Multi-Resource Fair Sharing for Datacenter Jobs with Placement Constraints

Wei Wang[†], Baochun Li[‡], Ben Liang[‡], Jun Li[‡]
[†]*Hong Kong University of Science and Technology,* [‡]*University of Toronto*
weiwa@cse.ust.hk, {bli, liang, junli}@ece.utoronto.ca

***Abstract*—Providing quality-of-service guarantees by means of fair sharing has never been more challenging in datacenters. Due to the heterogeneity of machine configurations, datacenter jobs frequently specify *placement constraints*, restricting them to run on a particular class of machines meeting specific hardware/software requirements. In addition, jobs have diverse demands across *multiple resource types*, and may saturate any of the CPU, memory, or storage resources. Despite the rich body of recent work on datacenter scheduling, it remains unclear how multi-resource fair sharing is defined and achieved for jobs with placement constraints. In this paper, we propose a new sharing policy called Task Share Fairness (TSF). With TSF, jobs are better off sharing the datacenter, and are better off reporting demands and constraints truthfully. We have prototyped TSF on Apache Mesos and confirmed its service guarantees in a 50-node EC2 cluster. Trace-driven simulations have further revealed that TSF speeds up 60% of tasks over existing fair schedulers.**

***Index Terms*—Cluster schedulers; multi-resource allocation; placement constraints; fairness**

I. Introduction

Datacenter-scale compute clusters running parallel processing frameworks such as MapReduce [5] and Spark [32] serve as the major powerhouses for data-intensive computation. As their workloads surge, providing quality-of-service guarantees for datacenter jobs, each consisting of many *tasks*, has become increasingly important. Fair sharing is a fundamental tool to achieve this objective, with which jobs are guaranteed to receive fair shares of cluster resources, irrespective of the behavior of others. Achieving fair sharing in datacenters, however, is particularly challenging, due to the heterogeneity of physical machines and the diversity of workload.

Over time, a datacenter typically evolves to include three to five generations of machine hardware, with 10–40 different configurations [6], [20], [22]. Incompatibilities between machine configurations and prerequisites of task execution are often encountered. As a result, an increasing number of datacenter jobs specify *placement constraints*, restricting their tasks to run on a particular class of machines that meet specific hardware (*e.g.*, GPU and SSD) and/or software (*e.g.*, a particular kernel version) requirements [11], [20], [22]. For example, a CUDA [1] task must run on machines with GPUs, while a DNS service requires machines with public IP addresses. According to Google, approximately 50% of its production jobs have simple, yet restrictive, constraints [22].

In addition to these constraints, datacenter jobs are characterized by a high degree of demand diversity across *multiple resource types*. For example, business analytics jobs typically have CPU-intensive tasks, while machine learning and graph analytics may have memory-bound tasks. Workload analyses on production traces from Facebook, Microsoft, and Google [10], [12], [20] have confirmed that jobs may require vastly different amounts of resources of memory, CPU cores, storage space, I/O bandwidth, and network bandwidth.

In the presence of placement constraints and multi-resource demands, scheduling tasks in datacenter jobs involves some non-trivial technical challenges. Prevalent schedulers, such as the Hadoop Fair Scheduler [31] and the Capacity Scheduler [2], allocate resources in the units of *slots*, each containing a fixed amount of memory or CPU cores. While slot schedulers can be easily extended to handle placement constraints using a recently proposed algorithm called Choosy [11], they suffer from poor utilization due to *resource fragmentation* [10], [12] — resources in these allocated slots, even when idle, are not available to the other tasks. Therefore, recent works, notably DRF [10] and its variants [3], [8], [23], [30], have turned to *multi-resource fair scheduling*. However, these algorithms do not explicitly model placement constraints. Worse, as we shall show in Sec. IV, directly applying them to constrained jobs provides no quality-of-service guarantees.

In this paper, we ask a fundamental question of resource management: how should fair sharing be defined and achieved for jobs with placement constraints and multi-resource demands? We propose a new sharing policy, called *Task Share Fairness* (TSF), that equalizes the *task share* of jobs as much as possible. Task share is defined for each job as the ratio between the number of tasks scheduled and the maximum number of tasks the job can run if we remove its constraints and allocate it the entire datacenter. We show that TSF satisfies a number of desirable sharing properties that are most important for datacenter scheduling [4], [10], [11]. In particular, with TSF, jobs are better off sharing the datacenter dynamically than running in some static, dedicated resource pools (*sharing incentive*); no job can receive more resources by lying about its demands and/or constraints (*strategy-proofness*); no job envies the allocation of another (*envy-freeness*); no job can increase its allocation without decreasing that of another (*Pareto optimality*). To our knowledge, TSF is the first multi-resource sharing policy meeting all these desirable properties in the presence of placement constraints.

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

We have prototyped TSF as a pluggable resource manager on Apache Mesos [14], using a simple yet effective *online* algorithm. Our experiments on a 50-node EC2 cluster have confirmed that TSF delivers its theoretically proven properties with predictable service guarantees. In addition, our large-scale simulation study with cluster traces from Google [20] has further revealed that, by allocating constrained jobs the "right" share of resources, TSF speeds up the completion of 60% of tasks over existing fair sharing algorithms in datacenters, including Choosy [11], DRF, and its variants [8], [30].

The remainder of this paper is organized as follows. We motivate the problem of multi-resource sharing with placement constraints in Sec. II. In Sec. III, we list the desirable properties that we aim to satisfy in this paper. We analyze existing fair sharing policies and illustrate their problems in Sec. IV. We present the design and analysis of TSF in Sec. V and evaluate its performance in Sec. VI. We survey related work in Sec. VII and conclude the paper in Sec. VIII.

II. Background and Model

In this section, we provide more background information and describe our model.

A. Modeling Placement Constraints

Placement constraints can be either *hard* or *soft*. Hard constraints must be satisfied by all means (*e.g.*, the requirement of GPUs or a particular kernel version); soft constraints, on the other hand, are simply preferences (*e.g.*, co-locating tasks to data) and can be violated at the expense of degraded performance. In this paper, we focus on hard constraints, as soft constraints can be effectively addressed using existing techniques such as Delay Scheduling [31] and cost- or utility-based scheduling algorithms [15], [24]. Hard constraints are more difficult to address [11] and may result in serious utilization and latency consequences if handled inappropriately [22]. To our knowledge, Choosy [11] is the only work that provides quality-of-service guarantees in the presence of hard constraints. However, Choosy is designed for single-resource sharing — scheduling tasks based on one resource type inevitably results in fragmentation and over-allocation [12].

We limit our discussion to *simple, non-combinatorial* constraints depending on the machine attributes only, *e.g.*, machines with GPUs and the Linux kernel version of $3.x.y$. Complex, combinatorial constraints are possible to be encountered (*e.g.*, never co-locating two tasks of a job) but are dominated by simple ones. For instance, at Google, 50% of production jobs have simple constraints, as opposed to 11% having combinatorial constraints [22]. We leave the exploration of combinatorial constraints as our future work.

We model placement constraints (hereafter constraints) as a bipartite *constraint graph* consisting of the *user vertices* and the *machine vertices*. A *user* corresponds to a datacenter job consisting of many parallel tasks. Unless otherwise stated, we do not differentiate between a user and a job in the remainder of the paper. In the constraint graph, an edge is connected between a user vertex and a machine vertex if the user can run tasks on the machine. Fig. 1 illustrates an example of the constraint graph where 4 users share a 4-node cluster. Without loss of generality, we assume that the constraint graph is *connected* — for a disconnected graph, its connected components can be considered separately.

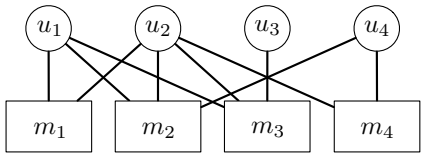


Fig. 1: An example of constraints modeled as a bipartite graph, where user u_1 can run tasks on all machines but m_4, user u_2 can run on all machines, user u_3 can only run on m_3, and u_4 can run on both m_2 and m_4.

