CS262 Report: Online Learning for Fair and Efficient Resource Allocation

Student: Wenshuo Guo
(Joint work with Romil Bhardwaj, Kirthevasan Kandasamy et al)

Abstract

Efficient resource allocation in distributed computing system and cloud computing has been an important and difficult task. For user satisfaction, these allocations are also expected to satisfy certain fairness criteria. Traditional systems for resource allocation in multi-tenant environments have either aimed at providing fairness (in data centers), relied on users to specify their resource requirements, or estimating the resource requirements via surrogate metrics (e.g. CPU utilization). These approaches fall short on what a user cares most: how well their job performs in the real world. In this project, we build Cilantro, a framework for performance-aware resource allocation in data centers and the cloud. At the core of Cilantro is an online learning mechanism which forms feedback loops with the job to estimate the resource to performance mappings. This relieves the users from the onerous task of profiling their jobs and collects more reliable real-time data. By translating performance requirements to resource demands, Cilantro can adopt a wide variety of performance-aware scheduling policies. In our evaluation on a 250-node cluster shared by 20 users, we find that three online learning policies implemented on Cilantro outperform seven baselines by $1.5-3\times$ with three different performance-aware evaluation criteria, while the utilities of some users increase by up to $4\times$.

1 Introduction

Efficient resource allocation in distributed computing system and cloud computing has been an important and difficult task for a resource manager.

Traditionally, scheduling policies have optimized on various objectives, such as providing fairness [13, 20], maximizing resource utilization [40], maximizing the amount of work done [20, 20], or minimizing queue lengths [1, 33]. However, these policies miss, or at best are imperfect proxies for what matters most to the users: the performance of their jobs in terms of actionable real-world metrics (e.g. P95 latency or throughput for a serving job, completion time for an analytic job). Since application (job) performance is crucial to user satisfaction, there has been extensive work in tracking and recording these metrics, giving rise to the popularity of monitoring tools such as Grafana, Prometheus and Datadog. Yet, resource allocation mechanisms are largely oblivious to them, instead requiring users to observe these metrics and determine how much resources to allocate for their jobs.

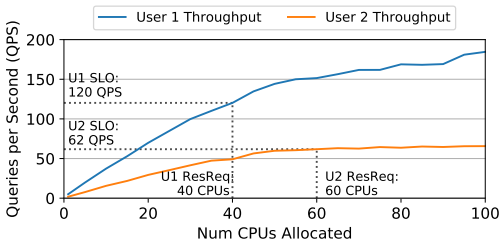


Figure 1. Two users serving TPC-DS benchmark queries with different resource-throughput mappings and performance goals (SLO). A user's ResReq is the amount of CPUs needed for her SLO. Fair equal allocation of resources causes user 2 to miss their SLO, while user 1 has already exceeded their SLO.

To illustrate the pitfalls of performance-oblivious scheduling, consider the example in Figure 1 where two users are sharing a cluster of 100 CPUs. They are running database queries from TPC-DS benchmark, and care about the throughput, i.e. the rate at which queries are processed. The jobs have different resource-to-throughput curves and service level objectives (SLO). The first user's SLO is 120 query-per-second (QPS) which requires 40 CPUs, and the second user's SLO is 62 QPS which requires 60 CPUs.

In this scenario, a performance-oblivious resource-based fair allocation policy will simply allocate 50 CPUs to each user. While this is resource-efficient, in that both jobs will be doing work, only the first user will achieve her SLO while the second one will not. Instead, if we allocate 40 CPUs to user 1 and 60 to user 2, they both achieve their SLOs. While this might seem unfair from a resource point of view, user 1 may not have complaints since her SLO is satisfied. In realistic situations, we often cannot satisfy the SLOs of all the jobs and the scheduler should decide how best to divide the resources among these competing jobs.

In this project, we introduce Cilantro, a performance-aware scheduling framework. At the core of Cilantro is an online learning mechanism which forms feedback loops with jobs to allocate resources and get performance feedback from jobs. A pool of independent learners analyze this feedback and learn increasingly accurate models of resource-performance

curves for each application. These models serve as a translation layer for scheduling policies which converts the performance goals to resource requirements and vice-versa.

We deployed the implementation of performance-aware policies in fixed clusters. We use Cilantro to implement strategy-proof policies which operate directly on the resource-performance curves (e.g. Kelly mechanism [28]) and policies which require resource-demand information (e.g. economic fair division methods [13, 20]). This is a marked departure from performance-oblivious resource-fairness policies which equally allocate resources, and those based on unreliable proxy metrics such as CPU utilization and queuing delays.

The main strength of Cilantro lies in the online nature of its learning mechanism. First, online learning methods are not bound by the amount of data used for learning - they can continually keep learning and improve over time. Second, since the learners learn from runtime data generated from live workloads, they can reliably account for real-world phenomenon, such as performance variability of servers [15]. Third, online learning reduces the burden on users since they are no longer required to manually estimate the resource demand and any updates to the workload are automatically accounted for by the learner.

1.1 Related Work

Large organizations typically have a pool of resources in the form of a cluster that must be shared among multiple users running different applications. In addition to sharing these resources, the organization usually also enforces resource allocation constraints in the form of policies which dictate constraints on the allocations.

Resource-sharing policies. In multi-tenant clusters, resource-sharing policies generally aim to achieve some notion of fairness. The majority of existing work in fair allocation fall into one of the two categories. One class of methods use *proportional fairness*, which simply divides the resource equally (or proportional to some weights) without accounting for resource requirements [24].

A second class of methods are *demand-based*, where each user specifies the amount of resources necessary to execute a unit of work for their job (e.g., based on resource requirements or other mechanisms like work-queue lengths), and then the mechanism allocates resources based on these submitted requests while accounting for their fair shares. Demers et al. [13] introduced the popular max-min fairness (MMF) procedure, subsequent work have applied the same principle for allocating resources in data centers. Some have also applied the MMF principle to design various queuing mechanisms such as lottery scheduling [39] and weighted fair queuing [6, 36]. Prior work also studied demand-based fairness for multi-resource environments. Examples include dominant resource fairness and its queuing variants [19, 20, 22] which extend MMF to multi-resource settings.

