# TR #2019-06: Fair Scheduling for Deadline Driven, Resource-Constrained Multi-Analytics Workloads

Stratos Dimopoulos, Chandra Krintz, Rich Wolski

Department of Computer Science
University of California, Santa Barbara
{stratos,ckrintz,rich}@cs.ucsb.edu

***Abstract*—In this paper, we analyze and empirically evaluate *Justice*, a new approach to fair-share, deadline-aware job scheduling for resource-constrained cloud deployments managed by big data resource negotiators. *Justice* provides admission control, which leverages historical traces and job deadlines to guide and adapt resource allocation decisions to changing workload conditions. We evaluate *Justice* using different deadline types and production analytics workloads. We find that it outperforms extant allocators in terms of fair allocation, deadline satisfaction, and useful work.**

***Keywords*—cloud for IoT; scheduling; big-data; fairness;**

## I. Introduction

Increasingly, cloud users deploy big data frameworks (e.g., Apache Hadoop and Spark) via resource negotiators such as Apache Mesos and YARN. Resource negotiators simplify deployment and enable multiple frameworks to execute concurrently using the same set of resources. They employ fair-share resource allocators [6, 8], which attempt to partition resources equally across frameworks in these multi-analytics settings.

In this paper, we investigate fair-share allocation for workloads with deadline and resource constraints. Deadline-driven workloads represent an important class of big data applications [12, 15, 20], which are unfortunately under supported in multi-analytic settings. Resource-limited deployments are those in which more resources (CPU, memory, local disk) cannot simply be added on-demand, in exchange for an additional charge, as they can in a public cloud. Such deployments include private clouds and IoT edge systems in which data analytics is performed near where data is collected to provide low-latency (deadline-driven) actuation, control, data privacy, decision support, and to reduce bandwidth requirements [19]. Because modern resource negotiators and big data frameworks were not designed for this combination of constraints, their use can result in low utilization, poor performance, missed deadlines, and unfair sharing [3].

To address these limitations, we design and implement admission control for resource negotiators that satisfies deadlines while preserving fairness. Our system, called *Justice*, uses historical job analysis and deadline information to assign the minimal fraction of resources required to meet a job's deadline. *Justice* estimates this fraction from a running tabulation of an expansion factor that it computes from an on-line, post-mortem analysis of all previous jobs executed.

We compare *Justice* to the baseline "fair" allocator employed by Mesos and YARN, to a simple extension of this allocator, and to an "oracle" allocator, which knows the exact minimum number of resources required for each job to meet its deadline. The metrics we use to do this comparison are fairness and equality [11] (we use separate formulas to measure how "fair" and how "equally" resources are shared), deadline satisfaction, productivity, wasted time and utilization. To evaluate our work, we use two large production workload traces from an industry partner that provides commercial big-data services using YARN.

Our results show that *Justice* performs similarly to the oracle in terms of fairness, deadline satisfaction, and effective use of resources. In addition, *Justice* performs significantly better than the Mesos and YARN allocators. This is because these resource negotiators attempt to preserve fairness without considering resource demand, which impedes performance when resources are constrained. We also find that *Justice* achieves greater productivity, wastes fewer resources, and has significantly better system utilization than its counter-parts for the workloads, deployment sizes, and deadlines that we consider.

## II. Background and Related Work

In resource-constrained deployments encountered in the private cloud or the IoT, the fair-share policies [6, 8] fail to preserve fairness [3, 13]. Also, current fair allocators are deadline-agnostic. The kind of fairness they try to preserve is based on the resources requested by the job submitters or their given quota and they ignore the actual demand of the job in order to meet its deadline. Moreover, to satisfy deadlines cluster administrators have to add extra resources, violate fairness by manually prioritizing one group of users over another, use solutions similar to a capacity scheduler [1], or require users to reserve resources in advance [2, 17]. Such solutions are costly, inefficient, or impractical for resource constrained clusters as they further limit peak cluster capacity.

Moreover, in these multi-analytic settings, the intra-job allocators of frameworks like Hadoop and Spark greedily occupy resources allocated to them even if they are not using them [3, 9, 22]. Authors in [9] attempt to address this issue by exploiting task-level resource requirements information and DAG dependencies. In contrast, *Justice* builds on the the admission control mechanism introduced in PYTHIA [5] without requiring job-repetitions and task-level information. Moreover, *Justice* adapts to changing cluster conditions to avoid over-provisioning and preserves "true fair-sharing" in addition to satisfying deadlines.

