# Karma: Resource Allocation for Dynamic Demands

Midhul Vuppalapati
Cornell University

Giannis Fikioris
Cornell University

Rachit Agarwal
Cornell University

Asaf Cidon
Columbia University

Anurag Khandelwal
Yale University

Éva Tardos
Cornell University

## Abstract

We consider the problem of fair resource allocation in a system where user demands are dynamic, that is, where user demands vary over time. Our key observation is that the classical max-min fairness algorithm for resource allocation provides many desirable properties (*e.g.*, Pareto efficiency, strategy-proofness, and fairness), but only under the strong assumption of user demands being static over time. For the realistic case of dynamic user demands, the max-min fairness algorithm loses one or more of these properties.

We present Karma, a new resource allocation mechanism for dynamic user demands. The key technical contribution in Karma is a credit-based resource allocation algorithm: in each quantum, users donate their unused resources and are assigned credits when other users borrow these resources; Karma carefully orchestrates the exchange of credits across users (based on their instantaneous demands, donated resources and borrowed resources), and performs prioritized resource allocation based on users' credits. We theoretically establish Karma guarantees related to Pareto efficiency, strategy-proofness, and fairness for dynamic user demands. Empirical evaluations over production workloads show that these properties translate well into practice: Karma is able to reduce disparity in performance across users to a bare minimum while maintaining Pareto-optimal system-wide performance.

## 1 Introduction

Resource allocation is a fundamental problem in computer systems, spanning private and public clouds, computer networks, hypervisors, etc. There is a large and active body of research on designing resource allocation mechanisms that achieve Pareto efficiency (high resource utilization) and strategy-proofness (selfish users should not be able to benefit by lying about their demands) while ensuring that resources are allocated fairly among users, *e.g.*, [30,32,39,57,59,66,67].

For a system containing a single resource, the two most popular allocation mechanisms are strict partitioning [9,72] and max-min fairness [30, 32, 36, 40, 49, 50, 57, 59, 66]. The former allocates the resource equally across all users ("fair share"), independent of their demands; this guarantees strategy-proofness and fairness, but not Pareto efficiency since resources can be underutilized when one or more users have demands lower than the fair share. Max-min fairness alleviates limitations of strict partitioning by taking user demands into account: it maximizes the minimum allocation across users while ensuring that each user's allocation is no more than their demand. A classical result shows that resource allocation based on max-min fairness guarantees each of the three desirable properties—Pareto efficiency, strategy-proofness, and fairness. These powerful properties have, over decades, motivated efforts in both systems and theory communities on generalizations of max-min fairness for allocating multiple resources [30–32], for incorporating application performance goals and deadlines [31,39,46,47], and for new models of resource allocation [17,22,25,33,59,66], to name a few.

This paper explores a complementary problem—resource allocation of a single elastic resource in a system where user demands are dynamic, that is, vary over time. Dynamic user demands are the norm in most real-world deployments [12,16, 41,45,60,63,70,72,79]; for instance, analysis of production workloads in §2 reveals that user demands vary by as much as $17\times$ within minutes, with majority of users having demands with standard deviation $0.5-43\times$ of the average over time. We show in §2 that, for systems with such dynamic user demands, resource allocation based on the max-min fairness algorithm fails to guarantee one or more of its properties: (1) if the allocation is done based on demands at $t=0$, Pareto efficiency and strategy-proofness are no longer guaranteed; and, (2) if the allocation is done periodically, *long-term* fairness is no longer guaranteed—for $n$ users with the same average demand, the max-min fairness algorithm may allocate some user as much as $\Omega(n)$ more resources than other users over time.

We present Karma, a new resource allocation mechanism for dynamic user demands. The key technical contribution of Karma is a credit-based resource allocation algorithm: in each quantum, users receive credits when they donate a part of their fair share of resources (*e.g.*, if their demand is less than their fair share); users can use these credits to borrow resources in any future quantum when their demand is higher than their fair share. When the supply of resources from donors is equal to the demand from borrowers, it is easy to exchange resources and credits among users. The key algorithmic challenge that Karma resolves is when supply is not equal to demand—in such scenarios, Karma carefully orchestrates resources and credits between donors and borrowers: donors are prioritized so as to keep credits across users as balanced as possible, and borrowers are prioritized so as to keep the resource allocation as fair as possible.

We theoretically establish Karma guarantees for dynamic user demands. Karma guarantees Pareto efficiency at all times: in each quantum, it allocates resources such that it is not possible to increase the allocation of a user without decreasing the allocation of at least another user. For strategy-proofness, Karma guarantees that a selfish user cannot increase their aggregate resource allocation by *over*-reporting their demands in any quantum. In addition, we show a new surprising phenomenon (that may primarily be of theoretical interest): if a user had perfect knowledge about the future demands of all other users, the user can increase its own aggregate allocation by a small constant factor by *under*-reporting its demand in some quanta; however, for $n$ users, imprecision in this future knowledge could lead to the user losing $\Omega(n)$ factor of their aggregate resource allocation by under-reporting their demand in any quantum. Put together, these results enable Karma to provide powerful guarantees related to strategy-proofness. Finally, for fairness, we prove that given a set of (past) allocations, Karma guarantees an optimally-fair resource allocation. We also establish that Karma guarantees similar properties even when multiple selfish users can collude, and even when different users have different fair shares.

We have realized Karma on top of Jiffy [41], an open-sourced multi-tenant elastic memory system; an end-to-end implementation of Karma is available at `https://github.com/resource-disaggregation/karma`. Evaluation of Karma over production workloads demonstrates that Karma's theoretical guarantees translate well into practice: it matches the max-min fairness algorithm in terms of resource utilization, while significantly improving the long-term fairness of resources allocated across users. Karma's fairer resource allocation directly translates to application-level performance; for instance, over evaluated workloads, Karma keeps the *average* performance (across users) the same as the max-min fairness algorithm, while reducing performance *disparity* across users by as much as $\sim 2.4\times$. Karma also incentivizes users to share resources: our evaluation shows that (1) Karma-conformant users achieve much more desirable allocation and performance compared to users who prefer a dedicated fair share of resources; and, (2) if users were to turn Karma-conformant, they can improve their performance by better matching their allocations with their demands over time.

## 2 Motivation

We begin by outlining our motivating use cases, followed by an in-depth discussion on the limitations of the classic max-min fairness algorithm for dynamic user demands.

**Motivating use cases.** Fair resource allocation is an important problem in private clouds where resources are shared by multiple users or teams within the same organization [12, 16, 17, 30–33, 36, 39, 40, 45, 46, 59, 60, 66, 70, 72, 79, 80]; our primary use cases are from such private clouds. Karma may also be useful for emerging use cases from multi-tenant public clouds where spare resources may be allocated to tenants while providing performance isolation [8, 14, 38, 41, 57, 63, 64, 66]. We discuss motivating scenarios in both contexts below.

One scenario is shared analytics clusters. For instance, companies like Microsoft, Google, and Alibaba employ schedulers [32, 35, 39, 69, 70, 80] that allocate resources across multiple internal teams that run long-running jobs (*e.g.*, for data analytics [23, 81]) on a shared set of resources. Consider memory as a shared resource; in many of these frameworks, main memory is used to cache frequently accessed data from slower persistent storage and to store intermediate data generated during job execution. Indeed, increasing the allocated memory improves job performance; however, since memory is limited and is shared across multiple teams, ensuring resource allocation fairness is also a key requirement. Moreover, since these jobs are usually long-running, their performance depends on long-term memory allocations, rather than instantaneous allocations [16, 32, 45].

Another use case is shared caches: many companies (*e.g.* Facebook [9, 12, 52] and Twitter [79]) operate clusters of in-memory key-value caches, such as memcached or Redis, serving a wide array of internal applications. In this use case, the memory demand of each application may be computed as the amount of memory that would be required to fit hot objects within the cache [18, 19, 52, 79]. In such settings, efficient and fair sharing of caches is of utmost importance [9, 19, 52, 72]: to maintain service level agreements, it is important to have consistently good performance over long periods of time, rather than excellent performance at some times and very poor performance at other times (see [9, 19, 52, 72] for more discussion on the importance of long-term performance).

Third, fair resource allocation while ensuring high utilization is also a goal in inter-datacenter bandwidth allocation [36, 40, 49]. Existing traffic engineering solutions used in production environments perform periodic max-min fair resource allocation to account for dynamic user demands [36, 40, 49]. Our work demonstrates that periodically performing max-min fair resource allocation over such dynamic demands leads to unfair resource allocation across users.