However, both proportional and demand-based fairness methods fail to account for application performance. As illustrated in Figure 1, fairness in allocating resources does not necessarily result in fairness of performance outcomes, where some users may obtain their SLOs while others may not. Moreover, in demand-based methods, users are forced to perform the translation of their performance requirements into resource requirements estimates, which is challenging due to the fluctuating loads and performance variability of both applications and servers.

Performance (utility)-based policies. Prior to the emergence of large scale cluster computing, a line of work in the networking literature has considered user performance in fair allocation. These policies typically work with a notion of *utility*, which is the value derived by the user from a given performance. The most popular utility-based policy class is the Kelly's model [28], which states that we should maximise the *social welfare*, i.e. (weighted) sum of user utilities, when determining allocations. An alternative approach to using utilities in fair allocation is to maximise the *egalitarian welfare*, i.e. (weighted) minimum of user utilities[1]. One recent work which directly considers a user's satisfaction-based utility under this framework is Minerva [32], who does so in a video streaming setting.

More recently, there have been works on using learning methods to optimise cluster performance but they do not consider users' utilities. Jockey [16] satisfies latency SLOs by modelling internal job dependencies to dynamically re-provision resources, but focuses doing so for a single job. Paragon [11] accounts for resource heterogeneity and interference among jobs to achieve performance guarantees. Quasar [12] prioritizes application performance in resource allocation and is similar to Cilantro in spirit, but requires offline profiling of jobs and has a fixed operator-centric policy that aims to maximize cluster utilization. Autopilot [34] performs application load and resource requirement prediction to minimize the amount of non utilized resources. But it does not account for actual application performance and instead uses surrogate metrics such as out-of-memory (OOM) rates to determine allocations. Morpheus [25] aims to mitigate performance unpredictability by defining SLOs and satisfying their resource demands by using models based on historical data. While Morpheus relies on resource reservation to re-provision resources, its automated SLO generation is synergistic with Cilantro towards achieving a zero-configuration cluster scheduler. Moreover, all these works highlight the

[1]This is different form max-min fairness (MMF) as described in [13] which allocates idle resources to other jobs, whereas egalitarian welfare can acquire utilized resources too.

need for automatically learning resource requirements since users tend to exaggerate their demands.

Cluster managers are performance-oblivious. To simplify resource coordination between users and to enforce policy-based constraints on resource sharing, cluster operators deploy cluster management frameworks such as Kubernetes [8], Mesos [23] and YARN [37]. All of these support common fairness policies, while also allow third party extensions to implement custom scheduling policies. To execute resource allocations from policies, Kubernetes and YARN use resource reservations while Mesos negotiates resource allocations through resource offers. However, in doing so, these frameworks do not provide any mechanisms for the policy to get an application's utility. Instead, they just focus on one-way allocations of resources and do not collect any feedback on the given allocation. This prevents policies from observing application performance, and policies are forced to make invasive changes to application logic to extract relevant metrics [40]. In the next section, we discuss how Cilantro overcomes this performance-obliviousness by designing and allowing policies to adjust resource allocations in response to application performance.

2 Cilantro Design

Cilantro is designed to support performance-aware resource allocation in a general manner. Cilantro incorporates performance-aware policies through online learning of the resource-performance mappings of jobs, while achieving generality by decoupling the scheduling policies from the resource allocation and job execution mechanisms of the framework.

Assumptions & terminology: In this work, we will focus on jobs which can scale elastically with the number of resources with corresponding gains in performance. Examples of such workloads include stateless or stateful distributed services (e.g web serving, prediction serving [10], memcached [18], Cassandra [29]), distributed computation (ML model training, ML hyperparameter tuning, MPI jobs) and distributed frameworks (e.g. Hadoop [35], Spark [41], Ray [30]). Some of these workloads can be viewed as a collection of several tasks whose job size may vary with time, such as in serving jobs where each task may refer to a query whose arrival rate may change with time. For such jobs, we will refer to the instantaneous rate of task arrival as the *load* (measured in queries per second (QPS)). We also assume that all resources are fungible.

Design overview. Figure 2 presents the architecture of Cilantro. The Cilantro framework is composed of two key components - the centralized Cilantro scheduler which is responsible for generating resource allocations, and the Cilantro clients, which fetch the job's performance state, compute local utility and send it the Cilantro scheduler.

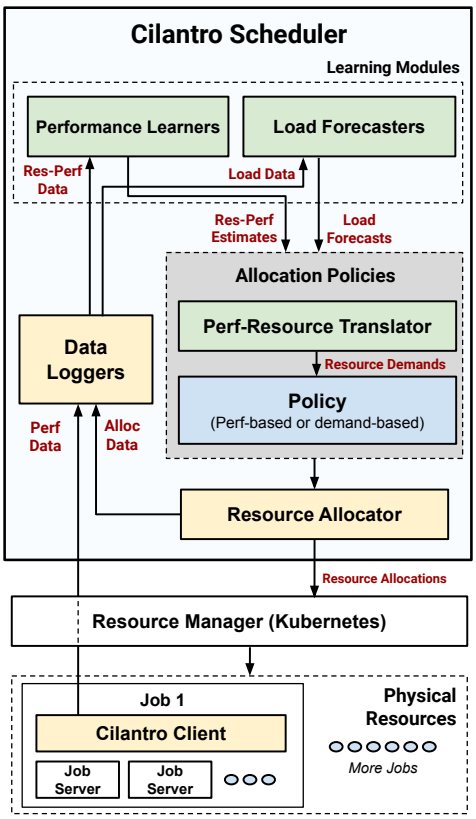


Figure 2. The Cilantro architecture. Arrows indicate data flow directions and red text highlights the objects flowing in.

2.1 Cilantro Scheduler

The Cilantro scheduler is designed as an asynchronous event driven system. Events trigger various modules in the system. Event sources include timers, performance updates received from the Cilantro clients, and cluster state updates from the underlying resource manager (e.g. Kubernetes in our implementation). We now describe different modules in the Cilantro scheduler.

1. Data loggers. Performance updates from Cilantro clients are stored in data loggers – memory-backed tables which are queried by the online learning mechanism. The Cilantro scheduler maintains a data logger per running application. The data logger offers a `submit-event` interface, which applications can use to push data, and a `get-data` interface, which the time series and performance learners can use to pull data. When the size of the data-logger exceeds a specified threshold, the table is spilled to disk to reduce memory footprint.

