# Extended Tech Report #2019-09: 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

**Abstract**—We present a new approach to fair-share, deadline-aware job scheduling for resource-limited cloud deployments that are managed by "big data" framework resource negotiators (e.g. YARN and Mesos), called *Justice*. *Justice* provides admission control that leverages historical traces and job deadline information 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, among other metrics.

**Keywords**—scheduling; analytics; resource-constraints; deadlines;

## 1 Introduction

Increasingly, cloud users deploy "big data" frameworks (e.g. Apache Hadoop [1] and Apache Spark [2]) via resource negotiators such as Apache Mesos [21] and YARN [40]. Resource negotiators simplify deployment and enable multiple frameworks to execute concurrently using the same set of resources. Resource negotiators employ fair-share resource allocators [14, 17], 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 [27, 29, 44], 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, and decision support, and to reduce bandwidth requirements [13, 42]. 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, unfair sharing, and long job turn-around times for multi-analytics workloads [9].

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. Further, *Justice* "risks" running some jobs with greater or fewer resources so that it can continuously adapt its admission control to changing workload characteristics.

We compare *Justice* to the baseline allocator employed by Mesos [21] and YARN [40], 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 [24] (we use separate formulas to measure how "equally" and how "fair" resources are shared), deadline satisfaction, productivity, 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[1]. The original jobs in these traces were not resource constrained nor did they require completion according to individual deadlines. For these reasons we use discrete-event, trace-driven simulation to represent how these workloads execute with significantly fewer compute resources using different deadline formulations from related work [15, 30, 43, 44, 51].

Our results show that for both traces, *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 allocator. We also find that *Justice* achieves greater productivity, wastes less resources, and has significantly better system utilization than its counter-parts for the workloads, deployment sizes, and deadlines that we consider.

1. The partner wishes to remain anonymous for reasons of commercial competitiveness.

## 2 JUSTICE

*Justice* is a resource allocator with admission control for resource negotiators that manage big data frameworks (Hadoop and Spark, among others) in multi-analytics settings. *Justice* is unique in that it supports deadline-driven workloads and attempts to provide fair sharing of resources, deadline satisfaction, and high productivity (completed useful work), regardless of deployment size (resource constraints). To enable this, *Justice* employs a black-box, framework-agnostic prediction technique to estimate the minimum number of CPUs that a job requires to meet its deadline.

Prior work shows that because fair-share allocators assume "infinite" resources available in a cloud they fail to preserve fairness when resources are limited [9, 19, 48]. This disadvantage is due in part to the use of greedy allocation (required because of their inability by the allocator to predict future demand) and the lack of corrective mechanisms (ex: job preemption or dropping). *Justice* addresses these drawbacks by proactively adapting to future demand and resource availability through its admission control mechanisms.

Existing fair-share allocators also do not support job deadline constraints. Instead, they assume that the result of a job submitted by a user has value regardless of how large the turn-around time may be. For *Justice* we assume that each job is submitted with a "maximum execution time" parameter that tells the resource negotiator when the completion of each job is no longer valuable. Henceforth, we refer to this parameter as the "deadline" for the job. The only assumption we make is that the deadline is feasible, i.e., there is an optimal allocation that is sufficient to complete the job before its deadline. Currently, cloud administrators statically divide resources with capacity schedulers [5], or require users to reserve resources in advance [7, 39] to create differentiated service classes with respect to turn-around time. Such approaches are inefficient and impractical when resources are limited, as they further restrict peak capacity. In contrast, *Justice* incorporates deadline information to drive its resource allocation, admission-control, and job dropping decisions.

### 2.1 Resource Allocation

To determine how many CPUs to allocate to a new job, *Justice* uses the execution time data from previously executed jobs. *Justice* analyzes each completed job and uses this information to estimate the minimum number of CPUs that the job *would have needed* to have finished by its deadline (represented as the `deadlineCPUs` variable in Algorithm 1). *Justice* assumes that this minimum required capacity utilizes perfect parallelism and that the number of tasks for a job is the maximum parallelization possible. We refer to this number as the `requestedTasks` for the job. Therefore, the maximum number of CPUs that can be assigned to any job (`maxCPUs`) at any given time is the minimum between the `requestedTasks` and the total deployment size (`cluster_capacity`).

To bootstrap the system, *Justice* admits all jobs regardless of deadline. For these jobs, *Justice* allocates `requestedTasks` CPUs to the job. For any job for which there are insufficient resources for the allocation, *Justice* allocates the number of CPUs available. When a job completes (either by meeting or exceeding its deadline), *Justice* invokes the pseudocode Function `TRACK_JOB` shown in Algorithm 1.

**Algorithm 1** *Justice* TRACK_JOB Algorithm

```
1: function TRACK_JOB(compTime, requestedTasks, deadline, numCPUsAllocd, success)
2:     deadlineCPUs = compTime/deadline
3:     maxCPUs = min(requestedTasks, cluster_capacity)
4:     minReqRate = deadlineCPUs/maxCPUs
5:     minReqRateList.add(minReqRate)
6:     MinCPUFrac = min(minReqRateList)
7:     MaxCPUFrac = max(minReqRateList)
8:     LastCPUFrac = numCPUsAllocd/maxCPUs
9:     LastSuccess = success
10:    fractionErrorList.append(minReqRate − LastCPUFrac)
11: end function
```

**Algorithm 2** Fraction Calculation

```
1: function CALCULATE_ALLOC_FRACTION
2:     if LastSuccess then
3:         CPUFraq = MinCPUFrac
4:     else
5:         CPUFraq = MaxCPUFrac
6:     end if
7:     fraction = (LastCPUFrac + CPUFraq)/2
8:     return fraction
9: end function
```

`TRACK_JOB` calculates the minimum number of CPUs required (`deadlineCPUs`) if the job were to complete by its deadline, using its execution profile available from system logs. Line 2 in the Function is derived from the equality:

$$numCPUsAllocd * jobET = deadlineCPUs * deadline$$

On the left is the actual computation time by individual tasks, which we call `compTime` in the algorithm. `numCPUsAllocd` is the number of CPUs that the job used during execution and `jobET` is its execution time without queuing delay. The right side of the equation is the total computation time consumed across tasks if the job had been assigned `deadlineCPUs`, given this execution profile (`compTime`). `deadline` is the time (in seconds) specified in

**Algorithm 3** Fraction Correction and Validation

```
1: function CORRECT_ALLOC_FRACTION(fraction)
2:     correction =CALC_SMOOTHED_AVG(fractionErrorList))
3:     correctedFraction = fraction + correction
4:     correctedFraction =VALIDATE_FRACTION(correctedFraction)
5:     return correctedFraction
6: end function
7: function VALIDATE_FRACTION(fraction)
8:     if fraction < min(minRequiredAllocationRatioList) then
9:         fraction = min(minRequiredAllocationRatioList)
10:    else if fraction > 1 then
11:        fraction = 1
12:    end if
13:    return fraction
14: end function
```