B. Modeling Machines and Demands

We model a datacenter as a common shared infrastructure consisting of a number of machines having multiple types of resources, *e.g.*, CPU, memory, storage space, I/O bandwidth, *etc.* Each machine is characterized by a *configuration vector*, which specifies the total amount of resources it has, *e.g.*, $\langle 8$ CPUs, 4 GB RAM$\rangle$.

Users run many parallel tasks. Each task is characterized by a *demand vector*, which specifies the amount of resources needed during the runtime, *e.g.*, $\langle 1$ CPU, 3 GB RAM$\rangle$. Because the tasks of a job are typically the same binary program running on different data blocks of similar sizes (*e.g.*, MapReduce [5]), they require the same amount of resources [20], [21], [26]. We therefore assume the same demand vector across a user's tasks [4], [10], [19]. This assumption has been widely confirmed in production workload traces [20], [21]. Finally, to support service differentiation, each user is assigned a *weight* — a positive real number — which indicates its relative importance in the system.

III. Desirable Sharing Properties

In this section, we motivate seven sharing properties that are key in providing quality-of-service guarantees.

A. Required Properties

As observed by DRF [3], [8], [10], there are four properties that *any* datacenter scheduler should satisfy: sharing incentive, strategy-proofness, envy-freeness, and Pareto optimality. While the latter two properties directly extend to constrained tasks, the former two require non-trivial generalization.

1) ***Sharing Incentive****: Sharing incentive* is the key to provide service guarantee and is particularly important to datacenters. In a nutshell, it ensures that each user obtained more resources by dynamically sharing the datacenter than running tasks in a static resource pool dedicated to it, irrespective of the behaviors of others.

Definition 1 (Sharing incentive): Assume originally each user is given a dedicated resource pool to run tasks. Suppose now the users share their resource pools with others. A new

allocation of the shared resources is said to provide *sharing incentive*, if it allows each user to run no fewer tasks than the user would have run in its original dedicated resource pool.

We stress that in our definition of sharing incentive, the dedicated resource pool given to users can be *arbitrary*. This generalizes the requirement of existing work where the dedicated pool is limited to *equal resource partitioning* (*i.e.*, each of N users is given $1/N$ of the total amount of resources) [3], [4], [8], [10], [23], [30].

2) ***Strategy-Proofness****:* It has been observed in production datacenters that users may attempt to manipulate the scheduler by lying about demands, in the hope of obtaining more resources at the expense of the other users [10], [11]. In addition to lying about resource demands, a user may manipulate the constraints as well. To eliminate the incentive of both kinds of strategic manipulation, it is important for a scheduler to be *strategy-proof*:

Definition 2 (Strategy-proofness): No user can run more tasks by lying about its resource demands and/or constraints.

From the perspective of game theory, users have *two-dimensional* strategies, one for the demands and another for the constraints. Existing schedulers, however, can only handle *one-dimensional* strategies, where users either game the demands [8], [10], [30] or the constraints [11], but not both.

3) ***Envy-Freeness****: Envy-freeness* embodies the basic means of fairness in the sense that no user "envies" the allocation of another. Intuitively, a user envies another user if it is able to run more tasks by taking that user's allocation. For weighted users, envy-freeness ensures that user i does not envy user j when the allocation of user j is scaled by w_i/w_j, where w_i and w_j are the weights of users i and j, respectively.

Definition 3 (Envy-freeness): Assume user i can run n_i tasks with its own allocation and can run $n_{i\leftrightarrow j}$ tasks after exchanging its allocation with another user j. We have $n_i \geq \frac{w_i}{w_j} n_{i\leftrightarrow j}$ for all i and j.

4) ***Pareto Optimality****:* In pursuit of high utilization, we require that no user can obtain more resources without decreasing another user's allocation. This ensures that no resource is wasted in idle unless there is a "good" reason.

Definition 4 (Pareto optimality): Any attempt to launch more tasks for a user results in fewer tasks for another.

B. Other Properties

In addition to the four required properties that must be satisfied by all means, we also require the solution to recognize two widely adopted sharing policies as special cases. In particular, when tasks do not have any constraints and can run on any machines, Dominant Resource Fairness (DRF) [10], [19] embodies the notion of fair sharing by modeling the entire datacenter as *one gigantic machine*; when sharing is limited to one resource type, Constrained Max-Min Fairness (CMMF) [11] serves as a natural extension to the conventional max-min fairness in the presence of constraints. Our solution should reduce to DRF and CMMF in the two special cases, respectively.

Definition 5 (Single machine fairness): When sharing is limited to one machine, the allocation reduces to DRF.

Definition 6 (Single resource fairness): When sharing is for one resource type, the allocation reduces to CMMF.

We shall use all seven properties as our general guideline to develop a sharing policy. Our objective is to satisfy all of them at the same time.

IV. Analysis of Existing Policies

In this section, we ask a question: can the required properties be satisfied by extending existing sharing policies? We analyze four policies: CMMF and three DRF variants, including Per-Machine DRF [8], [30], DRFH [30], and CDRF [8]. We show that they all suffer from serious problems and fail to retain all the required properties when applied to multi-resource sharing with constraints.

A. Constrained Max-Min Fairness

Ghodsi *et al.* [11] studied the problem of fair allocation for constrained jobs in the single-resource setting. The proposed scheduler, called Choosy, implements CMMF with which no user can increase its allocation without decreasing that of another with a less or equal allocation. It has been shown in [11] that CMMF is in essence a *market-based allocation* where users are given a budget and use it to buy or trade resources at given prices in a perfectly competitive market[1]. However, it has been shown in [10] that a market-based allocation is *not* strategy-proof for multi-resource sharing, even without placement constraints. We therefore exclude CMMF from further consideration.

B. DRF Variants

When tasks do not have constraints and can run on any machines, DRF [10], [19] serves as the *de facto* fair sharing policy for multi-resource allocation. The idea behind DRF is to equalize the share of dominant resources of all users. The *dominant resource* is defined, for each user, as the resource whose percentage share required by the user is maximum among all resources.

While DRF satisfies all the required properties described in Sec. III-A, it models the entire datacenter *as if* it were a gigantic machine that holds and dispatches all resources [10], [19]. We next study three variants that generalize DRF to multiple machines and constrained tasks.

1) ***Per-Machine DRF****:* The first variant we consider, called *per-machine* DRF, applies DRF to each machine *separately*, on which only users that can run tasks are considered. This policy has been analyzed in the following two simple scenarios. (1) When tasks do not have constraints and can run on any machines, per-machine DRF violates Pareto optimality with poor utilization [8], [30]. (2) When sharing is limited to one resource type, per-machine DRF reduces to *independent allocation* studied in [11] and is shown to violate sharing incentive. The lack of Pareto optimality and sharing incentive disqualifies per-machine DRF from further consideration.

[1]The definition follows the conventional micro-economic assumptions.

*2) **DRF in Heterogeneous Systems (DRFH)**:* The second DRF variant we consider, called DRFH [30], follows the same intuition of DRF but does not model a datacenter as a gigantic machine with all resources. DRFH can be naturally extended to constrained tasks by seeking a maximum feasible allocation to equalize the share of dominant resources of all users. An allocation is *feasible* if (1) no machine is allocated more tasks than it could accommodate, and (2) no task is scheduled onto a machine on which it cannot be executed.

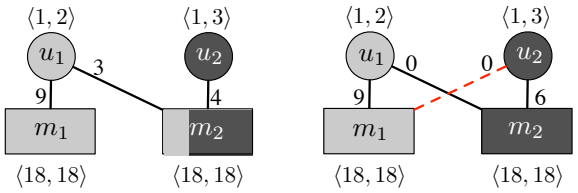
DRFH satisfies envy-freeness, strategy-proofness, and Pareto optimality. However, it violates sharing incentive, even for unconstrained tasks [8], [30]. We therefore rule it out as a desirable solution.