2. Performance learner and model. The goal of the performance learner is to learn an application's performance as a function of the amount of resources used and the load (if applicable) using an associated model. At regular periods, it polls the data logger for new data and updates the model. The

update frequency of a performance learner is constrained only by the speed at which the model can be updated. Like the data logger, one instance of a performer model is maintained per application. A performance learner provides a polling interface for policies; `get-perf-est`, which returns an estimate and confidence intervals for the performance as a function of the amount of resources and load.

In our implementations, we use the estimator described in [27] which bins the amount of resources into different bins and estimates the performance for each bin using the average. The bins are created adaptively such that more bins are created in a region if there is more data available in that region. For workloads with varying loads, we use the resources divided by the load as the input instead of directly using the amount of resources. While being simple, this estimator worked well on the workloads we stress-tested and has some desirable properties over conventional ML models. First, it is fast and light-weight, with `get-perf-est` typically taking only a few ms (see Table 2). Second, it is simple to impose monotonicity constraints on this estimator (e.g. more resources cannot hurt the performance). Third, its results are easy to interpret, which is useful if one wants to explain the allocations to the job's developer.

3. Load forecasters. In many real-world deployments, the job size could vary with time depending on the real-time traffic, which should be accounted for when allocating resources. The goal of the time series learner is to estimate this load for the duration of the allocation based on past observed loads. It offers a `get-load-est` interface for a policy which returns an estimate and confidence intervals for the future load. are regularly updated by polling from the data loggers.

In our current implementation, we use an auto-regressive integrated moving average (ARIMA) estimator which uses a moving average to estimate the load and correct for any linear trends. While quite rudimentary, we found that ARIMA worked sufficiently well in practice.

4. Performance-resource translator (PRT) and policies. While many performance-aware policies can directly utilize current application performance and load information to compute resource allocations, supporting performance-oblivious scheduling policies (such as max-min fairness) requires stating the resource demand of the job. To maintain the generality of Cilantro, we introduce a performance-resource translator (PRT), a thin layer which converts performance, load estimates and the job's performance goals into the resource demands for jobs.

Our implementation of PRT operates on a simple assumption that each task in the job requires the same amount of resources to complete. PRT exposes the `get-dem-est` interface, which, for a given performance goal and load, returns an estimate and confidence intervals for the resource demand (i.e. amount of resources required to meet the performance goal). PRT produces these estimates by querying the `get-perf-est` interface for the resource requirement to achieve the target performance. This value is scaled by the current load estimate received from `get-load-est` to get the resource demand required for satisfying the job's performance goal. We discuss the importance of returning confidence intervals in Section 3.3.

By polling the `get-dem-est` interface, the policy computes an allocation according to a pre-specified goal (see Section 3 for some goals and specific policies). It provides a `get-alloc` interface which, upon invocation, queries the performance-resource translation layer for estimates of resource demands and computes an allocation.

5. Resource allocator. The resource allocator is responsible for executing the resource allocations by interfacing with the underlying cluster manager, such as Kubernetes. This module is event driven: on receiving an allocation expiry event, it invokes the policy's `get-alloc` method and allocates the resources as specified by the policy. Allocation expiry events can be raised based on a time or when the load estimate for a running job changes significantly. However, in practice, the frequency of applying an allocation is limited by the agility of the environment. Since scaling jobs requires time, changing resource allocations too frequently can result in job thrashing (having to scale down before it has a chance to utilize new resources).

2.2 Cilantro client

The Cilantro client is a lightweight side-car container that is co-located with an associated job. The purpose of the Cilantro client is to poll the application to get its current performance, process it to obtain a standardized output, and publish it to the data loggers that are running on the Cilantro scheduler.

The primary challenge for Cilantro clients is to extract metrics from their assigned job. Many systems typically expose REST endpoints to query current system performance [3, 4], but often the applications also use monitoring tools such as Prometheus or Grafana. In such situations, the Cilantro client can directly query these services. Depending on the application, the performance metric extraction logic is specified by the users. If the Cilantro client has no visibility into the application metrics, we provide built-in fallback that can use surrogate metrics from the Kubernetes API, such as resource utilization, to report job performance. The user can also directly submit a resource demand via the client which will then be fed to the policy when determining allocations.

3 Policies

In this section, we describe our online learning policies for performance-aware resource allocation using Cilantro for

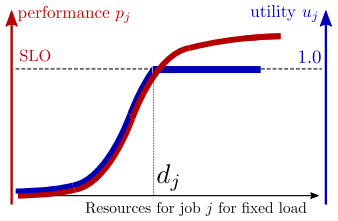


Figure 3. An SLO-aware utility (blue curve, y-axis on right) derived from the raw performance metric (red curve, y-axis on left). While we have used this form of utilities for our evaluation in this work, Cilantro can handle most utilities which are non-decreasing in the raw performance metric.

multiple competing jobs in a fixed cluster. We will first formally describe what we mean by performance.

Performance: The performance p_j of a job j refers to some raw metric of interest, which, say, can be obtained from a monitoring tool. We write the performance $p_j(a_j, \ell_j)$ as a function of the resources received a_j and the load ℓ_j. We will assume that for fixed loads, the performance p_j is (at least approximately) non-decreasing in the amount of resources received a_j, and for a fixed a_j, the performance is (at least approximately) non-increasing with the load. For example, in a serving job with a P95, 100 ms latency SLO, the load may refer to the arrival rate and performance may be the fraction of queries completed in under 100 ms.

In most cases, a compute cluster is over-subscribed and we need to decide how best to allocate the resources so that the overall goals of a group of users are achieved. In order to do this, we will first need to introduce the concept of utility, which is closely tied to a job's performance.

3.1 From raw performance metrics to utilities

Intuitively, the utility of a job is a practical value we derive due to its performance. The utility u_j is a non-decreasing function of the performance (higher performance does not hurt), and can be written as $u_j(a_j, \ell_j) = u'_i(p_j(a_j, \ell_j))$ for some non-decreasing function u'_i. One straightforward option is to simply set the utility to be equal to the performance, i.e. $u_j = p_j$. However, we might also choose a utility which captures diminishing returns of increased performance.

For our evaluations, where our primary focus is on long running jobs which have well-defined SLOs, we consider the following forms for the utilities. The maximum utility for any job is normalized as 1, which is achieved for any performance greater than or equal to the SLO; for performances below the SLO, the utility increases proportionally with the performance. We illustrate this in Figure 3.