**Algorithm 4** Admission Control and Resource Allocation

```
1: function ADMISSION_CONTROL(RequesterJob)
2:     for all j ∈ SubmittedJobs do
3:         Feasible = True, TTD = Deadline − ElapsedTime
4:         reqCpus = ESTIMATE_REQ(j, TTD)
5:         if reqCpus > min(taskCount, capacity) then
6:             Feasible = False
7:         end if
8:         if Share(j) < reqCpus then
9:             if Feasible == True then
10:                priority = reqCpus/TTD,ADD2HEAP(priority,j)
11:            else
12:                DROP_JOB(j)
13:            end if
14:        end if
15:    end for
16:    allocations = ALLOC_RESOURCES(heap)
17:    if RequesterJob ∉ allocations then
18:        Add RequesterJob to queue
19:    end if
20: end function

21: function ESTIMATE_REQ(Job)
22:    maxCpus = min(tasks, capacity), reqCpus = maxCpus
23:    if CompletedJobs > 1 then
24:        fraction = CALCULATE_ALLOC_FRACTION()
25:        fraction = CORRECT_ALLOC_FRACTION(fraction)
26:        fraction = (deadline/(deadline − queue)) ∗ fraction
27:        reqCpus = max(ceil(fraction ∗ maxCpus), 1)
28:    end if
29:    return reqCpus
30: end function

31: function ALLOC_RESOURCES(heap)
32:    offers = CREATE_OFFERS(heap)
33:    allocations = SEND_OFFERS(offers)
34:    return allocations
35: end function

36: function CREATE_OFFERS(heap)
37:    while availableCpus > 0 and heap not empty do
38:        for all Job j ∈ heap do
39:            offer = min(request(j), availableCpus)
40:            if offer < request(j) then
41:                offer = 0
42:            else
43:                availableCpus− = offer
44:                offersDict[j] = offer
45:            end if
46:        end for
47:    end while
48:    return offersDict
49: end function
```

the job submission. By dividing `compTime` by `deadline`, we extract `deadlineCPUs` for this job.

Next, *Justice* divides `deadlineCPUs` by the maximum number of CPUs allocated to the job. The resulting `minReqRate` is a fraction of the maximum that *Justice* could have assigned to the job and still have it meet its deadline. *Justice* adds `minReqRate` to a list of fractions (`minReqRateList`) that contains the minimum required rates (fractions of `deadlineCPUs` over `requestedTasks`) across all completed jobs. Then it calculates from this list the global minimum (`MinCPUFrac`) and maximum (`MaxCPUFrac`) fractions. It also tracks the observed fraction allocated to the last completed job (`LastCPUFrac`) and whether the job satisfied or exceeded its deadline (`LastSucess`). *Justice* then uses `MaxCPUFrac` and `MinCPUFrac` to predict the allocatable fractions of future jobs. `MaxCPUFrac` and `MinCPUFrac` are always less than or equal to 1. The tighter the deadlines, the more conservative (nearer to 1) these fractions and the corresponding *Justice*'s resource provisioning will be.

*Justice* computes the CPU allocation fraction (`alocCPUFrac`) for each newly submitted job as the average of the `LastCPUFrac` and either `MinCPUFrac` or `MaxCPUFrac`, as shown in Algorithm 2, 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., `alocCPUFrac` converges to `MinCPUFrac`), while deadline violations make *Justice* more conservative, causing it to increase the fraction in an attempt to prevent future violations (`alocCPUFrac` converges to `MaxCPUFrac`).

*Justice* uses a Kalman filter mechanism to correct inaccuracies of its initial estimations (Algorithm 3). 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, `validate_fraction` ensures that the corrected fraction remains with allowable limits (the fraction should not be less than the minimum observed `MinCPUFrac` or greater than 1).

After *Justice* computes, corrects, and validates `alocCPUFrac`, *Justice* considers the time that the job has spent in the queue (line 26 in Function `estimate_req` of Algorithm 4). *Justice* multiplies `alocCPUFrac` by the number of tasks requested in the job submission (rounding to the next largest integer value). It uses this value (or the maximum cloud capacity, whichever is smaller) as the number of CPUs to assign to the job for execution (Function `estimate_req` in Algorithm 4). *Justice* allocates resources to jobs (Function `alloc_resources`) by creating the 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). *Justice* sends these offers to the frameworks (line 33 in Algorithm 4 - we omit Function `SEND_OFFERS` for brevity, which provides communication between *Justice* and the resource negotiator (e.g. YARN or Mesos)). *Justice* performs this process each time a job is submitted or completes. It also updates the deadlines for jobs in the queue, reducing each by the time that has passed since submission (line 3 in Algorithm 4), recomputes the CPU allocation of each enqueued job and and as part of its admission control policy, it either drops any jobs in queue with infeasible deadlines or enqueues jobs

that cannot be admitted but are still feasible (lines 12 and 18 respectively in Algorithm 4).

### 2.2 Admission Control

After estimating job resource requirements, *Justice* implements a *proactive* admission control so that it can prevent infeasible jobs (jobs likely to miss their deadlines) from ever 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 (i.e. limited capacity or high utilization). *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 (which are expected to run longer) from wasting resources.

*Justice* admits jobs based on a pluggable priority policy. We have considered various policies for *Justice* and use a policy that prioritizes minimizing the number of jobs that miss their deadlines. For this policy (line 10 in Algorithm 4), *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 Experimental Methodology

We compare *Justice* to the fair-share allocator that currently ships with the open-source Mesos [21] and YARN [40] resource negotiators, using trace-based simulation. Our system is based on SimPy [36] and replicates the execution behavior of industry-provided production traces of big data workloads (cf Section 4).

The current Mesos and YARN fair-share allocator does not account for job deadlines. When making allocation decisions, it (tacitly) assumes that each job will use the resources allocated to it indefinitely and that there is no limit on the turn-around time a job's owner is willing to tolerate. We hypothesize a straight-forward modification to the basic allocator that allows it to consider job deadlines (which would need to be submitted with each job) when making decisions.

Finally, we implement an "oracle" allocator that has perfect foreknowledge of the minimum resource requirements each job needs to meet its deadline exactly. Note that the oracle does not implement system-wide prescience – its prediction is perfect on a per-job basis. That is, the oracle does not try all possible combinations of job schedules to determine the optimal allocation. Instead, the oracle makes its decision based on a perfect prediction of each job's needs. These allocation policies are summarized as follows:

`Baseline FS`: This allocator employs a fair sharing policy [4, 16, 17, 37, 46] Its behavior is similar to that of the default allocator in Mesos and YARN and, as such, runs all jobs submitted regardless of their deadlines and resource requirements.

`Reactive FS`: This allocator extends Baseline FS by allowing the allocator to terminate any job that has exceeded its deadline. That is, it "reacts" to a deadline miss by freeing the resources so that other jobs may use them.

`Oracle`: This allocator allocates the minimum number of resources that a job requires to meet its deadline. If sufficient resources are unavailable, the Oracle queues the job until the resources become available or until its deadline has passed (or is no longer achievable). For the queued jobs, Oracle gives priority to jobs with fewer required resources and longer time until the deadline.

*`Justice`*: As described in Section 2, this allocator proactively drops, enqueues, or admits jobs submitted. It estimates the share of each job as a fraction of its maximum demand. This fraction is based on the historical performance of jobs. For the queued jobs, *Justice* gives priority to jobs with fewer required resources and longer computation times. *Justice* drops any jobs that are infeasible based on a comparison of their deadlines with a prediction of the time to completion. Jobs that are predicted to miss their deadlines are not admitted (they are dropped immediately) as are any jobs that exceed their deadlines.

