6], [8].

In a multi-resource setting, users often have heterogeneous demands and dynamic workloads [3], [6]. Some mechanisms have been proposed to address the multi-resource allocation problem [6], [8], [9], most notably, Dominant Resource Fairness (DRF) [6]. DRF generalizes max-min to the multi-resource setting, by giving users an equal share of their mostly demanded resource — their *dominant resource*. Using this approach, DRF achieves several desirable properties. Despite the extensive literature on fair allocation, most allocation policies focus only on instantaneous or short term fairness, ensuring that users receive an equal share of the resources regardless of their past behaviors. DRF is no exception, it guarantees fairness only when users' demands remain constant. In practice, however, users' workloads are quite dynamic [3], [10] and ignoring this fact leads to unfairness in the long run.

This paper proposes Stateful Dominant Resource Fairness (SDRF), an extension of DRF that accounts for the past behavior of users and improves fairness in the long run. The key idea is to make users with lower average usage have priority over users with higher average usage. When scheduling tasks, SDRF ensures that users that only sporadically use the system have their tasks scheduled faster than users with continuous high usage. The intuition for SDRF is that when users use more resources than their rightful share of the system, they commit to use less in the future if another user needs. SDRF tracks users commitments and ensures that whenever system resources are insufficient, commitments are honored.

We conduct a thorough evaluation of SDRF and show that it satisfies the fundamental properties of DRF. SDRF is strategyproof as users cannot improve their allocation by lying to the mechanism. SDRF provides sharing incentives as no user is better off if resources are equally partitioned. Moreover, SDRF is Pareto efficient as no user can have her allocation improved without decreasing another. DRF can be efficiently implemented using a priority queue that determines which user has the highest allocation priority. When we consider the past, allocation priorities may change at any instant and the implementation cannot benefit from a priority queue. We mitigate this problem — being able to implement SDRF efficiently — introducing live tree, a data structure that keeps elements with predictable time-varying priorities sorted.

Besides the theoretical evaluation, we analyze SDRF using large-scale simulations based on Google cluster traces containing 30 million tasks over a one-month period, and compare it to regular DRF. Results show that SDRF reduces the average time users wait for their tasks to be scheduled. Moreover, it increases the number of completed tasks for users with lower demands, with negligible impact on high-demand users. We also use Google cluster traces to evaluate the performance of live tree, concluding that SDRF can be used in practice.

This paper is organized as follows. We review related work

ISBN 978-3-903176-08-9

in Sec. II and introduce the system model in Sec. III. In Sec. IV we present DRF and its allocation properties. We then introduce SDRF and show its properties in Sec. V. Sec. VI focus on the implementation of SDRF. We test SDRF and our implementation under trace-driven simulations in Sec. VII. Finally, we conclude the paper in Sec. VIII.

II. Related Work

Fair resource allocation is a prevalent research topic, both in the computer science and economics fields. Nonetheless, focus is often given to the single resource setting. Ghodsi *et al*. [6] are the first to investigate the multi-resource setting under a shared computing perspective, proposing DRF. Dolev *et al*. [8] propose an alternative based on "bottleneck fairness". Nevertheless, the alternative is not strategyproof and is computationally expensive [11]. Gutman *et al*. [12] develop polynomial-time algorithms to compute both DRF and "bottleneck fairness" for non-discrete allocations. Joe-Wang *et al*. [4] extend the notion of fairness introduced by DRF to develop a framework that captures the fairness-efficiency tradeoff. However, they assume a cooperative environment and as such do not evaluate strategyproofness. Wang *et al*. [13] generalize DRF for a scenario with multiple heterogeneous servers, relaxing the sharing incentives restriction. Friedman *et al*. [14] also look at the allocation on multiple servers but provide a randomized solution that achieves sharing incentives. Another extension of DRF is proposed by Parkes *et al*. [15] to account for users with different weights and zero demands. Zarchy *et al*. [16] also investigate multi-resource allocation, but their focus is when the same application may be developed differently to use different proportions of resource types. They propose a framework that allows users to submit multiple demands for the same application. Even though the aforementioned works consider the multi-resource setting, they ignore the dynamic nature of users' demands.

Bonald and Roberts [9] suggest Bottleneck Max Fairness (BMF), which also does not enforce strategyproofness, but improves resource utilization as compared to DRF. They consider dynamic demands in their analysis, arguing that for highly dynamic environments, such as networks, it is hard for users to manipulate the system. BMF convergence is proved in a later work [17]. Even though the analysis of BMF considers dynamic demands, the allocation itself considers only short term usage, ignoring fairness in the long run. Kash *et al*. [18] investigate a dynamic setting where users arrive and never leave, however, they also assume that demands remain constant. Friedman *et al*. [19] evaluate the scenario where multiple users arrive and leave the system. The focus, however, is on the fair division of resources as soon as the user arrives, limiting the number of task disruptions. There are also works that adapt DRF to packet processing [11], [20] and consider a recent past. Nevertheless, this is done to prevent limitations that arise when scheduling packets — in which resources must be shared in *time* — and not to ensure fairness and efficiency in the long run. Finally, other authors have focused on improving efficiency in the long run but not fairness [21], [22]. While some of these works consider users' dynamicity, they do not address fairness in the long run, our focus in this paper.

III. System Model

In this section, we model the multi-resource allocation problem in a multi-user system. We first formalize users and resource demands, and then define the general structure of an allocation mechanism. From this structure we formalize users' sequential interactions as a repeated game.

A. Multi-Resource Setting and Allocation Mechanism

The system consists of a set of users $\mathcal{N} = \{1, \ldots, n\}$ that share a pool of different hardware resources $\mathcal{R} = \{1, \ldots, m\}$. Without loss of generality, we normalize the total amount of every resource in the system to 1, *i.e*., if a system has a total of 100 CPU cores and 10 TB of memory, 0.1 CPU equals 10 cores while 0.1 memory equals 1 TB. For simplicity, we assume that the set of users and the amount of resources remain fixed. Every user i has a demand vector $\boldsymbol{\theta}_i^{(t)} = \langle \theta_{i1}^{(t)}, \ldots, \theta_{im}^{(t)} \rangle$ representing the user demand for every resource at instant t. We consider positive demands for every resource type, therefore at every instant t, $\theta_{ir}^{(t)} > 0, \forall i \in \mathcal{N}, r \in \mathcal{R}$.