Finally, an interesting use case in the public cloud context is that of burstable VMs [2, 4] that use virtual currency to enable resource allocation over dynamic user demands. These VMs share resources with VMs from other users and are charged

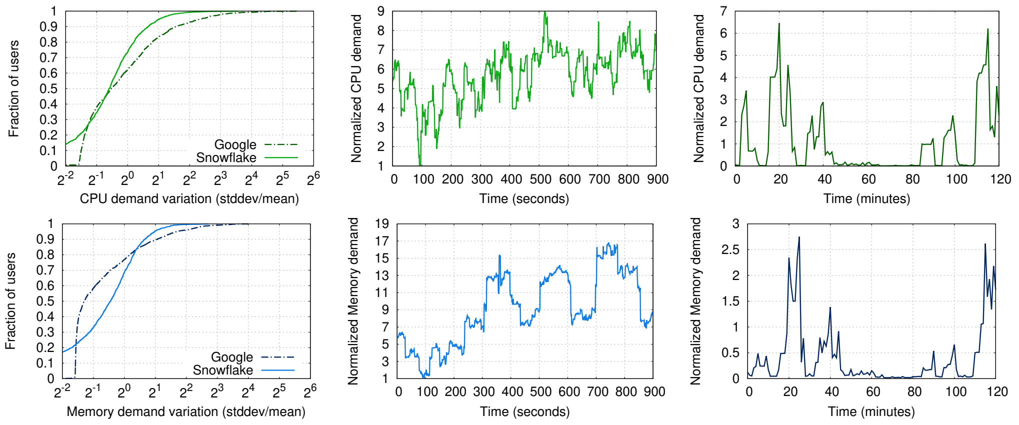


Figure 1: **Analysis of Google and Snowflake workloads suggests that a large fraction of users have dynamic demands (left) that can change dramatically over short timescales (center, right)** (Left) CDFs, across users, of the ratio of standard deviation and mean of each user's demand. (Center) For a randomly sampled user in the Snowflake trace, the variation in the user's CPU and memory demands (normalized by minimum demand) over a 15 minute period. (Right) For a randomly sampled user in the Google trace, the variation in the user's CPU and memory demands (normalized by minimum demand) over a 2 hour period.

on an instance-specific baseline. When resource utilization is below the baseline, users accumulate virtual currency that they can later use to gain resources beyond the baseline during periods of high demand. Given that Burstable VMs are primarily useful for dynamic user demands, they will likely need resource allocation mechanisms that guarantee high utilization, strategy-proofness, and fair resource allocation.

**Dynamic user demands.** Increasingly many applications running data analytics or key-value caches operate on data collected from social media, application and network logs, mobile systems, etc. A unique characteristic of these data is that they are less controllable by the organization because they are generated by entities outside of the organization. As a result, applications can observe highly time-varying dynamic resource demands [12, 16, 41, 45, 60, 63, 70, 72, 79].

To build a deeper understanding of variation in user demands over time, we analyze two publicly-available production workloads: (1) Google [60] resource usage information across 8 clusters ($1000-2000$ users per cluster) over a 30 day period; and, (2) Snowflake [72], a cloud-based database query engine that provides resource usage statistics for over 2000 users over a 14 day period. To characterize user demand variability over time, we compute—for each user—the ratio of the standard deviation and mean of their demands over the entire period. Figure 1 (left) shows that $40-70\%$ of all users in both Google and Snowflake workloads have a standard deviation in CPU and memory demands at least $0.5\times$ their mean, indicating high variability in demands for most users. Furthermore, the standard deviation in demands of as many as 20% of the users can be as high as their mean demand, with some users having extremely high variance in demands (standard deviations up to $12-43\times$ the mean). Similar observations have been made for time-varying user demands in inter-datacenter networks; for instance, production studies [5] show that, on average, user demands vary by 35% within 5-minute intervals, with some demands varying by as much as 45% within a short period of time.

Figure 1 (center) shows the CPU and memory demands for a randomly-sampled user from the Snowflake trace over a 15 minute window (we show only one user and only 15 minute window for clarity; analyzing a sample of 100 users, we find 87% of the users to have similar demand patterns). The figure shows that user demands can change dramatically over tens of seconds, by as much as $6\times$ and $2\times$ for compute and memory, respectively. Similarly, we see significant variation in demands even for a random user from the Google trace (shown in Figure 1 (right)).

**Max-min fairness guarantees fail for dynamic user demands.** The classical max-min fairness algorithm for resource allocation provides many desirable properties, *e.g.*, Pareto efficiency, strategy-proofness, and fairness. However, buried under the proofs is the assumption that user demands are static over time, an assumption that does not hold in practice (as demonstrated in Figure 1). For the realistic case of dynamic user demands, max-min fairness can be applied in two ways, each of which leads to violating one or more of its properties. We will demonstrate this using the example in Figure 2; here, time is divided into five quanta and three users have demands varying across quanta.

First, one can naïvely perform max-min fair allocation just once based on user demands at quantum $t = 0$. This results in max-min fairness losing both Pareto efficiency and strategy-proofness. In the example of Figure 2, since allocations will only be done based on the demands specified by the users at $t = 0$, if users were to specify their true demands, user C will obtain an allocation of 1 unit leading to a total useful allocation of 3 units over the entire duration (as shown in Figure 2 (middle, top)); if user C were to lie

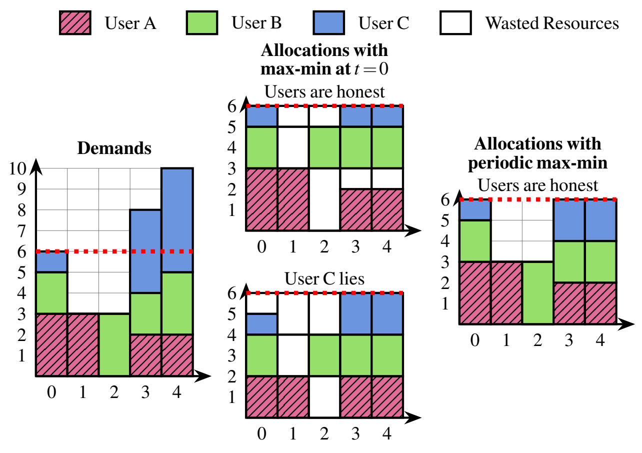


Figure 2: **Classical max-min fairness guarantees break for dynamic user demands.** Here, 6 units of a resource are shared by 3 users (fair share of 2). Discussion in §2.

and over-report their demand at $t=0$ as 2 units, then they can achieve a more desirable total useful allocation of 5 units (Figure 2 (middle, bottom)). This breaks strategy-proofness. In addition, max-min fairness is also not Pareto efficient: for many quanta, resources allocated to users will be underutilized as is evident in Figure 2 (middle).

A better way to apply max-min fairness for dynamic user demands is to periodically reallocate resources based on users' instantaneous demands (*e.g.*, every quantum of time periods, as in several operating systems and hypervisors [3, 73]). This trivially guarantees Pareto efficiency and strategy-proofness but results in extremely unfair allocation across users. Figure 2 (right, top) shows an example where max-min fairness can result in $2\times$ disparity between resources allocated to users over the 5 quanta—user A receives a total allocation of 10 slices, while user C receives a total allocation of only 5 slices, despite them having the same average demand; this example can be easily extended to demonstrate that max-min fairness can, for $n$ users, result in resource allocations where some user gets a factor of $\Omega(n)$ larger amount of resources than other users (proof in [71]). Such disparity in resource allocations also leads to disparity in application-level performance across users since, as discussed above in use cases, many applications require consistently good performance over long periods of time, rather than excellent performance at some times and very poor performance at other times [22, 28, 32, 68]. We will demonstrate, in the evaluation section, that users experience significant disparity in application-level performance due to such disparate resource allocations.

For the rest of the paper, we focus on long-term fairness; informally, an allocation is considered fair if all users have the same aggregate resource allocation over time. Our goal is to design a resource allocation mechanism that, for dynamic user demands, guarantees Pareto efficiency, strategy-proofness, and fairness.

## 3 Karma

Karma is a resource allocation mechanism for dynamic user demands. Karma uses *credits* (§3.1, §3.2)—users receive credits when they donate a part of their fair share of resources (*e.g.*, when their demand is less than their fair share), and can use these credits to borrow resources beyond their fair share during periods of high demand. Karma carefully orchestrates the exchange of resources and credits between donors and borrowers: donors are prioritized in a manner that ensures credit distribution across users remains as balanced as possible, and borrowers are prioritized in a manner that keeps the resource allocation as fair as possible. We will prove theoretically in §3.3 that, while simple in hindsight, this allocation mechanism simultaneously achieves Pareto efficiency, strategy-proofness, and fairness for dynamic user demands.

### 3.1 Preliminaries

We consider the following setup for the problem: we have $n$ users sharing a single resource (CPU, memory, GPUs, etc.); each user has a fair share of $f$ resource units (each unit is referred to as a *slice*), and thus the pool has $n\times f$ slices of the resource (as we discuss in §3.4, all our results hold for users having different fair shares). Time is divided into quanta, users demand a certain number of resource slices every quantum, and Karma performs resource (re)allocation at the beginning of each quantum. While user demands during each quantum can be arbitrary, unsatisfied demands in one quantum do not carry over to the next. Similar to prior work [30, 57, 59, 66], we assume that users are not adversarial (that is, do not lie about their demands simply to hurt others' allocations), but are otherwise selfish and strategic (willing to misreport their demands to maximize their allocations).

### 3.2 Karma design

Let $0\leq\alpha\leq 1$ be a parameter. Karma guarantees that each user is allocated an $\alpha$ fraction of its fair share $(=\alpha\cdot f)$ in each quantum; we refer to this as the guaranteed share. Karma maintains a pool of resource slices—karmaPool—that, at any point in time, contains two types of slices:

- **Shared slices** are the slices in the resource pool that are not guaranteed to any user. It is easy to see that the number of shared slices in the system is $n\cdot f-n\cdot\alpha\cdot f=n\cdot(1-\alpha)\cdot f$.
- **Donated slices**, that are donated by users whose demands are smaller than their guaranteed share.