**Algorithm 1** *Justice* TRACK_JOB Algorithm

```
function TRACK_JOB(requestedTasks, deadline, compTime, numCPUsAllocd, success)
    deadlineCPUs = compTime/deadline
    maxCPUs = min(requestedTasks, cluster_capacity)
    minReqRate = deadlineCPUs/maxCPUs
    minReqRateList.add(minReqRate)
    MinCPUFrac = min(minReqRateList)
    MaxCPUFrac = max(minReqRateList)
    LastCPUFrac = numCPUsAllocd/maxCPUs
    LastSuccess = success
    fractionErrorList.append(minReqRate − LastCPUFrac)
end function
```

Much related work [7, 18, 24] focuses on building job performance profiles and scalability models offline or exploits historic and runtime information [10, 14, 20, 23, 24]. These approaches are not suitable for resource constrained, multi-analytics settings. Sampling, simulations, and extensive monitoring, impose overheads and additional cost. Also, trace analysis [4, 7] shows that some production workloads have small ratio of repeated jobs and these jobs have often large execution times dispersion. Therefore, approaches based solely on past executions cannot predict with high statistical confidence for ad-hoc jobs or jobs that are not frequently repeated. Lastly, most of these approaches require task-level information, for the specific framework they target (e.g., Hadoop [10, 14, 23, 24] or Spark [21]) and for this reason they cannot be integrated into resource managers.

*Justice* focuses on preserving fairness while satisfying job deadlines without utilizing additional cluster resources or increasing scheduling latency. It is suitable for resource negotiators like Mesos and YARN to manage batch applications with deadlines on clusters with constrained resources. To achieve this, it employs black-box, framework-agnostic prediction to estimate the minimum number of CPUs a job needs to meet its deadline "just-in-time". It proactively applies admission control by dropping jobs that cannot make their deadlines and improves its predictions based on its historic performance. This paper presents a comprehensive analysis on the algorithm that powers *Justice*[4], introduces a new fairness metric to distinguish "equality" from "true fairness", and provides an extensive empirical evaluation versus competitive approaches on a significantly larger production workload.

## III. Justice Algorithm Analysis

*Justice* can be conceptually separated into two main operations that it performs in parallel. The job tracking operation as described in Algorithm 1 and the admission control and resource allocation operation as described in Algorithm 2. Admission control and resource allocation depends on the online statistical model the tracking operation builds to estimate the resources jobs need to satisfy their deadlines (Function `alloc_resources` in Algorithm 2) and to correct these estimations based on the Kalman filter mechanism described in Algorithm 3.

*Justice* performs these operations without delaying job scheduling as it can run on the background to calculate the new allocation fractions similarly to other window-based fairness algorithms. Its memory requirements scale linearly to the points used on the desired history window and, depending

**Algorithm 2** Admission Control and Resource Allocation

```
function ADMISSION_CONTROL(RequesterJob)
    for all j ∈ SubmittedJobs do
        Feasible = True, TTD = Deadline − ElapsedTime
        reqCpus = ESTIMATE_REQ(j, TTD)
        if reqCpus > min(taskCount, capacity) then
            Feasible = False
        end if
        if Share(j) < reqCpus then
            if Feasible == True then
                priority = reqCpus/TTD, ADD2HEAP(priority, j)
            else
                DROP_JOB(j)
            end if
        end if
    end for
    allocations = ALLOC_RESOURCES(heap)
    if RequesterJob ∉ allocations then
        Add RequesterJob to queue
    end if
end function

function ESTIMATE_REQ(Job)
    maxCpus = min(tasks, capacity), reqCpus = maxCpus
    if CompletedJobs > 1 then
        fraction = CALCULATE_ALLOC_FRACTION()
        fraction = CORRECT_ALLOC_FRACTION(fraction)
        fraction = (deadline/(deadline − queue)) ∗ fraction
        reqCpus = max(ceil(fraction ∗ maxCpus), 1)
    end if
    return reqCpus
end function

function ALLOC_RESOURCES(heap)
    offers = CREATE_OFFERS(heap)
    allocations = SEND_OFFERS(offers)
    return allocations
end function

function CREATE_OFFERS(heap)
    while availableCpus > 0 and heap not empty do
        for all Job j ∈ heap do
            offer = min(request(j), availableCpus)
            if offer < request(j) then
                offer = 0
            else
                availableCpus− = offer
                offersDict[j] = offer
            end if
        end for
    end while
    return offersDict
end function
```

**Algorithm 3** Allocation Calculation and Correction

```
function CALCULATE_ALLOC_FRACTION
    if LastSuccess then
        CPUFraq = MinCPUFrac
    else
        CPUFraq = MaxCPUFrac
    end if
    fraction = (LastCPUFrac + CPUFraq)/2
    return fraction
end function
function CORRECT_ALLOC_FRACTION(fraction)
    correction = CALC_SMOOTHED_AVG(fractionErrorList))
    correctedFraction = fraction + correction
    correctedFraction = VALIDATE_FRACTION(correctedFraction)
    return correctedFraction
end function
```

on the desired scheduling latency and estimation accuracy requirements, a limited amount of metrics could be stored in memory and the rest in database.

