# Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints

Ali Ghodsi Matei Zaharia Scott Shenker Ion Stoica
UC Berkeley
{alig,matei,shenker,istoica}@cs.berkeley.edu

## Abstract

Max-Min Fairness is a flexible resource allocation mechanism used in most datacenter schedulers. However, an increasing number of jobs have hard *placement constraints*, restricting the machines they can run on due to special hardware or software requirements. It is unclear how to define, and achieve, max-min fairness in the presence of such constraints. We propose Constrained Max-Min Fairness (CMMF), an extension to max-min fairness that supports placement constraints, and show that it is the only policy satisfying an important property that incentivizes users to pool resources. Optimally computing CMMF is challenging, but we show that a remarkably simple online scheduler, called *Choosy*, approximates the optimal scheduler well. Through experiments, analysis, and simulations, we show that Choosy on average differs 2% from the optimal CMMF allocation, and lets jobs achieve their fair share quickly.

## 1. Introduction

Large clusters running parallel processing frameworks like MapReduce [8] have become a key computing platform[1]. As their workloads become more diverse, efficient resource allocation becomes even more important. One of the most widely used resource allocation mechanisms has been *max-min fairness*. Many current datacenter schedulers, including Hadoop's Fair Scheduler [34] and Capacity Scheduler [1], Seawall [29], and DRF [12], provide max-min fairness. The attractiveness of max-min fairness stems from its generality. By enabling different weights to be set for different users, a wide range of policies can be implemented, including priority, reservation, and proportional sharing [31, 32].

While there has been much work on max-min fairness for datacenter schedulers, little focus has been on max-min fairness with *placement constraints*. As recently observed, over 50% of the jobs at Google have strict, albeit simple, constraints about the machines that they can run on [28]. For example, a job might require a machine with a public IP address, particular kernel version, special hardware such as GPUs, or large amounts of memory, and might be unable to run on machines that lack these requirements. These are simple constraints that define which machines the job can run on and not complex combinatorial constraints, such as requiring two tasks to be placed on two distinct machines. Placement constraints are not specific to Google. The upcoming Hadoop Next-Generation [2] is incorporating such constraints in the resource requests of their resource management system. Differences in machines arise naturally as an organization accumulates multiple generations of hardware or purchases specialized hardware like GPUs [28].

In this paper, we consider the problem of max-min fairness in the presence of constraints. We do so in three steps. First, we propose an allocation policy called *Constrained Max-Min Fairness* (CMMF) that naturally extends max-min fairness. We show that CMMF is the *only* policy that has two important sharing properties: it incentivizes pooling of resources in common clusters and it is robust to users lying about their constraints. Second, we provide an optimal *offline* algorithm for computing CMMF, based on iterative linear programming. Third, we show, through analysis and simulations, that a very simple greedy *online* algorithm approximates the optimal offline scheduler surprisingly well. We now elaborate on these three.

CMMF naturally extends max-min fairness. Specifically, CMMF recursively maximizes the allocation of the user with the lowest share, then of the user with the second-lowest share, etc, subject to satisfying all users' constraints. Compared to other allocation policies, CMMF has two desirable properties. First, CMMF is the only policy that satisfies the *sharing incentive property* [12], which ensures that users on a shared cluster do not get a smaller allocation than if they ran on a dedicated cluster of $\frac{1}{n}$ the size (assuming $n$ users). This ensures that users (or groups of users) are "bet-

[1] This research is supported in part by NSF CNS-1161813, NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware and Yahoo!.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*Eurosys'13* April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04…$15.00

ter off" pooling their resources in a common cluster. Other simple allocation policies lack this property. Second, CMMF is *strategy-proof*, that is, users cannot increase their shares by lying about their demands[2].

Computing optimal CMMF allocations is challenging. We give a solution that uses iterative linear programming. The solution is particularly useful for coarse-grained scheduling in an *offline* fashion, suitable in HPC/Grid contexts [25].

The offline solution does not scale to high rates of scheduling decisions per second. For this reason, we seek a simple approximation. Towards this end, we analyze the performance of the simplest and most natural online scheduler for this problem. We refer to it as *Choosy*: each time a machine is free, greedily assign it to the user with the lowest current share that can run on that machine. Despite the complexity of the optimal offline scheduler, Choosy turns out to approximate offline solutions well.

We analyze why the simple Choosy scheduler performs well with respect to the considerably more complex optimal CMMF, and confirm these results with simulations and experiments. For our experiments, we implement Choosy in the Mesos resource manager [14], and use data on placement constraints from Google [28]. The experiments and simulations show that Choosy closely approximates the optimal scheduler, providing response times that differ by less than 2% on average and letting new jobs acquire their fair share of the cluster quickly.

Choosy fits current datacenter schedulers—such as the Hadoop Fair Scheduler [34], Capacity Scheduler [1], and Mesos [14]—that quickly need to schedule large numbers of fine-grained tasks online. It is, thus, a practical choice for implementing max-min fairness with constraints.

## 2. Background and Model

We first motivate the problem and describe our model of placement constraints.

### 2.1 Motivation

The number of machines and applications running in datacenters is steadily increasing, leading to a diverse set of applications running over heterogeneous hardware. This has resulted in applications having constraints on the machines they can run on; for instance, a DNS service might need to run on machines that have a public IP address. While the nature of the constraints might vary, they can usually be classified into two categories: *hard* and *soft constraints*. A job cannot run if its hard constraints are violated. But it can run, possibly with degraded performance, if its soft constraints are not met. An example of a hard constraint is the presence of a GPU, while a soft constraint is data locality. In our work, we focus only on hard constraints, which are more difficult to handle (*c.f.,* §8). Furthermore, we do not consider *combinatorial constraints* that specify rules on when different combinations of machines are desirable.

According to Sharma *et al.* [28], most constraints in practice are simple. Approximately 50% of Google's jobs only have the simple non-combinatorial constraints that we consider, compared to 11% having complex ones. In addition, constraints have a significant impact on job performance, considerably inflating the scheduling time of jobs in Google's proprietary scheduler.

### 2.2 Modeling Job Constraints

We seek a simple job constraint model that captures the intricacies of job placement constraints. In practice, the conditions behind constraints can be complex, involving, for example, Boolean expressions about various machine attributes (*e.g.,* public IP AND memory $>$ 16 GB) [28, 30]. To support arbitrary such conditions, we will work, not with the conditions themselves, but with just the *results* of evaluating them. Specifically, we require, for each user (job), a Boolean specifying whether it can use each machine. We encode this information in a *constraint graph*, where the vertices are machines and users, and there is an edge between each user and each of the machines that user can run on.

Figure 1 illustrates the model. User $u_1$ specifies the constraint set $c_1 = \{m_1, m_4\}$ (*i.e.,* she can only run on $m_1$ or $m_4$), $u_2$ specifies $c_2 = \{m_3, m_4\}$, $u_3$ specifies $c_3 = \{m_2, m_3, m_4, m_6, m_7\}$, and $u_4$ specifies $c_4 = \{m_5, m_6, m_7, m_8, m_9, m_{10}\}$. We urge the reader to think about what the "right" allocation should be, assuming equally important users.

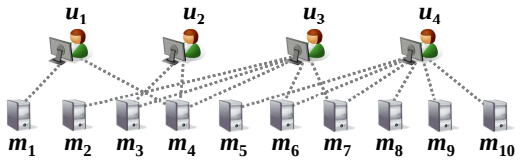


Figure 1: Example of 4 users with constraints on 10 machines.

## 3. Desirable Scheduler Properties

In this section, we motivate and define two desirable properties of a fair allocation policy in the presence of constraints: (i) constrained sharing incentive, and (ii) strategy proofness.

### 3.1 Sharing Incentive

As observed in DRF [12], a highly desirable property of any allocation policy in datacenters is *sharing incentive*. In a nutshell, this property ensures that users get more resources when pooling resources in a dynamically shared cluster than statically partitioning the resources among themselves. For example, in the case of $n$ identical users, each user should be able to get at least $\frac{1}{n}$ of the resources in a shared cluster regardless of the demand of other users. Otherwise, this user would be better off in a dedicated cluster containing only $\frac{1}{n}$ of the machines. This is especially important when

[2] From a microeconomics perspective, CMMF is characterized by Nash bargaining [24], which is different from previous fair schedulers—such as DRF [12]—that are built on the Kalai-Smorodinsky allocation [17]. Despite this, CMMF is strategy-proof.

sub-divisions of an organization might consider buying a dedicated cluster.

Unfortunately, it is easy to see that this version of the sharing incentive property cannot be guaranteed in the presence of arbitrary constraints. For example, consider ten machines and ten users, with the first nine users only able to use the first machine, whereas the last user is able to use any of the ten machines. It is impossible to guarantee that each user gets one machine, which is each user's share according to the sharing incentive property.