*3) **Containerized DRF (CDRF)**:* The last DRF variant we consider is Containerized DRF (CDRF) [8]. Similar to DRFH, CDRF models a datacenter as a shared cluster of heterogeneous machines. Better, CDRF retains the sharing incentive, a property violated by DRFH. Its intuition is to equalize the *work slowdown* of all users. The *work slowdown* is computed for each user as the ratio between the number of tasks allocated and the number of tasks the user can run when *monopolizing* the datacenter.

CDRF can be directly generalized to constrained tasks. The resulting algorithm, referred to as *constrained* CDRF, seeks a maximum feasible allocation[2] to equalize the work slowdown of all users. For example, we consider a cluster in Fig. 2a consisting of two machines each having $\langle 18$ CPUs, 18 GB RAM$\rangle$, or simply $\langle 18, 18\rangle$. There are two users, u_1 and u_2. Each task of u_1 demands $\langle 1, 2\rangle$ and can run on both machines; each task of u_2 demands $\langle 1, 3\rangle$ and must run on m_2. When monopolizing the cluster, u_1 can run 18 tasks, while u_2 can run 6, all on m_2. In this example, constrained CDRF schedules 12 tasks for u_1, among which 9 run on m_1 and 3 on m_2. It also schedules 4 tasks for u_2, all on m_2. We see that the work slowdown of each user is equalized to $\frac{2}{3}$, which is the maximum as no more tasks can be allocated.

CDRF is shown to satisfy all the required properties when tasks do not have constraints [8]. We might expect that the same result also holds for constrained CDRF by interpreting a constraint as the demand of some *virtual resource* exclusive to machines meeting the constraint. For instance, in the example of Fig. 2a, we add an infinite amount of virtual resource v to m_2 and let u_2 demand a unit of v per task. This changes the configuration vectors of m_1 and m_2 to $\langle 18, 18; 0\rangle$ and $\langle 18, 18; \infty\rangle$, respectively, and the demand vectors of u_1 and u_2 to $\langle 1, 2; 0\rangle$ and $\langle 1, 3; 1\rangle$, respectively. Here, we use a semicolon (";") to separate the real and virtual resources in the configuration/demand vector. Because m_1 does not have virtual resource demanded by u_2, we can safely remove the constraint of u_2 without worrying about misplacing its tasks onto m_1. This way, we transform a constrained sharing problem to an unconstrained problem, to which CDRF can be applied to obtain the same allocation as constrained CDRF.

[2]The definition follows the same description in Sec. IV-B2.



(a) Both users are truthful. (b) u_2 lies about constraints.

Fig. 2: An example of constrained CDRF. The user demand vectors and machine configuration vectors are given in the figure. The number of tasks allocated to a user on a machine is also presented along the edge connecting the user and the machine. (a) Constrained CDRF allocation when users are truthful. (b) User u_2 can increase its allocation by claiming that it can run tasks on m_1 (illustrated as a dotted line).

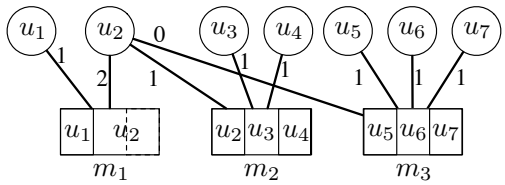


Fig. 3: An example showing that constrained CDRF is not envy-free, where u_1 envies the allocation of u_2.

However, to our surprise, constrained CDRF is *neither* strategy-proof *nor* envy-free. To see that it is not strategy-proof, we refer to the example of Fig. 2a and let u_2 lie about its constraints by claiming that its tasks can run on m_1 as well. Fig. 2b illustrates this scenario. According to the claim, u_2 can run 12 tasks when monopolizing the cluster. Constrained CDRF then schedules 6 tasks of u_2 on m_2, and 9 tasks of u_1 on m_1, equalizing the work slowdown of each user to $\frac{1}{2}$. This allows u_2 to increase its allocation by two more tasks than being honest, at the expense of u_1.

To illustrate the violation of envy-freeness, we consider a more complex example in Fig. 3, where each machine has 3 CPUs. There are 7 users each demanding one CPU per task. Their constraints and received allocations are shown in Fig. 3. In this example, when monopolizing the cluster, each user but u_2 can run 3 tasks; user u_2 can run 9 tasks. With constrained CDRF, each user but u_2 runs one task on the only machine it can use; user u_2 runs 3 tasks, two on m_1 and one on m_2. The job slowdown of each user is equalized to $\frac{1}{3}$. However, this allocation is not envy-free: user u_1 envies the allocation of u_2, with which u_1 can run 2 tasks.

The two examples in Figs. 2 and 3 reveal that placement constraints cannot be simply interpreted as the demand of some special kinds of resource exclusive to a subset of machines: such a transformation, while leading to the same allocation, is *not equivalent* from the perspective of property analysis.

Before concluding this section, we summarize in Table I the properties of the three DRF variants in the presence of

TABLE I: Properties of DRF variants and TSF in the presence of constraints: sharing incentive (SI), strategy-proofness (SP), envy-freeness (EF), Pareto optimality (PO), single machine fairness (SMF), and single resource fairness (SRF).

Property	Per-Machine DRF	DRFH	CDRF	TSF
SI			✓	✓
SP	✓	✓		✓
EF	✓	✓		✓
PO		✓	✓	✓
SMF	✓	✓	✓	✓
SRF				✓

constraints. More details are deferred to our technical report [28] due to space constraints. The failure of existing fair sharing policies motivates our design of a new alternative.

V. Task Share Fairness

In this section, we propose Task Share Fairness (TSF), a new multi-resource sharing policy that retains all the properties described in Sec. III for constrained tasks (see Table I). We start with the definition of TSF followed by an offline algorithm that achieves TSF in an idealized setting. We then analyze its properties and give a practical online algorithm that implements TSF in a dynamically shared datacenter.

A. Task Share Fairness

TSF computes the *task share* for each user, defined as the ratio between the number of tasks allocated and the maximum number of tasks the user can run *if we remove its constraints and allocate it the entire datacenter*. One can interpret task share as the *job slowdown* due to resource sharing and placement constraints. TSF applies *max-min fair allocation with respect to the users' task share*. That is, it always maximizes the lowest task share first, followed by the second lowest, *etc.*

While TSF seems to be a simple fix to CDRF, we will show through non-trivial analyses in Sec. V-C that such a simple idea is sufficient to satisfy all the desirable properties. Before we proceed, let us take a look at a running example that illustrates how TSF works.

A running example of TSF. We consider a 3-node cluster shown in Fig. 4, where machines m_1 and m_3 have the same configuration of $\langle 9 \text{ CPUs}, 12 \text{ GB RAM}\rangle$, while m_2 has $\langle 3 \text{ CPUs}, 4 \text{ GB RAM}\rangle$. There are three users. The task of u_1 demands $\langle 1 \text{ CPU}, 2 \text{ GB}\rangle$ and can run on all machines but m_3; the task of u_2 demands $\langle 3 \text{ CPUs}, 1 \text{ GB}\rangle$ and can run on m_2 only; the task of u_3 demands $\langle 1 \text{ CPU}, 4 \text{ GB}\rangle$ and can run on all machines. Suppose that we have allocated 6 tasks to u_1 on m_1, 1 to u_2 on m_2, and 3 to u_3 on m_3. We compute the task shares of three users as follows.

- For u_1, we remove its constraints and let it monopolize the entire cluster. In this hypothetical scenario, u_1 can run 14 tasks, 6 on m_1, 2 on m_2, and 6 on m_3. The task share of u_1 is $\frac{6}{14} = \frac{3}{7}$.

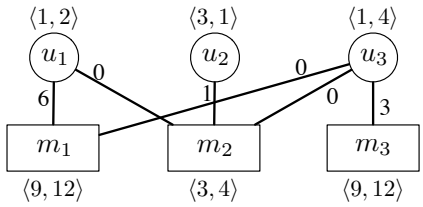


Fig. 4: An example of a TSF allocation.

- For u_2, we remove its constraints and let it monopolize the cluster. This allows u_2 to run 7 tasks, 3 on m_1, 1 on m_2, and 3 on m_3. The task share of u_2 is $\frac{1}{7}$.
- For u_3, its tasks can be executed on all machines. If monopolizing the cluster, u_3 can run 7 tasks, 3 on m_1, 1 on m_2, and 3 on m_3. The task share of u_3 is $\frac{3}{7}$.