We use these two sets of slices in the following manner. In any given quantum, if a user has demand less than its guaranteed share, then the user is said to be "donating" as many slices as the difference between the user's guaranteed share and demand in that quantum. A user that has demand larger than its guaranteed share is said to be "borrowing" slices beyond its guaranteed share, which the system can potentially supply using either shared slices or donated slices.

### 3.2.1 Karma credits

Karma allocates resources not just based on users' instantaneous demands, but also based on their past allocations. To maintain past user allocation information, Karma uses credits.

Users earn credits in three ways. First, each user is bootstrapped with a fixed number of initial credits upon joining the system (we discuss the precise number once we have enough context, in §3.4); second, each user is allocated $(1-\alpha)\cdot f$ free credits every quantum as compensation for contributing $(1-\alpha)$ fraction of its fair share to shared slices. Finally, users earn one credit when some other user borrows one of their *donated* slices (one credit per quantum per slice).

Unlike earning credits, there is only one way for any user to lose credits: for every slice borrowed from the karmaPool (donated or shared), the user loses one credit.

### 3.2.2 Prioritized resource allocation

We now describe Karma's resource allocation algorithm, that orchestrates resources and credits across users (Algorithm 1). To make the discussion succinct, we refer to the sum of user demands beyond their guaranteed share as "borrower demand"; that is, to compute borrower demand for any given quantum, we take all users with demand greater than their guaranteed share and sum up the difference between their demand (in that quantum) and $\alpha \cdot f$. In quanta when borrower demand is equal to the supply (number of slices in karmaPool), Karma's decision-making is trivial: simply allocate all slices in karmaPool to the borrowers, and update credits for all users as described in the previous subsection. The key algorithmic challenge that Karma resolves is when the supply is either more or less than the borrower demand. We describe Karma allocation mechanism for such scenarios next and then provide an illustrative example.

**Orchestrating resources and credits when supply > borrower demand.** When supply is greater than borrower demand, there are enough slices in karmaPool to satisfy the demands of all borrowers. In such a case, Karma prioritizes the allocation of donated slices over shared slices (so that donors get credits), and across multiple donated slices, prioritizes the allocation of a slice from the donor that has the smallest number of credits—this allows "poorer" donors to earn more credits, and moves the system towards a more balanced distribution of credits across users. Intuitively, credits capture the allocation obtained by a user until the last quantum—users who obtained lower allocations in the past will have a higher than average (across users) number of credits, while those who received a surplus of allocations will have a below-average number of credits. Hence, balancing the number of credits across users over time allows Karma to move towards a more equitable set of total allocations across users. Once all donated slices are allocated, Karma allocates shared slices to satisfy the remaining borrower demands.

**Algorithm 1 : Karma resource allocation algorithm.**

```
demand[u]: demand of user u in the current quantum
credits[u]: credits of user u in the current quantum
alloc[u]: allocation of user u in the current quantum
$f$: fair share
$\alpha$: guaranteed fraction of fair share

Every quantum do:
 1: shared_slices $\leftarrow n \cdot (1-\alpha) \cdot f$
 2: For each user u,
 3:     increment credits[u] by $(1-\alpha) \cdot f$
 4:     donated_slices[u] = max $(0, \alpha \cdot f -$ demand[u]$)$
 5:     alloc[u] = min (demand[u], $\alpha \cdot f$)
 6: donors $\leftarrow$ all users u with donated_slices[u] $> 0$
 7: borrowers $\leftarrow$ all users u with
 8:       alloc[u]<demand[u] & credits[u]>0

 9: while borrowers $\neq \phi$ and
10:       ($\sum_u$ donated_slices[u] $> 0$ or shared_slices $> 0$)
    do
11:     $b^\star \leftarrow$ borrower with maximum credits
12:     if donors $\neq \phi$ then
13:       $d^\star \leftarrow$ donor with minimum credits
14:       Increment credits[$d^\star$] by 1
15:       Decrement donated_slices[u] by 1
16:       Update the set of donors (line 6)
17:     else
18:       Decrement shared_slices by 1
19:     Increment alloc[$b^\star$] by 1
20:     Decrement credits[$b^\star$] by 1
21:     Update the set of borrowers (line 7)
```

**Orchestrating resources and credits when supply < borrower demand.** When supply is less than demand, karmaPool does not have enough slices to satisfy all borrower demands. In such a scenario, Karma prioritizes allocating slices to users with the maximum number of credits. This strategy essentially favors users that had fewer allocations in the past (and thus, a larger number of credits), hence moving the system towards a more balanced allocation of resources across users, promoting fairness. At the same time, reducing the credits for the users with the most credits also moves the system to a more balanced distribution of credits across users.

**Illustrative example.** We now illustrate through a concrete example. The running example in Figure 3 shows the execution of Karma's algorithm for the example from Figure 2 for $\alpha = 0.5$: that is three users A, B, and C, each with a fair share 2 slices ($f = 2$), and a guaranteed share of 1 slice. Recall that, since $(1-\alpha)\cdot f = 1$, each user receives 1 credit every quantum, and suppose all users are bootstrapped with 6 initial credits.

In the first quantum, C's demand is equal to the guaranteed share, while A and B request 2 and 1 slices beyond the guaranteed share, respectively. Since supply (= 3 shared slices in karmaPool) is equal to borrower demand, Karma uses the shared slices to allocate slices beyond the guaranteed share for

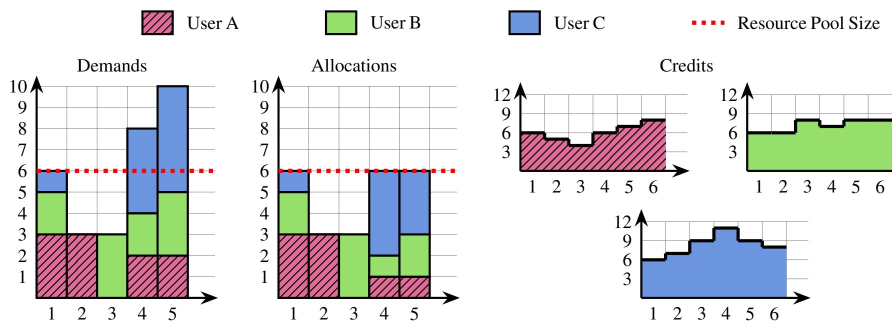


Figure 3: **Karma resource allocation for the running example of Figure 2:** Recall that there are 6 resource slices, 3 users each with average demand and fair share equal to 2. We show the case of the guaranteed share being 1 ($\alpha = 0.5$), with 6 bootstrapping (initial) credits for each user. Note that each user receives 1 free credit every quantum. Karma achieves significantly improved fair allocation than max-min fairness—it allocates each user an equal allocation of 8 resource slices over time.

A and B and satisfies their demands. This results in a final allocation of 3 slices for $A$, 2 slices for $B$, and 1 slice for $C$. $A$ loses 2 credits, and $B$ loses 1 credits, and no one gains any credits.

In the second quantum, A demands 3 slices, while B and C donate 1 slice each. The total supply (=5, with 2 donated slices and 3 shared slices) exceeds the borrower demand. A is allocated 3 slices and it loses 3 credits (since its allocation is 2 slices above its guaranteed share). B and C receive 1 credit each since their donated slices are used. Similarly, in the third quantum, B demands 3 slices, while A and C donate 1 slice each. Since total supply exceeds borrower demand, B receives the 3 slices it asked for, and loses 2 credits; A and C gain 1 credit each.

The fourth quantum is important: here, demand exceeds supply, and there are no donated slices. Now, unlike classic max-min fairness, Karma will prioritize the allocation of resources based on the credits of each tenant. Since at the start of this quantum, C has 11 credits, while A and B have only 6 and 7 credits respectively, C will be able to get 3 extra slices from the pool of shared slices by using 3 credits and achieve an allocation of 4. A and B will get their guaranteed allocation of 1 and do not gain or lose any credits.

In the fifth quantum, once again, demand exceeds supply. C has 9 credits, B has 8 credits, and A has 7 credits. Karma first prioritizes allocating to C giving it 1 extra slice, at which point both C and B have equal credits (8). Next, they both get 1 extra slice each, at which point the supply is exhausted. The final resulting allocation is 1 slice for A, 2 slices for B, and 3 slices for C.

In the end, A, B, and C end up with the exact same total allocation (8 slices) and number of credits (unlike max-min fairness where user allocations had a disparity of $2\times$).

## 3.3 Karma Properties & Guarantees

In this section, we present a theoretical analysis of Karma. Recall from §3.1 that, similar to all prior works, users are considered selfish and strategic (that is, are willing to misreport their demands to maximize their allocations), but not adversarial (that is, do not lie about their demands simply to hurt others' allocations). For the purpose of our theoretical analysis, we assume that Karma is initialized with a large enough number of initial credits so that users do not run out of credits during the execution of the algorithm (we discuss how to achieve this in practice in § 3.4). All our results hold for $\alpha = 0$; extending our results to $\alpha > 0$ is an interesting open question. Finally, while we provide inline intuition for each of our results, full proofs are presented in [71].

We define Pareto efficiency on a per-quantum basis. An allocation is said to be Pareto efficient if it is not possible to increase the allocation of a user without decreasing the allocation of at least one other user by a similar total amount during that quantum. Note that, Pareto efficiency on a per-quantum basis implies Pareto efficiency over time.

**Theorem 1.** *Karma is Pareto efficient.*

Karma's Pareto efficiency follows trivially from the observation that similar to max-min fairness, Karma allocation satisfies the two properties: (1) no user is allocated more resources than its demand, and (2) either all resources are allocated or all demands are satisfied.