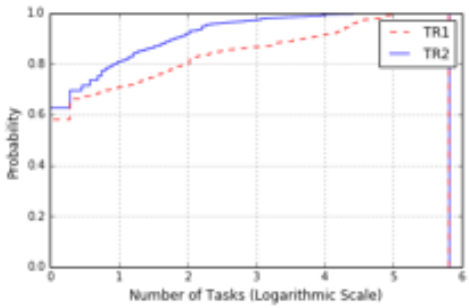
### 3.1 Deadline Types

We evaluate the robustness of our approach by running experiments using deadline formulations from prior works [15, 30, 43, 44, 51] and interesting variations on them. In particular, we assign deadlines that are multiples of the optimal execution time of a job (which we extract from our workload traces). We use two types of multiples: Fixed and variable.

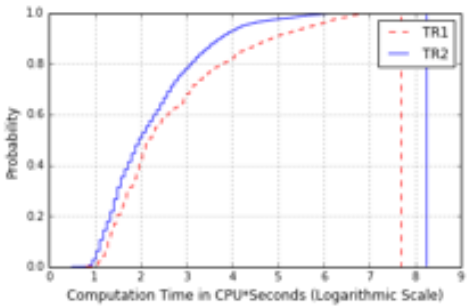
`Fixed Deadlines`: With fixed deadlines, we use a deadline that is a multiple of the optimal execution time (a formulation found in [30, 51]). Each deadline is expressed as $D_i = x \cdot T_i$, where $T_i$ is the optimal runtime of the job and $x >= 1.0$ is some fixed multiplicative expansion factor. 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. We consider the following variable deadline types:

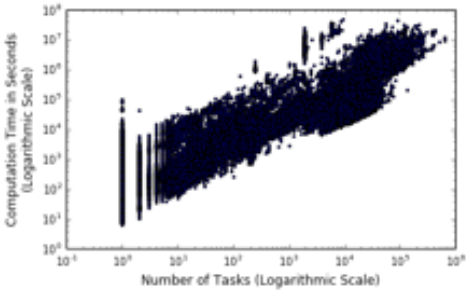
- *Jockey*: We pick with equal probability a deadline expansion factor $x$ from two possible values (a formulation described in [15]). In this work, we use the intervals from the sets with values $(1, 2)$ and $(2, 4)$ to choose $x$ and, again, compute $D_i = x \cdot T_i$, where $T_i$ is the minimum possible execution time. We refer to this variable deadline formulation as *Jockey1x2x* and *Jockey2x4x*.
- *90loose*: This is 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*: The deadline multiples of this type are uniformly distributed in the intervals $[1, 3]$ and $[2, 4]$ (as described in [43, 44]); we refer to these deadlines as *Aria1x3x* and *Aria2x4x*, respectively.



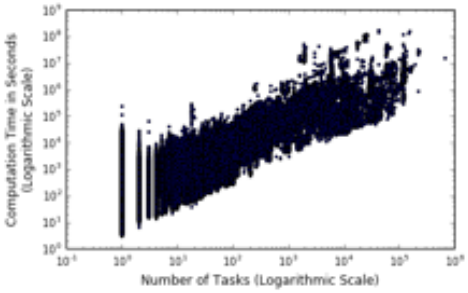

(a) CDFs of number of tasks per job




(b) CDFs of computation time per job




(c) Job computation time vs number of tasks for TR1




(d) Job computation time vs number of tasks for TR2

Fig. 1: **Workload Characteristics:** Number of tasks per job (Figure 1a) and computation time per job (Figure 1b) for TR1 and TR2 and computation time relative to jobs size in number of tasks (Figures 1c and 1d). Small jobs are large in number but consume a very small proportion of trace computation time.

| Trace | CPUs | Jobs | Comp. Time (Hours) | 1-Task Pct | 1-Task Time Pct |
|---|---|---|---|---|---|
| TR1 | 9345 | 159194 | 8585673 | 58% | 0.1% |
| TR2 | 24721 | 1140064 | 13301659 | 62% | 0.3% |

TABLE 1: Summary of Traces. Columns are trace name, peak cluster capacity, total number of jobs, total computation time in hours, percentage of 1-task jobs, and percentage of 1-task job computation time.

## 4 Workload Characterization

To evaluate *Justice*, we use two 3-month traces from production Hadoop deployments executing over different YARN clusters. The traces were recently donated to the *Justice* effort by an industry partner on condition of anonymity. Each trace contains a job ID, job category, number of map and reduce tasks, map and reduce time (computation time across tasks), job runtime, among other data. It does contain information about the scheduling policy or HDFS configuration used in each cluster. Thus we assume a minimum of one CPU per task and use this minimum to derive cluster capacity; we are considering sub-portions of CPUs as part of future work.

Table 1 summarizes the job characteristics of each trace. The table shows the peak cluster capacities (total number of CPUs), the total number of jobs, the total computation time across all tasks in the jobs, the percentage of jobs that have only one task, and the percentage of computation time that single-task jobs consume across jobs. We refer to the trace with 159194 jobs as TR1 and the trace with 1140064 jobs as TR2. The peak observed capacity (maximum number of CPUs in use) for TR1 is 9345 and for TR2 is 24721.

The table also shows that even though there are many single-task jobs, they consume a small percentage of the total computation time in each trace. To understand this characteristic better, we present the cumulative distribution of number of tasks in Fig. 1a and computation time in Fig. 1b per job in on a logarithmic scale. Approximately 60% of the jobs have a single task and 70-80% of the jobs have fewer than 10 tasks, across traces. Only 13% of the jobs in TR1 and 3% of the jobs in TR2 have more than 1000 tasks. Also, the vast majority of jobs have short computation times. Approximately 70% of jobs in TR1 and 80% in TR2 have computation time that is less than 1000 cpu*seconds, ie their execution would be 1000 seconds if they were running in one CPU core.

The right graph in the figure compares job computation time with the number of tasks per job (both axes are on a logarithmic scale) for the TR1 trace (TR2 exhibits a similar correlation). In both traces, 80% of the 1-task jobs and 60% of the 2-10 task jobs have computation time of fewer than 100 seconds. Their aggregate computation time is less than 1% of the total computation time of the trace. Jobs with more than 1000 tasks account for 98% and 94% of the total computation

time for TR1 and TR2, respectively. Finally, job computation time varies significantly across jobs.

We have considered leveraging the job ID and number of map and reduce tasks to track repeated jobs, but find that for these real-world traces such jobs are small in number. In TR1, 18% of the jobs repeat more than once and 12% of the jobs repeat more than 30 times. In TR2, 25% of the jobs repeat more than once and 16% of the jobs repeat more than 30 times. Moreover, we observe high performance variation within each job class. Previous research has reported similar findings and limited benefits from exploiting job repeats for production traces [15].

## 5 Results

We evaluate *Justice* using two production traces for different resource-constrained cloud deployments (number of CPUs). We compare *Justice* against different 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 3.

### 5.1 Fairness Evaluation

We use Jain's fairness index [24] applied to the fraction of demand each scheduler is able to achieve as a measure of fairness. 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. Jain's fairness index is then $\frac{|\sum_{i=1}^{n} F_i|^2}{n * \sum_{i=1}^{n} F_i^2}$.

Figure 2 presents the fairness index averaged over 60-sec intervals for all the allocation policies and deadlines considered in this study, for trace TR1 (top graphs) and trace TR2 (bottom graphs) and for two resource-constrained cloud deployments; highly constrained (left graphs) and moderately constrained (right graphs) settings.