The allocation mechanism should produce as output a resource allocation based on users' *declared demands*. We represent the declared demands vector for a user i at instant t analogously to the demands vector, $\hat{\boldsymbol{\theta}}_i^{(t)} = \langle \hat{\theta}_{i1}^{(t)}, \ldots, \hat{\theta}_{im}^{(t)} \rangle$. When users declare demands truthfully, $\hat{\boldsymbol{\theta}}_i^{(t)} = \boldsymbol{\theta}_i^{(t)}$. We also define the allocation vector for user i at instant t for every resource type as $\boldsymbol{o}_i^{(t)} = \langle o_{i1}^{(t)}, \ldots, o_{im}^{(t)} \rangle$. The allocation returned by the mechanism at instant t is represented by a matrix of all the individual allocation vectors: $\boldsymbol{o}^{(t)} = \langle \boldsymbol{o}_1^{(t)}, \ldots, \boldsymbol{o}_n^{(t)} \rangle$. We impose a feasibility restriction to the allocations so that they may never be greater than the total amount of resources in the system, *i.e*., at every instant t, $\sum_{i \in \mathcal{N}} o_{ir}^{(t)} \leq 1, \forall r \in \mathcal{R}$.

We represent user's preferences using a utility function. Given an arbitrary allocation $\boldsymbol{o}_i^{(t)}$, for every user i and time t, the utility function is

$$u_i^{(t)}(\boldsymbol{o}_i^{(t)}) = \min \left\{ \min_{r \in \mathcal{R}} \{ o_{ir}^{(t)} / \theta_{ir}^{(t)} \}, 1 \right\}. \tag{1}$$

Intuitively, users prefer allocations that maximize their number of tasks, being indifferent between different allocations that result in the same number of tasks (when the utility is 1, the user is able to allocate all the tasks she desires). This assumes tasks are arbitrarily divisible [6], [13], [15]. This assumption does not hold in practice and we evaluate its impact in Sec. VI-B. Note that we do not rely on the utility function for interpersonal comparison, we only use it to induce ordinal preferences [15], [18]. This means that, even though the utility function can be used to determine which allocation is better for a user, it cannot be used to determine if one user is doing better than another.

B. Repeated Game

In the previous sub-section we referred to an instant t when defining most notations, however we omitted the influence

time has in the allocation and in the user's preferences. In game theory, we typically say that at every instant t there is a *stage game* where users declare their demands ($\hat{\boldsymbol{\theta}}_i^{(t)}, \forall i \in \mathcal{N}$) and the allocation mechanism decides an allocation ($\boldsymbol{o}_i^{(t)}, \forall i \in \mathcal{N}$). The sequence of stage games defines the *repeated game*. To evaluate user's expected long-term utility, we consider that they discount future utilities using a discount factor $\delta_i \in [0, 1)$, *i.e.*, user i's *expected long-term utility* in the repeated game for the instant t is

$$u_i^{[t,\infty)} = \mathbb{E}_{u_i}\left[(1-\delta_i)\sum_{k=t}^{\infty}\delta_i^{k-t}u_i^{(k)}(\boldsymbol{o}_i^{(k)})\right]. \quad (2)$$

The normalization factor $(1-\delta_i)$ adjusts the units so that we can compare the stage-game and repeated-game utilities.[1] The discount factor δ_i is often called the "user patience"; the closer it is to 1, the more users care about future outcomes. Conversely, the closer it is to 0, the more users care about recent future and the stage-game outcomes.

IV. DRF and Allocation Properties

In this section, we quickly review the DRF mechanism and the static allocation properties DRF and DRF-based schedulers usually satisfy. We show that these properties alone are not enough to enforce fairness in the long run, requiring an alternative for the dynamic setting.

A. DRF Mechanism

Dominant Resource Fairness (DRF) [6] extends Max-Min Fairness (MMF) to the multi-resource setting. DRF calculates an allocation based on users' dominant resources (the most demanded resource for each user, relative to the total amount in the system). As we have normalized all the different kinds of resources to 1, we say $\tilde{r}_i^{(t)}$ is a dominant resource for user i at instant t, if

$$\tilde{r}_i^{(t)} \in \arg\max_{r\in\mathcal{R}} \theta_{ir}^{(t)}. \quad (3)$$

Given the dominant resource, we define the *normalized demand vector* for each user, in which the dominant resources become 1. The normalized demand vector for user i at instant t is denoted by $\tilde{\boldsymbol{\theta}}_i^{(t)} = \langle \tilde{\theta}_{i1}^{(t)}, \ldots, \tilde{\theta}_{im}^{(t)} \rangle$, where

$$\tilde{\theta}_{ir}^{(t)} = \frac{\hat{\theta}_{ir}^{(t)}}{\hat{\theta}_{i\tilde{r}_i^{(t)}}^{(t)}}, \forall i \in \mathcal{N}, r \in \mathcal{R}. \quad (4)$$

When users request an infinite number of tasks, DRF computes an allocation where each user receives an equal share of their dominant resource. For this particular case, DRF can be described using a simple linear program whose solution (x) is the share of dominant resource each user receives [15]:

$$\begin{aligned} \max_{x} \quad & x \\ \text{s.t.} \quad & \sum_{i\in\mathcal{N}} o_{ir}^{(t)} \le 1, \forall r \in \mathcal{R}, \\ & o_{ir}^{(t)} = x \cdot \tilde{\theta}_{ir}^{(t)}. \end{aligned} \quad (5)$$

[1] This is easy to verify by calculating $\sum_{t=0}^{\infty}\delta_i^t = \frac{1}{1-\delta_i}$.

Intuitively, we increase x — and consequently the share of dominant resource for every user — until we achieve a bottleneck and no task can be allocated. Given x, the allocation for every user and resource can be calculated as $o_{ir}^{(t)} = x \cdot \tilde{\theta}_{ir}^{(t)}$.

B. Static Allocation Properties

Ghodsi *et al*. [6] established that the DRF allocation satisfies the following properties. These properties have also been used in a variety of works [13], [15], [18] to measure both fairness and efficiency for a static resource allocation. For the following definitions we consider a stage game happening at time t.