For strategy-proofness, we make two important notes. First, if one assumes that the system has a priori knowledge of all future user demands, the resource allocation problem can be solved trivially using dynamic programming; however, for many use cases, it is hard to have a priori knowledge of all future user demands. This leads to our second note: Karma is solving an "online" problem (that is, it does not assume a priori knowledge of future user demands), and thus, we prove online strategy-proofness [7] defined as follows: assume that all users are honest during quanta 0 to $q-1$; then, a mechanism is said to be online strategy-proof if, for any quantum $q$, a user cannot increase its allocation during quantum $q$ by lying about its demand during quantum $q$.

**Theorem 2.** *Karma is online strategy-proof.*

To prove Theorem 2, we actually prove a stronger result stated below. Karma's online strategy-proofness trivially follows from this.

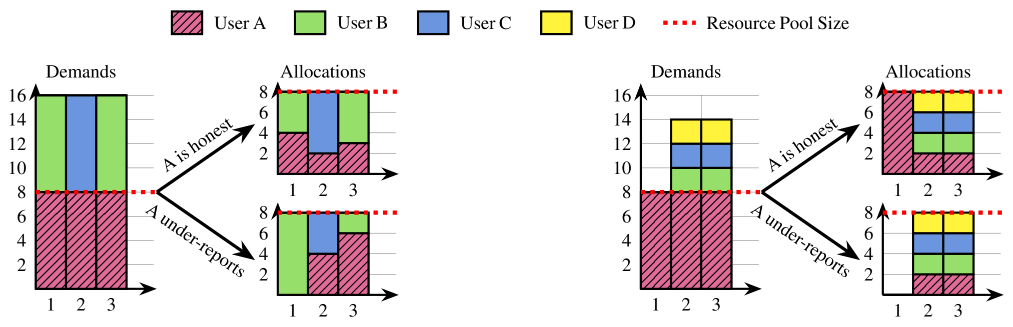


Figure 4: **The phenomenon of users (left) gaining a small factor of improvement in their allocations by specifying demands less than their real demands, by exploiting knowledge of all future demands of all users; (right) any imprecision in the knowledge of future demands of all users could result in a significant reduction in useful allocations of the lying user.** The resource pool has 8 slices, and 4 users with fair share of 2 and guaranteed share of 0 ($\alpha = 0$).

**Lemma 1.** *A user cannot increase its useful resource allocation by specifying a demand higher than its real demand in any quantum.*

The proof for the lemma is a bit involved, but intuitively, it shows the following. The immediate effect of a user specifying a demand higher than its actual demand is that if the user is allocated more resources than its actual demand, these extra resources do not contribute to its utility, but do put the user into a disadvantageous position: not only can this user lose credits (either because it's asking for resources beyond its guaranteed share, or because it could have gained credits if this extra resource could have been allocated to some borrower), but also because other users get fewer resources; this makes other users be favored by the allocation algorithm in the future while making the lying user less favored. Thus, the user cannot increase its long-term "useful" allocation by specifying a demand higher than the real demand in any quantum. Specifically, it is possible that when a user over-reports its demand during quantum $q'$, the user receives an increased instantaneous allocation during some future quantum $q > q'$; however, we are able to show that, in this case, the user will also receive reduced instantaneous allocation(s) during other quantum(s) in between $q'$ and $q$, leading to either a lower or equal total allocation over the period between $q'$ and $q$. The hardness in the proof stems from carefully analyzing such cascade effects: a small change in users' resource allocation in any quantum can result in complex changes in future allocations that may lead to higher instantaneous but equal or lower total allocations in future quanta. Once we prove this lemma, the proof for Karma's online strategy-proofness follows immediately.

While analyzing Karma properties, we encountered a new, surprising, phenomenon that may be of further theoretical interest: we show that a user that *knows all future demands of all other users* can report a demand that is lower than its actual demand in the current quantum to increase its allocation in future quanta by a small constant factor. However, any imprecision in the knowledge of all future demands of all other users could result in the user losing a factor of $\Omega(n)$ of its total allocation.

**Lemma 2.** *A user cannot increase its total useful allocation by a factor more than $1.5\times$ by specifying a demand less than its real demand in any quantum. Gaining this useful allocation requires the user to know the future demands of all users. If the user does not have a precise knowledge of all future demands of all users, it can lose its useful allocation by a factor of $\frac{n+2}{2}$ (for $n \geq 3$) by specifying a demand less than its real demand.*

We provide intuition for this phenomenon using an example (Figure 4). In the left figure, user A is able to gain 1 extra slice in its overall allocation by under-reporting its demand (reporting 0 instead of 8) in the first quantum. By under-reporting, its allocation in the first quantum reduces, enabling it to get more resources during the second quantum when it competes with user C. In the third quantum, it is able to recover the resources it lost in the first quantum from user B, resulting in an overall gain. To see the flip-side, if the demands of other users had been as shown in Figure 4 (right), then user A sees a $3\times$ degradation in overall allocation.

To prove the first part of the Lemma 2, we consider an arbitrary user Alice and an arbitrary time period, and compare two scenarios—one where Alice is truthful (hereby called the truthful scenario) and one where Alice is deviating by under-reporting her demand during some quantum (hereby called the deviating scenario).

Our key insight for the proof is that bounding the increase in total allocation of *all users* is easier than reasoning about the increase in total allocation of an individual user (Alice) since even a small change in Alice's demand during one quantum can result in cascading effects on the total allocation of other users as well. To that end, we prove the following claim: the total amount of resources all the users have earned in excess in the deviating scenario compared to the truthful one can be at most as large as Alice's total allocation in the truthful scenario. We prove this claim based on the following observation: whenever Alice under-reports her demand she is effectively "donating" the allocation she would have gotten in the truthful scenario to the other users whose allocations in the deviating scenario increase. Since Karma is Pareto efficient, the total gain in allocation across users during this quantum is limited by the amount donated by Alice which is in turn bound by Alice's own allocation during this quantum in the truthful scenario. By applying this reasoning iteratively across all quanta[1], we can show that the total increase in allocation across all users cannot exceed the total allocation of Alice in the truthful scenario. This already implies a $2\times$ upper bound on the maximum increase in total allocation that Alice can achieve.

To tighten the upper bound, we prove a second claim: if Alice receives higher total allocation in the deviating scenario compared to the truthful scenario, then there must exist some other user Bob who gained an even larger increase in total allocation than Alice. Putting together the above two claims allows us to establish the desired upper bound. Based on the first claim, the total gain in allocation across all users cannot exceed Alice's total allocation in the truthful scenario. This implies that the sum of total gains across Alice and Bob cannot exceed Alice's total allocation in the truthful scenario. Since Bob's gain is at least as large as Alice's gain (based on the second claim), this implies that Alice's gain is at most half the total allocation of Alice in the truthful scenario—a gain of at most $1.5\times$, thus proving the first part of Lemma 2.

The second part of the lemma is proven by first creating a set of demands where a user can under-report its demand during quantum $q$ to earn increased total allocation by some quantum $q' > q$. Then we create a set of demands that are identical up to quantum $q$ but vastly different from quanta $q+1$ to $q'$. If the user (in the hope of facing the first set of demands) under-reports its demand on quantum $q$ but ends up facing the second set of demands then this results in vastly different allocations by quantum $q'$. By correctly picking the two sets of demands we get the desired bounds.

In [71], we prove an even stronger result that extends Karma properties from Theorem 1, Theorem 2, Lemma 1 and Lemma 2 to the case of multiple colluding users:

**Theorem 3.** *No group of colluding users can increase their allocation by specifying a demand higher than their real demand. Additionally, for any group of colluding users, under-reporting demands cannot lead to more than a $2\times$ improvement in their useful resource allocation. Finally, even if users form coalitions, Karma is Pareto efficient and online strategy-proof.*

Recall that Karma focuses on long-term fairness without a priori knowledge of future user demands. To that end, the following theorem summarizes Karma's fairness guarantees:

**Theorem 4.** *For any quantum $q$, given fixed user allocations from quantum $0$ to quantum $q-1$, and user demands at quantum $q$, Karma maximizes the minimum total allocation from quantum $0$ to quantum $q$ across users.*

[1]It turns out that Alice under-reporting in a given quantum cannot cause cascading increases in total allocation across users in future quanta if Alice does not under-report in future quanta. This is because Karma prioritizes allocation to users with high credits (or equivalently low total allocations).

The proof for the above theorem follows from the prioritized resource allocation mechanism of Karma. Intuitively, given allocations from quantum 0 to $q-1$, the user with the least total allocation up to quantum $q-1$ will have the largest number of credits. In quantum $q$, Karma will prioritize the allocation of resources to this user (until it is no longer the one with the minimum total allocation, after which it will prioritize the next user with the minimum total allocation, and so on), thus maximizing the minimum total allocation from quantum 0 to $q$ across users—this is the best one can do in quantum $q$ given past allocations.

## 3.4 Discussion

Finally, we briefly discuss some additional aspects of Karma design not included in the previous subsections.