The results show that when resources are limited, fair-shair allocation policies generate substantially lower fairness indices compared to *Justice*. This occurs because these allocators do not anticipate the arrival of future workload. Thus, jobs that require large fractions of the total resource pool receive everything they ask for, causing jobs that arrive later to block or to be under-served [9]. Moreover, jobs waiting in queue may miss their deadlines (i.e. receive an $A_i$ value of zero) or receive an insufficient allocation once released.

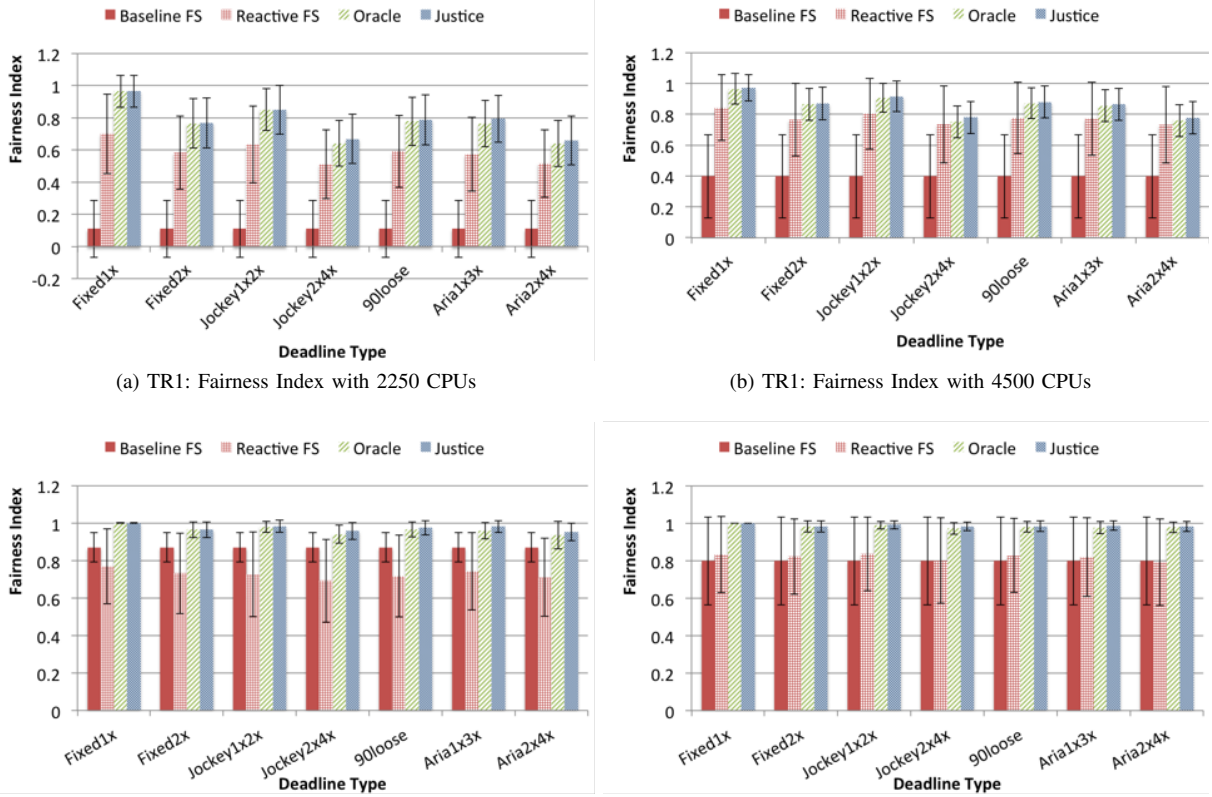
Note that adding the ability to simply drop jobs that have missed their deadlines does not alleviate the fairness problem. The Reactive FS policy (described in Section 3) achieves better fairness than the Baseline fair-share scheduler on TR1, but does not achieve the same levels as *Justice*. Also notice that the Baseline FS and Reactive FS are not directly comparable with each other as they both apply the same fairness policy while one of them (Reactive FS) drops the jobs that have already missed their deadlines leading to a different workload schedule. Thus, on TR2 the Baseline FS achieves better results than the Reactive FS. However, for both TR1 and TR2 the "fair" allocators do worse than *Justice* and the oracle. When a large job (one with a large value of $D_i$) can meet its deadline (i.e. it is not dropped by Reactive FS), it may only get a small fraction of its requested allocation (receiving a small value of $A_i$) thereby contributing to the fairness imbalance when compared to *Justice*. Because the confidence intervals between Reactive FS and *Justice* on TR1 and Baseline FS and *Justice* on TR2 overlap, we also conducted a Welch's t-test [47] for all deadline-types and resource capacities. We find that in all cases, the P-value is very small (e.g. significantly smaller than 0.01). Thus the probability that the means are the same is also very small.

The reason *Justice* is able to achieve fairness is because it uses predictions of future demand to implement admission control. *Justice* uses a running tabulation of the average fraction of $A_i/D_i$ that was required by previous jobs to meet their deadline to weight the value of $A_i/D_i$ for each newly arriving job. *Justice* computes this fraction globally by performing an on-line "post mortem" of completed jobs. Then, for each new job, *Justice* allocates a fraction of the demand requested using this estimated fraction. *Justice* continuously updates its estimate of this fraction so that it can adapt to changing workload conditions. As a result, every requesting job gets the same share of resources as a fraction of its total demand, which is by definition the best possible fairness according to Jain's formula.

Interestingly, *Justice* achieves a better fairness index than the Oracle for variable deadlines (e.g. Aria1x3x). The Oracle allocates to every job the minimum amount of resources required to meet the deadline. Consequently, when the deadline tightness across jobs differ, the fraction of resources that each job gets compared to its maximum resources will also differ. This leads to inequalities in terms of fairness. To avoid the paradox of an Oracle not giving perfect fairness, we could modify Jain's formula by replacing the maximum demand of a job with the minimum required resources in order to meet a deadline. However, we wish to use prior art when making comparisons to the existing fair-share allocators, and so the Oracle (under this previous definition) also does not achieve perfect fairness. In other words, Oracle is an oracle with respect to minimum resource requirements needed to satisfy each job's deadline and not a fairness oracle for the overall system.

Although *Justice* yields the best fairness results compared to other allocators, it is not optimal (i.e. the fairness index is not 1). In particular, when queued jobs are released they may miss their deadlines, but while doing so, cause other jobs to receive little or no allocation. To compensate for this, *Justice* attempts to further weight their allocation by the ratio of the deadline to the time remaining to the deadline ($\frac{deadline}{deadline - queueTime}$), or if achieving the deadline is not possible, *Justice* drops them to avoid wasted occupancy. The cost of this optimization is an occasional fairness imbalance but this cost is less than that for the other allocators we evaluate.

Integer CPU assignment is another source of fairness imbalance. Because jobs require an integer number of CPUs each allocation must be rounded up when it is weighted by the current success fraction. For small jobs, the additional fraction




(a) TR1: Fairness Index with 2250 CPUs

(b) TR1: Fairness Index with 4500 CPUs

(c) TR2: Fairness Index with 2500 CPUs

(d) TR2: Fairness Index with 5000 CPUs

Fig. 2: **Fairness Evaluation:** Average of Jain's fairness index (and 0.95 error bars) for trace TR1 (top graphs) and trace TR2 (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]

constitutes a significant overhead in terms of fairness. While the industry traces contain large numbers of small jobs, they are often short lived allowing *Justice* to adapt overall fairness quickly. We are considering sub-CPU allocations as part of future work.