Fortunately, there is a simple generalization of the sharing incentive property to handle constraints:

> *Constrained sharing incentive*. Assume each user $i$ contributes $k_i$ machines to a common pool of machines, and that each user can use at least the machines she contributed. Then, user $i$ should be able to get at least $k_i$ machines.

Note that this definition assumes that user $i$ is "entitled" to at least $k_i$ machines, *i.e.,* her weight is $k_i$. In addition, user $i$ can also get more than $k_i$ machines if some of the other users do not currently need their full entitlements.

### 3.2 Strategy-Proofness

Prior work has shown that users are willing to manipulate the scheduling system to gain more resources, even within a single organization [12]. For example, in one company, jobs were required to have high utilization in order to be given dedicated machines, so users artificially inflated their utilization by adding busy-loops to their code. At another company, users of a Hadoop cluster observed that the slots for map tasks were often contended whereas slots for reduce tasks were usually free, so they rewrote their job to run the entire computation in the reduce task, acquiring nodes faster but bypassing all the efficiency benefits associated with data locality-aware map scheduling [12].

Allocation policies can avoid manipulation by ensuring that they are robust to misreported requirements. This is captured by the *strategy-proofness* property:

> *Strategy-proofness*. A user should not be able to raise her allocation of desirable machines by misreporting the set of machines she is constrained to run on.

An example of a policy that does not meet strategy-proofness would be one that tries to assign each user a share proportional to the total number of machines that user can run on. In this case, a user might be better off misreporting her job as being able to run on any machine, as this might let her receive more machines that she wants in addition to some that she does not want. Such scenarios have been observed for extensions of max-min fairness to settings with multiple resource types [11, 12], so we wish to avoid them for the constrained sharing case.

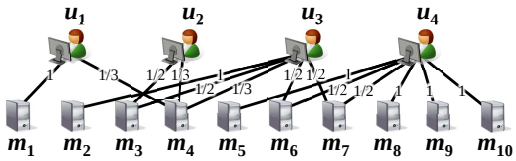


Figure 2: Independent allocation for the problem in Figure 1. The numbers show the fraction of a machine that each user is given.

## 4. Policies for Allocation with Constraints

In this section, we study three resource allocation policies. First, we consider a simple natural policy, *independent allocation*, that, unfortunately, does not satisfy the sharing incentive property, demonstrating how this property can be violated. Next, we introduce our approach, Constrained Max-Min Fairness (CMMF). Finally, we discuss how CMMF relates to market-based allocation. Some of the proofs are in the appendix of this paper.

### 4.1 Independent Allocation

An intuitive allocation policy, which we call *independent allocation*[3], is to consider each machine in isolation and give each of the $k$ users that can use that machine $\frac{1}{k}$ of it. In effect, this treats each machine as a separate *type* of resource.[4]

One advantage of the independent allocation policy is that it is *additive*. This means that the independent allocations can be computed by allocating each machine in isolation and then taking the union of the results.

For the example problem in Figure 1, independent allocation gives the following shares, shown in Figure 2: $a_1 = \{m_1, \frac{m_4}{3}\}$, $a_2 = \{\frac{m_3}{2}, \frac{m_4}{3}\}$, $a_3 = \{m_2, \frac{m_3}{2}, \frac{m_4}{3}, \frac{m_6}{2}, \frac{m_7}{2}\}$, and $a_4 = \{m_5, \frac{m_6}{2}, \frac{m_7}{2}, m_8, m_9, m_{10}\}$.

Unfortunately, this policy lacks an important property:

THEOREM 1. *Independent allocation does not satisfy the constrained sharing incentive property.*

**Proof** Consider two machines and two users, each user contributing one machine, where the first user can use both machines, while the second user can use only the second machine. Suppose both users have infinite demands. Independent allocation will then assign the first machine to the first user, as the second one cannot use it. In addition, it will assign half of the second machine to the first user, and half to the second user. This violates the constrained sharing incentive, as user two only gets half a machine. □

Thus, pooling resources in a common cluster may *hurt* a user's allocation with independent allocation—an undesirable property in a shared cluster. Independent allocation does

[3] This allocation policy matches the game theory concept of *Shapley Value*, which is important in cost- and gain-sharing theory [23].

[4] In practice, allocation does not have to be performed by subdividing machines. There are often multiple machines satisfying each equivalence class of constraints (*e.g.,* having both a public IP and a GPU), and these can be divided $k$ ways among the interested users.

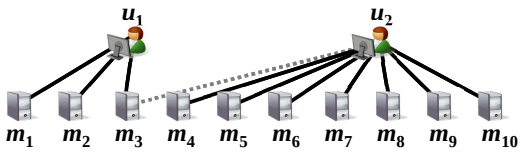


Figure 3: A simple CMMF example with two users and ten machines. Solid lines indicate allocations, whereas dashed lines indicate that a machine can be used by a particular user.

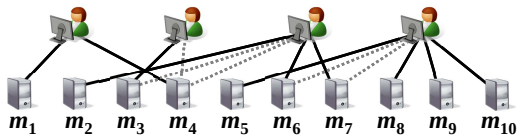


Figure 4: CMMF allocation of the example in Figure 1.

turn out to be strategy-proof (see Appendix), but we view the lack of sharing incentive as a serious reason not to use it.

We thus propose an alternate policy, CMMF, that does provide the sharing incentive, and is also strategy-proof.

### 4.2 Constrained Max-Min Fairness (CMMF)

Our policy, *Constrained Max-Min Fainess* (CMMF), attempts to recursively maximize the allocation of the worst-off user, then the second-worst-off user, and so on.

Specifically, we define a CMMF allocation to be an assignment in which *it is not possible to increase the minimum allocation within any subset of users by reshuffling the machines given to these users*. This captures the intuitive notion that we cannot increase the share of any user without decreasing the share of another user with a lower share.[5]

For example, in the simple two-user setting in Figure 3, CMMF allocates $a_1 = \{m_1, \cdots, m_3\}$ to user $u_1$, and $a_2 = \{m_4, \cdots, m_{10}\}$ to user $u_2$. These assignments are represented by full lines in the figure. Note that while both users can use machine $m_3$, this machine is given to $u_1$, as $u_1$ cannot use as many other machines as $u_2$. It is easy to check that this allocation satisfies CMMF: user $u_1$ cannot use more than the three machines she already has, while $u_2$ cannot increase her allocation without reducing the allocation of user $u_1$, who already has a lower allocation.

To better understand CMMF, we consider the more complex example given by Figure 1. If we assume that only whole machines can be allocated (*i.e.,* machines are *indivisible*) then one constrained max-min fair allocation is given by Figure 4: $a_1 = \{m_1, m_4\}$, $a_2 = \{m_3\}$, $a_3 = \{m_2, m_6, m_7\}$, and $a_4 = \{m_5, m_8, m_9, m_{10}\}$. To see that this is indeed a CMMF allocation, note that it is not possible to increase the minimum allocation across any subset of users by reshuffling the allocations between the users in that subset. For example, consider the subset of users $\{u_1, u_2\}$: while one could give $m_4$ to $u_2$ instead of $u_1$, this does not increase the minimum allocation across the two users, which still remains one (before re-allocation, $u_2$ has one machine, while after $m_4$'s reallocation, $u_1$ has one machine). Note that this alternative allocation is also a CMMF allocation, which shows that the CMMF allocation is not always unique.

If we allow fractional allocation of the machines, the CMMF allocation is the same as above except that the first two users get assigned half of the second machine each, *i.e.,* $a_1 = \{m_1, \frac{m_4}{2}\}$ and $a_2 = \{m_3, \frac{m_4}{2}\}$.

#### 4.2.1 Weighted CMMF

CMMF can be easily generalized to include a weight $w_i$ for each user $u_i$. For each allocation vector $\langle x_1, \cdots, x_n \rangle$, define a corresponding weighted allocation vector $\langle \frac{x_1}{w_1}, \cdots, \frac{x_n}{w_n} \rangle$, where $x_i$ is the total number of machines allocated to user $i$, and $w_i$ is the weight of user $u_i$. Then a weighted CMMF allocation is one in which, for each subset of users, one cannot reshuffle the machines assigned to that subset to increase the minimum value of $\frac{x_i}{w_i}$ across those users.

#### 4.2.2 Properties

THEOREM 2. *When users have infinite demands (*i.e., *can launch an arbitrary number of tasks), the constrained sharing incentive property is equivalent to weighted CMMF.*

**Proof** Suppose that each user $i$ brought $k_i$ machines that she can use to a shared cluster. Then the only allocation vector that satisfies the constrained sharing incentive is $\langle k_1, k_2, \ldots, k_n \rangle$, *i.e.,*, giving user $i$ exactly $k_i$ machines. If we set the weight of each user in weighted CMMF to $k_i$, then the weighted allocation vector for this allocation is $\langle 1, 1, \ldots, 1 \rangle$. This is a feasible allocation, and it is not possible to increase one of the components without decreasing another (since all $k_1 + \cdots + k_n$ machines have been allocated), so this is the weighted CMMF allocation. □

Note that when some users do not have infinite demand (*e.g.,* a user brought 10 machines to the cluster but currently only needs 5), there are multiple possible allocations that satisfy the constrained sharing incentive. However, CMMF is the "natural" extension of max-min fairness in this case, as it maximizes the share of the least well-off user.