**Bootstrapping Karma with initial credits.** Recall that, to bootstrap users, Karma allocates each user an initial number of credits. The precise number of initial credits has little impact on Karma's behavior; after all, credits in Karma essentially capture a relative ordering between users, rather than having any absolute meaning. The only importance of the number of credits is to ensure that no user runs out of credits at any quantum (which, in turn, could lead to violation of Karma's Pareto efficiency guarantees): even if spare resources are available, a user with high demand may not be able to borrow resources beyond the guaranteed share (line 7 of Algorithm 1) due to running out of credits. Thus, Karma sets the number of initial credits to a large numerical value to ensure that no user ever runs out of credits[2],

**User churn.** Fairness is relatively ill-defined when users can join and leave the system on a short-term basis (*e.g.*, when a user runs a short query with large parallelism, and then leaves the cluster). Also, recall from our motivating scenarios, fair resource allocation in private clouds is usually performed for long-running services. However, Karma still handles user churn since, in many realistic scenarios, the set of all users of the system may not be known upfront during system initialization. For users that join and leave over longer timescales, Karma handles user churn with a simple mechanism: its credits. When a new user joins, either the resource pool size remains fixed and the fair share of all users is reduced proportionally or the resource pool size increases and the fair share of users remains the same. The credits of the existing $n-1$ users do not change, and the new user is bootstrapped with initial credits equal to the current average number of credits across the existing $n-1$ users. Intuitively, users who have donated more resources than they have borrowed will have above-average credits, and those who have borrowed

[2]For example, in a system with 100 users with fair share of 100 slices, setting initial credits to say $10^{13}$ will ensure that even a worst-case user with highest possible demand (10000 slices) during all quanta cannot run out of credits for $\sim 31$ years, which is good enough for all practical purposes.

more than they have donated will have below-average credits. As such, initializing the new user with the average number of credits (heuristically) puts the new user on equal footing with an existing user that has borrowed and donated equal amounts of resources over time. When a user leaves the system, the fair share of the remaining users is increased proportionally (or resource pool size reduces while maintaining the same fair share), and there is no change in their credits.

**Users with different fair shares.** We have presented Karma's algorithm for the case of users having the same fair share merely for simplicity: all our results extend to the case of users having different fair shares. To generalize the algorithm to users with different fair shares, users with larger weights are charged fewer credits to borrow resources beyond their guaranteed share when compared to users with smaller weights. Intuitively, this enables users with larger weights to obtain more resources than users with smaller weights for the same number of credits. We achieve this by updating Line 20 of Algorithm 1 to decrement credits by $\frac{1}{n \cdot w_i}$ instead of 1, where $w_i$ is the normalized weight of the corresponding user, and $n$ is the number of users. For users with different fair shares, this generalization leads to the same properties and guarantees as discussed in §3.3 (the only difference, is that the upper bound factor in Lemma 2 changes from $1.5\times$ to $2\times$). A full description of the weighted version of the algorithm along with proofs of guarantees can be found in [71].

**System parameters, and interpretation for $\alpha$.** Karma has only one parameter: $\alpha$; one can think of resource slice size and quantum duration as parameters, but these are irrelevant to Karma's guarantees: they hold for any slice size and quantum duration, as long as demands change at coarse timescales than the quantum duration. The $\alpha$ parameter in Karma provides a tradeoff between instantaneous and long-term fairness. Providers can choose any $\alpha$ depending on the desired properties. Intuitively, an $\alpha$ smaller than 1 leads to a larger portion of shared slices, giving Karma's algorithm more flexibility in adjusting allocations to achieve better long-term fairness.

## 4 Karma Implementation Details

We have implemented Karma on top of Jiffy [41], an open-sourced elastic far memory system. Jiffy has a standard distributed data store architecture (Figure 5(a)): resources are partitioned into fixed-sized slices (blocks of memory) across a number of resource servers (memory servers), identified by their unique sliceIDs (referred to as blockIDs in Jiffy). A logically centralized controller tracks the available and allocated slices across the various resource servers and stores a mapping that translates sliceIDs to the corresponding resource server. We have implemented Karma as a new resource allocation algorithm at the Jiffy controller[3].

[3]Karma can thus directly piggyback on Jiffy's existing mechanisms for controller fault tolerance [41, Section 4] to persist its state across failures.

Users interact with the system through a client library that provides APIs for requesting resource allocation and accessing allocated resource slices. Users express their demands to the controller through resource requests which specify the number of slices required. The controller periodically performs resource allocation using the Karma algorithm and provides users with the sliceIDs of the resource slices that are allocated to them. Users can then directly access these slices from the resource servers through read or write API calls without requiring controller interposition. In the rest of this section, we discuss the key data structures and mechanisms required to integrate Karma with Jiffy.

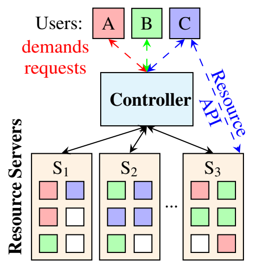
Karma employs three key data structures to efficiently implement the policies and mechanisms outlined in §3: karmaPool, a credit map, and a rate map.

**karmaPool.** Recall from §3.2 that the karmaPool tracks the pool of donated slices and shared slices, and needs to be updated when resource allocations change. Also, the resource allocation algorithm should be able to efficiently select donated slices from a particular user while satisfying borrower demands (§3.2.2). To this end, the karmaPool is implemented as a hash map, mapping userIDs to the list of sliceIDs corresponding to slices donated by them. The list of sliceIDs corresponding to shared slices is stored in a separate entry of the same hash map. When resource allocations change, the corresponding sliceIDs are added to or removed from the corresponding lists. As such, karmaPool supports all updates in $O(1)$ time.

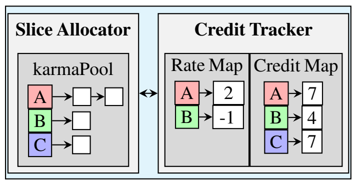
**Credit Tracking.** Karma employs two data structures for tracking and allocating credits across various users: a rate map and a credit map. The rate map maps each user to the *rate* at which it earns or spends its credits every quantum, that is, the difference between the user's guaranteed share and the number of its allocated slices in that quantum. The rate is positive when the user is earning (that is, has donated slices) and negative when it is spending credits (that is, has borrowed slices), respectively. The credit map, on the other hand, maps each user to a counter corresponding to its current credits.

Separating the rate map and credit map facilitates efficient credit tracking at each quantum: Karma simply iterates through the rate map entries, and updates the credit counters in the credit map based on the corresponding user credit rates. Since the rate map only contains entries for users with non-zero rates, Karma can efficiently update credits for only the relevant users. At the same time, Employing a hash-map for each of them permits $O(1)$ updates to the user credit rate or number of credits while performing resource allocation.

**Borrowing and donating slices.** Karma realizes its credit-based prioritized allocation algorithm (§3.2) using two modules at the controller. First is a *slice allocator* that maintains the karmaPool to track and update slice allocations across users, and, second a *credit tracker* that maintains the current number of credits for any user (via Credit Map) and



(a) System Architecture



(b) Karma state in the Controller

Figure 5: **Karma Design.** See §4 for details.

how it should be updated (via Rate Map). Figure 5(b) shows these modules along with the data structures they manage.

The slice allocator intercepts resource requests from users, periodically executes the Karma resource allocation algorithm (Algorithm 1) to compute allocations based on the user demands, and updates slices in the karmaPool accordingly. It interacts with the credit tracker to query and update user credits. A naïve implementation of Algorithm 1 runs in $O(n \cdot f \cdot \log n)$ time, where $n$ is the number of users, and $f$ is the fair share[4]. Instead of computing allocations one slice at a time, we use an optimized implementation that carefully computes them in a batched fashion (full details are provided in [71]). This enables the slice allocator to support resource allocation at fine-grained timescales.

**Consistent hand-off of resources.** Since users are allowed to directly access slices from resource servers, we need to ensure consistent hand-off of slices from one user to another when slices are reallocated. For example, say user $U_1$ has a slice during a given quantum, and in the next quantum, this slice is allocated to user $U_2$. We need to ensure that (1) $U_1$'s data is flushed to persistent storage before $U_2$ overwrites it (2) $U_1$ should not be able to read/write to the slice after $U_2$ has accessed it (for example, there could be in-flight read/write requests to the slice which were initiated before $U_1$ gets to know it's allocation changed).

Karma ensures the above by maintaining a monotonically increasing sequence number and current userID for each slice, at both the controller (within the karmaPool) and the resource servers (as slice metadata). On slice allocation, its userID is updated and its sequence number is incremented at the controller, and the sequence number is returned to the user. Subsequent user reads and writes to the slice specify this userID and sequence number. A slice read succeeds only if the accompanying sequence number is the same as the current slice sequence number, while a slice write succeeds only if the accompanying sequence number is the same or greater than the current sequence number. If a write necessitates an overwrite of the current slice content and metadata, the old slice content is transparently flushed persistent storage (*e.g.*, S3) before the overwrite. In our example above, $U_2$'s first access to the slice after re-allocation will trigger a flush of $U_1$'s data to S3 and update the slice sequence number. Following this $U_1$'s accesses to this slice will fail since the current sequence number of the slices is higher. $U_1$ can then read/write this data from persistent storage. Implementing consistent resource hand-off in Jiffy required minor changes to the controller (to track sequence numbers per slice), memory servers (to perform sequence number checking), and the client library (to tag requests with sequence numbers).

[4]The loop in Line 10 of Algorithm 1 takes $O(n \cdot f)$ iterations and each iteration would take $O(\log n)$ time to find the donor/borrower with the minimum/maximum credits (if we were to maintain min/max heaps for the donor and borrower sets).