We consider the above form for the utility due to the following reasons. *(i)* First, it is a meaningful notion of utility for workloads we consider in this work, with well defined SLOs. For instance, when one job is part of a larger service, the overall quality may be bottlenecked by other jobs and external factors; in such cases, there is little value derived in exceeding a carefully chosen SLO. *(ii)* SLOs feature prominently in service level agreements (SLAs) with external customers. Usually a service provider may have to pay a penalty if they violate the SLOs (e.g. [2]) and might lose credibility with their customer if they repeatedly do so. Using the SLOs as a basis for the utilities is a natural way to capture such effects. *(iii)* Third, for the purposes of our evaluation, it provides a simple way to compare the values derived from jobs with heterogeneous performance metrics and SLOs, such as those based on latencies, throughput, or completion time.

We wish to emphasize that the above choice of utility is not a fundamental limitation of Cilantro and is solely for the purpose of our evaluation. Provided that the utility is non-decreasing with performance, Cilantro can incorporate them into many allocation policies as we will discuss in Sec. 3.3.

3.2 (Oracular) policies with known utilities

Before we delve into the online learning policies which learn the performance curves on the fly, we will first review three common allocation methods when the performance curves (and hence the utilities) are known.

Notation: We will denote the number of jobs by n and the amount of resources by R. We will denote an allocation by $a = (a_1, \ldots, a_n)$, where a_j denotes the amount of resources allocated to job j. Note that the sum of allocations is always less than the total amount of resources, i.e. $\sum_{i=1}^{n} a_j \leq R$.

(i) Social welfare (a.k.a Kelly mechanism [28]): We choose the allocation a which maximizes the social welfare (or equivalently the average utility), i.e. $a = \text{argmax}\, W_S$, where,

$$W_S = \frac{1}{n} \sum_{j=1}^{n} u_j(a_j, \ell_j). \tag{1}$$

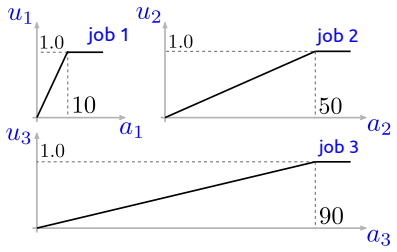
Typically, but not always, this allocates more resources to jobs which have a high utility-to-resource ratio, i.e. can generate large utility with a small amount of resources (Fig. 4).

(ii) Egalitarian welfare: Here, we choose the allocation a which maximizes the egalitarian welfare (minimum of all utilities), i.e. $a = \text{argmax}\, W_E$, where

$$W_E = \min_{j \in \{1,\ldots,n\}} u_j(a_j, \ell_j). \tag{2}$$

Typically, this allocates more resources to jobs which have a low utility-to-resource ratio (see Fig. 4).

(iii) No justified complaints (NJC) fair division [14, 22]: A naive form of ensuring fairness across all jobs is to simply allocate R/n resources per job. While this is fair, it can result in wasteful allocations; for instance, if a job j requires less than R/n resources, the excess resources would be either sitting idle or doing work that does not meaningfully increase the job's utility. Fair division mechanisms which adopt the



Policy	allocation (utility) job 1		job 3		job 3		metrics W_S	W_E	F_{NJC}
Soc. welf.	10	(1.0)	50	(1.0)	0	(0.0)	0.67	0.0	0.0
Egal. welf.	4	(0.4)	20	(0.4)	36	(0.4)	0.4	0.4	0.4
NJC fair.	10	(1.0)	25	(0.5)	25	(0.28)	0.59	0.28	1.0
Equal	20	(1.0)	20	(0.4)	20	(0.22)	0.54	0.22	1.0

Figure 4. Comparison of the (oracular) policies described in Sec. 3.2 via a synthetic example with 60 total resources. *Left:* Utility curves for three jobs. The y axis is the utility and the x-axis is the number of resources. For simplicity, we have ignored the loads and assumed that utilities increase linearly up to some resource demand for each job and do not increase thereafter. The total demand is 150, whereas only 60 resources are available; hence, an allocation policy should decide how best to divide the resources. *Right:* The allocations for each job and the corresponding utility in paranetheses for the social welfare maximizing policy (Soc. welf.), egalitarian welfare maximizing policy (Egal. welf.), the NJC fairness policy (NJC fair.) and when allocating the resources equally (Equal). We have also shown the W_S (1), W_E (2), and F_{NJC} (3) metrics for each policy. Different policies might be appropriate for different use cases. Cilantro provides tools to allocate according to many such policies including the above when the utilities are unknown a priori.

NJC principle allocate such excess resources to other jobs in order to achieve a Pareto-efficient allocation. While a job j may receive less than its fair share, it will have *no complaints*, since its ability to execute the job or the utility derived from it has not deteriorated. While there are many methods for NJC fair division, in this work, we adopt the max-min fairness procedure [13, 27], which takes excess resources from jobs whose demand is less than R/n and divides it equally among other jobs[2]. One additional advantage of this method is that it also has desirable strategy-proofness properties, i.e. users cannot manipulate this policy to obtain more resources and increase their utilities [20, 27]. We show that this strategy-proofness properties also translate to the online learning setting. For evaluating policies on fairness, we define the following NJC fairness metric:

$$F_{\text{NJC}} = \min_{j \in \{1,\ldots,n\}} \frac{u_j(a_j, \ell_j)}{u_j(R/n, \ell_j)} \tag{3}$$

The term inside the minimum can be interpreted as a fairness metric for job j as it measures the utility achieved relative to its fair share of R/n resources. In contrast to quantities such as the Jain's index, the above metric accounts for users' utilities when evaluating fairness. F_{NJC} metric has a maximum value of 1 and is trivially maximized by equally allocating R/n resources to each user. However, several economic fair division policies can also achieve a value of 1 while simultaneously achieving better utilities for all users.

As we have illustrated in Fig. 4, the above policies can yield very different allocations with different outcomes. The best policy might depend on the specific use case. Cilantro's goal is to allow a practitioner to choose an appropriate policy and provide an online learning mechanism to learn resource-performance curves while simultaneously allocating according to this policy. We discuss this further in Section 3.3.

[2]This is not to be conflated with maximizing the minimum utility, a.k.a egalitarian welfare. To avoid confusion, we will use the terms egalitarian welfare and NJC fairness throughout this manuscript.