1) *Sharing Incentives* (SI). Users should be better off participating in the system than having a proportional and exclusive share of all the resources. Formally, we say that an allocation mechanism satisfies sharing incentives if for every user $i \in \mathcal{N}$, it outputs an allocation $\boldsymbol{o}_i^{(t)}$ such that, $u_i^{(t)}(\boldsymbol{o}_i^{(t)}) \ge u_i^{(t)}(\langle 1/n, \ldots, 1/n \rangle)$. This assumes users have the right to an equal share of all the resources.[2]
2) *Strategyproofness* (SP). Users should not benefit by misreporting their demands to the mechanism. Formally, if we denote the allocation returned by the mechanism when the user i reports her demands truthfully ($\hat{\boldsymbol{\theta}}_i^{(t)} = \boldsymbol{\theta}_i^{(t)}$) as $\boldsymbol{o}_i^{(t)}$ and when the user lies ($\hat{\boldsymbol{\theta}}_i^{(t)} \neq \boldsymbol{\theta}_i^{(t)}$) as $\boldsymbol{o}_i'^{(t)}$, then $u_i^{(t)}(\boldsymbol{o}_i^{(t)}) \ge u_i^{(t)}(\boldsymbol{o}_i'^{(t)})$.
3) *Pareto Optimality* (PO). The allocation should be optimal in the sense that if it can be changed to make a user better, it must make at least another user worse (in other words the allocation cannot be Pareto dominated by another). Formally, an allocation mechanism is Pareto optimal if it returns an allocation $\boldsymbol{o}^{(t)}$ such that for any other feasible allocation $\boldsymbol{o}'^{(t)}$, if there is a user $i \in \mathcal{N}$ such that $u_i^{(t)}(\boldsymbol{o}_i'^{(t)}) > u_i^{(t)}(\boldsymbol{o}_i^{(t)})$ then there must be a user $j \in \mathcal{N}$ such that $u_j^{(t)}(\boldsymbol{o}_j'^{(t)}) < u_j^{(t)}(\boldsymbol{o}_j^{(t)})$.

In addition to the above properties, DRF also satisfies *envy-freeness*, which ensures that users never prefer other user's allocation to their own. Unfortunately satisfying both Pareto optimality and envy-freeness is impractical under indivisibilities [15]. Moreover, as we will see in the next subsection, while envy-freeness is usually desirable for static allocations, it does not ensure fairness in the dynamic setting.

C. Fairness in the Dynamic Setting

We now design an allocation policy that is fair in the long run. Previous work [6], [13] modeled users as having an infinite number of tasks with the same demand for each resource type. When this happens, only the share of resources each task needs is considered — time becomes irrelevant and the allocation is equivalent to a static one. In practice, however, while some users have workloads with repeated jobs, most users have quite dynamic workloads [3], [10].

To illustrate the importance of considering the past in an allocation, we present an example with users A, B and C

[2] It is also possible to consider a weighted version, where users have a lower or higher share depending on their weights.

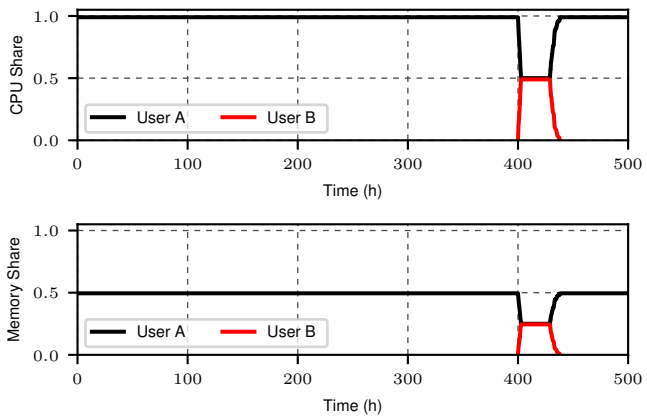


Fig. 1. Unfairness in the long run. User B hardly uses the system but receives the same shares as user A.

sharing a system with a DRF scheduler (see Fig. 1). There are two resources in the system, CPU and memory. User A's dominant resource is CPU and her normalized demand is $\langle 1, 0.5 \rangle$. User A is eager for resources and submits a huge amount of tasks. Nevertheless, the other users only use the system sporadically, with usage spikes. After user A is using the entire system for a while, user B has a spike with normalized demand $\langle 1, 0.5 \rangle$ as well. Even though user B never used her rightful share, the share she receives is the same as user A, *i.e.*, equal to 1/2. This demonstrates that the properties of fairness defined for a static allocation are not enough to enforce fairness in the long run. Satisfying sharing incentives guarantees that users will receive their rightful share but does not reward users for their lower usage. Envy-freeness assumes users are only aware of the present allocation and do not envy other users based on their past allocations.

D. Users' Commitments

To distinguish between users who constantly require more resources than their proportional share from users who only use the system sporadically, we introduce the concept of *commitment*. Commitment is a measure of users propensity to overuse their shares. The key intuition is that users who use more resources than their share, commit to use less if other users need. Users who overuse their shares for a short period of time should have lower commitment than users who constantly overuse. Also, users who overuse less resources should get lower commitment than users who overuse more resources. Every user $i \in \mathcal{N}$ has a separate commitment for each resource $r \in \mathcal{R}$. We define commitment using an exponential moving average of overused resources. The user i's commitment for resource r at time t is given by:

$$c_{ir}^{(t)} = (1-\delta) \sum_{k=-\infty}^{t} \delta^{t-k} \bar{o}_{ir}^{(k)}, \tag{6}$$

where

$$\bar{o}_{ir}^{(k)} = \max \left\{ \left(o_{ir}^{(k)} - \frac{1}{n^{(k)}} \right), 0 \right\}. \tag{7}$$

The term $n^{(k)}$ is the number of users in the system at instant k. Therefore, the term $\bar{o}_{ir}^{(k)}$ represents how much user i overused her share for resource r on instant k. When this term is zero, the user did not overused her share. The more in the past users overused their share the less it influences their commitments.

V. Stateful Dominant Resource Fairness

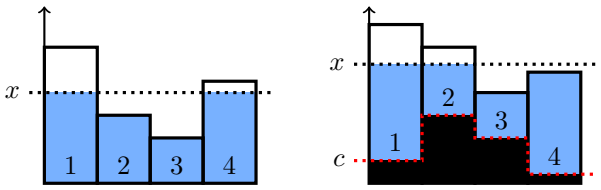
In this section, we introduce *Stateful Dominant Resource Fairness* (SDRF), a generalization of DRF that improves fairness in the long run by enforcing users' commitments. First we develop a simpler version of SDRF for a single resource type. Then, we extend this version and obtain an optimization problem that yields an SDRF allocation. From this problem we proceed to prove that it satisfies the desired properties introduced in Sec. IV.