## 5 Evaluation

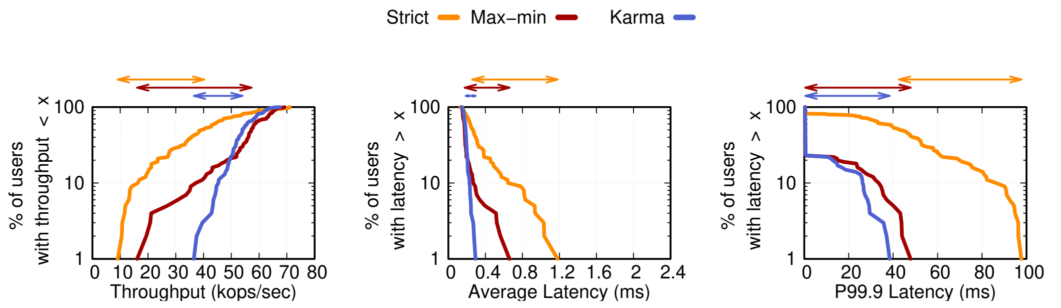
We have already established Karma properties theoretically in §3. In this section, we evaluate how Karma's properties translate to application-layer benefits over an Amazon EC2 testbed with real-world workloads. Our evaluation demonstrates that:

- Karma reduces the performance disparity between different users by $\sim 2.4\times$ relative to classic max-min fairness, without compromising on system-wide utilization or average performance (§5.1);
- Karma incentivizes users to share resources, quantifying Karma's online strategy-proofness property (§5.2);

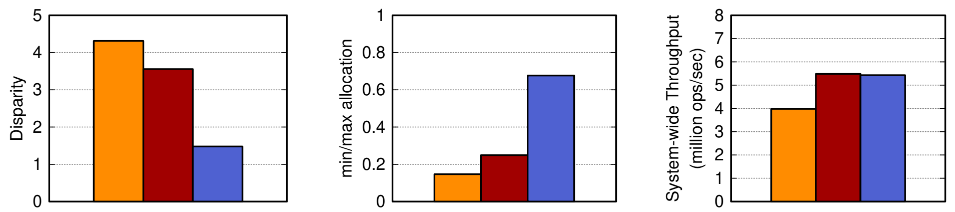
We primarily focus on the shared cache use case from §2 for the following reason. While datasets for the shared data analytics clusters use case are publicly available (*e.g.*, Google and Snowflake datasets), they do not provide user queries that may impact our final conclusions. For the shared cache use case, we do have all the information we need: these datasets provide information on the working set size of each user over time, which can be fed into an end-to-end multi-tenant in-memory cache system running on Amazon EC2. We, thus, focus on this use case.

**Experimental setup.** Our experimental setup consists of a distributed elastic in-memory cache shared across multiple users backed by a remote persistent storage system. For the cache, we use Jiffy [41], augmented with our implementation of Karma (§4) and other evaluated schemes. If the evaluated scheme does not allocate sufficient slices to a user on Jiffy to fit its entire working set, the remaining data is accessed from remote persistent storage. When slices are reallocated between users across quanta, the corresponding data is moved between Jiffy and persistent storage through the consistent hand-off mechanism described in §4. We deployed our setup on Amazon EC2 using c5n.9xlarge instances (36 vCPUs, 96GB DRAM, 50Gbps network bandwidth). We host the Jiffy controller and resource servers across 7 instances and use 25 instances for the users/clients that issue queries to Jiffy. We use Amazon S3 as the persistent storage system.

**Workload.** We use the publicly available Snowflake dataset [72] that provides dynamic user demands in terms of memory usage for each customer from Snowflake's



(a) CDF of throughput across users (b) CCDF of average latency across users (c) CCDF of P99.9 latency across users



(d) Disparity in throughput across users (median/min) (e) Fairness in overall allocations (min/max allocation) (f) System-wide average throughput (million ops/sec)

Figure 6: **Understanding Karma benefits.** (a) Karma enables a much tighter throughput distribution across users (colored arrows show the absolute gap between median and minimum throughput across users). (b, c) It also enables a tighter distribution of average and tail latencies across users (again, colored arrows show the absolute gap between median and maximum latency across users). (d) Karma achieves much lower throughput disparity—ratio of median to minimum values of throughput across users—than classic max-min fairness. (e) It also significantly reduces the gap between the users with minimum and maximum overall allocations, (f) while achieving similar system-wide performance as max-min fairness.

production cluster. We use these demands as the dynamic working set size for individual users. For each user, we issue data access queries using the standard YCSB-A workload [20] (50% read, 50% write) with uniform random access distribution, with queries during each quantum being sampled (according to the YCSB parameters) within the instantaneous working set size of that user. If a query references data that is currently cached in Jiffy, then it is serviced directly from the corresponding resource server; otherwise, it is serviced from the persistent storage.

**Default parameters.** Unless specified otherwise, we randomly choose 100 users (out of $\sim$ 2000 users) over a randomly-chosen 15 minute time window (out of a 14-day period) in the Snowflake workload. To test for extreme scenarios, we set the length of each quantum to be one second (that is, a total of 900 quanta). The fair share of each user is 10 slices, and the total memory capacity of the system is set to the number of users times the fair share (1000 slices). Each slice is 128MB in size, while each query corresponds to a read or write to a 1KB chunk of data (the default size in the YCSB workload).

**Compared schemes.** We compare Karma to strict partitioning and max-min fairness, since they correspond to the two most popular fair allocation schemes, and represent extremes in resource allocation and performance. When evaluating Karma, we set the number of initial credits to a large value[5]. The fraction of fair share that is guaranteed ($\alpha$) is 0.5 by default.

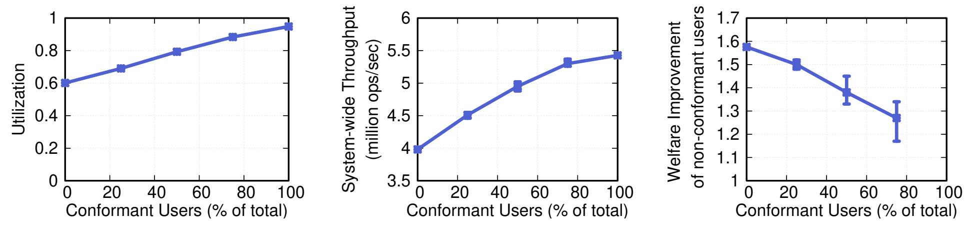
**Metrics.** We evaluate system-wide resource utilization, along with both per-user and system-wide performance—key metrics for any resource allocation mechanism. For performance, we measure both throughput and latency (average and 99.9th percentile tail). We define performance *disparity* for an allocation scheme as the ratio of median to minimum performance (that is, throughput or latency) observed across various users. For any given user, we define *welfare* over time $t$ as $\frac{\sum_t \text{allocations}}{\sum_t \text{demands}}$, that is, the fraction of its total demands satisfied by the allocation scheme. We define *fairness* as $\frac{min_{users}\text{welfare}}{max_{users}\text{welfare}}$ (higher is better, 1 is optimal), as a measure of welfare disparity between users.

## 5.1 Understanding Karma Benefits

We now evaluate Karma's benefits in terms of reducing disparity across users' application-level performance as well as resource allocation.

**Karma reduces performance disparity between users.** Figure 6(a) shows the throughput distribution across users for our compared schemes; the y-axis is presented in log-scale to

[5]As discussed in §3.4, the precise value is unimportant. Here, we set it to 900,000, so that even if a user was allocated the full system capacity for the entire duration ($1000 \times 900$) it would not run out of credits.



(a) Utilization (b) Performance (c) Welfare improvement

Figure 7: **Karma incentivizes resource sharing.** All metrics are computed as averages (with error bars) for three random selections of users being non-conformant. See §5.2 for details.

focus on the users at the tail of the distribution, which observe the most performance disparity. Since Karma strives to balance fairness over time, it significantly narrows the throughput distribution across users compared to the two baselines: the ratio between the maximum and minimum throughput across all users is 7.8× with strict partitioning and 4.3× with max-min fairness, but only 1.8× for Karma. As Figure 6(d) shows, Karma lowers the throughput disparity across users by 2.4× compared to max-min fairness. Karma also reduces average latency disparity (Figure 6(b)) by 2.4× and 99.9th percentile latency disparity (Figure 6(c)) by 1.2× compared to max-min fairness by enabling a tighter distribution for both latencies.

Equitability in performance across users for a scheme is closely tied to how fairly resources are allocated across users. Specifically, because of the large gap between elastic memory (Jiffy) and S3 latencies (50–100×), accesses to slices in S3 result in significantly lower throughput than accesses to slices in elastic memory. As a result, users' average throughput ends up being roughly proportional to their total allocation of slices in elastic memory over time. Similarly, since a larger total allocation results in a smaller fraction of requests going to S3, average and tail latencies also reduce.

**Karma reduces disparity in allocations.** We now quantify disparities in overall allocations obtained by users across our compared schemes via our fairness metric in Figure 6(e). Due to dynamic demands, strict partitioning exhibits very poor fairness, since users with very bursty demands end up getting much lower total allocations than users who have steady demands[6]. While, max-min fairness observes better fairness compared to strict partitioning, the best-off user still receives 4× higher allocation than the worst-off user, resulting in poor absolute fairness. Karma achieves significantly better fairness with the best-off user receiving only 1.5× higher allocation than the worst-off user. It is able to achieve this by prioritizing the allocation of resources beyond the fair share to users with more credits (§3.2.2).