3.3 Online learning policies

We now derive some online learning policies for Cilantro based on the above oracular policies, when the jobs' performance mappings (and hence utilities) and the load are unknown. Instead, we need to rely on the utility learners and time series learners for estimates (Fig. 2).

(i) Cilantro-SW: Here, we aim to design an online learning policy which resembles the social welfare policy in Sec 3.2. On each allocation round, we choose

$$a = \text{argmax} \sum_{i=1}^{n} \widehat{u}_j(a_j, \widehat{\ell}_i),$$

Above, $\widehat{\ell}_i$ is a 90% upper confidence bound for the load (measured in QPS) using the time series model. We found that the arrival rate of queries could vary significantly within an allocation round, and slightly over-estimating the load via an upper confidence bound, instead of directly using the estimate, performed well in practice. Next, $\widehat{u}_j$ is computed using $\widehat{u}_j(a_j, \widehat{\ell}_i) = u_j(\widehat{p}_i(a_j, \widehat{\ell}_i))$, where $\widehat{p}_i$ is an upper confidence bound on the performance curves obtained using our performance learners. While it is tempting to directly use an estimate, the stochastic bandit optimization literature suggests that doing so can fail spectacularly; instead, using a upper confidence bounds is optimal in online settings [7, 26]. Intuitively, using an upper confidence bound incentivizes a policy to explore other allocations instead of quickly coming to false conclusions with insufficient data.

(ii) Cilantro-EW: Here, we design an online learning policy which resembles the egalitarian welfare policy in Sec 3.2. On each allocation round, we choose

$$a = \text{argmax} \min_{i \in \{1,\ldots,n\}} \widehat{u}_j(a_j, \widehat{\ell}_i),$$

We use an upper confidence bound $\widehat{\ell}_i$ for the load and $\widehat{u}_j$ for the utility for similar reasons described above.

(iii) Cilantro-NJC: Unlike the two previous cases, in NJC fair sharing, there is no well-defined welfare function that we

can maximise to choose the next allocation. Here, we adopt the method proposed in [27] which is shown to be able to asymptotically generate fair and Pareto-efficient allocations. Intuitively, this method uses the upper and lower confidence bounds on the performance curves to estimate the amount of resources at which a job achieves its maximum utility (e.g. achieves its SLOs). Then it invokes the NJC fairness procedure to allocate resources based on this estimate.

For NJC fair division policies, we require the utility to be clipped so that we may define a *resource demand*, i.e. the point at which a user's utility does not increase further (d_j in Fig. 3). Applying an online learning version of a fair division policy in Cilantro is also straightforward: on each allocation round, we first estimate the resource demand via the `get-dem-est` interface of the performance learners and then invoke the corresponding policy using the estimated demands.

4 Experiments

The main highlights of our evaluation are:

1. The Cilantro-SW, Cilantro-EW, and Cilantro-NJC policies, which do not start with any prior data, are competitive with oracular policies which have access to jobs' resource to performance mappings obtained after several hours of profiling. Moreover, they outperform 7 other baselines on relevant metrics.
2. We qualitatively compare the three allocation paradigms discussed in Section **??**, via Cilantro's online learning framework. While Cilantro's goal is to enable a wide variety of policies, we find that NJC fair division mechanisms provide an excellent trade-off between the overall performance and fairness. Moreover, they are strategy-proof which disincentivises manipulative users from trying to get more resources for themselves.
3. We show that Cilantro's learning mechanism is lightweight, making it suitable for real-time scheduling.

4.1 Implementation & Workloads Details

The Cilantro Scheduler and the Cilantro client are implemented in 7600 lines of python code. The scheduler runs as a standalone scheduler for Kubernetes. Any Kubernetes pods created with the `scheduler=cilantro` label are forwarded by the kubernetes control plane to the Cilantro scheduler which assigns a node to the pending pod.

Resource reallocation events are triggered by a timer-based event, which is rasied every 2 minutes in our experiments. This window was chosen based on the fact that Kubernetes pods could be created and destroyed in 5-15 seconds. A 2 minute allocation window is long enough for the pod to reach its steady state that performance metrics from the job would be reliable while at the same time frequent enough to adapt to changes in the load and learned performances.

To execute updated resource allocations received from policies, we horizontally scale the workloads by adding more replicas to their kubernetes deployment. Newly created pods then rely on the kubernetes service discovery mechanism to connect to the workload's ring of servers. The workload is responsible for updating local resource count and load balancing queries onto the new servers.

We implement the cilantro client by sharing disk volumes between the cilantro client and the workload. The workload peridiocally (every 2s) writes STDOUT to logs in the shared volume, which are then read by the cilantro client and published to the scheduler over gRPC. These gRPC messages also act as heartbeats to inform liveness to the scheduler.

Workloads: We evaluate Cilantro on three classes of workloads – database querying, prediction serving and machine learning training – which are used to create multiple jobs. We have shown the resource-to-performance mappings of all workloads in Fig. 5, which were obtained by profiling each job for 4 hours in the presence of other jobs in cluster. Queries to the database querying and prediction serving workloads are periodically dispatched by a trace-driven workload generator. We use the Twitter Sample API[5] to collect a scaled trace of tweet arrival rate at Twitter over a period of 24 hours at Twitter's Asia datacenters. To bring the trace at parity with our experiment cluster, we subsample the arrival rate by a factor of 10. The workload generator parses this trace and dispatches a query at every arrival.

For the ML training workload, we draw queries from an essentially infinite pool to create a constant stream of work. All these workloads are made to report their performance to the cilantro client every 10 seconds. Tasks which are running when the performance is reported are discarded to avoid affecting results for next reporting round.

Database querying: We use the TPC-DS [31] benchmark suite as the workload backed by replicated instances of sqlite3 database. Each job's database is populated with the TPC-DS data generator with the scale factor parameter set to 100. The TPC-DS suite consists of 99 query templates out of which 27 were not compatible with the sqlite dialect and were discarded. Of the remaining workloads, we considered those that had an average completion time of under 300 ms on an Amazon m5xlarge instance. We created two workloads using this subset of queries: DB-0, which had queries that completed in under 100 ms and DB-1 which had queries that had a completion time between 100 and 300 ms. When a query is requested, we randomly pick a relevant query from this database and dispatch it according to the trace. The performance metric of interest is query latency.