We also show that CMMF is robust to misreporting (proof in Appendix):

THEOREM 3. *CMMF is strategy-proof.*

In fact, a user that misreports a superset of the machines she can run on can actually get hurt under CMMF. Consider two users and two machines. The first user can only run on $m_1$, whereas the second user can run on either $m_1$ or $m_2$. If the first user misreports that she can run on either machine, then CMMF might allocate the $m_2$ to this user and the $m_1$ to the second user. But the first user cannot $m_2$. This is not true under independent allocation, where additivity ensures that the first user still gets the same share of $m_1$.

[5] The reason for the more complicated definition is to handle the discrete case, where we *could*, for example, increase the allocation of a user with 1 machine by giving it a machine from a user with 2 of them, but this would not change the overall fairness of the allocation.

Finally, we look at the relationship of CMMF to pricing.

### 4.3 Market-Based Allocation

An alternate approach to scheduling is to use a market-based solution. Users are given a budget, prices are set for each machine, and users use their budget to buy machines.

The key question in this model is how to set the prices. For a perfectly competitive market,[6] the prices and allocation can be computed without resorting to trading, through the Nash bargaining solution [27, Thm 3]. This solution first assigns machines that are only usable by one user to that user. It then finds the allocation that maximizes the product of the utilities of the users, where each user's utility is defined as the number of contended machines she receives.

After removing the non-contended machines, the Nash bargaining solution turns out to be equivalent to CMMF. Thus, market-based allocation and CMMF produce the same assignments as long as each machine can be used by at least two users. Note that this is unlike in other datacenter scheduling settings, such as the multi-resource setting considered in Dominant Resource Fairness [12], where market-based allocation is *not* strategy-proof, and schedulers use a different Kalai-Smorodinsky solution instead [17].

In summary, CMMF is the only policy that provides the sharing incentive property, and is additionally strategy-proof. We view both requirements as essential.

## 5. Computing CMMF Allocations Offline

In this section, we explore how to compute constrained max-min fair (CMMF) allocations. We initially assume that there are a fixed set of users with constraints on the machines, and that all the machines are currently idle. This represents the *offline* setting [13]. We later derive an online algorithm that approximates this solution.

Like other max-min fair solutions [4, 12], the general approach we take is *progressive filling*. The algorithm starts by increasing all users' allocations equally until the maximum possible level, $M_1$. Once this maximum value has been found, we determine which users can continue increase their allocation beyond $M_1$ without decreasing other users below $M_1$. We call these users *active*. Round 2 of the algorithm freezes the inactive users' number of machines at $M_1$, while raising the allocations of the active users equally to a new maximal level $M_2$. This process repeats until there are no more active users. Note that the freezing only applies to the quantity of allocated machines and not the actual machine allocation, which may change.

Before going through the details, the following example demonstrates the algorithm. Consider five machines and two users with constraints $c_1 = \{m_1, m_2\}$ and $c_2 = \{m_2, m_3, m_4, m_5\}$. In the first round, one could end up with the allocation $a_1 = \{m_1, m_2\}$ and $a_2 = \{m_3, m_4\}$. Thus, $M_1 = 2$ since both users have been given two machines. Since it is impossible to increase user 1's allocation further, but it is possible to increase the user 2's allocation without decreasing user 1's, the second round only has user 2 as active and requires that user 1 maintains two machines. Thus the second round ends with $a_1 = \{m_1, m_2\}$ and $a_2 = \{m_3, m_4, m_5\}$. Now it is impossible to increase any user without hurting others, so we are done.

[6] With the usual micro-economic assumptions, such as price-taking users.

### 5.1 Algorithm for Divisible Resources

The actual allocation in each round can be found in different ways depending on whether resources are *divisible* (*i.e.,* users can receive fractional shares of a machine).

With divisible resources, we can solve for $M_i$ in each round using a linear program. The linear program has non-negative variables $x_{i,j}$, describing how much of machine $j$ user $i$ is allocated, and two types of constraints: *machine constraints* and *user constraints*. The machine constraints are $\sum_i x_{i,j} \leq 1$ for all $j$, *i.e.,* they ensure that all users are together allocated at most one unit of each machine. In the first round, the user constraints are $\sum_j x_{i,j} \geq M_1$ for all $i$, *i.e.,* that each user gets at least $M_1$ number of machines, and the linear program attempts to maximize $M_1$. In subsequent rounds, the user constraints are updated for the active users to $\sum_j x_{i,j} \geq M_2$, and the program aims to maximize $M_2$. Thus, inactive users are guaranteed to get $M_1$ units, while all active users are increased simultaneously.

Finally, to determine which users remain active after each round, we can solve a linear program for each user $i$ that leaves all users except $i$ at their previous rounds' allocations, while maximizing the allocation of $i$. If it is possible to increase $i$'s allocation, then $i$ is considered active. This test can be done independently for each user because if $k$ users can *separately* increase their allocation without hurting any other user, then they can also increase their allocation *together* without hurting others. This can be done by taking a linear combination of the allocations that increased each user's share individually: the resulting allocation will still be feasible, but will simultaneously increase all their shares.

**Example** The following example, illustrated in Figure 5, shows how the algorithm works. Consider 9 machines and 3 users with constraints $c_1 = \{m_1, m_2\}$, $c_2 = \{m_2, \ldots, m_5\}$, and $c_3 = \{m_5, \ldots, m_9\}$. In the first round there are 9 machine constraints of the form $x_{1,j} + x_{2,j} + x_{3,j} \leq 1$ for every $j$. Each user $i$ has a constraint of the form $x_{i,1} + \cdots + x_{i,9} \geq M_1$. The program attempts to maximize $M_1$. It finds the allocation in Figure 5a, where each user has been given two machines ($M_1 = 2$). Now a saturation test is done for each of the three users, and both $u_2$ and $u_3$ are determined to be active as they can be allocated machines $m_5$ and $m_8$ respectively. Thus the algorithm enters round 2, with the same machine constraints but these two users' constraints updated to: $x_{1,1} + x_{1,2} + x_{1,3} \geq M_1$ and $x_{i,1} + x_{i,2} + x_{i,3} \geq M_2$. The new program maximizes $M_2$ and finds the allocation given by Figure 5b. Note that the

```
procedure OFFLINE_SOLVER(C)
    r := 1                                          ▷ Current round
    M_0 := 0, S_0 := ∅                              ▷ Max level and users stuck
    while true do
        (M_r, x_{i,j}) := LP(C, r, M_1, .., M_{r-1}, S_1, .., S_{r-1})
        S_r := SATURATED(r, M_1, .., M_r, S_1, .., S_r)
        if |S_1 ∪ ··· ∪ S_r| = N then               ▷ All users saturated?
            return x_{i,j}                          ▷ Return matrix of allocations
        r := r + 1

procedure SATURATED(r, M_1, .., M_r, S_1, .., S_{r-1})
    U := {1, ..., n} \ (S_1 ∪ ··· ∪ S_{r-1})        ▷ Active users
    S := ∅                                          ▷ Users saturated in this round
    for u ∈ U do
        S_r := U\{u}                                ▷ Saturate all but u and solve LP
        (M_{r+1}, x_{i,j}) := LP(C, r + 1, M_1, .., M_r, S_1, .., S_r)
        if M_{r+1} = M_r then                       ▷ Did u fail to improve?
            S := S ∪ {u}
    return S

procedure LP(C, r, M_1, .., M_{r-1}, S_1, .., S_{r-1})
    maximize M_r subject to:
        Σ_{j=1}^{m} x_{i,j} ≤ C_{i,j}                 (1 ≤ i ≤ n)
        Σ_{j=1}^{m} x_{i,j} ≥ M_l                     (1 ≤ l ≤ r − 1, i ∈ S_l)
        Σ_{i=1}^{n} x_{i,j} ≤ 1                       (1 ≤ j ≤ m)
        M_i, x_{i,j} ≥ 0                              (1 ≤ i ≤ n, 1 ≤ j ≤ m)
    return (M_r, x_{i,j})                           ▷ Value and allocations found
```

Algorithm 1: Offline CMMF scheduler for divisible resources.