### A. Job Metrics Tracking

*Justice* invokes Algorithm 1 every time a job completes its execution. The algorithm takes as inputs a number of metrics provided by the user at submission time and a number of metrics extracted by the job's execution profile that is available in system logs after the job completes. In return, the algorithm produces a number of metrics that are used as inputs to the functions of Algorithm 2 and Algorithm 3. The inputs of the algorithm are the number of tasks a job has (`requestedTasks`), its `deadline` as defined in seconds by the job submitter, the total computation time of the job (`compTime`) expressed in `CPU*Seconds`, the number of CPUs allocated to the job (`numCPUsAllocd`), and a boolean `success` indicating whether the job was successful (completed its job before its deadline) or failed (exceeded its deadline or got dropped before completing).

Based on these five inputs, the algorithm derives a number of intermediate metrics for all jobs in order to produce its final results. These metrics are, the minimum number of CPUs a job would have needed to finish by its deadline (`deadlineCPUs`), the maximum parallelism of the job (`maxCPUS`), the minimum required rate (`minReqRate`) which is the fraction of resources the job would have needed compared to its maximum resources, in order to meet its deadline just in time. These rates are stored for all job encountered in the system (`minReqRateList`).

After producing these derived metrics, the algorithm calculates the desired results. These are `MinCPUFrac` and `MaxCPUFrac`, which correspond to the minimum and maximum request rates encountered across all jobs respectively and the `LastCPUFrac`, which is the last given rate observed for the job that completed and triggered the algorithm. It also stores whether the last job completed successful `LastSuccess`. Lastly, a historic fraction error is produced across all jobs `fractionErrorList` as the difference between the minimum required rate the job would have needed to meet its deadline and the fraction of resources *Justice* assigned to it. Note that `deadlineCPUs` cannot be greater than `maxCPU` (assuming feasible deadlines) and `MinCPUFrac` and `MaxCPUFrac` are always less or equal to 1.

### B. Resource Estimation, Admission Control and Allocation

After a bootstrapping period in which the job tracking operation runs without any admission control in order to gather enough data and produce estimations with statistical significance, the admission control and resource allocation mechanism of *Justice*'s kick in. Algorithm 2 takes as input a submitted job (`RequesterJob`), and based on the metrics that the job tracking operation continuously produces and stores, as discussed on the previous section, it creates an allocation for the job. This allocation should be sufficient to meet its deadline just in time or if, based on the cluster conditions, it estimates that it is impossible for the job to complete before its deadline, then, it drops the job.

*1) Resource Estimation:* To achieve this, it updates the deadlines for jobs in the queue, reducing each by the time that has passed since submission (line 3 in Algorithm 2). Then, it estimates the minimum amount of resources the job requires to meets its deadline (Function `estimate_req` in Algorithm 3). That happens by a subsequent call to Function `calc_alloc_fract` in Algorithm 3 that computes the CPU allocation fraction (`fraction`) for each newly submitted job as the average of the `LastCPUFrac` and either `MinCPUFrac` or `MaxCPUFrac`, as shown in Algorithm 3, depending on whether the last completed job met or missed its deadline, respectively. In other words, consecutive successes make *Justice* more aggressive, causing it to allocate smaller resource fractions (i.e., `fraction` converges to `MinCPUFrac`), while deadline violations make *Justice* more conservative, causing it to increase the fraction in an attempt to prevent future violations (`fraction` converges to `MaxCPUFrac`).

The fraction produced by Function `calc_alloc_fract` is further improved by a Kalman filter mechanism (Function `admission_control`. Every time a job completes its execution, *Justice* tracks the estimation error and uses it to correct the CPU allocation fraction. Estimation error is the difference between the allocation fraction and the ideal minimum fraction (`deadlineCPUs`). *Justice* calculates a weighted average of the historical errors (Function `correct_alloc_fraction`) and adds it to the allocation fraction. *Justice* can be configured to assign the same weights to all past errors or to use exponential smoothing (i.e., to weigh recent values higher than those that occurred in the distant past). Lastly, a validate function (that we do not include on the algorithm for brevity) ensures that the corrected fraction remains within allowable limits (no less than the minimum observed `MinCPUFrac` or greater than 1).

After *Justice* computes, corrects, and validates `alocCPUFrac`, it considers the time that the job has spent in the queue (line 26 in Function `estimate_req` of Algorithm 2). *Justice* multiplies `alocCPUFrac` by the number of tasks requested on job submission and uses this value as the number of CPUs to assign to the job (Function `estimate_req` in Algorithm 2).

*2) Admission Control:* *Justice* recomputes the CPU allocation of each enqueued job and, as part of its admission control policy, it either drops the ones with infeasible deadlines or keeps those that cannot be admitted but are still feasible (lines 12 and 18 respectively in Algorithm 2). *Justice* implements a *proactive* admission control mechanism to prevent jobs likely to miss their deadline from entering the system and consuming resources wastefully. This way, *Justice* attempts to maximize the number of jobs that meet their deadline even under severe resource constraints. *Justice* also tracks jobs that violate their deadlines and selectively drops some of them to avoid further waste of resources. It is selective in that it terminates jobs when their `requestedTasks` exceed a configurable threshold. Thus, it still able to collect statistics on "misses" to improve its estimations by letting the smaller violating jobs complete their execution while at the same time it prevents the bigger violators from wasting resources.