Prediction serving: In prediction serving (e.g. [10]), a job processes arriving queries to output a prediction, usually obtained via a machine learning model. In our set up, we

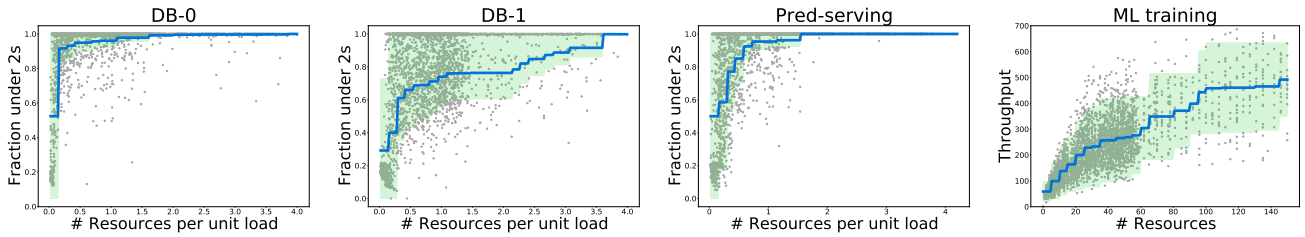


Figure 5. The resource to performance mappings obtained via the adaptive binning estimator (see Sec. 2) for the four workloads after approximately 6 hours of profiling. The blue curve is the average performance value. Shaded in blue is the 95% (2-σ) confidence region for the average performance value and shaded in green is the 95% region for the noisy performance value. For the latency-based workloads (DB-0, DB-1, and prediction serving), we have shown the number of resources per unit load (arrival QPS) in the x-axis and the fraction of queries completed under 2s on the y-axis. For the ML training workload, we have shown the number of resources on the x axis the amount of data processed per second on the y-axis. To obtain accurate estimates, when profiling, we sampled low resources allocation values more densely.

use a random forest regressor and the the news popularity dataset [17]. Half of the dataset is used to train the model (before the experiment), while the remaining half (test set) is used to generate queries for the workload. Queries are issued in batches of 4, which are picked randomly from the test set. The metric of interest here is the serving latency.

ML training: Many real world machine learning deployments require that we continuously update the model from an input stream of data so as to account for changes in the data distribution [40]. In our set up, we use a neural network, with four hidden layers of size 64 each. We train our model using the naval propulsion [9] dataset using stochastic gradient descent (SGD). Each task in this workload consists of training a batch of 16 points for 100 iterations. The performance metric of interest here is throughput, or the number of samples processed per second.

4.2 Fixed cluster experiments

Experimental set up: We use a cluster of 1000 CPUs composed of 250 AWS m5.xlarge instances which have 4 vCPUs each. The Cilantro scheduler runs on its own dedicated m5.xlarge instance. We use the above 4 workloads to create 20 jobs as follows: 3 DB-0 users with P90, P90, and P95 latency SLOs of 2s; 7 DB-1 users with P90, P90, P95, P95, P95, P99, P99 latency SLOs of 2s; 3 prediction serving users with P90, P90, and P95 latency SLOs of 2s; 7 machine learning training users with throughput SLOs of 400, 400, 450, 450, 500, 500, and 500 QPS. The estimated total amount of resources based on the median demand was 1637 CPUs; hence, even at full capacity, not all users can satisfy their SLOs. We evaluate all baselines for 4 hours.

4.2.1 Baselines

Oracular policies: We implement the following oracular policies which have access to the performance curves, obtained by exhaustively profiling each workload for at least 4 hours.

1) Oracle-SW, 2) Oracle-EW, 3) Oracle-NJC: The oracular policies for maximizing the Nash welfare, maximizing the egalitarian welfare, and the max-min fairness procedure for NJC fairness respectively as described in Sec. 3.2. For Oracle-SW and Oracle-EW, on each round, we maximize the welfare computed via the profiled performance curves using an evolutionary algorithm. For Oracle-NJC, we compute the demands using the profiled performance curves

Cilantro policies: The following three policies use Cilantro's online learning framework as described in Sec. 3.3.

4) Cilantro-SW, 5) Cilantro-EW, 6) Cilantro-NJC: For Cilantro-SW and Cilantro-EW, on each round, we maximize an upper confidence bound (UCB) on the welfare computed via the UCBs on the performance curves using an evolutionary algorithm. Since the UCBs are available in memory analytically this can be done quickly.

Other heuristics: We implement the following methods for maximizing the social/egalitarian welfare, or for fair sharing. While they are not based directly off specific prior work, such methods are common in the scheduling literature [10, 21].

7) EvoAlg-SW, 8) EvoAlg-EW: We design evolutionary algorithms to maximize the social and egalitarian welfare. On each allocation round, it stochastically samples one of the past allocations so that those with higher welfare are more likley to be sampled. It then perturbs the sampled point slightly to obtain a new allocation.

9) Greedy-EW: This method starts by allocating an equal amount of resources to all jobs. On each round, it evaluates the utility achieved by each job during the previous allocation round. Then it takes away one resource each from the top half of the users who had high utility

10) Resource-Fair: This method simply allocates an equal amount of resources to all jobs.

Baselines from prior work: To the best of our knowledge, there are no prior methods for the above criteria, especially when the resource-performance mappings are unknown. Therefore, we adapt the following methods from prior work.

Policy	Social Welfare, (W_S)	Egalitarian Welfare (W_E)	NJC Fairness (F_{NJC})	Useful resource usage
Oracle-SW	0.939 ± 0.003	0.315 ± 0.010	0.740 ± 0.005	0.961 ± 0.002
Oracle-EW	0.825 ± 0.004	0.572 ± 0.011	0.706 ± 0.002	0.997 ± 0.000
Oracle-NJC	0.860 ± 0.002	0.482 ± 0.010	1.000 ± 0.000	0.991 ± 0.000
Cilantro-SW	**0.910 ± 0.002**	0.325 ± 0.011	0.739 ± 0.010	0.872 ± 0.005
Cilantro-EW	0.811 ± 0.004	**0.573 ± 0.012**	0.740 ± 0.010	0.886 ± 0.003
Cilantro-NJC	0.826 ± 0.002	0.492 ± 0.005	**0.982 ± 0.006** *	**0.954 ± 0.003**
EvoAlg-SW	0.807 ± 0.004	0.209 ± 0.009	0.768 ± 0.023	0.761 ± 0.006
EvoAlg-EW	0.801 ± 0.003	0.220 ± 0.008	0.747 ± 0.028	0.777 ± 0.005
Resource-Fair	0.667 ± 0.002	0.201 ± 0.004	**1.000 ± 0.000** *	0.764 ± 0.001
Greedy-EW	0.754 ± 0.005	0.287 ± 0.003	0.90735 ± 0.021	0.807 ± 0.003
Ernest	0.753 ± 0.003	0.192 ± 0.006	0.865 ± 0.012	0.752 ± 0.001
Quasar	0.833 ± 0.002	0.344 ± 0.007	0.747 ± 0.006	0.810 ± 0.003
Minerva	0.733 ± 0.030	0.246 ± 0.031	0.609 ± 0.002	0.418 ± 0.067