A. Stateful Max-Min Fairness

The intuition for SDRF is better understood if we first look at the single resource setting. Suppose we have a finite amount of a particular resource, *e.g.*, CPU cores, and we want to equally divide it among the users. The fairest way to divide it is to give an equal share of the resource for every user, *e.g.*, same number of CPU cores. Nonetheless, some users may not need their entire share, in that case it can be redistributed among the other users. This is the main principle behind *Max-Min Fairness* (MMF). One way to achieve MMF is to use a *water-filling algorithm* [5]. Water-filling progressively gives resources for every user until their demands are met. When a user demand is met, she stops receiving resources and the algorithm continues to give resources for the other users. Fig. 2a shows the water-filling diagram for the MMF allocation. Each column (or tank) represents the total amount of resource each user demands. The resource is finite and progressively fills the tanks, until there is no more resource left. In the example, users 2 and 3 have their demands fulfilled while users 1 and 4 only have it partially fulfilled.

Even though MMF is fair for a static allocation, directly applying MMF to the dynamic setting causes the same problem as DRF — it does not consider the past and therefore cannot enforce fairness in the long run. To modify MMF to account for commitments, we introduce Stateful Max-Min Fairness (SMMF). The intuition behind SMMF is better illustrated by an example. "*If the equal share for the resource is 3 CPUs and the user has a commitment of 1 CPU, then the user should have the right to receive at least 2 CPUs*". This notion can be directly implemented using the water-filling algorithm just by adding commitments as a "base for the tanks". Fig. 2b shows the water-filling diagram for the SMMF allocation. Demands are the same as in Fig. 2a, but now there is a base layer of arbitrary commitments c (black layer). Note how user 2 has a lower allocation than she would have without commitments, on the other hand, the demand for user 4 is now met.

Formally, the SMMF allocation can be defined using an optimization problem. Since SMMF allocates a single resource, the resources set becomes a singleton $\mathcal{R} = \{1\}$ and each user $i \in \mathcal{N}$ has a single allocation $o_{i1}^{(t)}$ at time t. The



(a) MMF. (b) SMMF.

Fig. 2. Water-filling diagram for (a) MMF and (b) SMMF.

optimization problem maximizes x, the water level in Fig. 2b, as long as there are resources left in the system:

$$\begin{aligned} \max_{x} \quad & x \\ \text{s.t.} \quad & \sum_{i\in\mathcal{N}} o_{i1}^{(t)} \leq 1, \\ & o_{i1}^{(t)} = \max\left\{0, \min\left\{\hat{\theta}_{i1}^{(t)}, (x - c_{i1}^{(t)})\right\}\right\} . \end{aligned} \tag{8}$$

Given x, each user receives an allocation $o_{i1}^{(t)} = \max\{0, \min\{\hat{\theta}_{i1}^{(t)}, (x - c_{i1}^{(t)})\}\}$ which ensures that allocations are never above demands and remain nonnegative. When commitments are zero, SMMF is equivalent to MMF.

Having defined SMMF, we now generalize it to the multiple resources setting to finally obtain the SDRF mechanism.

B. SDRF Mechanism

SDRF generalizes SMMF similarly to the way DRF generalizes MMF to multiple resources. We use the same concept of dominant resource as DRF, defined in (3). Differently from DRF, though, we must deal with different commitments for different resources. We define the *dominant commitment* for a user i at time t as the user's largest commitment relative to the system total. As we have normalized all the resources to 1, the dominant commitment is simply the largest commitment for the user, *i.e.*,

$$\tilde{c}_i^{(t)} = \max_{r\in\mathcal{R}}\{c_{ir}^{(t)}\} . \tag{9}$$

Having defined the dominant commitment, we define SDRF using ideas from both DRF (5) and SMMF (8). Like DRF, SDRF increases the share of dominant resource for every user until a bottleneck is achieved. Like SMMF, users only start receiving resources when x is above their (dominant) commitment. SDRF is formally defined as:

$$\begin{aligned} \max_{x} \quad & x \\ \text{s.t.} \quad & \sum_{i\in\mathcal{N}} o_{ir}^{(t)} \leq 1, \forall r \in \mathcal{R} , \\ & o_{ir}^{(t)} = \max\left\{0, \min\left\{\hat{\theta}_{ir}^{(t)}, (x - \tilde{c}_i^{(t)}) \cdot \tilde{\theta}_{ir}^{(t)}\right\}\right\} . \end{aligned} \tag{10}$$

Recall $\tilde{\theta}_{ir}^{(t)}$ is the normalized demand for user i and resource r, defined in (4). From x, we may calculate the allocation for every user and resource by $o_{ir}^{(t)} = \max\{0, \min\{\hat{\theta}_{ir}^{(t)}, (x - \tilde{c}_i^{(t)}) \cdot \tilde{\theta}_{ir}^{(t)}\}\}$. In the next subsection we analyze the properties of SDRF that prove it behaves well in both the stage game and in the long run.

C. Analysis of SDRF Allocation Properties

We start our analysis of SDRF proving that it satisfies the desirable properties introduced in Sec. IV-B, namely: strategyproofness, Pareto optimality and sharing incentives. The proof of all propositions is provided in the technical report [23].

First, we show that SDRF increases the share of dominant resource for every user until a resource runs out (the bottleneck resource). This is indicated in the following proposition.

Proposition 1 (Bottleneck). *The SDRF allocation obtained by solving (10) is such that all users have their demands fulfilled or there is a bottleneck resource. Formally, $\boldsymbol{o}_i^{(t)} = \hat{\boldsymbol{\theta}}_i^{(t)}, \forall i \in \mathcal{N}$ or $\exists r \in \mathcal{R}$ such that $\sum_{i\in\mathcal{N}} o_{ir}^{(t)} = 1$.*

Although simple, Proposition 1 is useful to demonstrate the following properties. One of the fundamental properties of DRF is strategyproofness. Without it, users may try to manipulate the system by, *e.g.*, faking their usage, which results in inefficiencies [6], [11]. Propositions 2 and 3 show SDRF is also strategyproof.

Proposition 2 (Strategyproofness in the Stage Game). *When users consider only the stage game utility (1), the SDRF allocation obtained by solving (10) is strategyproof.*

Proposition 2 shows that when users consider only stage game utilities, SDRF is strategyproof. However, the fact that we consider past allocations may create new incentives for users to manipulate their declared demands. It may be possible that some users would not use the system when they actually need, hoping that this would improve their future allocations — this would also bring inefficiencies to the system. Fortunately, Proposition 3 shows that this is not possible.