To see that this allocation realizes TSF, we first show that the minimum task share (*i.e.*, $\frac{1}{7}$ for u_2) is maximized. Because u_2 is allocated the entire m_2, the only machine u_2 can run tasks on, its task share cannot be further increased. We next see that the second lowest task share is also maximized, because both u_1 and u_3 receive the same share $\frac{3}{7}$, and no more task can be allocated.

TSF can be directly generalized to weighted users as follows.

Definition 7 (Weighted TSF): An allocation is said to achieve *weighted TSF* if any attempt to allocate more tasks to a user would result in fewer tasks allocated to another with an equal or lower weighted task share. *Weighted task share* is defined, for each user, as the task share normalized by weight.

Unless otherwise specified, we assume weighted users and do not differentiate TSF from weighted TSF in the remainder of the paper. We also do not distinguish between task share and weighted task share.

B. Computing TSF Allocation Offline

Assuming *divisible* tasks, TSF can be ideally achieved by conventional *progressive filling* in multiple rounds. In the first round, the algorithm *equally* raises the task share of every user until it reaches a maximum. Users whose task shares cannot be further increased[3] become *inactive*, and their task shares are frozen. In the second round, the algorithm continues to act on the other users that remain *active*, raising their task shares equally without decreasing those of inactive users. The algorithm repeats round by round, until no more user is active.

Progressive filling is an idealized *offline* algorithm — it recomputes the allocation upon a user arrival or departure. While not implementable, it serves as an important tool for our property analyses (Sec. V-C) and can be well approximated by a practical *online* algorithm (Sec. V-D). We next formalize its description.

*1) **Notations**:* We assume that there are N users sharing a datacenter, which consists of M machines with R resource types. For each machine m, we normalize its resource capacity to the total availability of the datacenter and denote by $\mathbf{C}_m = \langle C_{m1}, \ldots, C_{mR}\rangle$ the normalized configuration vector,

[3]It happens when machines on which a user can run tasks are saturated.

where C_{mr} is the share of resource r available on machine m. Similarly, we normalize the task demand to the total availability of the datacenter and denote by $\mathbf{d}_i = \langle d_{i1}, \ldots, d_{iR} \rangle$ the normalized demand vector of a task of user i, where d_{ir} is the share of resource r required by the task at runtime.

Each user specifies a *constraint vector* to indicate if its task can run on a machine. Specifically, let $\mathbf{p}_i = \langle p_{i1}, \ldots, p_{iM} \rangle$ be the constraint vector of user i, where $p_{im} = 1$ if user i can run tasks on machine m, and $p_{im} = 0$ otherwise.

We assume *divisible* tasks in the offline algorithm. We denote by h_i the number of tasks user i can run in a hypothetical scenario where it monopolizes the datacenter without constraints. Suppose that user i is allocated n_i tasks and has weight w_i, its task share is computed as $s_i = \frac{n_i}{h_i w_i}$.

*2) **Offline Progressive Filling**:* We now present progressive filling in Algorithm 1. The algorithm runs in rounds. In each round t, it determines s^t, which is the maximum task share achieved for all active users, using a linear program. The linear program has non-negative variables n_{im}, indicating the number of tasks scheduled for user i on machine m. It also has three types of constraints for active users, inactive users, and machines, respectively. For *active users*, the constraints (2) ensure that their task shares are increased *equally*, *i.e.*, $\frac{1}{h_i w_i} \sum_m n_{im} p_{im} = s^t$ for all i in the active user set $\mathcal{A}^t$. For *inactive users*, the constraints (3) ensure that they should not be penalized with fewer tasks in later rounds after becoming inactive. Finally, we impose the *machine constraints* (4) to ensure that the amount of resources allocated on each machine m does not exceed its capacity.

Once the maximum task share s^t is allocated to all active users in round t, the algorithm checks each active user j to determine whether its task share can be further increased. To do so, the algorithm freezes the total number of tasks allocated to all users but j. It then tries to raise the task share of user j to the maximum, using the same linear program mentioned above. If user j cannot further increase its task share, it is saturated and is marked inactive in the following rounds, and the number of tasks allocated to it is frozen.

Algorithm 1 Computing TSF using Progressive Filling.

procedure TSF($\{\mathbf{C}_m\}, \{\mathbf{d}_i, \mathbf{p}_i, w_i\}$)
  $t \leftarrow 1$ ▷ The current round
  $\mathcal{A}^t \leftarrow \{1, \ldots, N\}$ ▷ The set of active users in round t
  $n_i \leftarrow 0$ for all i ▷ # tasks scheduled for i when it becomes inactive
  while $\mathcal{A}^t \neq \emptyset$ **do**
    $(s^t, \{n_{im}\}) \leftarrow \text{LP}(t, \mathcal{A}^t, \{\mathbf{C}_m\}, \{n_i, \mathbf{d}_i, \mathbf{p}_i, w_i\})$
    $(\mathcal{A}^{t+1}, \{n_i\}) \leftarrow \text{Freeze}(t, s^t, \mathcal{A}^t, \{\mathbf{C}_m\}, \{n_i, \mathbf{d}_i, \mathbf{p}_i, w_i\})$
    $t \leftarrow t + 1$
  return $\{n_{im}\}$
procedure LP($t, \mathcal{A}^t, \{\mathbf{C}_m\}, \{n_i, \mathbf{d}_i, \mathbf{p}_i, w_i\}$)

$$\max_{\{n_{im}\}} \quad s^t \tag{1}$$

$$\text{s.t.} \quad \frac{1}{h_i w_i} \sum_m n_{im} p_{im} = s^t, \qquad i \in \mathcal{A}^t, \tag{2}$$

$$\sum_m n_{im} p_{im} \geq n_i, \qquad i \notin \mathcal{A}^t, \tag{3}$$

$$\sum_i n_{im} d_{ir} \leq C_{mr}, \qquad \text{for all } m \text{ and } r. \tag{4}$$

  return $(s^t, \{n_{im}\})$
procedure FREEZE($t, s^t, \mathcal{A}^t, \{\mathbf{C}_m\}, \{n_i, \mathbf{d}_i, \mathbf{p}_i, w_i\}$)
  $\mathcal{I}^t \leftarrow \emptyset$ ▷ Users becoming inactivate after round t
  for $j \in \mathcal{A}^t$ **do**
    for $i \in \mathcal{A}^t \setminus \{j\}$ **do**
      $n_i \leftarrow \sum_m n_{im}$ ▷ Freeze # tasks for all but j
    ▷ Increase user j's task share to the maximum
    $(s', \{n'_{im}\}) \leftarrow \text{LP}(t, \{j\}, \{\mathbf{C}_m\}, \{n_i, \mathbf{d}_i, \mathbf{p}_i\})$
    if $s' == s^t$ **then** ▷ If task share cannot be increased
      $\mathcal{I}^t \leftarrow \mathcal{I}^t \cup \{j\}$ ▷ User j becomes inactive
    for $i \in \mathcal{A}^t \setminus \{j\}$ **do**
      $n_i \leftarrow 0$ ▷ Unfreeze # tasks of active users
  for $j \in \mathcal{I}^t$ **do**
    $n_j \leftarrow \sum_m n_{jm}$ ▷ Freeze # tasks of inactive users
  return $(\mathcal{A}^t \setminus \mathcal{I}^t, \{n_i\})$

C. Properties of TSF

As summarized in Table I, TSF satisfies all the required and desirable properties described in Sec. III. We discuss these properties and provide intuitive explanations to our analyses. We begin by showing that TSF provides service guarantees with the sharing incentive.

Theorem 1: Assume that originally each user i is given a dedicated resource pool and can run k_i tasks. If the users instead share their resources and re-allocate them using TSF with weight $w_i = k_i/h_i$ for user i, user i is able to run at least k_i tasks.