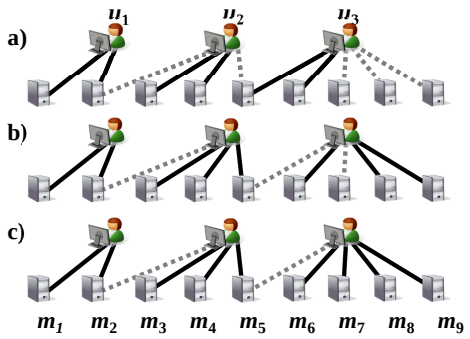


Figure 5: (a) Allocation after round 1 of the offline CMMF algorithm, in which all users are given 2 machines. (b) Allocation after round 2, where $u_2$ and $u_3$ are active and each gets 3 machines. (c) Final allocation after round 3, where $u_3$ is active.

*set* of machines allocated to both active and inactive users can shift, but their *number* of machines will obey the user constraints. For the third round, only $u_3$ is active, which leads to the final allocation in Figure 5c.

**Optimization: Coalescing Identical Machines** While a cluster can consist of tens of thousands of machines, the total number of *unique* machine configurations is substantially lower [28]. We refer to each such configuration as a *machine type*. Thus, we can reduce the number of variables in our linear program by having one variable for each machine type, and thus make the problem faster to solve. Resources with identical constraints also arise in when each machine is partitioned into multiple "slots" that can run tasks [1, 34].

**Algorithm for Non-Divisible Resources** When resources are not divisible, it is possible to find the allocations at each round using a network flow algorithm instead of a linear program, which will efficiently compute *integer* allocations for the types of constraints in our matching problem. The details of this are out of scope for this paper.

## 6. Choosy: An Online CMMF Scheduler

The previous section outlined an offline algorithm for CMMF. Offline scheduling is often used in HPC and Grid environments, in which jobs are typically *coarse-grained*, using several machines for long periods of time. In these environments, jobs tell the scheduler an upper bound on how long they will run, and the scheduler can make a plan for when to launch each job [22].

In datacenter environments, however, schedulers need to make *online* decisions on which task to allocate resources to whenever a new job is added or when resources free up. This is because data-intensive computing frameworks like MapReduce and Dryad divide jobs into *fine-grained* tasks, each of which uses a slice of a machine for a short amount of time that is not known to the scheduler in advance [14].[7] Typical datacenter schedulers, including the Hadoop Fair Scheduler and Mesos, make thousands of scheduling decisions per second as tasks continuously finish and their resources need to be allocated to new tasks [14, 34].

The offline scheduler is not applicable in this environment because, if we simply called it each time a resource freed up, we might have to reallocate a large number of machines to obtain the configuration it returns. This could include both migrating tasks and preempting (pausing) them. In general, current data-intensive computing frameworks do not support migration or preemption, and such facilities come at a cost.[8] In addition, the offline solution is expensive to compute; we show in §7.4 that it can take more than 1s for large clusters, which is too slow to make thousands of decisions per second.

Nonetheless, we can approximate the optimal allocation by looking at *what the offline scheduler would do if it were restricted to not migrate or preempt existing tasks*. In particular, suppose that a cluster is full with tasks from $n$ users, and one machine, $m_i$, frees up. Then to obtain the "fairest" (lexicographically largest) allocation vector subject to the constraint of not touching existing tasks, we need to allocate $m_i$ to the user with the lowest share that is capable of using the machine. This is precisely our simple online scheduler, which we call *Choosy*:

[7] The fine-grained model is necessary to achieve good data locality in these environments, by allowing jobs to take turns accessing data distributed across machines, and facilitates load balancing and fault recovery [14].

[8] Some systems can kill tasks [16, 34], but do it rarely to avoid wasted work.

Whenever a resource frees up, assign it to the user with the lowest current allocation whose constraints the resource satisfies.

To our surprise, this remarkably simple scheduler approximates the optimal offline scheduler quite well; a user receives her fair share within a few seconds of joining the cluster (see Section 7). The rest of the paper investigates why this is so, and how well Choosy approximates CMMF.

### 6.1 Convergence Properties

How well does Choosy approximate CMMF, given that it only schedules one machine at a time? The following theorem (and prove it in the appendix), shows that for any initial allocation, Choosy will eventually converge close to the CMMF allocation.

THEOREM 4. *Consider a cluster with $n$ users, each of which have infinite demand and finite task lengths. Then no matter what allocation it starts with, Choosy converges to an allocation where each user's share is at most $2n$ machines less than its share in some optimal allocation.*

Note that the number of users, $n$, is typically far smaller than the number of machines in a cluster. Thus, being $2n$ machines from the optimal global allocation is for practical purposes close to the CMMF allocation.

### 6.2 Ramp-up time

The previous sub-section showed that that Choosy eventually converges close to the optimal CMMF allocation, here we analyze how long convergence takes.

A newly joined user must wait for her machines to free up in her constraint set so that she can ramp-up to her fair share. In an offline scheduler, the optimal allocation can be computed as soon as the user joins, and preemption can be used to ensure that the user ramps up quickly. In the online setting, resources free up one by one and are greedily assigned to whoever can use them. Here we analytically show that in this simple online setting, without preemption, users quickly reach their fair share.

The ramp-up time depends on how many machines a user wants and how many machines she can run on. The *pickiness* $p$ of a user is defined to be the ratio between her *target count* $k$ and *constraint count* $n$. The target count of a user is the number of machines the user is entitled to according to her fair share. The *constraint count* of a user is the number of machines she is constrained to use. For example, if a user can only run on $n = 20$ machines and has a fair share of $k = 5$ machines, her pickiness is $p = 0.25$. The *ramp-up time* is the time the user has to wait until she has been allocated her target count.

Computing the ramp-up time requires computing the probability distribution of the remaining task duration, given a user that joins at time $t$. For exponentially distributed task durations the remaining task duration is also exponentially distributed due to memorylessness. However, analysis from existing clusters show that task durations have been observed to be Pareto distributed [10], complicating the analysis due to the *waiting-time paradox* [6].

**Distribution of Remaining Task Duration.** Assume the cluster is full when a new user $u$ arrives at time $t$. Assume further that the task duration in the cluster is given by the CDF $F_T$ (and PDF $f_T$). The average duration of the tasks running at time $t$ is longer than the average duration of all tasks due to the aforementioned waiting-time paradox [6]. Let $F_Y$ denote the CDF for the *entire* duration of the tasks at time $t$, calculated as follows. The likelihood of user joining during an interval of length $y\,dy$ is proportional to the length of the interval times the frequency that the interval occurs, *i.e.,* $f_Y(y)dy \propto y f_T(y)dy$. Normalizing $f_Y(y)$ such that it integrates to 1 gives the task duration distribution observed at time $t$:

$$f_Y(y) = \frac{y f_T(y)}{E[T]}$$

We are interested in the distribution of the task duration *remaining* at time $t$. Conditioned on the duration of the task length $y$ then $t$ is distributed uniformly over $y$, *i.e.,* $f_{R|Y}(r, y) = \frac{1}{y}$ for $0 < r \leq y$, where $R$ is a random variable of the remaining task duration. Thus, the joint distribution is $f_{R,Y}(r, y) = \frac{f_T(y)}{E[T]}$. By convoluting we get the PDF for the remaining task durations at time $t$ to be:

$$f_R(r) = \int_r^\infty \frac{f_T(y)}{E[T]} dy = \frac{1 - F_T(r)}{E[T]}$$

The CDF for the Pareto distribution with shape $\alpha$ and minimum value $m$ is:

$$F_T(t) = \begin{cases} 0 & \text{if } t < m \\ 1 - \left(\frac{m}{t}\right)^\alpha & \text{if } t \geq m \end{cases}$$

For $\alpha > 1$ the expected value of $T$ is:

$$E[T] = \frac{\alpha m}{\alpha - 1}$$

This lets us compute the distribution of the remaining time $R$ for Pareto distributed task durations with shape $\alpha$ and minimum value $m$:

$$f_R(x) = \begin{cases} \frac{\alpha-1}{\alpha m} & \text{for } x < m \\ \frac{m^{\alpha-1}}{x^\alpha} \frac{\alpha-1}{\alpha} & \text{for } x \geq m \end{cases}$$

The CDF of $R$ is:

$$F_R(x) = \begin{cases} \frac{\alpha-1}{\alpha m} x & \text{if } x < m \\ 1 - \frac{m^{\alpha-1}}{x^{\alpha-1}\alpha} & \text{if } x \geq m \end{cases}$$

The $k$–th *order statistic* [7] can then be applied to $F_R$ to compute the ramp-up time. Though the closed formula of the order statistic is somewhat involved, it is known to be approximated for large $n$ by the normal distribution with mean $F_R^{-1}\left(\frac{k}{n}\right)$.