Proposition 3 (Strategyproofness in the Repeated Game). *When users evaluate their utilities using the expected-long-term utility (2), the SDRF allocation obtained by solving (10) is strategyproof, regardless of users' discount factors.*

The following two propositions demonstrate that SDRF is efficient. Proposition 4 shows that SDRF do not waste resources while Proposition 5 shows that the allocation is Pareto optimal, ensuring that it is not possible to increase a user's allocation without decreasing another.

Proposition 4 (Non-wastefulness). *The SDRF allocation $\boldsymbol{o}^{(t)}$ is such that, if there is a different allocation $\boldsymbol{o'}^{(t)}$ where $o'^{(t)}_{ir} \leq o_{ir}^{(t)}, \forall i \in \mathcal{N}, r \in \mathcal{R}$ and for a user $i^* \in \mathcal{N}$ and resource $r^* \in \mathcal{R}$, $o'^{(t)}_{i^*r^*} < o_{i^*r^*}^{(t)}$, then it must be that $u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{(t)}) > u_{i^*}^{(t)}(\boldsymbol{o'}_{i^*}^{(t)})$. In other words, SDRF is non-wasteful.*

Proposition 5 (Pareto optimality). *The SDRF allocation obtained by solving (10) is Pareto optimal.*

The last property indicates that users are better off if they participate in the system. More specifically, it shows that users receive a utility at least as good as if they had access to $1/n$ of resources in the system.

Proposition 6 (Sharing incentives). *The SDRF allocation obtained by solving (10) satisfies sharing incentives.*

VI. Implementation Using a Live Tree

In this section, we study how SDRF can be implemented in practice. We first consider the effect continuous time and indivisible tasks have in the model defined in Sec. III. We then develop a *water-filling* algorithm to schedule tasks. Nevertheless, the algorithm requires users' priorities to be recalculated and sorted at every execution. To mitigate this problem we introduce *live tree* — a data structure that keeps elements sorted even with time changing priorities — and show how it can be used to improve the SDRF scheduling algorithm.

A. Continuous Time

In the model defined in Sec. III we assume time progresses as a sequence of repeated games, suggesting a discrete time. The definition for commitment in (6) is compatible with this notion. In an actual system, however, tasks may arrive and finish at any instant, therefore we need an expression that allows us to compute commitment at continuous time. First we redefine (6) recursively using a difference equation [24],

$$c_{ir}^{(t)} = (1-\delta)\bar{o}_{ir}^{(t)} + \delta c_{ir}^{(t-\Delta t)} \tag{11}$$

where the commitment at time t can be calculated from commitment at time $t-\Delta t$. This assumes $\bar{o}_{ir}^{(t)}$ remains constant within the interval $(t-\Delta t, t]$. It turns out that (11) can be seen as an exponential smoothing and can be closely approximated in the continuous time [24], leading to the expression

$$\begin{aligned} c_{ir}^{(t)} &= (1-\mathring{\delta})\bar{o}_{ir}^{(t)} + \mathring{\delta} c_{ir}^{(t_0)} \\ \mathring{\delta} &= e^{-(t-t_0)/\tau}, \quad \tau = -\frac{\Delta t}{\ln(\delta)} \end{aligned} \tag{12}$$

where we may calculate $c_{ir}^{(t)}$ from any previous $c_{ir}^{(t_0)}$ as long as $o_{ir}^{(t)}$ remains constant from t_0 to t. Fortunately, o_{ir} only changes when a task for user i requiring resource r starts or finishes. In any other instant, o_{ir} remains constant, making (12) useful in practice. This expression is analogous to the discrete version, using $\mathring{\delta}$ instead of δ. When $t - t_0 = \Delta t$, (12) becomes (11).

B. Indivisible Tasks

So far, we have assumed that tasks are arbitrarily divisible. This allowed us to give arbitrarily small amounts of resources to users. In practice, however, tasks are often not divisible [1], [2]. To schedule indivisible tasks we use the same approach as previous works [6], [13] — applying water-filling to tasks.

Algorithm 1 summarizes the task scheduling procedure. We define a set A of *active users* (users with at least one task waiting to be scheduled), and keep track of the total amount of resources allocated for every user. If the system is not full and if there is at least one user with a pending task, *i.e.*, $A \neq \emptyset$, we schedule the next task for the user with the lowest share of dominant resource compensated for commitments.

Algorithm 1 SDRF task scheduling

```
A = {1, ..., k}                                   ▷ set of active users
o_i = ⟨o_i1, ..., o_im⟩, ∀i ∈ A                   ▷ resources given to user i
c_i = ⟨c_i1, ..., c_im⟩, ∀i ∈ A                   ▷ commitments for user i
while A ≠ ∅ do
    i ← arg min_{i∈A} { max_{r∈R} {o_ir + c_ir} }  ▷ pick user
    ∀r, θ_ir ← demand for r in user i's next task
    if ∀r, ( θ_ir + Σ_{j∈N} o_jr ) ≤ 1 then
        ∀r, o_ir ← o_ir + θ_ir
        if no more pending tasks for user i then
            remove i from A
    else
        return                                     ▷ the system is full
```

Whenever a task arrives or finishes, we rerun Algorithm 1 with updated set A, and vectors $\boldsymbol{o}_i$, $\boldsymbol{c}_i$, $\forall i \in A$. The smaller tasks are, the closer Algorithm 1 approximates (10).

Performance is a major concern in the design of a task scheduler. In peak hours, a scheduler may need to make hundreds of task placement decisions per second [3]. The most expensive part of Algorithm 1 is picking a user. While plain DRF can be implemented using a priority queue that stores the dominant resource share for every user[3], this is not possible for SDRF. In DRF, users' priorities only change when $\boldsymbol{o}_i$ changes, in SDRF users' commitments change at any instant and so do users' priorities. Recomputing priorities for every user and resource at every task scheduling decision would be too costly. The next subsection shows how to solve this problem.

C. Live Tree

When scheduling tasks, we are not really interested in the specific value of commitments, but in which user has the *highest priority*. Live tree is a data structure that keeps elements with predictable time-varying priorities ordered. The key idea is to focus on position-change events, instead of element priorities. When priorities follow a continuous function, elements change position whenever their priorities intersect. A live tree always has a *current time* associated with it — for this current time, it guarantees that elements are sorted. When the current time is updated, instead of updating every element priority, we see if any position-change event happened from the last update to the current time.