**Karma achieves Pareto efficiency and high system-wide performance.** Karma achieves the same overall resource utilization as max-min fairness ($\sim$ 95%). This is because Karma is Pareto efficient (§3.3) similar to max-min fairness and thus achieves near-optimal utilization. We find that the optimal utilization is $<$ 100% since some quanta observe total user demands less than system capacity.

Max-min fairness observes 1.4× higher system-wide throughput (that is, throughput aggregated across all users) than strict partitioning (Figure 6(f)) since it permits allocations beyond the fair share, allowing more requests to be served on faster elastic memory. Karma observes system-wide performance similar to max-min fairness for similar reasons; the slight variations are attributed to variance in S3 latencies.

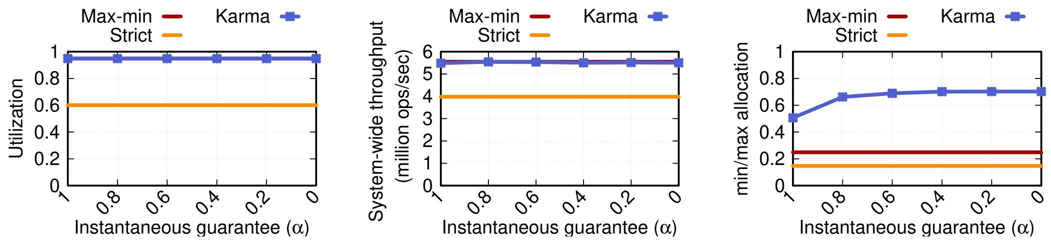
## 5.2 Karma Incentives

We now empirically demonstrate that Karma incentivizes users to donate resources instead of hoarding them, to improve their own as well as overall system welfare. To this end, we vary the fraction of users using Karma that are *conformant* or *non-conformant*. A conformant user is truthful about its demands and donates its resources when its demand is less than its fair share. A non-conformant user, on the other hand, always asks for the maximum of its demand or its fair share (that is, it over-reports its demand during some quanta).

**Resource utilization and system-wide performance improve with more conformant users.** Figure 7(a) and Figure 7(b) show that Karma's system-wide utilization and performance improve as the fraction of conformant users increases. This is because as more users donate resources when they do not need them, other users can use these resources, improving overall utilization and performance. When none of the users are conformant, since no one ever donates any resources, Karma essentially reduces to strict partitioning, hence achieving low overall utilization and performance. When all users are conformant, Karma achieves optimal utilization and performance, similar to classic max-min fairness.

**Becoming conformant improves user welfare.** Figure 7(c) shows the average welfare gain non-conformant users would achieve if they were to become conformant. When non-conformant users become conformant, it leads to significant (1.17–1.6×) welfare gains for them, empirically

[6]Note that only *useful* allocations are considered—strict partitioning guarantees a fixed allocation at all times, but resources may remain unused when demand is low.



(a) Utilization (b) Performance (c) Fairness

Figure 8: **Sensitivity analysis with varying instantaneous guarantee (α)** (a, b) Karma matches the resource utilization and system-wide performance of max-min fairness independent of α (c) Smaller values of α result in improved long-term fairness.

validating Karma's property that users have nothing to gain by over-reporting their demand (§3.3). Note that the gain varies with the number of conformant users in the system—the gains from non-conformant users becoming conformant are higher when the percentage of conformant users is low. As expected, the gains show diminishing returns as more users in the system become conformant as overall utilization is already high.

## 5.3 Karma Sensitivity Analysis

We now show sensitivity analysis with the only parameter in the Karma algorithm–the instantaneous guarantee (α). Figure 8 shows the resource utilization, system-wide performance, and fairness with α varying between 0 and 1. Karma continues to match the resource utilization and system-wide performance of max-min fairness independent of α (Figure 8(a) and Figure 8(b)). Varying α has an impact on the long-term fairness achieved by Karma (Figure 8(c)), with smaller values of α resulting in improved fairness, thus validating our discussion in §3.4. Even for $\alpha = 1$, Karma is able to achieve significantly better fairness compared to max-min fairness. This is because, while it allocates resources up to the fair share identically to max-min fairness, it prioritizes allocation beyond the fair share based on credits.

## 6 Related Work

There is a large and active body of work on resource allocation and scheduling, exploring various models and settings; it would be a futile attempt to compare Karma with each individual work. We do not know of any other resource allocation mechanism that guarantees Pareto efficiency, strategy-proofness, and fairness similar to Karma for the case of dynamic user demands; nevertheless, we discuss below the most closely related works.

**Max-min fairness variants in cloud resource allocation and cluster scheduling.** Many works study variants of max-min fairness for cloud resource allocation and cluster scheduling [8,10,17,30–33,44,46,57–59,64,66,77], including recent work on ML job scheduling [15,34,47,50,55]. We make three important notes here. First, while dominant resource fairness (DRF) [30] has generalized max-min fairness to multiple resources, it makes the same assumptions as max-min fairness: user demands being static over time; our goals are different: we have identified and resolved the problems with max-min fairness for the case of a single resource but over dynamic user demands. It is an interesting open problem to generalize Karma for the case of multiple resources.

Second, cluster scheduling has been studied under several metrics beyond fair resource allocation (*e.g.*, job completion time, data locality, priorities, etc.). Themis [47] considers long-term fairness but defines a new ML workload-specific notion of fairness, and is therefore not directly comparable to Karma. Our goals are most aligned with those works that study fair allocation under strategic users while guaranteeing Pareto efficiency. To that end, the closest to Karma is CARBYNE [32]. However, CARBYNE not only assumes non-strategic users but also, for the single-resource case (the focus of this paper), CARBYNE converges to max-min fairness. As discussed earlier, generalizing Karma to multiple resources remains an open problem; a solution for that problem must be compared against CARBYNE.

Finally, fairness in application-perceived performance is only indirectly related to fairness in resource allocation: other factors like software systems (*e.g.*, hypervisors and storage systems) and resource preemption granularity can impact performance. Similar to other mechanisms [9, 10, 16, 30–33, 39, 45, 46, 59, 60, 66, 67, 72, 77, 80], Karma's properties are independent of these system-level factors; while our evaluation shows that Karma properties translate to application-level benefits, absolute numbers depend on the underlying system implementation.

**Allocation of time-shared resources.** Generalized Processor Sharing (GPS) [54] is an idealized algorithm for sharing a network link which assumes that traffic is infinitesimally divisible (fluid model). For equal-sized packets and equal flow weights, GPS reduces to Uniform Processor Sharing [54, Section 2], which is equivalent to max-min fairness. GPS guarantees fairness over arbitrary time intervals only under the assumption that flows are *continuously backlogged* [54, Section 2]. This assumption implies that flows always have demand greater than their fair share, making it trivial to guarantee a max-min fair share of the network bandwidth over arbitrary time intervals. Classical fair-queueing algorithms [11, 24, 48, 65, 83] in computer networks approximate GPS with the constraint of packet-by-packet scheduling. Under this constraint, varying-sized packets and different flow weights make it hard to realize fairness efficiently; thus, the technical question that these algorithms solve is to achieve fairness approximately equal to GPS with minimal complexity. Karma focuses on a different problem—we show that GPS guarantees (equivalent to max-min fairness) are not sufficient when demands are dynamic and present new mechanisms to achieve fairness while maintaining other properties for such dynamic demands.

Stride [74] scheduling essentially approximates GPS in the context of CPU scheduling [74, Section 7], and thus the above discussion applies to it as well. DRF-Q [29] generalizes DRF to support both space and time-shared resources, but is explicitly designed to be memoryless similar to max-min fairness, and therefore suffers from similar issues for long-term fairness. Least Attained Service (LAS) [13,43,53] is a classical job scheduling algorithm that has been applied to packet scheduling [13], GPU cluster scheduling [34], and memory controller scheduling [42]. For $\alpha = 0$, Karma behaves similarly to LAS, and for $\alpha > 0$, Karma generalizes LAS with instantaneous guarantees. Moreover, our results from §3.3 establish strategy-proofness properties of LAS for dynamic user demands, which may be of independent interest.

**Theory works.** Several recent papers in the theory community study the problem of resource allocation for dynamic user demands. Freeman et al. [26] and Hossain et al. [37] consider dynamic demands under a different setting, where users can benefit when they are allocated resources above their demand; under this setting, they focus on instantaneous fairness (which is non-trivial since users can be allocated resources beyond their demand). Karma instead focuses on long-term fairness under the traditional model, where users do not benefit from resources beyond their demands. Sadok et al. [62] present minor improvements over max-min fairness for dynamic demands. Their mechanism allocates resources in a strategy-proof manner according to max-min fairness while marginally penalizing users with larger past allocations using a parameter $\delta \in [0,1)$. For both $\delta = 0$ and $\delta \to 1$, the penalty goes to 0 for every past allocation, and the mechanism becomes identical to max-min fairness; for other values of $\delta$, the penalty is at most a $\delta(1-\delta) \leq 1/4$ fraction of past allocation surplus, and it reduces exponentially with time (users who were allocated large amounts of resources further in the past receive an even smaller penalty). Thus, for all values of $\delta$, and in particular, for $\delta = 0$ and $\delta \to 1$, their mechanism suffers from the same problems as max-min fairness. Aleksandrov et al. [7] and Zeng et al. [82] consider dynamic demands, but in a significantly different setting than ours where resources arrive over time.