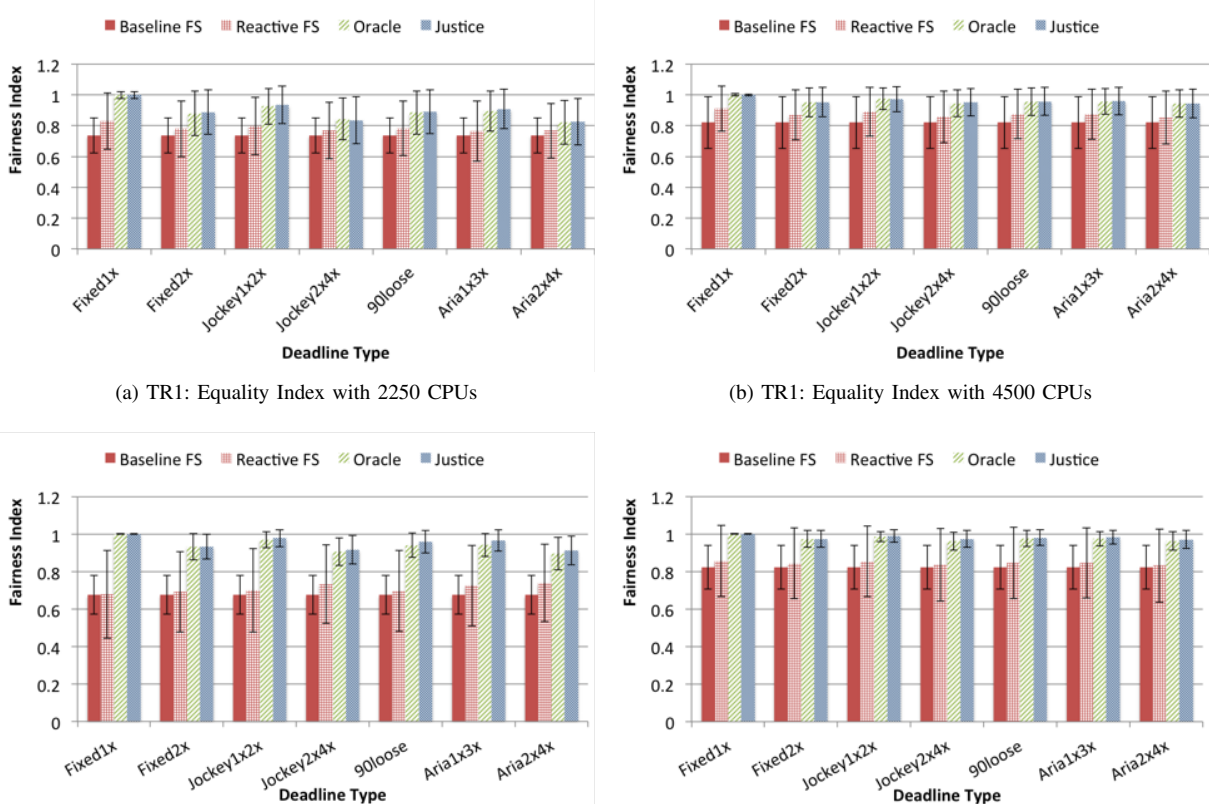
We next evaluate fairness using a second metric, which we refer to as "equality". Fair-share allocators attempt to give an "equal" share of resources to concurrently executing jobs. To evaluate, the degree to which the allocators achieve this goal in resource-constrained settings, we modify Jain's fairness index so that $F_i$ corresponds to the resource allocation of each job $i$ instead of to the job's fraction of demand.

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 differences in maximum demand (a job cannot be allocated more CPUs than it demands).

Figure 3 presents equality results across all allocators, workloads, and capacities that we consider. The results show that in resource constrained settings, fair-shair allocation policies preserve equality better than they do fairness. *Justice* again achieves a better fairness score than the fair-share allocators by up to 23% and 17% for the two capacities for TR1 (TR2 results are similar). Even though the goal of *Justice* is not to preserve equality but instead to prioritize 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 (when they do not get any resources at all). Second, due to constrained resources, *Justice* drops large jobs more frequently which provides more opportunity for it to facilitate fairness at a finer grain across frameworks.

## 5.2 Deadline Satisfaction

We next evaluate how well the allocators perform in terms of deadline satisfaction. Our goal with this set of experiments




(a) TR1: Equality Index with 2250 CPUs

(b) TR1: Equality Index with 4500 CPUs

(c) TR2: Equality Index with 2500 CPUs

(d) TR2: Equality Index with 5000 CPUs

Fig. 3: **Equality Evaluation:** Average equality indexes (and 0.95 error bars) for trace TR1 (top graphs) and trace TR2 (bottom graphs) with highly constrained capacities (left graphs) and moderately constrained capacities (right graphs).

is to verify that *Justice* is not simply achieving fairness by dropping a large fraction of jobs – so that those that remain receive a fair allocation.

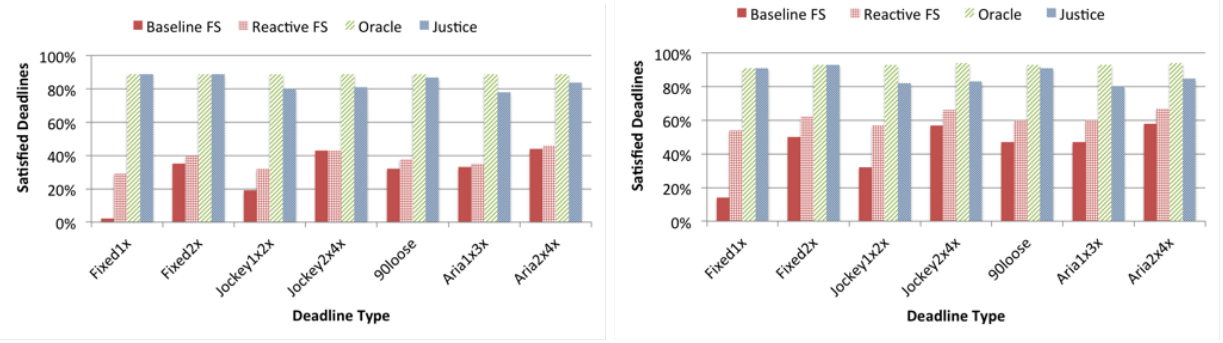
To investigate this, we compute the *Satisfied Deadline Ratio (SDR)* as the fraction of the jobs that complete before their deadline over the total number of submitted jobs. For the set of all the submitted jobs $J_1, J_2, ..., J_n$, if $m < n$ is the subset of successful jobs $J_1, J_2, ...J_m$, then SDR is: $\frac{\sum_{i=1}^{m} J_i}{\sum_{j=1}^{n} J_j}$.

Figure 4 presents the SDR for each combination of allocator and deadline type. For all deadline types, *Justice* meets significantly more deadlines than the fair-share policies and performs similarly to the Oracle. *Justice* satisfies at least 88% more deadlines than Baseline FS and from 83% to 207% more deadlines than Reactive FS. *Justice* outperforms fair-share policies because these policies do not consider deadline information and share resources naively and greedily. Because *Justice* is able to use both job deadlines and historical job behavior in its allocation decision, it is able to meet a larger fraction of deadlines than existing allocators while achieving greater fairness.