Live tree can be seen as a combination of two red-black trees [25], [26] and an array (see Fig. 3). We call one red-black tree *elements tree*, as it keeps elements sorted by priority, while the other is the *events tree*, as it tracks position-change events sorted by their time. The array is used for element lookups. For simplicity, we assume that each of the input elements has a distinct integer id that can be an index for the array[4]. Each position in the array has a pointer to an element in the elements tree (or NIL if there is no element for the given index). This allows us to retrieve elements by id in the tree in $O(1)$ time. If two neighboring elements in the elements tree are to change

[3]The DRF implementation on Mesos [1] uses an `std::set` from the C++ Standard Library, which is usually implemented as a binary tree.

[4]When elements do not have integer ids, or they are too sparse, the array may be replaced by a hash table and still present amortized $O(1)$ lookups.

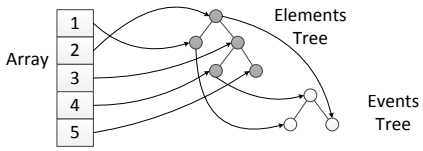


Fig. 3. Illustration of a live tree with its data structures. Positions in the array link to elements in the tree. Some elements link to events in the events tree.

position in the future, the left element will have a pointer to a position-change event in the events tree.

We assume that priorities for all elements can be calculated using the same continuous function $p(t, \kappa)$ based on time t and in the element attribute κ. Every element has a different attribute that dictates how its priority changes over time. It may be a number, a vector or even a tuple. Our description does not depend on the definition of κ, in the next subsection we better define κ for our setting. It also helps to introduce an order notation, we say that element i precedes element j for an instant t, $i \prec_t j$, if $p(t, \kappa_i) < p(t, \kappa_j)$. The elements tree compares elements using $[\prec_t]$. This is useful since, whenever we insert a new element, it is compared to the others consistently with the time t. Live tree also needs a function to calculate priority intersections. We denote by $t_{\text{int}}(t, \kappa_i, \kappa_j)$ the function that calculates the priority intersection time based on two element attributes (κ_i, κ_j) and the time t.

We now briefly describe the basic operations of a live tree[5]:

INSERT(i, κ_i). To insert an element i in the live tree, we first insert i in the elements tree. Since the elements tree compares elements using $[\prec_t]$, i will be placed in the correct position relative to time t. Once inserted, we set a pointer from position i in the array to the element in the tree. Then, we call UPDATEEVENT for i and for its predecessor in the tree. When i is the minimum element, we only call UPDATEEVENT for i. INSERT can be accomplished in $O(\log n)$ time.

UPDATEEVENT(i). If an element i will change position with its successor in the future, it must have a position-change event associated with it. To update an event we first check if the element i has an event in the events tree and remove it if so. Then, we check if i and its successor j will switch places in the future by calculating their priorities intersection $t_{\text{int}}(t, \kappa_i, \kappa_j)$. If t_{int} exists and is positive, we add an event for element i and time $t_{\text{int}} + t$ in the events tree. Then we add a pointer from element i in the elements tree to the event in the events tree. When i is the maximum element, it has no successor and thus cannot have a position-change event (note this does not imply it cannot change position, as its predecessor can have an event). UPDATEEVENT can be done in $O(\log n)$ time.

UPDATE(t). Whenever the current time changes, we must update the tree. We assume that time progresses forward and live tree can only be updated to the future. To update the tree to a new time t, we look at all events that happen before t. If there is no event, *i.e.*, the first event in the events tree has time greater than t, then no element should change position and the tree is already updated, otherwise we must consider the events. We remove events from the events tree in order until the next event has time greater than t or the events tree becomes empty. For every removed event, we remove its correspondent element as well as its successor from the elements tree calling DELETE. Once we finish removing events we reinsert each removed element calling INSERT. Since elements are compared using $[\prec_t]$, the reinsertion places elements in their correct position relative to time t. UPDATE can be accomplished in $O(n \log n)$ time. The worst case happens when every element must change position and therefore must be reinserted in the tree. In Sec. VII we show that, for SDRF, the actual time is much smaller than the worst case.

DELETE(i). To delete an element i, we first check position i in the array. From position i we get a pointer to the elements tree. If the element has an event, we get a pointer to its event as well. We then remove the event from the events tree, the element from the elements tree and set NIL at position i in the array. If i was the minimum element, we are done, otherwise we must call UPDATEEVENT to the predecessor of i in the elements tree. DELETE can be accomplished in $O(\log n)$ time.

MINIMUM/MAXIMUM. The minimum (maximum) in the live tree is the minimum (maximum) in the elements tree. MINIMUM/MAXIMUM can be accomplished in $O(1)$ time.

We omitted from our description corner cases, such as if an element being deleted does not exist, or if the element being inserted is already in the tree.

Live tree performance depends heavily on the priority function used and the frequency of UPDATE calls. When elements change position often, UPDATE has to process more events. Nevertheless, the higher the frequency of UPDATE calls, the less events each call has to process. In Sec. VII we evaluate how live tree performs when used to implement SDRF.

D. Live Tree Applied to SDRF

We now apply live tree to Algorithm 1. In Algorithm 1, we pick the user with the minimum value of $\max_{r \in \mathcal{R}}\{o_{ir} + c_{ir}\}$, therefore we use a live tree to sort users by this value. Using (12), we define the priority function p as

$$\begin{aligned} p(t, \kappa_i) &= \max_{r \in \mathcal{R}} \left\{ o_{ir} + (1 - \mathring{\delta})\bar{o}_{ir} + \mathring{\delta} c_{ir}^{(t_i)} \right\} \\ \mathring{\delta} &= e^{-(t - t_i)/\tau} \end{aligned} \tag{13}$$

$\kappa_i = (t_i, \tau, o_{i1}, \ldots, o_{im}, \bar{o}_{i1}, \ldots, \bar{o}_{im}, c_{i1}^{(t_i)}, \ldots, c_{im}^{(t_i)})$, τ is defined as in (12) and is the same for all users. t_i is the time user i is inserted in the live tree. Obtaining the intersection function is a bit more involved (further details are in the technical report [23]). We define it using a set $\mathcal{I}_{ij}$ of all intersections between resources from users i and j:

$$\mathcal{I}_{ij} = \left\{ \tau \ln \left(\frac{\bar{o}_{ir_1} - \bar{o}_{jr_2} + c_{jr_2}^{(t_0)} - c_{ir_1}^{(t_0)}}{\bar{o}_{ir_1} - \bar{o}_{jr_2} + o_{ir_1} - o_{jr_2}} \right) \middle| (r_1, r_2) \in \mathcal{R}^2 \right\},$$

where $t_0 = \max\{t_i, t_j\}$. We define the intersection function getting the minimum intersection after the current time t,

$$t_{\text{int}}(t, \kappa_i, \kappa_j) = \min\{k + t_0 - t | k \in \mathcal{I}_{ij} \wedge k + t_0 > t\} \tag{14}$$