Table 1. The social welfare (1), egalitarian welfare (2), NJC fairness metric (3), and the effective resource usage (4) for all 13 methods. Higher is better for all four metrics, and the maximum and minimum possible values for all metrics are 1 and 0. The values shown in bold have achieve the highest value for the specific metric, besides the oracular policies. (*) Resource-Fair has NJC fairness $F_{NJC} = 1$ by definition.

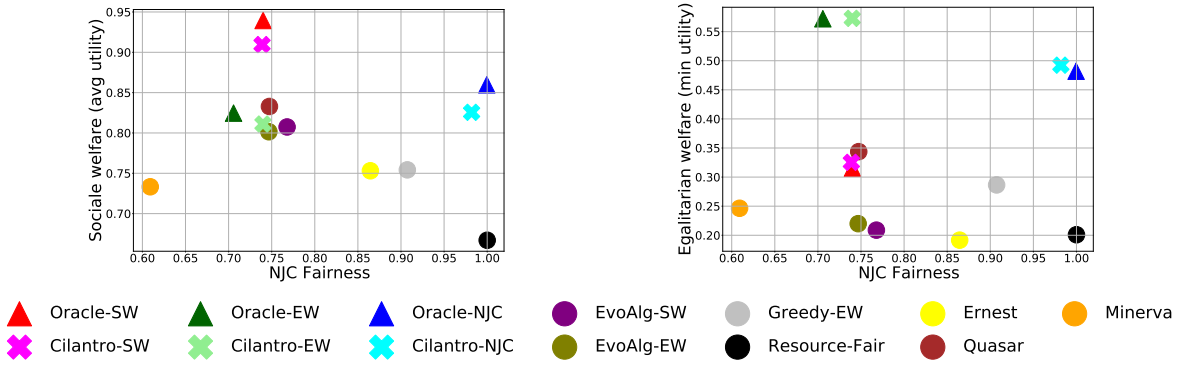


Figure 6. An illustration of the results in Table 1 where we have compared the NJC fairness metric achieved by all policies against the social and egalitarian welfare. Higher is better for all metrics so methods closer to the top right corner do well on balance.

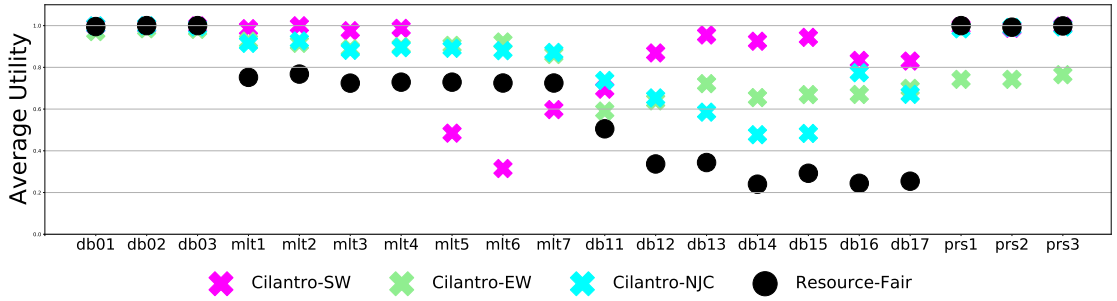


Figure 7. The average utility achieved by the 20 jobs across our experiment for the three online learning methods in Cilantro and Resource-Fair. Here, `db0x`, `mltx`, `db1x`, and `prsx` refers to jobs using the DB-0, ML training, DB-1, and prediction serving workloads (Fig. 5).

Updating model	`get-alloc` for Cilantro-SW	`get-alloc` for Cilantro-EW	`get-alloc` for Cilantro-NJC
0.0413 ± 0.0048	2.8823 ± 0.3155	2.1239 ± 0.0212	0.8132 ± 0.162

Table 2. Avg time taken by Cilantro to update the performance model, and for computing a new allocation for each of the three policies.

11) Ernest [38]: Ernest uses a featurized linear model to estimate the time taken to run a job as a function of the load amount of resources allocated. Equipped with this estimate, we approximate the resource demand for a job, i.e. the amount of resources required, to satisfy a given target SLO for a given load. On each round, we use the estimated demand as inputs to the max-min fairness procedure to compute allocations for all jobs.

12) Quasar [12]: Quasar uses collaborative filtering to estimate the amount of resources required to achieve a given SLO. We use this estimated deand in max-min fairness to compute the allocation for all jobs. The authors also describe procedures for vertical scaling and co-locating workloads based on interference and heterogeneity. We do not incorporate these methods in order to be consistent across all methods and since our experimental platform does not support extracting these metrics.

13) Minerva [32]: Minerva, which was proposed for video streaming, stipulates that we set the allocation for job j at each step to be proportional to a_j/u_j where a_j and u_j are the allocation and utility at the previous round. The authors also propose several video-streaming specific optimizations in addition to this core algorithm which are not applicable in our setting.

4.3 Results & Discussion

Evaluation on social welfare, egalitarian welfare, NJC fairness and resource usage: In our first experiment in this section, we compare all baselines above on the social welfare (1), egalitarian welfare (2), and the NJC fairness criteria (2). Additionally, we also evaluate on the following the following metric which measures the useful resource usage.

$$\text{Useful resource usage} = \sum_{j=1}^{m} \min(a_j, d_j) \tag{4}$$

Here, the "resource demand" d_j is the value of the resource at which the utility is clipped (see Fig. 3). This measures how much *useful* work is being done by the cluster.