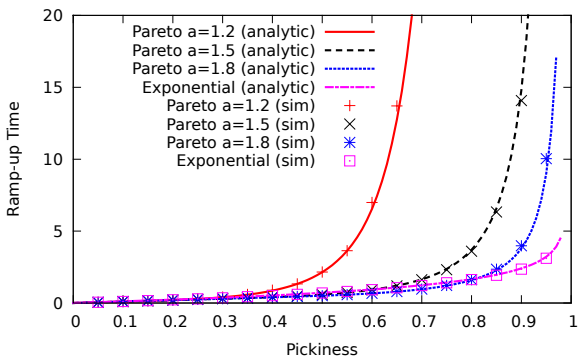


Figure 6: Ramp-up time for varying levels of pickiness. The y-axis shows the ramp-up waiting time as a factor of the mean task duration for different levels of pickiness on the x-axis.

For the Pareto distribution with shape $\alpha$ and minimum value $m$ the inverse CDF for $R$ is:

$$F_R^{-1}(x) = \begin{cases} \frac{\alpha m}{\alpha - 1} x & \text{if } 0 \leq x < \frac{\alpha-1}{\alpha} \\ m\left(\alpha(1-x)\right)^{\frac{1}{1-\alpha}} & \text{if } x \geq \frac{\alpha-1}{\alpha} \end{cases}$$

Figure 6 shows the analytically computed ramp-up times for different pickiness degrees (x-axis). The y-axis shows the factor of the mean task duration that a user has to wait. The figure also includes simulation results that confirm that the derivations are correct. We see that for larger values of $\alpha$, the ramp-up time is much less than $2x$ the mean task duration.

We analyzed tasks from two large production clusters at Bing and Facebook and found that the task duration is roughly (with $R^2$ fit around $0.8$) Pareto distributed with $\alpha$ greater than $1.9$ (see Figure 7). For such high values, the ramp-up time (see Figure 6) is less than twice the mean task duration for pickiness values as high as $0.85$. Note that the ramp-up time is the time until the user gets all of her fair-share. She can, however, start running tasks long before the ramp-up time has been reached.

**Summary.** Ramp-up time with Choosy will be quick for any $\alpha > 1.9$ and pickiness below $0.8$. The reason is that with Pareto distributions (and exponential), the vast majority of tasks are small and finish quickly. Only highly picky users ($> 0.9$) will have to wait for the very few long tasks that need to finish for them to ramp-up to their fair-share.

## 7. Evaluation

We evaluated Choosy through simulations based on existing workloads and an implementation using the Mesos resource manager. We start with simulations of a large cluster based on workload traces from Facebook and Google (Section 7.1). We then use a smaller set of microsimulations to compare Choosy to offline schedulers in various scenarios (Section 7.2). Next, we show results for our implementation in Mesos (Section 7.3). Finally, we report the performance of our offline CMMF solver (Section 7.4).

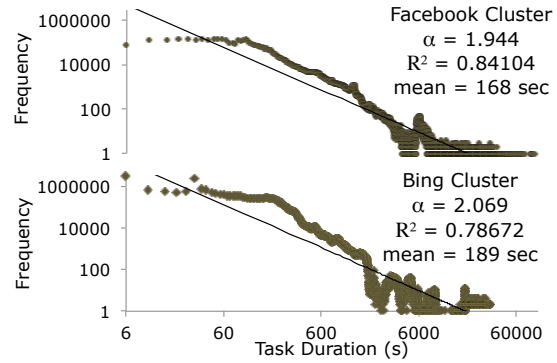


Figure 7: Tasks in Facebook and Bing production clusters are Pareto distributed with $\alpha > 1.9$ and fit $\approx 0.8$.

### 7.1 Macrobenchmark

We simulated a 1000-node cluster, running workloads based on traces from Facebook and Google, to evaluate Choosy for large-scale datacenter workloads. Specifically, we used the distribution of MapReduce job and task sizes at Facebook [34] to generate a schedule for a series of MapReduce jobs. We ran 100 different simulations, each covering one hour of simulated time with 300 submitted jobs.

We assigned constraints to jobs using the model proposed by Sharma *et al.* for constraints at Google [28]. In this model there are 21 different constraints, several machine types, and a frequency of each machine type meeting each constraint. Frequencies range from 100% for one constraint to 7-8%. Jobs require each constraint with varying probabilities, with each job having 1.8 constraints on average. In our experiments, the average job could use 38% of the machines; 40% of the jobs could use less than 20% of the machines; and 30% of them could use less than 10% of the machines. This constitutes a nontrivial workload, with a significant fraction of highly constrained jobs.

We compared Choosy against an optimal offline scheduler that is not permitted to migrate or preempt tasks (Restricted Offline), as well as two offline schedulers that are granted *more* freedom than typical systems to give an upper bound on how fair a system can be. One of these (Migration) could migrate tasks at no cost, and another (Preemption) could also preempt them at no cost (the preempted task is simply paused, and it resumes from where it left off whenever it is rescheduled). We note that these latter two schedulers are quite unrealistic: no current cluster programming framework supports pausing or migrating tasks, and these actions will certainly come at a cost. Some systems do support killing tasks, but this causes computation to be lost. However, we compare against these schedulers to show how close Choosy gets even to these ideal CMMF allocations.

As a side note, the absolute majority of jobs in the workload are small. Thus, the majority of jobs can fit all their tasks within their fair share, implying that their job response time is similar to other scheduling principles that run whole jobs, such as shortest job first.

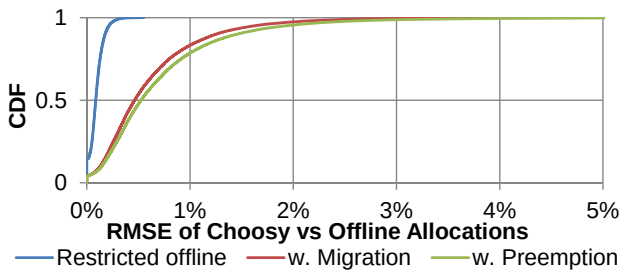


Figure 8: CDF of the Root Mean Square Error (RMSE) between Choosy's allocation vectors and those of the offline schedulers.

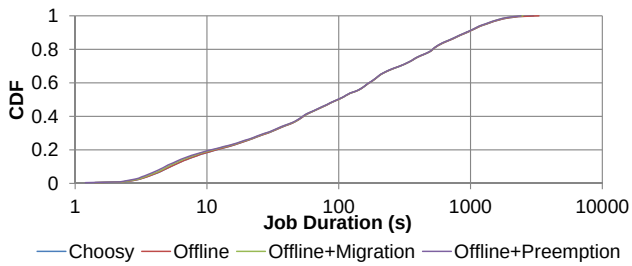


Figure 9: CDFs of job durations in Choosy vs. offline schedulers. The CDFs match closely, with Choosy and Restricted Offline being farthest right (having slightly longer jobs) and with the scheduler that can preempt tasks being farthest left.

| Job size | Slowdown versus: | | |
|---|---|---|---|
| | Restricted offline | Migration | Preemption |
| <30s | 1.02 (0.25) | 1.07 (0.38) | 1.13 (0.58) |
| 30–120s | 1.00 (0.06) | 1.01 (0.06) | 1.02 (0.07) |
| 120–600s | 1.00 (0.03) | 1.01 (0.04) | 1.01 (0.04) |
| >600s | 1.00 (0.01) | 1.01 (0.04) | 1.01 (0.04) |

Table 1: Slowdown of jobs in Choosy relative to the offline schedulers, binned by job duration in the offline scheduler (*e.g.,* a job that takes 29s with an offline scheduler but 31s with Choosy is in the first bin). Standard deviations in parentheses.

We evaluate Choosy on two metrics: similarity of its allocation vectors to those of the offline schedulers, and job response times. Ideally, an online scheduler would provide the same allocation vectors and the same response time for each job as an offline scheduler.

To measure similarity of the allocation vectors, we plot a CDF of Root Mean Square Error (RMSE) between Choosy's allocation vectors and the ones of each offline scheduler in Figure 8. That is, if Choosy gives a sorted allocation vector $\langle c_1, c_2, \ldots, c_n \rangle$ at a particular time $t$, and an offline scheduler gives $\langle o_1, o_2, \ldots, o_n \rangle$ for the same workload, the RMSE is $\sqrt{\frac{1}{n}((c_1 - o_1)^2 + \cdots + (c_n - o_n)^2)}$. We see that the average RMSE is 0.09% for the restricted offline scheduler (that cannot migrate or preempt tasks), and stays below 0.71% even if we allow preemption and migration.

The discrepancy in job response times between Choosy and the offline schedulers is even lower. Figure 9 plots CDFs of the job response times in each system, showing that their distributions are nearly identical. However, looking at the overall distribution might hide discrepancies in the response times of particular jobs (*e.g.,* if some jobs are slowed down at the expense of others in Choosy). To measure the differences in response times of individual jobs, we also computed the *slowdown* of each job against each offline scheduler, defined as the ratio of the job's running time on Choosy vs. the offline scheduler. We report the mean slowdowns for jobs of different sizes in Table 1. In general, Choosy's job response times are almost identical to those of the restricted offline scheduler across all size ranges. When comparing with schedulers that can migrate or pause tasks, the main differences are in the smallest bin (jobs that take less than 30s). This occurs because jobs submitted at a time when all the machines they can use are full will take longer to get scheduled without migration than if migration is allowed, and a fixed increase in response time has a greater effect on the slowdown metric for a small job. Longer jobs have nearly identical response times even with migration.

| Scheduler | Avg. Execution Time (s) |
|---|---|
| Choosy | 1.5 |
| Restricted offline | 228.5 |
| Offline w. migration | 266.8 |
| Offline w. preemption | 590.0 |

Table 2: Running time of each simulation with each scheduler, showing the computational costs of the schedulers. Choosy is two orders of magnitude faster than the offline schedulers.