[5]Our implementation of SDRF and Live Tree is open source and is available at https://github.com/hugombarreto/sdrf

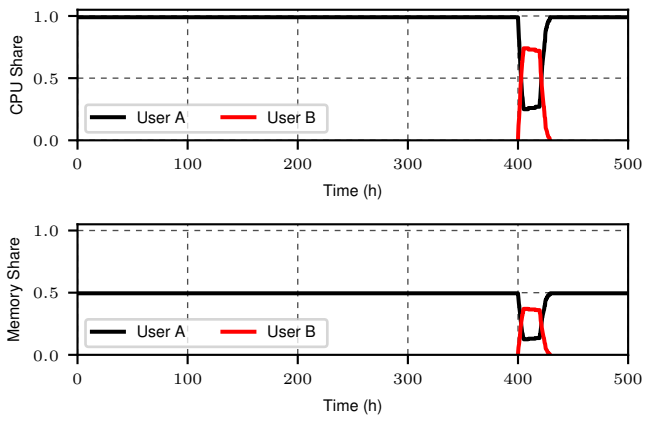


Fig. 4. Same example as Fig. 1 but using SDRF ($\delta = 1 - 10^{-6}$). Note how user B receives more resources and is able to complete her workload faster.

When there is no intersection after the time t, t_{int} does not exist and live tree will add no event. Note this intersection function may indicate intersections between resources that do not cause an intersection in priorities, *i.e.*, commitments may intersect without changing the dominant commitment. Although non-optimal it performs correctly, as false events do not change the order in the tree. In the next section, we show how SDRF and live tree perform when scheduling tasks.

VII. Simulation Results

In this section, we evaluate SDRF and live tree using trace-driven simulations based on Google cluster traces [3]. The traces contain information from workloads (from either Google services or engineers) running in a cluster over a month-long period. Workloads are submitted in the form of jobs, and each job may have multiple tasks. The traces contain events for every time a task is submitted, is scheduled or finishes. From these events we extract the CPU and memory demands as well as task submission and running times using them as input for our simulation. We remove tasks with 0 demand, as well as tasks that were evicted by the Google system, but leave tasks that failed due to user errors. After that, we are left with around 32 million tasks from 627 users.

We run simulations for different values of δ and system overload. The values of δ are relative to a Δt of 1 second (see (12)). We vary δ by making it exponentially closer to 1, *i.e.*, $\delta = 1-10^{-1}, \ldots, 1-10^{-7}$. This is equivalent to exponentially increasing τ from (12). To verify how SDRF performs under different levels of system load we also perform simulations for multiple values of total resources (*i.e.*, CPU and memory). We use the average system usage in the original trace, called hereinafter as R, as a baseline for our results. We then run simulations with the total amount of resources in the system varying from 50% to 100% of R, in steps of 10%.

First, we run SDRF for the same example presented in Sec. IV (Fig. 1). Fig. 4 shows how user B receives more resources (both CPU and memory) than user A and is able to complete her workload faster than using DRF. Since user A is constantly using the system, receiving less resources for a short period will have a low impact in her overall workload.

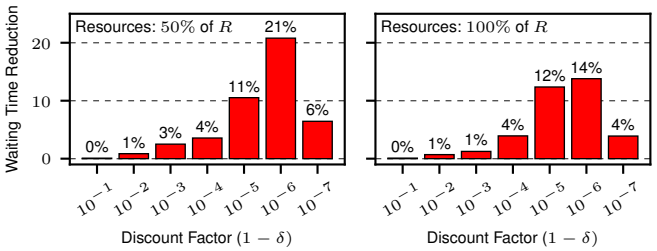


Fig. 5. Mean wait time reduction for every user relative to DRF.

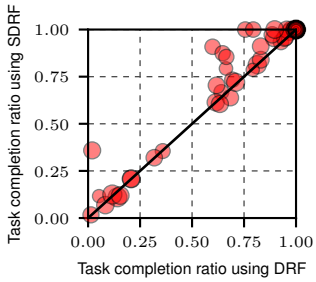


Fig. 6. Task completion ratio using DRF and SDRF. Each bubble is a different user. The bubble's size is logarithmic to the number of tasks submitted by the user. Users above the $y = x$ are better with SDRF.

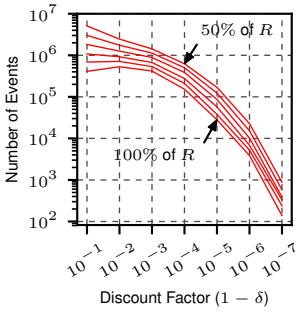


Fig. 7. Live tree events for different values of discount factor and system resources (50% to 100% of R from top to bottom).

We now evaluate the simulation results. Fig. 5 presents the mean waiting time reduction for every user under different values of δ and system load when compared to DRF. When δ is small enough, SDRF performs close to DRF. Also, for δ sufficiently close to 1, SDRF approaches DRF. This is justified inspecting (12): when δ is sufficiently close to 1, commitments never accumulate, alternatively, when δ is sufficiently close to 0, commitments are simply the last allocation, and therefore tasks are scheduled just like in DRF. The best waiting time reduction was observed for the discount factor $\delta = 1-10^{-6}$ for all levels of system load evaluated. Even though the advantage of SDRF is more evident when the system is overloaded, for $\delta = 1 - 10^{-6}$, SDRF consistently outperforms DRF by more than 10%. Since the intermediate cases (60% to 90% of R) perform in between the two extreme cases (50% and 100% of R), we defer those plots to the technical report [23].