Proof: We consider a trivial allocation where each user i simply receives its own dedicated pool and runs k_i tasks. Considering user weights given by $w_i = k_i/h_i$, this allocation equalizes the task share of every user to 1:

$$s_i = \frac{k_i}{h_i w_i} = 1, \quad i = 1, \ldots, N,$$

meaning that the minimum task share of all users is 1. By definition, TSF maximizes the minimum task share of all users. Therefore, with TSF, each user is allocated at least k_i tasks, and the minimum task share is at least 1, no less than the trivial allocation. □

Theorem 1 provides a viable means to compute the correct weights from the original dedicated resource pools given to users. This is different from the case where the weights are given *a priori*. Such a subtle difference does not affect the analyses of the other properties except strategy-proofness, which is addressed separately in two cases. We begin with a simple case where the weights are given and are not manipulable.

Theorem 2: TSF is strategy-proof when users' weights are given *a priori*.

We next consider a more complex case where the weights are computed from Theorem 1 based on the dedicated resource pools given to the users. By lying about demands and/or constraints, user i can manipulate both k_i and h_i, gaming its weight w_i. This challenge is unique to TSF and has never been explored in the existing work [3], [4], [8], [10], [23],

[30]. Nonetheless, we show that TSF is immune to this kind of manipulation.

Theorem 3: TSF is strategy-proof when the weights are computed from the dedicated resource pools given to the users, using Theorem 1.

To prove Theorems 2 and 3, we consider a user and compare its allocations made in each round by progressive filling with different strategies, truthful and untruthful. Our key observation is that the allocations given by the two strategies remain the same (in terms of the task share received) until the round in which the user becomes inactive with either strategy. We can then show that the untruthful strategy is always worse off from that round onward. The complete proofs are deferred to our technical report [28] due to space constraints.

We next show that with TSF, no user would envy the allocation of another.

Theorem 4: TSF is envy-free.

The proof of Theorem 4 is based on the observation that for any two users i and j, user i does not envy user j unless the latter has a higher task share. However, by the definition of TSF, a user *cannot* increase its allocation by taking the resources allocated to another user having a higher task share. User i therefore does not envy user j. The complete proof is deferred to the appendix.

As the last required property, we show that TSF is Pareto optimal.

Theorem 5: TSF is Pareto optimal.

Proof: Let us suppose the opposite. In particular, we assume that we can improve a TSF allocation by allocating more tasks to a user, say i, without decreasing the number of tasks allocated to the other users. Because user i can be allocated more tasks at no expense of the other users' allocations, by the definition of TSF, it must be the *only one* with the minimum task share. We see in this case that the minimum task share can be strictly increased. This, however, contradicts the fact that the minimum task share is maximized with TSF. □

In addition to the four required properties, we consider the following two special cases and show that TSF possesses the two nice-to-have properties as well. First, when the cluster consists of a single machine, the task share of a user is equivalent to the share of dominant resource [10], and TSF reduces to DRF.

Theorem 6: TSF reduces to DRF when there is only one machine in the system.

Second, when sharing is limited to one resource type, the task share of a user is equivalent to the share of resource allocated to it, and TSF reduces to CMMF [11].

Theorem 7: TSF reduces to CMMF when sharing is limited to one resource type.

To summarize, TSF retains all the desirable properties described in Sec. III. We therefore use it as the notion of fairness for multi-resource sharing with placement constraints.

D. Implementing TSF Online

So far, our discussion assumes divisible tasks in an offline setting, where the allocation is recomputed whenever a user arrives or departs, and tasks that are preempted due to the change of allocation can resume computation from where they stopped at no cost. This is neither practical nor efficient. Datacenter tasks are *indivisible* and must be executed as entities. Frequent preemption leads to a waste of computation, resulting in a lower task goodput and a longer job completion delay.

Similar to existing datacenter fair schedulers [3], [10], [11], [14], [18], [31], TSF can be implemented efficiently using a simple *online algorithm* by offering idle resources to the user that is the furthest from its fair share (*i.e.,* the most "unfair" user), without preempting tasks. In particular, the algorithm serves users in an ascending order of task shares. Upon task arrival, if the datacenter is not full, the algorithm schedules the task onto a machine satisfying both the task demands and constraints; otherwise, the task is queued up for later scheduling. Upon task completion on a machine m, the algorithm offers the available resources on m to users that can run tasks on m. The algorithm keeps serving the user with the least task share, until no more task can be scheduled. The algorithm then proceeds to the next user with the second least task share, and the entire process repeats.

The online algorithm can be easily implemented in the prevalent datacenter management systems, such as Mesos [14] and YARN [25]. Its performance can be further optimized by incorporating existing scheduling techniques, such as Delay Scheduling [31] and multi-resource packing [12]. The discussion to these optimizations is, however, out of the scope of this paper. We therefore evaluate TSF using the simple online implementation.

VI. Evaluation

We have evaluated TSF using prototype implementation and trace-driven simulation. We start with our TSF implementation on Mesos as a *micro-benchmark* deployed in a 50-node Amazon EC2 cluster. We then use trace-driven simulation as a *macro-benchmark* to evaluate TSF with respect to existing fair sharing policies in large clusters.

A. Prototype Implementation

We have implemented online TSF as a pluggable job scheduler on Apache Mesos (version 0.18.1) [14]. Mesos is a cluster resource manager that allows jobs of different processing frameworks to dynamically share the cluster. In particular, Mesos launches the `mesos-slave` process on each cluster node, who monitors the resource usage and reports the available resources to the `mesos-master` over time. The `mesos-master` offers available resources to the job that is the furthest below its deserved fair share, who launches tasks on demand. The remaining resources, if any, are offered to the job that is the second furthest below the fair share, and the entire process repeats. In our implementation, the TSF scheduler keeps track of the task share of each job and offers resources in an ascending order of the task share. We have ported Mesos to allow each job to specify the placement constraints as a *whitelist* (*blacklist*) of cluster nodes (in terms of the IP addresses): only

TABLE II: Configurations of jobs for a micro-benchmark.

Config.	Job 1	Job 2	Job 3	Job 4
Start time (s)	0	10	150	150
# tasks	1000	150	100	100
CPU cores	1	0.5	0.5	1
Memory (MB)	512	512	512	512
Mean task runtime (s)	23.2	18.3	21.3	55.6
Whitelisted nodes	1-50	1-25	1-10, 26-35	1-10, 26-35
h_i	75	100	100	75

nodes in the whitelist (blacklist) can (cannot) be used to run the job's tasks.

*1) **Experiment Setup**:* We have deployed a 50-node cluster in Amazon EC2 (instance type: `t2.medium`). To create a non-trivial cluster with heterogeneous machines, we have configured the `mesos-slave` to manage 1 CPU core and 1 GB RAM in each of nodes 1-25, and 2 cores and 1GB RAM in each of nodes 26-50. The `mesos-master` runs on an additional node exclusively. We have run our experiments in the cluster with various numbers of jobs. Each job is characterized by the start time, the number of tasks, the resource demands of each task (cores and memory), the mean task runtime, and the placement constraints specified as a whitelist of nodes. All tasks of a job have the same resource demand, but their runtime may fluctuate up and down randomly by up to 20% from the mean, so as to emulate the task execution in production datacenters [20]. The mean task runtime of a job is randomly generated following the distribution of the MapReduce jobs from Facebook traces [31].

*2) **Micro-Benchmark**:* For a micro-benchmark, we highlight the behavior of our TSF implementation in a simple, easy-to-verify scenario where 4 jobs dynamically share the cluster. We summarize in Table II the job configurations as well as the maximum number of tasks a job can run when monopolizing the cluster without constraints (h_i). Fig. 5 illustrates the allocation results over time, including the CPU, memory, and task share, and is explained below.