Finally, we compare the computational cost of each algorithm. Table 2 lists average running times of a simulation with each scheduler. The offline schedulers are more than two orders of magnitude slower than Choosy, because they solve a flow problem to make each scheduling decision. In fact, with a larger cluster than the 1000-node one we used, the running times of the offline schedulers would exceed the time period we simulate (about 55 minutes), meaning that they would be too slow to schedule tasks in a real cluster.

### 7.2 Microbenchmarks

For the microbenchmarks, we simulate the behavior of Choosy with respect to different offline schedulers in various simple scenarios, such as a user joining or leaving. We use a synthetic workload with random constraints to clearly highlight Choosy's behavior.

We explore how the system behaves when users join and leave while tasks are continuously starting and finishing. When a user joins, her tasks are immediately ready to be launched. For the case of leaves, there are two scenarios to consider. First, a user's tasks may finish one by one and once the user has no more tasks to run, she is done and can leave. The second case is if a user suddenly leaves and relinquishes all her resources at once. While the former scenario is more common in Hadoop and Dryad, it is also the most optimistic for Choosy, as there is only one task to assign at a time and Choosy will assign it the same way as an offline scheduler. We therefore do not include those results,

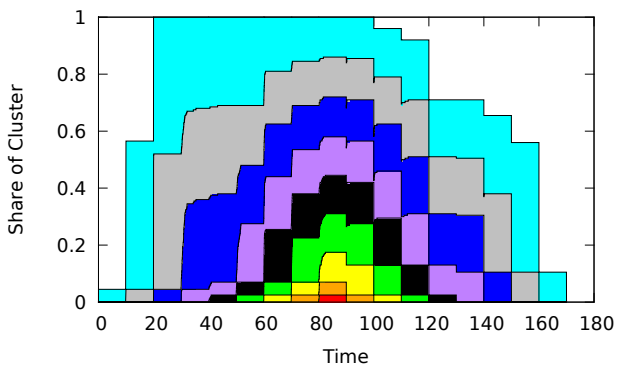


Figure 10: Choosy vs. restricted offline (no preemption/migration) for 9 users joining and leaving. Dark lines show offline allocations and colored filled curves give the Choosy allocation.

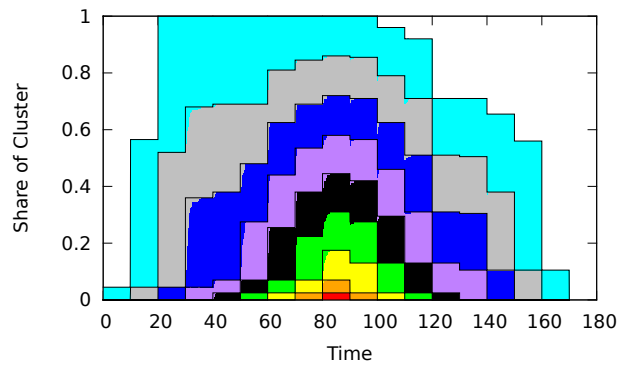


Figure 11: Choosy vs. an ideal offline scheduler that can perform preemption and migration for free. (Color legend in Figure 10.)

as they look similar to the case of joining users. Instead, we test the more pessimistic scenario for Choosy, where many resources free up at once and an offline scheduler can make a better decision than Choosy's greedy approach. This scenario occurs if, for instance, a job is canceled.

Our first three simulations show the same workload, in which nine users join the cluster over time and then leave, and tasks have exponential durations with mean 1.

**Choosy vs Restricted Offline** Figure 10 compares Choosy with the offline optimal allocator when it is restricted to not move or preempt tasks. Much like in our macrobenchmark, Choosy closely matches the offline scheduler, as the offline scheduler has little choice but to allocate any resource to the user with the lowest share.

**Choosy vs Offline with Migration+Preemption** Figure 11 compares Choosy with an offline scheduler for an ideal system that can pause or move tasks for free. As we see from the dark lines, the optimal allocation has a step-like shape as it immediately adjusts when a new user joins. We see that for joins, there is a slight discrepancy between allocations, because the offline scheduler can immediately preempt tasks. For leaves, the two schedules are almost identical. We also compared a scheduler with just migration, but its results were similar to this one.

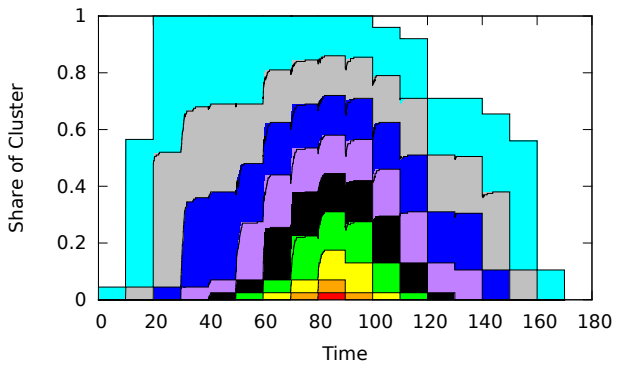


Figure 12: Choosy vs. periodic offline scheduling. (Color legend in Figure 10.)

**Choosy vs Periodic Offline Scheduling** Finally, Figure 12 compares Choosy with a hybrid scheduler, which uses the optimal offline schedule to make online decisions. This scheduler solves for a new offline allocation when a user joins or leaves, and then attempts to converge to that allocation by reassigning machines as tasks finish so that each user receives the correct number of machines of each type.

While this periodic offline policy might seem good at first glance, somewhat surprisingly, we see that Choosy performs better than the periodic offline scheduler. The reason is that the offline scheduler decides on *one* particular allocation, whereas there might be many different allocations that satisfy the joining users constraints. Thus, even though a machine that frees up can go to a joining user, the hybrid scheduler might decide to give it back to the user that just finished using it, because it has another machine in mind for the joining user. Depending on the order that machines finish in, the hybrid scheduler can thus be far from optimal.

**All Users Joining an Empty Cluster** Finally, we test what happens if all users join an empty cluster. As seen from Figure 13, the optimal scheduler immediately reaches the optimal CMMF allocation. This is true for all variations of the optimal scheduler, *i.e.,* with or without preemption/migration. In contrast, Choosy assigns machines greedily, and can take some time to converge. We see, however, that even when task durations are on average 1 time unit, most users are very close to their fair share within 1 time unit, so Choosy quickly converges to the right allocation.

### 7.3 Mesos Implementation

To evaluate Choosy with real jobs, we implemented it in the Mesos resource manager [14] as a new pluggable allocation module. The allocation module then ensures that whenever resources are free, Mesos *offers* them to the application with the lowest share of the cluster that can use them.

We set up a cluster on Amazon EC2 with a total of 160 cores, divided equally among 4 types of machines:

- *Standard* nodes with 8 cores and 7 GB RAM.
- *High-memory* nodes with 8 cores and 68 GB RAM.

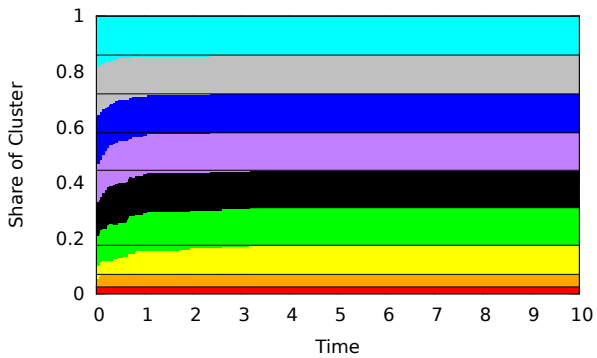


Figure 13: Choosy vs. all offline schedulers. Nine users with random constraints join an empty cluster. (Color legend in Figure 10)

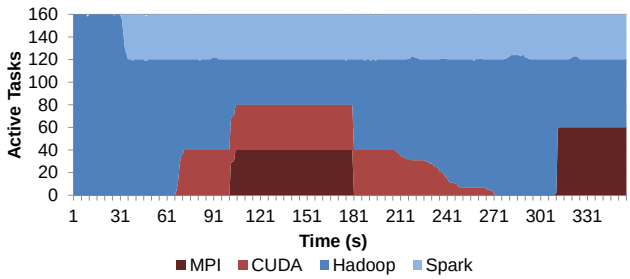


Figure 14: Resource share of each job over time in our experiment running Choosy on Mesos.