The priority policy *Justice* uses is pluggable. In the evaluations of this paper we use a policy that aims to minimize the number of jobs that miss their deadlines. For this policy (line 10 in Algorithm 2), *Justice* prioritizes jobs with a small number of tasks and greatest time-to-deadline (`TTD`). However, all of the policies that we considered (including shortest time-to-deadline) perform similarly. Once *Justice* has selected a job for admission, it allocates the CPUs to the job and admits it to the system for execution. Once a job run commences, its CPU allocation does not change.

*3) Resource Allocation:* Finally, *Justice* allocates the calculated resources to jobs (Function `alloc_resources` in Algorithm 2) by creating offers according to job priorities. Function `offer_resources` creates offers for jobs until there are no other jobs to be scheduled or the available resources are exhausted. Lastly, *Justice* sends these offers to the frameworks (line 33 in Algorithm 2 - we omit Function `SEND_OFFERS` for brevity.

## IV. EXPERIMENTAL METHODOLOGY

We compare *Justice* against the fair-share allocator implemented in open-source resource negotiators like Mesos and YARN, using trace-based simulation on industry-provided production traces. We refer to this allocator as `Baseline FS`. This allocator lacks any deadline information and therefore continues executing a job even after its deadline is exceeded. We also implement an extension of this allocator named `Reactive FS`, that enforces the same FS policy but reactively terminates a job that has exceeded its deadline. Lastly, we implement an "oracle" allocator that predicts the exact amount of resources a job requires to meets its deadline without having, however, knowledge of the optimal schedule in terms of deadline satisfaction or fairness.

We evaluate the robustness of our approach by running experiments using deadline formulations from prior works [7, 16, 20] and variations on them. In particular, we assign deadlines that are multiples of the optimal execution time of a job as we extract it from our workload traces. We use two types of multiples: Fixed and variable.

`Fixed Deadlines`: With fixed deadlines, we use a deadline that is a multiple of the optimal execution time as described in [16]. Each deadline is expressed as $D_i = x \cdot T_i$, where $T_i$ is the optimal runtime of the job and $x >= 1.0$ is some fixed multiplicative expansion factor. In our experiments, we use constant factors of $x = 1$ and $x = 2$, which we refer to as *Fixed1x* and *Fixed2x* respectively.

`Variable Deadlines`: For variable deadlines, we compute deadline multiples by sampling distributions. *Jockey* deadlines pick randomly a factor $x$ from two possible values as described in [7]. In this work, we use the intervals from the sets with values $(1, 2)$ and $(2, 4)$ to choose $x$ and, again, compute $D_i = x \cdot T_i$, where $T_i$ is the minimum possible execution time. We refer to this variable deadline formulations as *Jockey1x2x* and *Jockey2x4x*. *90loose* are a variation of the Jockey1x2x deadlines, in which the deadlines take on the larger value (i.e. are loose) with a higher probability (0.9) while the other uses the smaller value. *Aria* deadlines are uniformly distributed in the intervals $[1, 3]$ and $[2, 4]$ as described in [20]; we refer to these deadlines as *Aria1x3x* and *Aria2x4x*, respectively.

## V. RESULTS

We evaluate *Justice* in terms of fairness, deadline satisfaction, and effective resource utilization, using two production traces gathered over a 3-month period, for different resource-constrained cloud deployments (number of CPUs). We compare *Justice* against fair share schedulers and an oracle using multiple deadline strategies: a fixed multiple (Fixed), a random multiple (Jockey), a uniform multiple (Aria) of the actual computation time, and mixed loose and strict deadlines (90loose), as described on Section IV.

### A. Fairness Evaluation

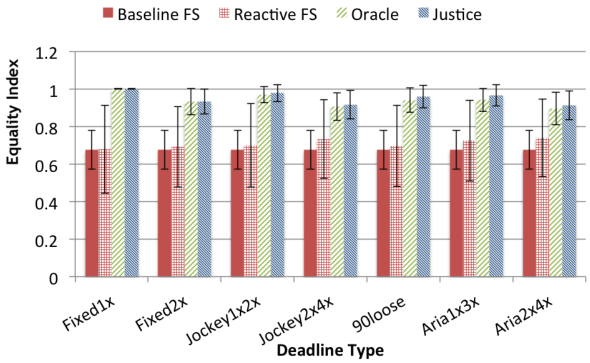
Traditional fair-share allocators [6, 8] attempt to give an "equal" share of resources to concurrently executing jobs regardless of whether this share is sufficient to meet their deadlines. Herein, we will refer to this form of fairness as "equality". To evaluate, the degree to which these allocators achieve this goal in resource-constrained settings, we use Jain's fairness index $\frac{|\sum_{i=1}^{n} F_i|^2}{n * \sum_{i=1}^{n} F_i^2}$ with $F_i$ corresponding to the resource allocation of each job $i$.