**Pricing- and credit-based resource allocation.** Another stream of work related to Karma is pricing-based and bidding-based mechanisms for resource allocation, *e.g.*, spot instance marketplace and virtual machine auctions [1,6,27,76,84,85]. While interesting, this line of work does not focus on fair resource allocation and is not applicable to use cases that Karma targets. XChange [75] proposes a market-based approach to fair resource allocation in multi-core architectures but focuses on instantaneous fairness rather than long-term fairness, unlike Karma. It assigns a "budget" of virtual currency to each user which can be used to bid for resources. This budget is however reset during every time quantum, and therefore information about past allocations is not carried over.

Credits are used in many other game theoretic contexts [25,51,61], *e.g.*, in peer-to-peer and cooperative caching settings to incentivize good behavior among participants with static demands [21,56,78]. However, we are not aware of any credit-based mechanisms that deal with resource allocation in the context of dynamic user demands.

## 7 Conclusion

This paper builds upon the observation that the classical max-min fairness algorithm for resource allocation loses one or more of its desirable properties—Pareto efficiency, strategy-proofness, and/or fairness—for the realistic case of dynamic user demands. We present Karma, a new resource allocation mechanism for dynamic user demands, and theoretically establish Karma guarantees related to Pareto efficiency, strategy-proofness, and fairness for dynamic user demands. Experimental evaluation of a realization of Karma in a multi-tenant elastic memory system demonstrates that Karma's theoretical properties translate well into practice: it reduces application-level performance disparity by as much as $2.4\times$ when compared to max-min fairness while maintaining high resource utilization and system-wide performance.

Karma opens several exciting avenues for future research. These include (but are not limited to) extending Karma theoretical analysis for $\alpha > 0$, generalizing Karma to allocate multiple resource types (similar to DRF), extending Karma to handle all-or-nothing or gang-scheduling constraints which are prevalent in the context of GPU resource allocation [15,47], and applying Karma to other use cases such as inter-datacenter network bandwidth allocation and resource allocation for burstable VMs in the cloud.

## Acknowledgements

We thank our shepherd, Sebastian Angel, and the OSDI reviewers for their insightful feedback. This research was supported in part by NSF CNS-1704742, CNS-2047220, CNS-2047283, CNS-2104292, CNS-2143868, AFOSR grants FA9550-19-1-0183, FA9550-23-1-0068, a NetApp Faculty Fellowship, an NDSEG fellowship, a Sloan fellowship, and gifts from Samsung, VMware, and Enfabrica.

## References

[1] Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[2] B-series Burstable Virtual Machine Sizes. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable.

[3] CFS: Completely Fair Process Scheduling in Linux. https://opensource.com/article/19/2/fair-scheduling-linux.

[4] Key Concepts and Definitions for Burstable Performance Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html.

[5] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting Wide-Area Network Topologies to Solve Flow Problems Quickly. In *NSDI*, 2021.

[6] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. Ginseng: Market-Driven Memory Allocation. In *VEE*, 2014.

[7] Martin Aleksandrov and Toby Walsh. Strategy-proofness, Envy-freeness and Pareto efficiency in Online Fair Division with Additive Utilities. In *PRICAI*, 2019.

[8] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, 2014.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *SIGMETRICS*, 2012.

[10] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.

[11] Jon CR Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. *Transactions on Networking*, 1997.

[12] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *NSDI*, 2020.

[13] Ernst W Biersack, Bianca Schroeder, and Guillaume Urvoy-Keller. Scheduling in practice. *Performance Evaluation Review*.

[14] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards $\mu$s Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *SIGCOMM*, 2022.

[15] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *EuroSys*, 2020.

[16] Yue Cheng, Ali Anwar, and Xuejing Duan. Analyzing Alibaba's Co-located Datacenter Workloads. In *Big Data*, 2018.

[17] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *NSDI*, 2016.

[18] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *NSDI*, 2016.

[19] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a Dynamic Multi-tenant Key-value Cache. In *ATC*, 2017.

[20] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[21] Landon P Cox and Brian D Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. *Operating Systems Review*, 2003.

[22] Alexander D'Amour, Hansa Srinivasan, James Atwood, Pallavi Baljekar, D Sculley, and Yoni Halpern. Fairness is Not Static: Deeper Understanding of Long Term Fairness via Simulation Studies. In *FAccT*, 2020.

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. 2008.

[24] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*, 1989.

[25] Joan Feigenbaum and Scott Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Current Trends in Theoretical Computer Science: The Challenge of the New Century Vol 1: Algorithms and Complexity Vol 2: Formal Models and Semantics*. 2004.

[26] Rupert Freeman, Seyed Majid Zahedi, Vincent Conitzer, and Benjamin C. Lee. Dynamic Proportional Sharing: A Game-Theoretic Approach. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[27] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-Driven LLC Allocation. In *ATC*, 2016.

[28] Yingqiang Ge, Shuchang Liu, Ruoyuan Gao, Yikun Xian, Yunqi Li, Xiangyu Zhao, Changhua Pei, Fei Sun, Junfeng Ge, Wenwu Ou, and Yongfeng Zhang. Towards Long-Term Fairness in Recommendation. In *WSDM*, 2021.

[29] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *SIGCOMM*, 2012.

[30] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*, 2011.

[31] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. In *SIGCOMM*, 2014.

[32] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-Resource Clusters. In *OSDI*, 2016.

[33] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *OSDI*, 2016.

[34] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.

[35] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[36] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

[37] Ridi Hossain. Sharing is Caring: Dynamic Mechanism for Shared Resource Ownership. In *AAMAS*, 2019.

[38] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for $\mu$s Latency and High Throughput. In *OSDI*, 2021.

[39] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.

[40] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[41] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In *EuroSys*, 2022.

[42] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

[43] Leonard Kleinrock. *Queueing Systems, Volume 1: Theory*. 1975.

[44] Haikun Liu and Bingsheng He. Reciprocal Resource Fairness: Towards Cooperative Multiple-Resource Fair Sharing in IaaS Clouds. In *SC*, 2014.

[45] Chengzi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Big Data*, 2017.

[46] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. Privacy Budget Scheduling. In *OSDI*, 2021.

[47] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *NSDI*, 2020.

[48] Paul E McKenney. Stochastic Fairness Queueing. In *INFOCOM*, 1990.

[49] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *SOSP*, 2021.

[50] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*, 2020.

[51] Noam Nisan and Amir Ronen. Algorithmic Mechanism Design. *Games and Economic Behavior*, 2001.

[52] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[53] Misja Nuyens and Adam Wierman. The Foreground–Background Queue: A Survey. *Performance Evaluation*, 2008.

[54] Abhay K Parekh and Robert G Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. *Transaction on Networking*, 1993.

[55] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler For Deep Learning Clusters. In *EuroSys*, 2018.

[56] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do Incentives Build Robustness in BitTorrent. In *NSDI*, 2007.

[57] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[58] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*, 2013.

[59] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. FairRide: Near-Optimal, Fair Cache Sharing. In *NSDI*, 2016.

[60] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *SoCC*, 2012.

[61] Tim Roughgarden. Algorithmic Game Theory. *Communications of the ACM*, 2010.

[62] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. Stateful DRF: Considering the Past in a Multi-Resource Allocation. *Transactions on Computers*, 2021.

[63] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. `https://arxiv.org/abs/2003.03423`.

[64] Alan Shieh, Srikanth Kandula, Albert G Greenberg, and Changhoon Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud*, 2010.

[65] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. In *SIGCOMM*, 1995.

[66] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *OSDI*, 2012.

[67] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *RTSS*, 1996.

[68] Shanjiang Tang, Bu-Sung Lee, Bingsheng He, and Haikun Liu. Long-Term Resource Fairness: Towards Economic Fairness on Pay-as-you-use Computing Systems. In *ICS*, 2014.

[69] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[70] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *EuroSys*, 2015.

[71] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Eva Tardos. Karma: Resource Allocation for Dynamic Demands. `https://arxiv.org/abs/2305.17222`.

[72] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an Elastic Query Engine on Disaggregated Storage. In *NSDI*, 2020.

[73] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, 2002.

[74] Carl A Waldspurger and William E Weihl. *Stride Scheduling: Deterministic Proportional Share Resource Management*. 1995.

[75] Xiaodong Wang and José F Martínez. XChange: A Market-Based Approach to Scalable Dynamic Multi-Resource Allocation in Multicore Architectures. In *HPCA*, 2015.

[76] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic Guarantees of Execution Duration for Amazon Spot Instances. In *SC*, 2017.

[77] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *SIGCOMM*, 2012.

[78] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative Caching with Return on Investment. In *MSST*, 2013.

[79] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *OSDI*, 2020.

[80] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[81] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[82] David Zeng and Alexandros Psomas. Fairness-Efficiency Tradeoffs in Dynamic Fair Division. In *EC*, 2020.

[83] Hui Zhang and Jon CR Bennett. WF2Q: Worst-Case Fair Weighted Fair Queueing. In *INFOCOM*, 1996.

[84] Liang Zheng, Carlee Joe-Wong, Christopher G Brinton, Chee Wei Tan, Sangtae Ha, and Mung Chiang. On the Viability of a Cloud Virtual Service Provider. In *SIGMETRICS*, 2016.

[85] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to Bid the Cloud. In *SIGCOMM*, 2015.