- *Cluster* nodes with 8 cores, 23 GB RAM, and a fast 10 Gbps Ethernet network.
- *Cluster GPU* nodes with 8 cores, 23 GB RAM, 10 Gbps Ethernet, and GPUs.

We then ran four jobs with different constraints:

- *Hadoop*: can use any kind of node.
- *Spark:* an in-memory computation framework that can only run on the high-memory nodes [35].
- *CUDA:* a GPU-based Black-Scholes solver.
- *MPI*: a communication-intensive Linpack benchmark that needed 10 Gbps Ethernet.

Figure 14 shows the shares of each application over time with Choosy. We see that initially, only Hadoop is active, and has a full share of the cluster. After about 30 seconds, a Spark job is submitted and ramps up to its CMMF share ($\frac{1}{4}$ of the machines) within 5 seconds. At time 65, the CUDA job is submitted and also ramps up within 5s as well. At time 100, the MPI job is submitted and ramps up within 4s. Both MPI and CUDA eventually finish, giving their resources back to Hadoop. Finally, at time 305, we submit a second MPI job that is now allowed to take $\frac{3}{8}$ of the machines instead of $\frac{1}{4}$ because CUDA is no longer active; thus, at this point, the resource allocations are $\frac{1}{4}$ for Spark, $\frac{3}{8}$ for Hadoop, and $\frac{3}{8}$ for MPI, which is again the CMMF allocation.

In summary, the experiment shows that Choosy converges to CMMF allocations quickly in practice in environments running fine-grained tasks.

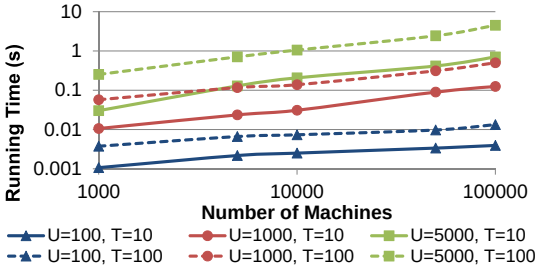


Figure 15: Performance of the offline CMMF solver versus the numbers of users ($U$), distinct machine types ($T$), and total machines in the cluster. Standard deviations were less than 10%.

### 7.4 Performance of Offline Solver

We implemented our flow-based offline CMMF solver (Section 5.1) in C++ using the push-relabel network flow package in the Boost Graph Library [5]. We made several optimizations to the algorithm to identify frozen users faster and search for the next level to increase the flows to efficiently.

We evaluated its performance using scheduling problems with various numbers of users, machines, and distinct machine types based on the distribution of constraints reported at Google [28]. We plot the performance of the solver in Figure 15, averaging results for ten problem instances of each type. We see that the algorithm runs in less than 5 seconds even with 5000 users, 100,000 machines and 100 distinct machine types. The running time is roughly linear with the number of users and machine types, but grows slower with the number of actual machines (only growing by a factor of 4–18$\times$ when we increase the number of machines by 100$\times$). While these running times are too high for scheduling fine-grained tasks, they indicate that offline CMMF is viable for environments with coarser-grained scheduling.

## 8. Related Work

**Datacenter Schedulers** Quincy [16] and the Hadoop Fair Scheduler [34] are fair schedulers for datacenters that take into account data locality. However, these schedulers treat locality as a *preference* rather than a hard constraint, and can assign tasks non-locally if a suitable machine is not available. In contrast, our work tackles the problems of defining a fair allocation in the presence of *hard* constraints, and of efficiently *computing* it online. Hard constraints are more difficult to handle than soft because they require careful machines selection. For instance, Choosy may need to iteratively swap machines among multiple users to improve the allocation (see, for example, Figure 5 in §5).

Of these schedulers, Quincy is perhaps closest to our work, as it defines a cost model using an optimization problem. However, Quincy does not handle hard constraints—it only seeks to equalize the *number* of machines each user gets, and to make as many tasks local as possible. For instance, in a cluster of 20 machines, where user 1 and user 2 both prefer the same subset of 10 of the nodes, Quincy is not guaranteed to split these ten equally; it may give 10 pre-

ferred nodes to one user and 10 non-preferred to the other. In addition, Quincy is an offline scheduler that takes several seconds to make a decision for 2500 nodes [16], while we provide a fast online scheduler for fine-grained tasks.[9]

Another recent scheduler that takes into account constraints is alsched [30]. alsched asks users for utility functions capturing constraints, and maximizes the sum of these utilities using an offline solver. Unfortunately, maximizing the total utility may provide neither sharing incentives nor strategy-proofness. First, the allocation that maximizes total utility might well be one that gives some users zero resources, which goes against the sharing incentive property. Second, users that provide their own utility functions may overstate how important some resources are to them to get a higher share (*e.g.,* claim that their utility is 0 unless they get at least 10 nodes). CMMF provably avoids these problems.

**Max-Min Fairness** Max-min fairness has been widely studied in networks, operating systems, and queuing systems [3, 4, 26, 31–33]. However, previous work assumes that resources are identical (*e.g.,* they are units of bandwidth on a link or cycles on a CPU), and does not take into account placement constraints. Our work proposes a generalization of max-min fairness that retains the attractive properties of single-resource fairness.

Dominant Resource Fairness (DRF) [12] extends max-min fairness to multiple resource *types*, such as CPU and memory, where tasks might have different requirements on different resource dimensions. However, it still assumes that the resources of a given type (*e.g.,* CPUs) are identical. While DRF identifies several important allocation properties, such as sharing incentive, and studies allocation from a microeconomic perspective, its setting and solution turn out to be quite different than in our constrained sharing problem: DRF selects an allocation based on the Kalai-Smorodinsky solution, and in its multi-resource setting, market-based (Nash bargaining) allocation is not strategy-proof, while CMMF selects an allocation similar to Nash bargaining that *does* turn out to be strategy-proof (§4.3).

**Scheduling Theory** In the theory literature, Kleinberg *et al.* [19, 20] solved and approximated the *single-source unsplittable flow problem*, which is to route flows in a network in a weighted fair manner assuming that each flow must follow a single path. They noted that this problem is closely related to a job scheduling problem, in which a set of *indivisible* jobs are given that have constraints on the machines they can use. This work differs from ours in that we are concerned with parallel jobs that are composed of *multiple* tasks and can thus be assigned multiple machines, as opposed to the unsplittable jobs in Kleinberg *et al.*'s work.

[9] Quincy does not provide max-min fairness. Max-min fair algorithms typically require solving multiple optimization problems as each set of users is saturated, as in our algorithm in §5, but Quincy only solves one optimization problem, which is to maximize locality while meeting certain limits on user shares.

The above work is part of a long effort by the computer science theory community to model, analyze, and construct algorithms for job scheduling [18]. Many different variations of the problem has been studied. The *identical*, *uniformly related*, and *unrelated* machine models capture the *efficiency* of each job on each machine, but do not consider hard placement constraints. The *open*, *job shop*, and *flow* machine models capture the fact that a job might consist of smaller *operations* (*e.g.,* steps in assembling a product). This is closer to the parallel job model we are concerned with [8, 14, 15], but most of this work assumes that only one operation from a job can execute at a time (*i.e.,* jobs are not parallel), and that it runs on one machine [18, 21].

**HPC and Grids** Job scheduling has also been studied extensively in the HPC and grid computing fields. However, in these environments, jobs are typically coarse-grained, using a fixed set of machines for a long period of time, and an upper bound on each job's duration is given [9]. This knowledge of job durations allows schedulers to plan ahead using offline algorithms and use backfilling [22] to let smaller jobs ahead of the queue if they do not affect longer ones. In contrast, data-intensive workloads in current datacenters consist of thousands of *fine-grained* tasks with unknown durations [14], necessitating online scheduling. Our contribution is an online algorithm for constraint-based scheduling in this setting.

**Choosy vs. Other Scheduling Problems** Many scheduling related problems have been considered in the literature. It is, therefore, natural to ask how Choosy can be modified or combined with other schedulers to solve other problems than those of hard constraints. First, we note that soft constraints, *i.e.,* preferences, can be satisfied using orthogonal mechanisms together with Choosy, such as Delay Scheduling [34]. Second, Choosy can be modified to handle multi-resources, by modifying the multi-resource scheduler, DRF [12], to schedule the user with the smallest dominant share that can run on a particular machine. Finally, Choosy should work well with existing hierarchical schedulers, as an important property of a hierarchical scheduler is to support composition of existing schedulers in its hierarchy.