To compute equality, we classify jobs based on their maximum demand. We then calculate the index for each job and the weighted average across indexes. Weights correspond to the number of jobs in each class (e.g., all jobs with demand of Y CPUs). We classify jobs in this way to avoid considering "unfair" (or "unequal") allocations that correspond to different maximum demand classes because a job cannot be allocated more CPUs than it demands.

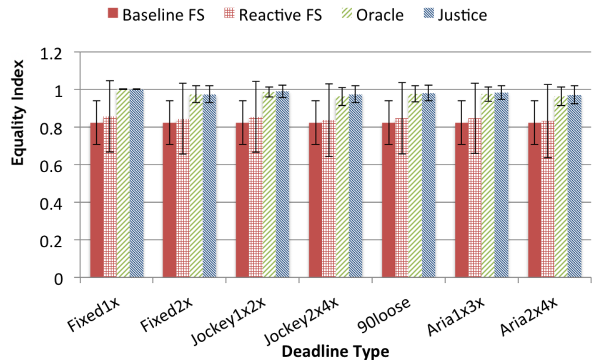
The top graphs of Figure 1 present equality results across all allocators and cluster capacities that we consider. The fairness index is averaged over 60-sec intervals on the lifetime of the workload. *Justice* achieves better fairness scores than the fair-share allocators by up to 23% and 17% for the two capacities. Even though the goal of *Justice* is not to preserve equality but instead to prioritize for what we consider actual fairness, it performs better than the fair-share allocators for two reasons. First, *Justice* keeps the system less utilized and therefore fewer jobs wait in the queue, which contributes negatively to equality. Second, due to constrained resources, *Justice* drops large jobs more frequently which provides opportunities for it to facilitate fairness at a finer grain across frameworks.

We argue that "equality" is not the desired property for deadline-driven workloads. Equality treats all jobs similarly regardless of their actual resource requirements. In practice, jobs have different priorities, max demands, and diverse deadline tightness. Instead, "true" fairness can be measured by using Jain's fairness index with $F_i$ corresponding to the fraction of demand of each job $i$. For each job $i$, among $n$ total jobs, we define the fraction of demand as $F_i = \frac{A_i}{D_i}$ where $D_i$ is the resource request for job $i$ and $A_i$ is the allocation given to job $i$. When $A_i >= D_i$ the fraction is defined to be 1.

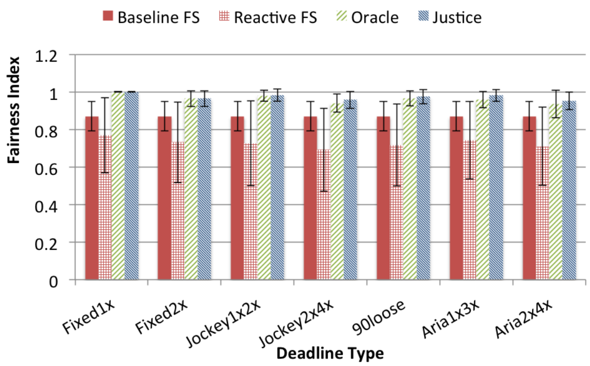
By using this "true" fairness metric, *Justice* outperforms all "equality" allocators we evaluate in this study in clusters with constrained resources. It achieves this by applying admission control instead of greedily allocating resources to jobs and by predicting the amount of resources jobs require to meet their deadlines "just in time". In contrast, the existing fair-share allocators cannot be fair under constrained resources as



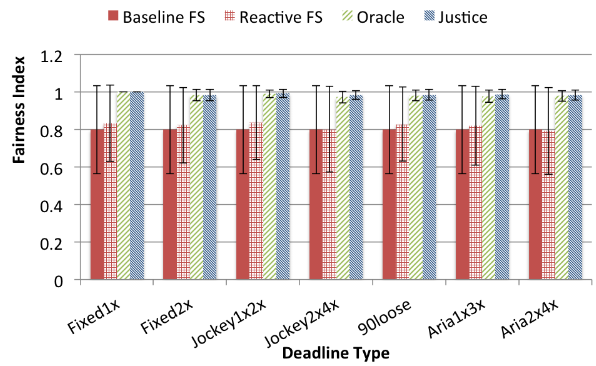

(a) Equality Index with 2500 CPUs




(b) Equality Index with 5000 CPUs




(c) Fairness Index with 2500 CPUs




(d) Fairness Index with 5000 CPUs

Fig. 1: **Equality Vs Fairness:** Average of Jain's fairness index adapted for equality (top graphs) and fairness (bottom graphs) with highly constrained capacities (left graphs) and moderately constrained capacities (right graphs). Experiments denoted as "Fixed" have deadlines multiples of 1 and 2. Experiments denoted as "Jockey" have multiples picked randomly from a set with two values (1, 2) and (2, 4). Experiments denoted as "90loose" have 90% deadlines with a multiple of 2 and 10% deadlines with a multiple of 1. Experiments denoted as "Aria" have multiples drawn from uniformly distributed intervals [1, 3] and [2, 4]

they cannot prevent larger jobs from taking over a significant portion of the cluster [3, 9, 22].