In particular, without admission control, the Baseline and Reactive FS allocators must admit a large fraction of jobs that ultimately do not meet their deadlines. This "wasted" work has two consequences on deadline performance. First, it causes unnecessary queuing of jobs that, because of the time spent in queue, may also miss their deadlines. Second, it causes resource congestion, thereby reducing the fraction of resources allocated to all jobs. Consequently, some jobs, which would otherwise succeed, miss their deadlines. By attempting to identify those jobs most likely to miss and dropping those jobs proactively, *Justice* is able to achieve a larger fraction of deadline successes overall.

Fair-share policies fail to meet deadlines when resources are constrained also because of their greedy allocation. They allocate as many resources as are available until they run out regardless of what jobs require to meet their deadlines. As a consequence, jobs with looser deadlines get more resources than what they actually need to finish on time, wasting valuable resources that are needed for future jobs with tighter deadlines. In contrast, *Justice* attempts to identify, based on the fraction of demand that previous successful jobs needed in order to meet their deadlines, the *minimum number* of resources required to meet their deadlines "just in time."

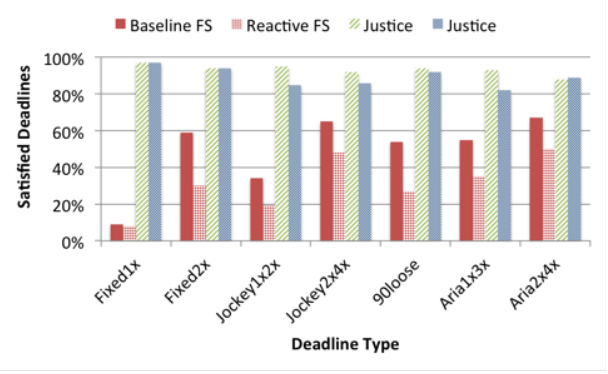
Finally, as noted previously, the Oracle does not have perfect information (i.e., it does not have a global optimal schedule). Instead it knows the actual job computation time (`compTime`). Thus, it is able to assign the minimum number of CPUs to



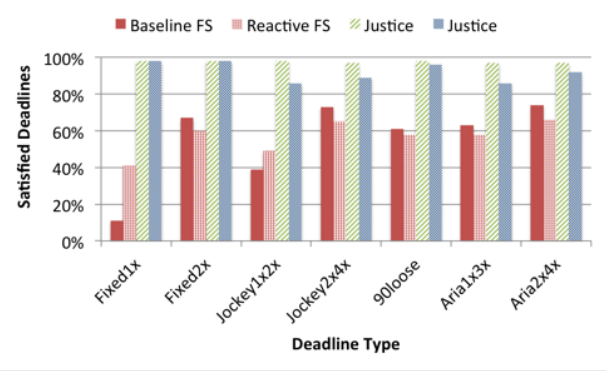

(a) TR1: Satisfied Deadlines with 2250 CPUs




(b) TR1: Satisfied Deadlines with 4500 CPUs




(c) TR2: Satisfied Deadlines with 2500 CPUs




(d) TR2: Satisfied Deadlines with 5000 CPUs

Fig. 4: **Deadline Satisfaction:** Satisfied Deadlines Ratio (SDR) for trace TR1 (top graphs) and TR2 (bottom graphs) with highly constrained capacities (left graphs) and moderately constrained capacities (right graphs) for different deadline types.

each job to satisfy its deadline. SDR for Oracle is not 100% because it must drop (refuse to admit) jobs for which there is insufficient capacity to meet their deadline.

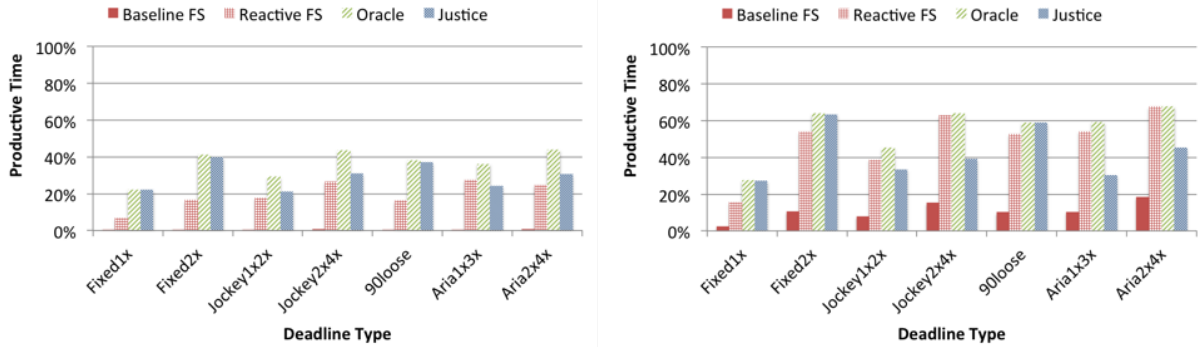
### 5.3 Efficient Resource Usage

We next evaluate workload productivity, i.e. the measure of productive time (i.e. the work done by jobs that complete by their deadlines) and wasted time (i.e. work done by jobs that miss their deadline) via the metrics *Productive Time Ratio (PTR)* and *Wasted Time Ratio (WTR)*. For the set of all the 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 $n = m + k$. PTR is $\frac{\sum_{i=1}^{m} T_i}{\sum_{j=1}^{n} T_j}$ and WTR is $\frac{\sum_{i=1}^{k} T_i}{\sum_{j=1}^{n} T_j}$.

Figure 5 and Figure 6 present PTR and WTR, respectively, for different allocation policies and deadline type for two resource constrained settings for trace TR1 (top graphs) and TR2 (bottom graphs). For all cases, Baseline FS spends a very small ratio of computation time productively, i.e. it spends almost all the computation time on jobs that missed their deadlines. Reactive FS improves over Baseline FS by reactively dropping jobs that have already violated their deadlines. *Justice*, performs significantly better (up to 221% higher PTR and up to 100% lower WTR than Reactive FS) and slightly worse than the Oracle (up to 33% lower PTR) for the 2250 CPUs deployment. *Justice* outperforms fair-share policies because it proactively drops jobs with violated deadlines and jobs that it predicts are likely to miss their deadline.

Our experiments also show that the more constrained or utilized the system, the better *Justice* performs in terms of PTR and WTR, relative to the other allocators that we consider. Baseline FS fails to satisfy deadlines of bigger jobs because it shares a very limited resources equally between bigger and smaller jobs. This share, under resource constrained settings, is not sufficient for the bigger jobs to complete on time. Reactive FS improves PTR and WTR because it drops jobs that violate their deadlines, freeing up resources for other jobs. *Justice* wastes significantly fewer resources compared to Reactive FS because it drops jobs with large expected computation times using its pluggable priority policy (Section 2), as soon as they become infeasible.