At the beginning, Job 1 is the only active job. Because it can run tasks on all nodes, it monopolizes the cluster with task share of 1. At the 10$^{\text{th}}$ second, Job 2 starts, whose tasks are restricted to run on nodes 1-25. Ideally, with TSF, Job 2 should be allocated all 25 nodes, each running 2 tasks. This raises its task share to $2\times25/100 = \frac{1}{2}$, which is the maximum that Job 2 can achieve. Meanwhile, Job 1 is allocated nodes 26-50, each running 2 tasks, and the task share is $2 \times 25/75 = \frac{2}{3}$. After Job 2 completes, Job 1 monopolizes the cluster until Job 3 and Job 4 start. Because both jobs can run on the same set of nodes, they should receive the same task share. As expected, our TSF scheduler lets them evenly share the 20 whitelisted nodes, equalizing their task shares to $\frac{1}{5}$. The other 30 nodes remain allocated to Job 1, whose task share is stabilized at $2 \times 30/100 = \frac{3}{5}$. The allocation maintains until Job 4 starts to complete, and the relinquished resources are correctly offered to Job 3, the one having the lowest task share. Finally, we see that Job 1 gradually takes the resources released by Job 3, until it finishes.

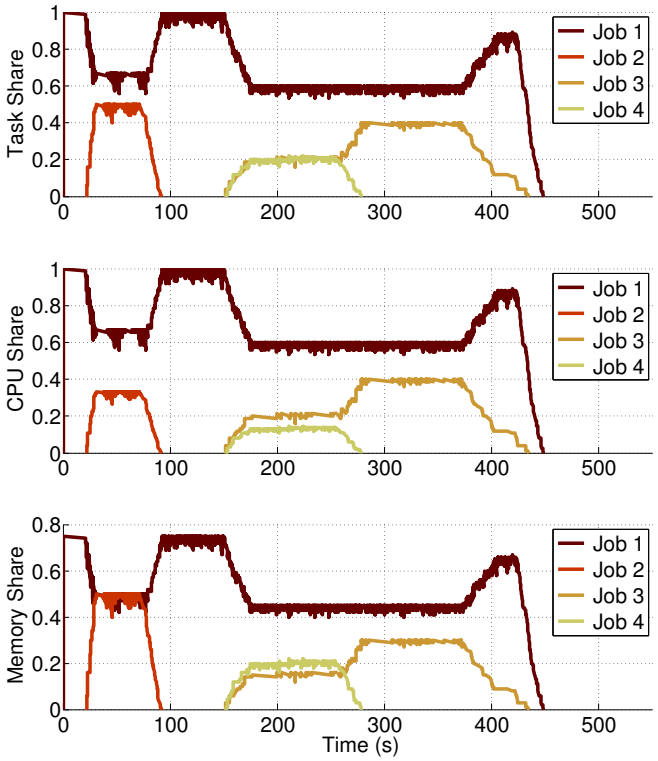


Fig. 5: CPU, memory, and task shares for four jobs over time in the experiment on Mesos.

*3) **Predictable Service Guarantees**:* We next use our implementation to confirm that with TSF, jobs are allocated more resources and finish faster than if they were run in dedicated resource pools (Theorem 1). To do so, we have run two experiments, one with static, dedicated pools and another with TSF. In the first experiment, we have divided the cluster into 4 dedicated pools for 4 jobs. Specifically, Pool 1 consists of nodes 1-10; Pool 2 consists of nodes 11-25; Pool 3 consists of nodes 26-35; Pool 4 consists of nodes 36-50. We have launched 4 jobs whose demands and task runtime follow Table II. The first two jobs can run in nodes 1-25, and the other two can run on all nodes. Nonetheless, we have restricted each job to run in its dedicated pool only. In the second experiment, we have run the same jobs but let them share the cluster using TSF. We see from Fig. 6 that TSF speeds up job completion by up to 22% over static partitioning, validating Theorem 1 in real systems.

Finally, we show that TSF provides desirable isolation for "mice" jobs, protecting their resources from being taken by "elephants". In the first experiment, we have launched two "elephants" and two "mice" (the resource demands and task runtime follow the previous setup). Each elephant job consists of 250 tasks and can run on 40 nodes (nodes 1-40 and 11-50, respectively). Mice Job 1 is very picky about the placement (100 tasks runnable on 10 nodes, 1-5 and 25-30); Mice Job 2 is less picky but has a very small number of tasks (10 tasks runnable

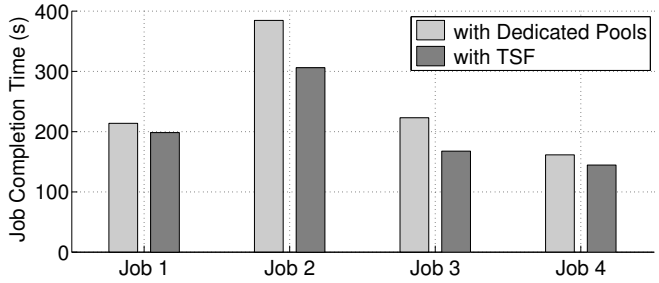


Fig. 6: The comparison of completion time of 4 jobs with static partitioning and with TSF, respectively.

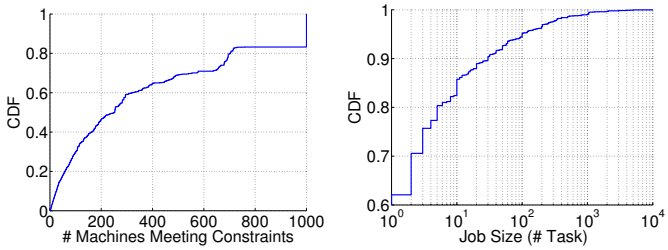
Fig. 7: Comparison of completion time of two elephant and two mice jobs, running with and without four additional elephant jobs, respectively.

on all nodes). In the second experiment, we have launched 4 more elephant jobs to congest the cluster, each having 250 tasks runnable on all nodes. Fig. 7 compares the completion time of the original 4 jobs in the two experiments. We see that while injecting more workload delays the completion of the two elephants significantly, the two mice have not been affected at all. TSF hence prevents elephants to starve mice.

B. Trace-Driven Simulation

We next evaluate TSF through a macro-benchmark in a large cluster using trace-driven simulation. In particular, we compare TSF with FIFO and four fair sharing policies discussed in Sec. IV: DRF, constrained CDRF, CMMF w.r.t. CPU share (hereafter CPU), and CMMF w.r.t. memory share (hereafter Mem). Our implementation for CPU and Mem follows Choosy [11]. We compare all the six policies with respect to performance metrics related to job/task completion. We emphasize that TSF is not designed specifically to optimize these job/task performance metrics. Rather, as shown in Table I, the main benefit of TSF is that it possesses important fairness properties that elude all of the alternatives above. Nevertheless, in this section, we would like to explore whether our "fairer" allocation is achieved at the expense of performance loss.

*1) **Simulation Setup**:* We have simulated a 1000-node cluster and fed it the workloads from a collection of Google traces [20]. The machine configurations are obtained by randomly sampling the 12k nodes in the traces. We have adopted a similar approach to [22] to generate placement



(a) Distribution of job constraints. (b) Distribution of job size.

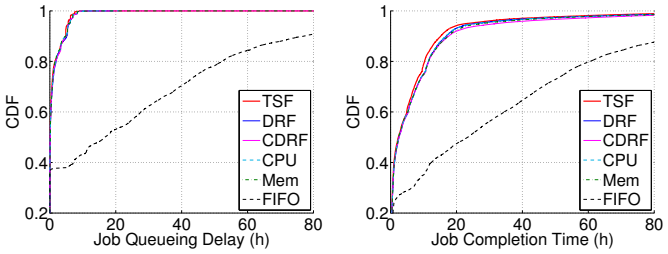
Fig. 8: Statistics of input workload used in simulations.

constraints. In particular, machines are classified into classes and are associated with *attributes* following the distribution measured in Google's production clusters (4 classes and 21 attributes in total). Tasks specify required machine attributes based on the measured distribution and can only run on machines satisfying all the required attributes. This leads to synthesized workloads with similar constraint distribution as in practice [22]. As shown in Fig. 8a, less than 20% of jobs can run on all 1000 machines, and 50% can run on 200. For the input workloads, we have sampled the tasks submitted within a 1-hour interval in the Google traces [20]. Fig. 8b gives the distribution of job size measured by the number of tasks, from which we see that the job population is dominated by mice ($\leq$ 10 tasks). Overall, the synthesized workloads consist of 180k tasks across 4,500 jobs, heavily loading the simulated cluster. We report results averaged over 50 different simulations.