We also investigate how the waiting time reduction affects the number of tasks each user is able to complete. We compute the task completion ratio for every user (*i.e.*, the number of tasks completed divided by the number submitted) and compare it when running DRF and SDRF. Fig. 6 shows the results for the simulation with $\delta = 1-10^{-6}$ and total resources 50% of R. Each bubble represents a different user: when above the black $y = x$ line, the user is able to complete more tasks under SDRF than under DRF. Most users perform better under SDRF, in fact, only 9 out of 627 users completed less tasks under SDRF. Also note that, even though these users com-

pleted less tasks, their task completion ratio had low impact. This happens because users that use the system in small bursts complete their workloads earlier and, consequently, have the opportunity to complete more tasks. On the other hand, users that use the system continually experience a low impact.

Next we evaluate how live tree performs under the same simulations. The theoretical worst case complexity for the update operation is $O(n \log n)$. This is driven by the maximum number of events an update may trigger, when there is no event, updates are performed in $O(1)$. In practice, however, the average number of events is much shorter. Fig. 7 shows the number of live tree events that happened during the entire simulation period for every simulation. Each curve represents a different value of system load, 50% of R to 100% of R, from top to bottom. The number of events increases when the amount of resources in the system decreases. Also, the closer δ is to 1, the less events we observe. This makes sense, since commitments vary slower the closer δ is to 1. When $\delta = 1 - 10^{-6}$ and the total resources 50% of R, there is a total of $22{,}718$ events, which is about 7 events for every $1{,}000$ scheduled tasks. Since every task scheduling triggers one update, this indicates that updates happen fast for this scenario. Even for the worst scenario ($\delta = 1 - 10^{-1}$, 50% of R), the total number of events is $5{,}094{,}167$, which is approximately 2 events for every 10 tasks. If live tree performed close to its theoretical worst case complexity, it would offer low advantage compared to sorting elements on every update. But with the number of updates observed, live tree operations will perform close to the ones of a red-black tree where weights are static.

VIII. Conclusion and Future Work

In this paper, we introduced SDRF, an extension of DRF that enforces fairness in the long run. SDRF looks at past allocations and benefits users with lower average usage. We show that SDRF satisfies the fundamental properties of DRF while enforcing fairness in the long run. To efficiently implement SDRF, we introduced live tree, a general-purpose data structure that keeps elements with predictable time-varying priorities ordered. We simulated SDRF using Google cluster traces for a month-long period. Results have shown that under SDRF, users with low utilization can complete their workloads faster. Meanwhile, users with high utilization suffer a low impact in their overall workload. We also used the simulations to evaluate the live tree performance concluding that SDRF can be implemented efficiently.

There are different future investigation directions. First, we believe live tree may benefit other applications, *e.g.*, Dijkstra's algorithm applied to graphs with time-variable weights. Second, SDRF can be extended to cover other applications. Although we mentioned the possibility of using weights for users, we did not evaluate it formally. Another possibility is the use of a different function to measure commitments.

Acknowledgment

This work was partially supported by CAPES, CNPq, FAPERJ, and FAPESP grants #15/24494-8 and #15/24490-2.

References

[1] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX NSDI*, 2011.

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. ACM SoCC*, 2013.

[3] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM SoCC*, 2012.

[4] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multiresource allocation: Fairnes-efficiency tradeoffs in a unifying framework," *IEEE/ACM Transactions on Networking*, vol. 21, no. 6, pp. 1785–1798, Dec. 2013.

[5] B. Radunovic and J.-Y. Le Boudec, "A unified framework for max-min and min-max fairness with applications," *IEEE/ACM Transactions on Networking*, vol. 15, no. 5, pp. 1073–1083, Oct. 2007.

[6] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX NSDI*, 2011.

[7] J. Jaffe, "Bottleneck flow control," *IEEE Transactions on Communications*, vol. 29, no. 7, pp. 954–962, Jul. 1981.

[8] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial, "No justified complaints," in *Proc. Conference on Innovations in Theoretical Computer Science*, 2012.

[9] T. Bonald and J. Roberts, "Multi-resource fairness: Objectives, algorithms and performance," in *Proc. ACM SIGMETRICS*, 2015.

[10] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proc. ACM SoCC*, 2010.

[11] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," in *Proc. ACM SIGCOMM*, 2012.

[12] A. Gutman and N. Nisan, "Fair allocation without trade," in *Proc. Intern. Conference on Autonomous Agents and Multiagent Systems*, 2012.

[13] W. Wang, B. Li, and B. Liang, "Dominant resource fairness in cloud computing systems with heterogeneous servers," in *Proc. IEEE INFOCOM*, 2014.

[14] E. Friedman, A. Ghodsi, and C.-A. Psomas, "Strategyproof allocation of discrete jobs on multiple machines," in *Proc. ACM Conference on Economics and Computation*, 2014.

[15] D. C. Parkes, A. D. Procaccia, and N. Shah, "Beyond dominant resource fairness," *ACM Transactions on Economics and Computation*, vol. 3, no. 1, pp. 3:1–3:22, Mar. 2015.

[16] D. Zarchy, D. Hay, and M. Schapira, "Capturing resource tradeoffs in fair multi-resource allocation," in *Proc. IEEE INFOCOM*, 2015.

[17] T. Bonald, J. Roberts, and C. Vitale, "Convergence to multi-resource fairness under end-to-end window control," in *Proc. IEEE INFOCOM*, 2017.

[18] I. Kash, A. D. Procaccia, and N. Shah, "No agent left behind: Dynamic fair division of multiple resources," *Journal of Artificial Intelligence Research*, vol. 51, no. 1, pp. 579–603, Sep. 2014.

[19] E. Friedman, C.-A. Psomas, and S. Vardi, "Controlled dynamic fair division," in *Proc. ACM Conference on Economics and Computation*, 2017.

[20] W. Wang, B. Liang, and B. Li, "Low complexity multi-resource fair queueing with bounded delay," in *Proc. IEEE INFOCOM*, 2014.

[21] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *Proc. USENIX OSDI*, 2016.

[22] C. Chen, W. Wang, S. Zhang, and B. Li, "Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees," in *Proc. IEEE INFOCOM*, 2017.

[23] H. Sadok, M. E. M. Campista, and L. H. M. K. Costa, "Stateful dominant resource fairness: Considering the past in a multi-resource allocation," GTA/PEE/UFRJ, Tech. Rep. GTA-18-06, 2018. [Online]. Available: https://www.gta.ufrj.br/ftp/gta/TechReports/SCC18.pdf

[24] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Prentice-Hall, 1999.

[25] R. Bayer, "Symmetric binary B-Trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, Dec. 1972.

[26] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. IEEE Symposium on Foundations of Computer Science*, 1978.