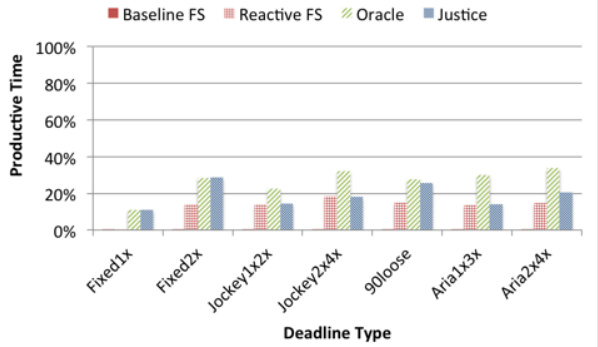
When the system is less constrained (e.g. 4500 CPUs in the right graphs), *Justice*'s PTR is significantly better than Baseline FS (from 146% on Aria2x4x up to 926% on Fixed1x). It also outperforms Reactive FS up to 72% and performs similarly to the Oracle, for deadline types with less variation (Fixed and 90loose). However, it achieves slightly (14% for Jockey1x2x) or moderately (44% for Aria1x3x) less PTR for high variable



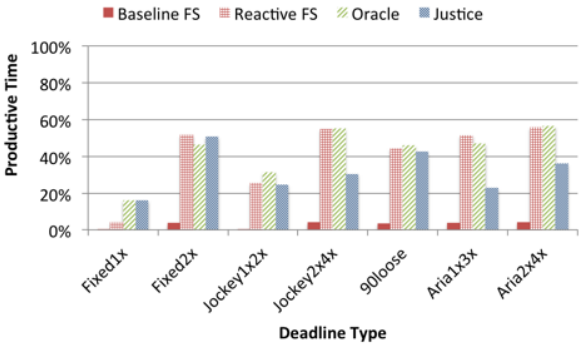

(a) TR1: Productive Time with 2250 CPUs




(b) TR1: Productive Time with 4500 CPUs




(c) TR2: Productive Time with 2500 CPUs




(d) TR2: Productive Time with 5000 CPUs

Fig. 5: **Productivity:** Productive Time Ratio (PTR) for trace TR1 (top graphs) and TR2 (bottom graphs) with highly constrained capacities (left graphs) and moderately constrained capacities (right graphs) for different deadline types.

deadline types, even though it still satisfies significantly more deadlines compared to Reactive FS for the these deadline types (recall *Justice*'s SDR on Figure 4a is 44% and 33% higher than reactive FS for Jockey1x2x and Aria1x3x respectively).

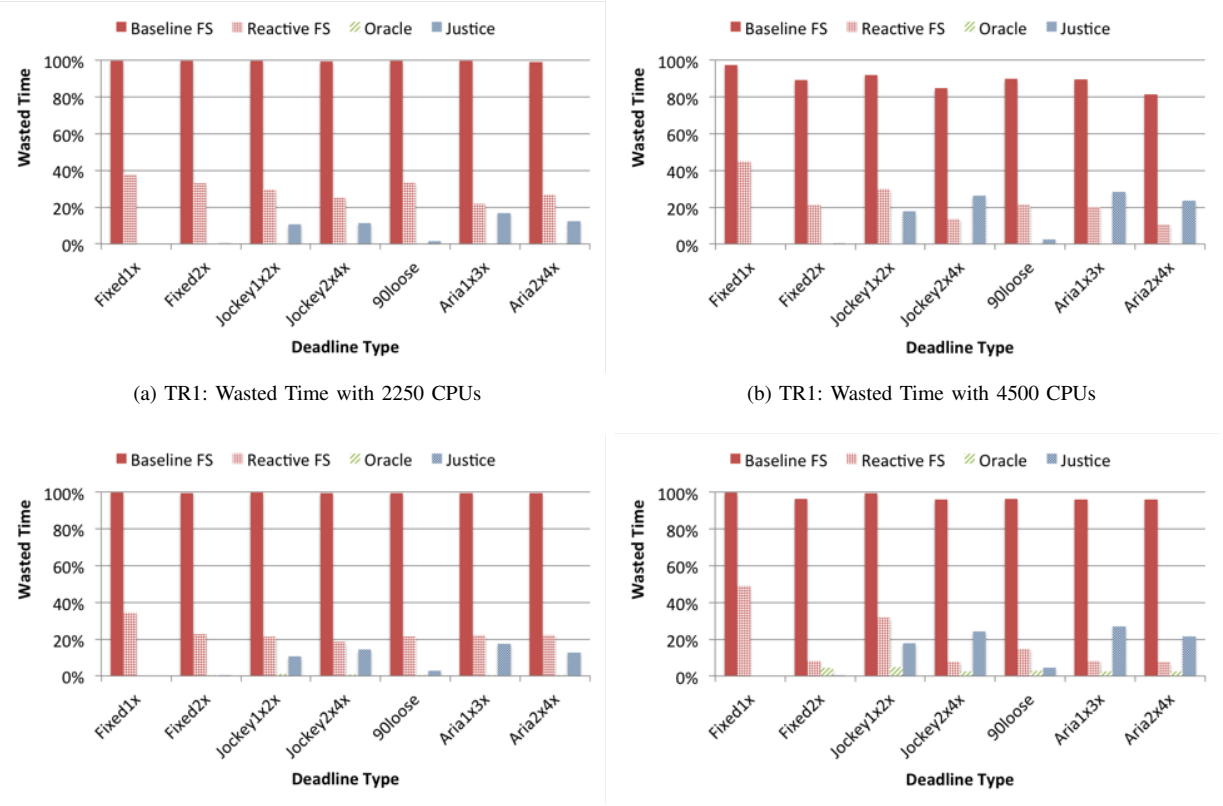
Specifically, as resource scarcity is reduced for a fixed workload, large jobs that are admitted by the Baseline FS and Reactive FS allocators stand a better chance of getting the "extra" resources necessary to complete, and thus, add to the PTR compared to *Justice*, which might have excluded them due to admission control. However, when deadlines are variable, *Justice*'s admission control is conservative, prioritizing fairness and deadline success over resource saturation. This result indicates that extant fair-share allocators may be more appropriate for maximizing productive work when resources are more plentiful and the need to meet deadlines less of a concern. Put another way, when resources are plentiful, the cost of meeting a higher fraction of deadlines with greater fairness is a lower PTR due to admission control.

## 5.4 Cluster Utilization

The final set of experiments investigates how allocation policies for resource constrained settings impact overall utilization and CPU idle times. *Justice* considers a CPU to be *idle* when the allocator has not assigned to it any tasks to run and and to be *busy* when the CPU is running a task. We then define *Cluster Utilization* as $\frac{busy}{idle+busy}$ where $busy$ is the total busy time and $idle$ is the total idle time across a workload.

Figure 7 shows resource utilization for trace TR1 (top graphs) and trace TR2 (bottom graphs) with highly constrained capacity (left graphs) and moderately constrained capacity (right graphs), for the different deadline types that we consider. The results in the left graphs are particularly surprising and somewhat counter-intuitive. Given severe resource constraints, *Justice* achieves lower utilization than the other allocators, but (as presented previously in Figures 5 and 6 respectively) exhibits higher PTR and lower WTR. Thus, *Justice* enables more productive work with less waste and lower utilization. One might assume that the utilization difference is due to less productivity or more overhead. However, the results show that this is not the case. *Justice* is able to achieve both a greater fraction of deadlines (cf Figure 4) and better fairness (cf Figure 2), with fewer resources.