In addition, *Justice* outperforms the oracle for variable deadlines. This is an artifact of the oracle's use of maximum job demand in the formula instead of the minimum required resources. Under this definition, our oracle is not a fairness oracle in terms of preserving fairness globally on the system. It is instead an oracle with respect to the minimum resource requirements needed to satisfy each job's deadline.

### B. Deadline Satisfaction

Being just fair in deadline-driven workloads is not enough. The main goal of a resource allocator in such settings is to satisfy deadlines. To investigate this, we compute the *Satisfied Deadline Ratio (SDR)* as the fraction of the jobs that complete before their deadline over the total number of submitted jobs.

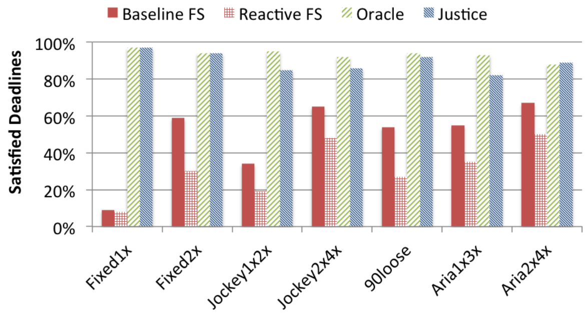
Figures 2a and 2b show that fair-share allocators, fail to satisfy job deadlines as they lack deadline information and assign resources solely based on what they consider as "fair". In contrast, *Justice* builds a statistical model based on previous job executions and assigns to them the amount of resources they need to satisfy their deadlines "just-in-time". As a result, *Justice* is able to achieve 80% of optimal allocation. Note that even the oracle doesn't achieve a perfect SDR ratio, because, as previously mentioned, it does not have knowledge of the perfect global schedule. Therefore, it also has to drop jobs that cannot achieve their deadlines under the current cluster conditions.

### C. Effective Resource Usage and Cluster Utilization

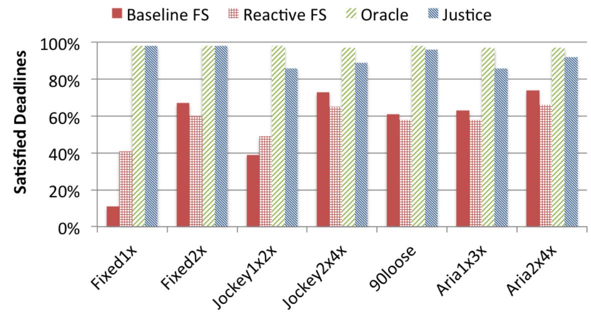
We next evaluate the resource allocators using three metrics that in combination show how effectively each utilizes cluster resources. For the set of submitted jobs $J_1, J_2, ..., J_n$ and their corresponding runtimes $T_1, T_2, ..., T_n$, we consider the subset of $m < n$ successful jobs $J_1, J_2, ..., J_m$ and the subset of $k < n$ failed or dropped jobs $J_1, J_2, ..., J_k$ where $n = m + k$.

We define Productive Time Ratio (PTR) as $\frac{\sum_{i=1}^{m} T_i}{\sum_{j=1}^{n} T_j}$ and Wasted Time Ratio WTR as $\frac{\sum_{i=1}^{k} T_i}{\sum_{j=1}^{n} T_j}$. Lastly, cluster utilization is $\frac{busy}{idle+busy}$ where $busy$ is the total busy time and $idle$ is the total idle time across a workload.

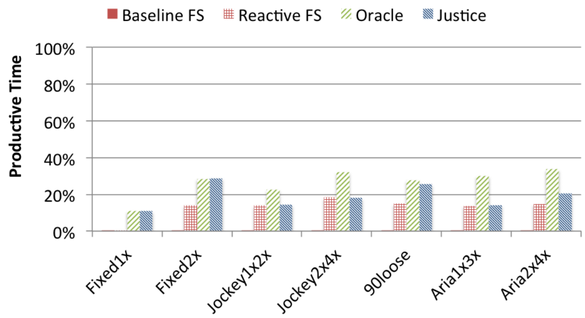
Under severe resource constraints (left graphs on Figure 2) *Justice* spends more time productively (i.e. has a higher PTR) and wastes fewer resources (i.e. has a lower WTR). It does so by dropping jobs that are likely to violate their deadlines according to its predictions. In contrast, fair-share