## 9. Conclusion

This paper shows how to extend max-min fairness to handle hard task placement constraints, which are becoming a commonplace in datacenters. We defined Constrained Max-Min Fairness (CMMF), a generalization of max-min fairness that has several desirable properties: First, CMMF is the only policy providing a sharing incentive property, which is important to motivate users to pool resources. Second, CMMF is strategy-proof, incentivizing users to truthfully report their needs. The main challenge with CMMF is that it is not efficiently implementable in today's online schedulers. We therefore presented and implemented a simple greedy online scheduler, called Choosy, that closely approximates CMMF.

Our evaluation shows that for constraints generated from a recent Google workload, Choosy's performance differs by less than 2% from an optimal scheduler.

We have focused on evaluating Choosy's convergence in a practical setting with current workloads. In the future, theoretical aspects on Choosy should be investigated, such as better bounds on its convergence time. Furthermore, defining a generalization of max-min fairness that takes both placement constraints and multiple resources into account is both challenging and interesting.

## References


[1] Hadoop Capacity Scheduler. `http://hadoop.apache.org/common/docs/r0.20.2/capacity_scheduler.html`.

[2] The Next Generation of Apache Hadoop MapReduce. `http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen`.

[3] B. Avi-Itzhak and H. Levy. On measuring fairness in queues. *Advanced in Applied Probability*, 36:919–936, 2004.

[4] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, second edition, 1992.

[5] The Boost Graph Library. `www.boost.org/doc/libs/release/libs/graph`.

[6] R. B. Cooper, S.-C. Niu, and M. M. Srinivasan. Some reflections on the Renewal-theory paradox in queueing theory. *Journal of Applied Mathematics and Stochastic Analysis*, 11:355–368, 1998.

[7] H. A. David and H. N. Nagaraja. *Order Statistics, Third Edition*. Wiley, New York, 2003.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997.

[10] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX NSDI*, 2012.

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

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

[13] T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, and D. Zufferey. Static scheduling in clouds. In *HotCloud*, June 2011.

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

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07*, 2007.

[16] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP '09*, 2009.

[17] E. Kalai and M. Smorodinsky. Other Solutions to Nash's Bargaining Problem. *Econometrica*, 43(3):513–518, 1975.

[18] D. Karger, C. Stein, and J. Wein. Scheduling algorithms, 1997.

[19] J. Kleinberg, Y. Rabani, and va Tardos. Fairness in routing and load balancing. In *J. Comput. Syst. Sci*, pages 568–578, 1999.

[20] J. M. Kleinberg. Single-Source Unsplittable Flow. In *FOCS*, 1996.

[21] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46:259–271, February 1990.

[22] D. A. Lifka. The ANL/IBM SP scheduling system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 1995.

[23] H. Moulin. *Fair Division and Collective Welfare*. MIT Press, 2004.

[24] J. Nash. The Bargaining Problem. *Econometrica*, 18(2):155–162, April 1950.

[25] R. Raman, M. Solomon, M. Livny, and A. Roy. The ClassAds language. pages 255–270, 2004.

[26] D. Raz, H. Levy, and B. Avi-Itzhak. A resource-allocation queueing fairness measure. *SIGMETRICS Perform. Eval. Rev.*, 32:130–141, June 2004.

[27] W. Shafer and H. F. Sonnenschein. Market demand and excess demand functions. In K. J. Arrow and M. Intriligator, editors, *Handbook of Mathematical Economics*, volume 2, chapter 14, pages 671–693. Elsevier, 4 edition, 1993.

[28] 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.

[29] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *NSDI*, pages 23–23, 2011.

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

[31] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional Share Resource Management*. PhD thesis, MIT, Laboratory of Computer Science, Sept. 1995. MIT/LCS/TR-667.

[32] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI '94*, 1994.

[33] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. In *SIGMETRICS'03*, 2003.

[34] 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 *EuroSys '10*, 2010.

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

## 10. Appendix

Here we provide the proofs of earlier stated theorems.

THEOREM 3. *CMMF is strategy-proof.*

**Proof** Let $C_i$ be the set of machines that user $i$ can actually use. Let assume user $i$ lies about her constraints by claiming that she can use a set of machines $C'_i$, instead. Let $A_i$, and $A'_i$ be the allocations of user $i$ under constraints $C_i$, and $C'_i$, respectively. Let $B_i = A'_i \cap C_i$, and $B'_i = A'_i \setminus B_i$. In other words, $B_i$ represents the number of allocated machines that user $i$ can use, and $B'_i$ the number of allocated machines she cannot use, when user $i$ lies about her constraints. Assume $B'_i = \emptyset$, and assume user $i$ benefits from lying, *i.e.,* $|B_i| > |A_i|$. If this were the case, allocation $A'_i = B_i$ would be also feasible under the original constraint set $C_i$, which means that user $i$ would get no benefit by lying. If $B'_i \neq \emptyset$, user $i$ may hurt other users, but won't get any benefit since she cannot use these machines. This proves our claim. □

THEOREM 4. *Consider a cluster with $n$ users, each of which have infinite demand and finite task lengths. Then no matter what allocation it starts with, Choosy converges to an allocation where each user's share is at most $2n$ machines less than its share in some optimal allocation.*

**Proof** First, note that each time a task finishes and its machine is reassigned, Choosy can only *improve* the leximin vector. In particular, the machine freed can only go to the user that owned it before, in which case the leximin vector stays the same, or to a user with a lower share, in which case the leximin vector improves.

Then, the only question is whether Choosy can keep improving the vector, or whether it gets "stuck" at an allocation far from the optimal. Fortunately, it is possible to show that for $n$ users, if Choosy reaches an allocation $A$ where the number of machines owned by some user is less than the user's share some optimal allocation $O$ by at least $2n$, then there is a task $t$ that can finish that will improve the leximin vector when its machine is reassigned. Because we assume finite task lengths, $t$ will eventually finish, allowing us to move to a better allocation.

Suppose that there is an allocation $A$ in which some user, $u_1$, has at least $2n$ fewer machines than in some optimal allocation, $O$. For each user $u$, let $S_A(u)$ denote its number of machines in $A$ and $S_O(u)$ denote its number of machines in $O$. Then there are three cases:

**Case 1:** There is an unallocated machine $m$ that can be used by at least one user. Then, Choosy will improve the allocation by simply giving $m$ to a user who can use it.

**Case 2:** There is a machine $m$ that at least two users $u$ and $u'$ can use, and $u$ currently owns $m$ but $S_A(u') < S_A(u) - 1$. We call such a machine *reassignable*. Then, when the task on $m$ finishes, Choosy will reassign $m$ it to $u'$ or to some other user with fewer machines than $u'$. This will result in the recipient's allocation increasing from a value less than $S_A(u) - 1$ to at most $S_A(u) - 1$, and in $u$'s allocation falling to $S_A(u) - 1$, so the lexicographic allocation vector will strictly improve.

**Case 3:** All the machines usable by at least one user are allocated, but there are no reassignable machines. We will show that this is impossible by contradiction. Suppose that it were true, and consider the user, $u_1$, who has at least $2n$ fewer machines in $A$ than in $O$ (that is, $S_A(u_1) \leq S_O(u_1) - 2n$). Then there is at least one machine, $m$, that is owned by $u_1$ in $O$ but by some other user, $u_2$, in $A$. Because $m$ is not reassignable in $A$, we have $S_A(u_2) \leq S_A(u_1)+1$. Furthermore, because $O$ is an optimal allocation, we also know that $m$ is not reassignable in $O$, so $S_O(u_1) \leq S_O(u_2) + 1$. Putting these inequalities together, we get:

$$\begin{aligned} S_A(u_2) &\leq S_A(u_1) + 1 \leq S_O(u_1) - 2n + 1 \\ &\leq S_O(u_2) - 2n + 2 \leq S_O(u_2) - 2(n-1) \end{aligned}$$

Now consider the user set of users $U = \{u_1, u_2\}$. We know that the users in $U$ have more machines in $O$ than in $A$, so there must be at least one machine $m$ owned by a user $u_3 \notin U$ inside $A$ and by a user $u_i \in U$ in $O$. We also know that $S_A(u_i) \leq S_O(u_i) - 2(n-1)$, because $u_i$ is either $u_1$ or $u_2$. Thus, by a set of inequalities similar to the ones above, we can obtain that $S_A(u_3) \leq S_O(u_3) - 2(n-2)$. Proceeding this way, we can define a series of users $u_1, u_2, u_3, u_4, \ldots$ and a series of sets $U_i = u_1, \ldots, u_i$ such that the users in set $U_i$ all have at least $2(n - i + 1)$ fewer machines in $A$ than they do in $O$. But then, because $U_n$ contains all the users in the cluster, this means that *every* user has at least 2 fewer machines in $A$ than in $O$. This contradicts our assumption that all the machines usable by at least one user are allocated, and concludes the proof by contradiction. □

THEOREM 5. *Independent allocation is strategy-proof.*

**Proof** This follows from the fact that independent allocation is given by the Shapley Value [23], which known to be additive. That is, the allocation of each machine is independent of how the other machines are assigned. □