*2) **Job Performance**:* We start with the comparison of *job queueing delay*, defined as the time elapsed from the job arrival to the time when its first task is scheduled. Fig. 9a gives the CDF of the job queueing delay of the six algorithms. As expected, FIFO suffers from job starvation with intolerably long queueing delay; fair sharing avoids this problem as newly submitted jobs have zero resource share and are of the highest priority to be served. Nonetheless, there remain 40% of jobs suffering from salient queueing delay because the cluster is heavily loaded, and jobs are picky about their placements.

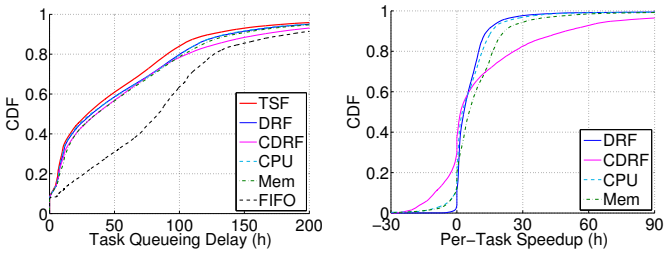
We next evaluate the job completion time using different algorithms. Fig. 9b shows the CDF. We see that by avoiding starvation, fair sharing algorithms outperform FIFO unanimously, speeding up the completion of 80% of jobs by up to 6×. Somehow surprisingly, it seems that jobs experienced similar completion time across all fair sharing algorithms! We attribute this phenomenon to the dominance of the population of mice jobs. As shown in Fig. 8b, over 60% of jobs have only one task. No matter which fair sharing algorithm is used, these single-task jobs are always scheduled at the highest priority. In general, mice jobs are less sensitive to different fair sharing algorithms. Given that their population dominates, directly comparing the CDF of job completion time does not tell a difference between TSF and the other alternatives.

To address this problem, we have divided jobs into four bins: small jobs ($\leq$ 10 tasks), medium jobs (11-100 tasks), big jobs



(a) Job queueing delay. (b) Job completion time.

Fig. 9: Job queueing delay and job completion time.



(a) Task queueing delay. (b) Per-task speedup.

Fig. 11: Task queueing delay and per-task speedup.

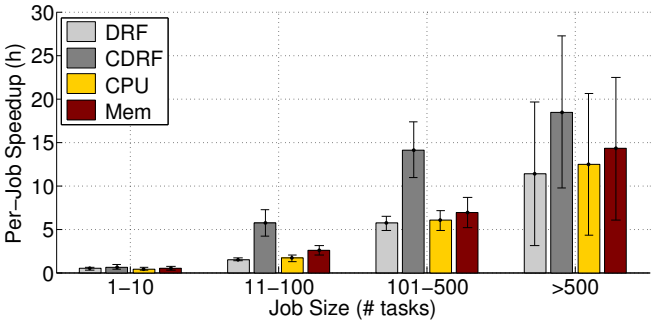


Fig. 10: Job completion speedup of TSF over four alternative fair sharing algorithms. The range of error bars corresponds to one standard deviation.

(101-500 tasks), and huge jobs (> 500 tasks). We use TSF as the baseline and compare it with the other four alternative fair sharing algorithms. In particular, we have measured per-job speedup using TSF over an alternative. *Per-job speedup* is defined for each job as the difference between its completion time using an alternative algorithm and TSF. Fig. 10 shows the average speedup of TSF over the four alternatives. The range of error bars measures one standard deviation. We see that TSF provides little speedup for small jobs. Yet, as the job size increases, the speedup of TSF becomes more salient. For medium and big jobs, the speedup is almost certain, and is on average of 10% faster. For huge jobs, however, both speedup and slowdown have been observed, because huge jobs are more likely to receive "unfair" shares (too much or too less) using alternative algorithms.

*3) **Task Performance**:* Small jobs, while dominating the population, contribute only a tiny portion of the task population. In our workloads, the biggest job consists of 20k tasks. In comparison, adding up all the tasks of small jobs (account for 86% of job population) is less than 8k. Therefore, measuring *task queueing delay* — defined as the wait time from the task submission to scheduling — would better illustrate the difference of five fair sharing algorithms. Fig. 11a shows the CDF of task queueing delay using different scheduling algorithms. As expected, FIFO suffers from the longest queueing delay due to the job starvation problem. Among the five fair sharing algorithms, TSF has the least queueing delay. This is more clearly illustrated in Fig. 11b, where we measure per-task speedup of TSF over alternative fair sharing algorithms. *Per-task speedup* is defined for each task as the reduction of task queueing delay when using TSF as compared to the other algorithm. We see in Fig. 11b that CDRF performs the worst: the speedup of 30% of tasks cannot compensate for the substantial slowdown of the other 60%. We attribute this problem to CDRF preferring jobs with more placement options, at the expense of more constrained jobs (see Sec. IV-B3). Mem does not suffer from the same design drawback and is better than CDRF. However, because the Google workload is CPU-intensive [20], fairly sharing CPU would be more efficient than Mem. This also explains why the performance of DRF and CPU are very close to each other. Nonetheless, failing to address constraints efficiently results in 60% of tasks with longer queueing delays than TSF.

VII. Related Work

Job scheduling with placement constraints. Most datacenter schedulers can only handle placement *preferences*, which are *soft* constraints that can be violated at the expense of performance penalty. For example, Quincy [15] and the Hadoop Fair Scheduler [31] treat data locality as a preference and can schedule tasks onto remote machines if those with the local data are not available. *alsched* is another scheduler supporting soft constraints [24]. It asks users to specify utility functions for constraints and schedules tasks to maximize the sum of these utilities. Compared with placement preferences, *hard* constraints are more difficult to handle [11], and may increase average task scheduling delays by 2 to 6 times [22]. Existing works, such as Choosy [11], address hard constraints only for single-resource scheduling in the units of *slots*. However, it has been widely observed that slot schedulers inevitably result in severe resource fragmentation and over-allocation [10], [12].

Multi-resource allocation. Dominant Resource Fairness (DRF) [10] is proposed as the *de facto* fairness notion for *multi-resource allocation*. DRF retains a number of desirable sharing properties, and has been widely studied in both theory and practice. Theoretically, Joe-Wong *et al.* [16] incorporated DRF into a unifying framework to characterize the fairness-efficiency tradeoff for multi-resource allocation. Gutman *et al.* [13] extended DRF to a larger family of user utilities, such as Leontief preferences. Parkes *et al.* [19] fine-tuned the analysis of DRF by allowing users to have zero demands for certain

resources and weighted endowments. The case of indivisible tasks has also been considered. Kash *et al.* [17] extended DRF to a dynamic setting where users dynamically arrive over time but never depart. Wang *et al.* [30] and Friedman *et al.* [8] extended DRF's all-in-one resource model to distributed systems with heterogeneous machines. Dolev *et al.* [7] and Bonald *et al.* [4], on the other hand, suggested other fairness notions for multi-resource allocation to achieve a better fairness-efficiency tradeoff. Tan *et al.* [23] generalized DRF to allow users to have multi-class tasks with different demands. The same technique can also be applied to TSF.

DRF has been widely implemented in systems like Apache Mesos [14] and YARN [25]. Notably, Bhattacharya *et al.* [3] extended DRF to a hierarchical scheduler in YARN to achieve fair sharing at various organizational priorities. Grandl *et al.* [12] incorporated DRF (or one of its variants) into a multi-resource packing scheduler that speeds up job completion without compromising fairness much. DRF has also been implemented as a packet scheduler in middleboxes and software routers [9], [27].

However, none of these works considers placement constraints, and their theoretical foundation, DRF, provides no fair sharing guarantee for constrained jobs. Our work bridges this gap and thus complements existing studies. The preliminary results of this work have been briefly reported in [29].