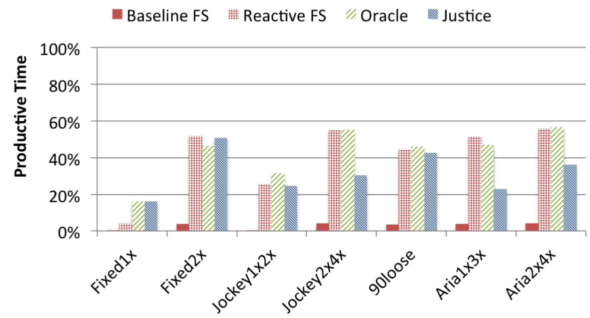

(a) Satisfied Deadlines with 2500 CPUs



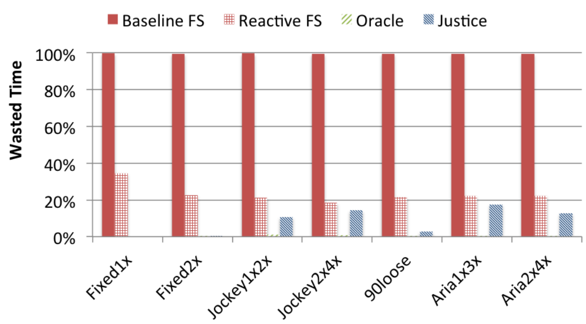

(b) Satisfied Deadlines with 5000 CPUs



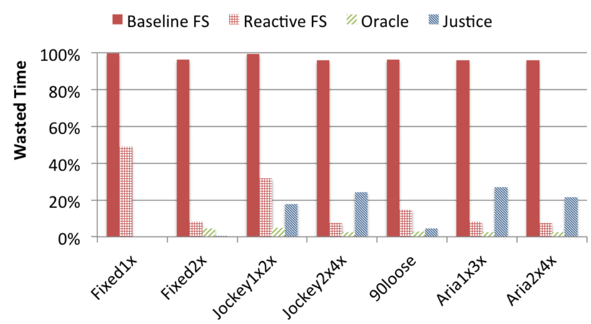

(c) Productive Time with 2500 CPUs



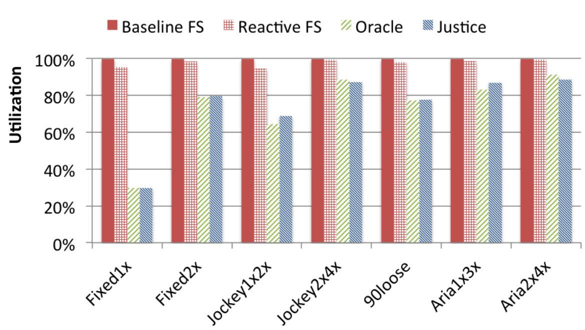

(d) Productive Time with 5000 CPUs



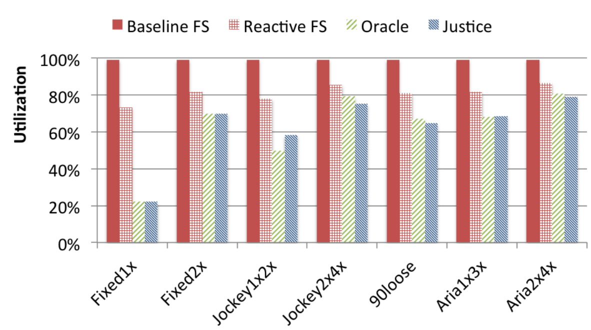

(e) Wasted Time with 2500 CPUs




(f) Wasted Time with 5000 CPUs




(g) Utilization with 2500 CPUs




(h) Utilization with 5000 CPUs

Fig. 2: **Deadline Satisfaction and Efficient Resource Utilization:** Satisfied Deadlines Ratio (SDR), Productive Time Ratio (PTR), Wasted Time Ratio (WTR), and cluster utilization with highly constrained cluster capacities (left graphs) and moderately constrained capacities (right graphs) for different deadline types.

policies attempt to share resources equally between smaller and bigger jobs. When resources are constrained, this share is insufficient for the bigger jobs to meet their deadlines. Moreover, Baseline FS wastes time on jobs that have already missed their deadlines, while Reactive FS avoids doing so by retroactively dropping such jobs.

More importantly, *Justice* outperforms the existing fair-share allocators in all these metrics, while at the same time it satisfies more deadlines and achieves better fairness than the fair-share allocators as previously discussed in Sections V-B and V-A. This means that the smaller utilization is not a by-product of added overhead but the result of effective admission control that filters out jobs that would not satisfy their deadlines under these constrained cluster resources. It also reveals an opportunity for running more small jobs through back filling, an option we want to explore in future work.