Table 1 summarizes the results for all 13 methods on all 4 criteria above, where we have reported the average value of the metrics from start to finish of the experiment. Fig. 7 visually illustrates these results via fairness vs welfare plots. While the oracular methods perform best on their respective metrics, we find that the online learning policies in Cilantro come close to matching them. Resource-Fair achieves a perfect NJC score by definition, but performs poorly on all other metrics as it is performance oblivious. Among other baselines, Quasar performs better than Ernest or Minerva, although we should emphasize that none of these methods were designed to optimize for these metrics. It is worth pointing out that Cilantro-NJC achieves an favourable trade-off between all criteria considered here, making it an attractive option, especially in instances when the overall goals of resource allocation cannot be explicitly stated.

To delve deeper into the various trade-offs involved with the three paradigms, we show the individual utilities achieved by these three policies in Fig. 7. We see that both the social and egalitarian welfare policies result in some users being worse off than receing their fair allocation of $1000/20 = 50$ CPUs. This results in a fairness violation. In contrast, in Cilantro-NJC, users are at most marginally worse off than their fair share. However, for most users, they achieve a higher utility than their fair share. Cilantro-NJC is able to learn the resource demands of all users, and is able to achieve a fair and efficient allocation by transferring resources from users whose fair share is in excess of their demand to the rest. We also see that Cilantro-EW tried maximize the egalitarian welfare by taking resources away from those who achieve high utility and giving it to those who do not, while Cilantro-SW tried to maximize the social welfare by allocating more resources to jobs that can quickly achieve high utility.

Cilantro Overheads: Table 2 evaluates the time taken for Cilantro to process the feedback and compute the allocations for the three policies. This shows that Cilantro is fairly light-weight. For comparison, the average time it took to de-allocate a kubernetes pod and assign it to a different job was on the order of 5-10s depending on the workload. Cilantro-SW and Cilantro-EW are slightly more expensive since finding the optimum social/egalitarian welfare requires running an evolutionary algorithm to optimize the upper confidence bound.

5 Discussions & Future Directions

In this project, we have designed and implemented Cilantro for performance-aware resource allocation in clusters. Our main ideas are: *(i)* cluster resource allocation policies should be performance-aware; *(ii)* since user's resource to performance mappings are difficult and burdensom to estimate via offline profiling, they should be done in an online fashion. We designed Cilantro under this tenets which enables the realization of several performance-aware policies, including, but not limited to social/egalitarian welfare maximization and NJC fair division policies. Cilantro's policies are competitive with oracular polices which know workload characteristics a priori and outperforms other baselines in data center and cloud settings. Empirically, we demonstrated that it outperforms other fair allocation methods.

For future work, we would like to apply Cilantro to different tasks, such as autoscaling on the cloud (i.e., increasing the size of the cluster by paying for additional resources) under a performance constraint while minimizing costs. to meet the performance goals of running applications. It is also test Cilantro with a larger number of various resources.

References

[1] [n. d.].

[2] [n. d.]. https://aws.amazon.com/compute/sla/. ([n. d.]). AmazonComputeServiceLevelAgreement

[3] [n. d.]. Kubernetes API health endpoints | Kubernetes. https://kubernetes.io/docs/reference/using-api/health-checks/. ([n. d.]). (Accessed on 10/09/2021).

[4] [n. d.]. Ray Dashboard — Ray v1.7.0. https://docs.ray.io/en/latest/ray-dashboard.html. ([n. d.]). (Accessed on 10/09/2021).

[5] [n. d.]. Twitter Streaming API. ([n. d.]). https://developer.twitter.com

[6] Jon CR Bennett and Hui Zhang. 1996. WF/sup 2/Q: worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*, Vol. 1. IEEE, 120–128.

[7] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvari. 2010. X-armed Bandits. *arXiv preprint arXiv:1001.4475* (2010).

[8] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *ACM Queue* 14 (2016), 70–93. http://queue.acm.org/detail.cfm?id=2898444

[9] Andrea Coraddu, Luca Oneto, Aessandro Ghio, Stefano Savio, Davide Anguita, and Massimo Figari. 2016. Machine learning approaches for improving condition-based maintenance of naval propulsion plants. *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment* 230, 1 (2016), 136–153.

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 613–627.

[11] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.

[13] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review* 19, 4 (1989), 1–12.

[14] Danny Dolev, Dror G Feitelson, Joseph Y Halpern, Raz Kupferman, and Nathan Linial. 2012. No justified complaints: On fair sharing of multiple resources. In *proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. 68–75.

[15] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 99–112. https://doi.org/10.1145/2168836.2168847

[16] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 99–112. https://doi.org/10.1145/2168836.2168847

[17] Kelwin Fernandes, Pedro Vinagre, and Paulo Cortez. 2015. A proactive intelligent decision support system for predicting the popularity of online news. In *Portuguese Conference on Artificial Intelligence*.

[18] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 124 (2004).

[19] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-resource Fair Queueing for Packet Processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 1–12.

[20] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.. In *Nsdi*, Vol. 11. 24–24.

[21] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 65–80.

[22] Avital Gutman and Noam Nisan. 2012. Fair allocation without trade. *arXiv preprint arXiv:1204.4286* (2012).

[23] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*, Vol. 11. 22–22.

[24] Junchen Jiang, Vyas Sekar, and Hui Zhang. 2012. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 97–108.

[25] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 117–134.

[26] Kirthevasan Kandasamy, Jeff Schneider, and Barnabás Póczos. 2015. High dimensional Bayesian optimisation and bandits via additive models. In *International conference on machine learning*. PMLR, 295–304.

[27] Kirthevasan Kandasamy, Gur-Eyal Sela, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. 2020. Online Learning Demands in Max-min Fairness. *arXiv preprint arXiv:2012.08648* (2020).

[28] Frank P Kelly, Aman K Maulloo, and David KH Tan. 1998. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society* 49, 3 (1998), 237–252.

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[31] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 1049–1058.

[32] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. 2019. End-to-end transport for video QoE fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*. 408–423.

[33] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[34] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[35] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[36] Dimitrios Stiliadis and Anujan Varma. 1998. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM*

Transactions on networking 6, 5 (1998), 611–624.

[37] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

[38] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 363–378.

[39] Carl A Waldspurger and William E Weihl. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. 1–es.

[40] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 595–610.

[41] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.