VIII. Conclusion

In this paper, we considered the open problem of multi-resource sharing for datacenter jobs with placement constraints. We showed that prevalent resource allocation policies, including Choosy, DRF and its variants, suffer from serious fairness concerns without quality-of-service guarantees. We addressed this problem with a new sharing policy, Task Share Fairness (TSF). TSF provides the *sharing incentive*, ensuring that each job receives more resources in a dynamically shared cluster than in a static, dedicated resource pool. TSF is also *strategy-proof*, in that no job can run more tasks by lying about its demands and/or constraints. TSF is *envy-free* and *Pareto optimal* as well. Our prototype implementation on Apache Mesos confirmed that TSF provides predictable service guarantees in real-world systems. Our trace-driven simulation studies further showed that TSF speeds up the completion of constrained jobs/tasks over existing fair schedulers.

Acknowledgments

We thank the anonymous reviewers for their helpful comments on improving this paper. This work was supported in part by a Microsoft Research Asia Collaborative Research Grant, and by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] CUDA. http://www.nvidia.com/object/cuda_home_new.html, 2015.

[2] Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html, 2015.

[3] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *ACM SoCC*, 2013.

[4] T. Bonald and J. Roberts. Multi-resource fairness: Objectives, algorithms and performance. In *ACM SIGMETRICS*, 2015.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX OSDI*, 2004.

[6] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM ASPLOS*, 2013.

[7] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial. No justified complaints: On fair sharing of multiple resources. In *ACM ITCS*, 2012.

[8] E. Friedman, A. Ghodsi, and C.-A. Psomas. Strategyproof allocation of discrete jobs on multiple machines. In *ACM EC*, 2014.

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

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

[11] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *ACM EuroSys*, 2013.

[12] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM*, 2014.

[13] A. Gutman and N. Nisan. Fair allocation without trade. In *Proc. 11th Intl. Conf. Autonomous Agents and Multiagent Systems (AAMAS'12)*, 2012.

[14] 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 *USENIX NSDI*, 2011.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM SOSP*, 2009.

[16] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. *IEEE/ACM Trans. Netw.*, 21(6):1785–1798, 2013.

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

[18] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *ACM SOSP*, 2013.

[19] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: extensions, limitations, and indivisibilities. *ACM Transactions on Economics and Computation*, 3(1):3, 2015.

[20] 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 *ACM SoCC*, 2012.

[21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *ACM EuroSys*, 2013.

[22] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in Google compute clusters. In *ACM SoCC*, 2011.

[23] J. Tan, L. Zhang, M. Li, and Y. Wang. Multi-resource fair sharing for multiclass workflows. *ACM SIGMETRICS Performance Evaluation Review*, 42(4):31–37, 2015.

[24] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *ACM SoCC*, 2012.

[25] 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 *ACM SoCC*, 2013.

[26] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *ACM EuroSys*, 2015.

[27] W. Wang, C. Feng, B. Li, and B. Liang. On the fairness-efficiency tradeoff for packet processing with multiple resources. In *ACM CoNEXT*, 2014.

[28] W. Wang, B. Li, B. Liang, and J. Li. Multi-resource fair sharing for datacenter jobs with placement constraints. Technical report, HKUST, https://www.cse.ust.hk/~weiwa/papers/tsf.pdf, 2016.

[29] W. Wang, B. Li, B. Liang, and J. Li. Towards multi-resource fair allocation with placement constraints. In *ACM SIGMETRICS (poster paper)*, 2016.

[30] W. Wang, B. Liang, and B. Li. Multi-resource fair allocation in heterogeneous cloud computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 26(10):2822–2835, 2015.

[31] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *ACM EuroSys*, 2010.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, 2012.

APPENDIX

In this section, we prove that TSF is envy-free. We start with the following lemma.

Lemma 1: For any two users i and j, we have

$$h_i \geq \rho_{ji} h_j, \tag{5}$$

where

$$\rho_{ji} = \min_{r:d_{ir}>0} \frac{d_{jr}}{d_{ir}}. \tag{6}$$

Proof: We consider a hypothetical scenario where user j removes its constraints and monopolizes the cluster. Let h_{jm} be the number of tasks user j executes on machine m in this hypothetical scenario. The share of resource r allocated to user j on machine m is $h_{jm}d_{jr}$. Now remove the constraints of user i and allocate it the resources that have been allocated to user j in the hypothetical scenario, with which user i can execute n_i tasks. We have

$$n_i = \sum_m \min_{r:d_{ir}>0} \frac{h_{jm}d_{jr}}{d_{ir}} = \rho_{ji} h_j. \tag{7}$$

On the other hand, by the definition of h_i, we have

$$h_i \geq n_i. \tag{8}$$

Combining (7) and (8), we see that the statement holds. □

Having established Lemma 1, we next prove envy-freeness of TSF.

Proof of Theorem 4: For the ease of presentation, we add a superscript t to the variables of the linear program solved in round t of Algorithm 1. Specifically, we denote by n_{im}^t the number of tasks scheduled for user i on machine m at the end of round t. Let $\mathbf{a}_i^t = [a_{imr}^t]$ be the allocation matrix of user i at the end of round t, where a_{imr}^t is the share of resource r allocated to user i on machine m. We have

$$a_{imr}^t = n_{im}^t d_{ir}.$$

Finally, we denote by $n_i(\mathbf{a})$ the number of tasks user i can execute given allocation $\mathbf{a}$.

For any two users i and j, let t_i and t_j be the rounds in which they become inactive, respectively. The allocations they receive are $\mathbf{a}_i^{t_i}$ and $\mathbf{a}_j^{t_j}$, respectively. Envy-freeness ensures that user i, if it takes user j's allocation scaled by $\frac{w_i}{w_j}$, runs fewer tasks, *i.e.*,

$$\frac{w_i}{w_j} n_i(\mathbf{a}_j^{t_j}) \leq n_i(\mathbf{a}_i^{t_i}). \tag{9}$$

To see this, we first show that user i does not envy user j before either of them becomes inactive, *i.e.*,

$$\frac{w_i}{w_j} n_i(\mathbf{a}_j^t) \leq n_i(\mathbf{a}_i^t), \quad t \leq \min\{t_i, t_j\}. \tag{10}$$

We derive as follows:

$$\begin{aligned}
n_i(\mathbf{a}_j^t) &= \sum_m \min_{r:d_{ir}>0} \frac{a_{jmr}^t p_{im}}{d_{ir}} \\
&\leq \sum_m \min_{r:d_{ir}>0} \frac{n_{jm}^t d_{jr}}{d_{ir}} && (p_{im} \leq 1) \\
&= \rho_{ji} \sum_m n_{jm}^t && \text{(substituting (6))} \\
&= \rho_{ji} h_j s^t w_j && (t \leq \min\{t_i, t_j\}) \\
&\leq h_i s^t w_j && \text{(Lemma 1)} \\
&= \frac{w_j}{w_i} n_i(\mathbf{a}_i^t), && (t \leq \min\{t_i, t_j\})
\end{aligned}$$

which is exactly (10). We next consider two cases.

Case 1: user j becomes inactive first, *i.e.*, $t_j \leq t_i$. In this case, we have

$$\begin{aligned}
\frac{w_i}{w_j} n_i(\mathbf{a}_j^{t_j}) &\leq n_i(\mathbf{a}_i^{t_j}) && \text{(by (10))} \\
&\leq n_i(\mathbf{a}_i^{t_i}). && (t_j \leq t_i)
\end{aligned}$$

Case 2: user i becomes inactive first, *i.e.*, $t_i < t_j$. By definition, user i cannot increase its task share after round t_i and hence cannot utilize any resources allocated to user j afterwards. This essentially suggests the following equality:

$$n_i(\mathbf{a}_j^t) = n_i(\mathbf{a}_j^{t_i}), \quad t_i < t \leq t_j.$$

Taking $t = t_j$, we have

$$\frac{w_i}{w_j} n_i(\mathbf{a}_j^{t_j}) = \frac{w_i}{w_j} n_i(\mathbf{a}_j^{t_i}) \leq n_i(\mathbf{a}_i^{t_i}),$$

where the last inequality is derived from (10). □