The right side of Figure 2 shows that fair-share policies might be more suitable for optimizing productive work for clusters for which resource scarcity is not severe. In such conditions, and in combination with higher deadline variability, *Justice* might sacrifice (deny admission) to some bigger jobs in order to preserve fairness and to satisfy deadlines for other (smaller) jobs. This effect is depicted both in a smaller PTR value (Figure 2d) and lower cluster utilization (Figure 2h).

## VI. CONCLUSIONS

*Justice* tracks job deadlines and runtime information to automatically adapt resource allocation and admission control mechanisms for big data frameworks. By doing so, it is able to achieve fairness and satisfy deadlines even when resource availability is scarce (e.g. as in private clouds and edge, fog, or IoT computing settings). We analyze the different parts of the *Justice* algorithm and empirically evaluate it using trace-based simulation of deadline-driven, production YARN workloads using resource-constrained clusters. We compare *Justice* to the existing fair-share allocator that ships with Mesos and YARN and find that *Justice* is able to achieve better "traditional" (equality) as well as "true" fairness, deadline satisfaction, and better resource utilization for resource-constrained clouds.

This work is supported by NSF (CNS-1703560, CCF-1539586, ACI-1541215) and AWS Cloud Credits for Research.

## REFERENCES

[1] *YARN Capacity Scheduler*. https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[2] C. Curino et al. Reservation-based Scheduling: If You're Late Don't Blame Us! In: *ACM Symposium on Cloud Computing*. 2014, pp. 1–14.

[3] S. Dimopoulos, C. Krintz, and R. Wolski. Big Data Framework Interference In Restricted Private Cloud Settings. In: *IEEE International Conference on Big Data. IEEE*. 2016.

[4] S. Dimopoulos, C. Krintz, and R. Wolski. Justice: A Deadline-aware, Fair-share Resource Allocator for Implementing Multi-analytics. In: *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE. 2017, pp. 233–244.

[5] S. Dimopoulos, C. Krintz, and R. Wolski. PYTHIA: Admission Control for Multi-Framework, Deadline-Driven, Big Data Workloads. In: *International Conference on Cloud Computing*. IEEE. 2017.

[6] *YARN Fair Scheduler*. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[7] A. D. Ferguson et al. Jockey: guaranteed job latency in data parallel clusters. In: *ACM European Conference on Computer Systems*. ACM. 2012, pp. 99–112.

[8] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resource types. In: *NSDI*. 2011.

[9] R. Grandl et al. Altruistic scheduling in multi-resource clusters. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2016.

[10] Z. Huang et al. RUSH: A RobUst ScHeduler to Manage Uncertain Completion-Times in Shared Clouds. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 242–251.

[11] R. Jain, D.-M. Chiu, and W. R. Hawe. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Vol. 38. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.

[12] K. Kc and K. Anyanwu. Scheduling hadoop jobs to meet deadlines. In: *International Conference on Cloud Computing*. 2010, pp. 388–392.

[13] J. Khamse-Ashari et al. An efficient and fair multi-resource allocation mechanism for heterogeneous servers. In: *IEEE Transactions on Parallel and Distributed Systems* 29.12 (2018), pp. 2686–2699.

[14] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In: *ACM International Conference on Autonomic Computing*. 2012, pp. 63–72.

[15] S. Li et al. WOHA: deadline-aware map-reduce workflow scheduling framework over hadoop clusters. In: *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE. 2014, pp. 93–103.

[16] J. Liu, H. Shen, and H. S. Narman. CCRP: Customized Cooperative Resource Provisioning for High Resource Utilization in Clouds. In: *IEEE International Conference on Big Data. IEEE*. 2016.

[17] A. Tumanov et al. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In: *European Conference on Computer Systems*. 2016, p. 35.

[18] S. Venkataraman et al. Ernest: efficient performance prediction for large-scale advanced analytics. In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016, pp. 363–378.

[19] T. Verbelen et al. Cloudlets: bringing the cloud to the mobile user. In: *ACM workshop on Mobile cloud computing and services*. ACM. 2012.

[20] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In: *ACM International Conference on Autonomic Computing*. 2011, pp. 235–244.

[21] K. Wang and M. M. H. Khan. Performance Prediction for Apache Spark Platform. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*. IEEE. 2015, pp. 166–173.

[22] Y. Yao et al. Admission control in YARN clusters based on dynamic resource reservation. In: *IEEE International Symposium on Integrated Network Management*. 2015, pp. 838–841.

[23] N. Zaheilas and V. Kalogeraki. Real-time scheduling of skewed mapreduce jobs in heterogeneous environments. In: *11th International Conference on Autonomic Computing (ICAC 14)*. 2014, pp. 189–200.

[24] W. Zhang et al. Mimp: Deadline and interference aware scheduling of hadoop virtual machines. In: *IEEE Cluster, Cloud and Grid Computing*. 2014, pp. 394–403.