These results are also interesting in that they reveal a potential opportunity to introduce *more* workload (to take advantage of the available utilization that is not used by *Justice* ) when resources are severely constrained. To investigate this potential, we extract and analyze the number and duration of idle CPUs




(a) TR1: Wasted Time with 2250 CPUs

(b) TR1: Wasted Time with 4500 CPUs

(c) TR2: Wasted Time with 2500 CPUs

(d) TR2: Wasted Time with 5000 CPUs

Fig. 6: **Resource Waste:** Wasted Time Ratio (WTR) for trace TR1 (top graphs) and TR2 (bottom graphs) with highly constrained capacities (left graphs) and moderately constrained capacities (right graphs) for different deadline types. Lower is better.

that correspond to the experiments shown in Figure 7a for the Aria1x3x deadline type on trace TR1. Figure 8 presents the cumulative distribution of idle time for CPUs that are simultaneously idle in groups of 10 (red, dotted curve) and 100 (blue, solid curve). We find that for deadline types that yield lower utilizations, idle time durations are even larger; we omit these results for brevity.

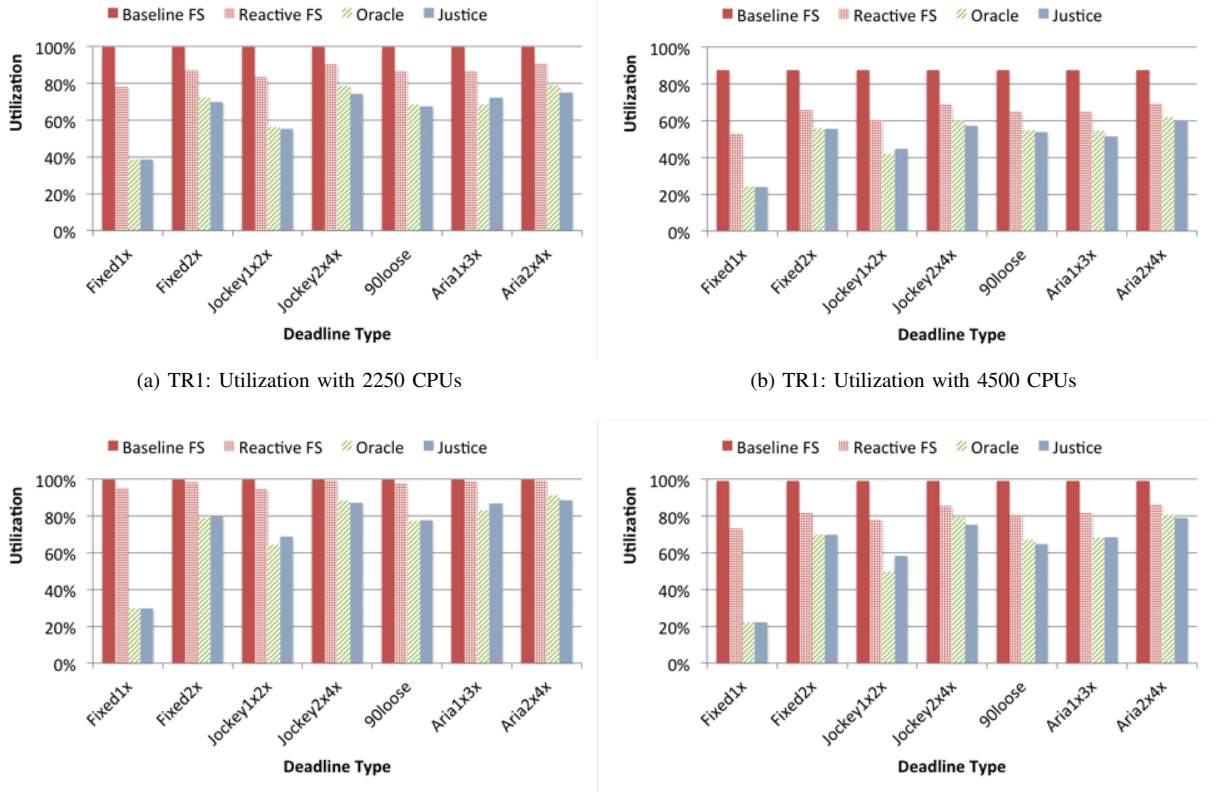
From these results, we observe that 81% of the 10-CPU groups remain idle more than 100 seconds, 68% more than 500 seconds and 59% more than 1000 seconds. Similarly for 100-CPU groups, 80%, 52%, and 41% have idle times of 100 seconds, 500 seconds and 1000 seconds, respectively. From these results, we can derive that 10-CPU and 100-CPUS idle groups exist at any given time of the traces duration with probabilities 98% and 76% respectively.

We next consider the workload characteristics of the traces that we study (Section 4). We have shown (Figure 1b) that 40% of jobs compute for less than 100 CPU*seconds, 60% compute for less than 500 CPU*seconds, and 70% compute less than 1000 CPU*seconds. Moreover, approximately 60% of the jobs employ a single task, 70% of the jobs have fewer than 10 tasks, and 80% less than 100 tasks (Figure 1a). As a result, *Justice* is able to free up enough capacity for sufficient durations to as to admit significant additional workload. That is, if the traces contained more jobs with these characteristics, *Justice* would likely have been able to achieve similar fairness, deadline, and productive work results via increased utilization. We are currently investigating this potential and how to best exploit it as part of on-going and future work.

## 6 Related Work

This paper is an extension to an early version of this work, entitled *Justice : A Deadline-aware, Fair-share Resource Allocator for Implementing Multi-analytics* [10], which overviews the approach. Herein, we significantly extend the original work via the inclusion of a comprehensive algorithm for *Justice*, and a more extensive empirical evaluation that adds a second, larger production workload trace. We also compare the different allocators using a second new fairness analysis, which we refer to as "equality" in Section 5. Other related work includes multi-tenant resource allocators and job performance prediction.

**Sharing in Multi-tenant Resource Allocators**: Cluster managers like Mesos [21] and YARN [40] enable the sharing of cloud and cluster resources by multiple data processing frameworks. Recent research [12, 18, 35] builds on this sharing, to allow users to run jobs without knowledge of the




(a) TR1: Utilization with 2250 CPUs

(b) TR1: Utilization with 4500 CPUs

(c) TR2: Utilization with 2500 CPUs

(d) TR2: Utilization with 5000 CPUs

Fig. 7: **Cluster Utilization:** Utilization for trace TR1 (top graphs) and TR2 (bottom graphs) with highly constrained capacities (left graphs) and moderately constrained capacities (right graphs) for different deadline types.

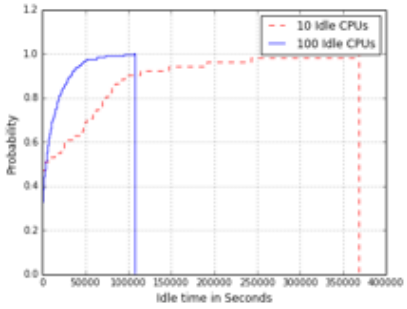



Fig. 8: CDFs of idles times of 10 and 100 CPU groups on TR1 (Similar distribution holds for TR2 as well).

underlying data processing engine. In these multi-analytics settings, the goal of the resource allocator is to provide performance isolation to frameworks by sharing the resources between them [4, 5, 14, 17]. However, for resource-constrained deployments, the fair-shair policies [14, 17] fail to preserve fairness (Section 5.1). Also, all the sharing policies in these works are deadline-agnostic. To meet deadlines, administrators add cluster resources, use a capacity scheduler [5], or require users to reserve resources in advance [3, 7, 39]. Such solutions are costly, inefficient, or impractical for resource constrained clusters.

Another issue encountered in multi-analytics systems, is that frameworks like Hadoop and Spark, which run on top of these resource allocators, have their own intra-job schedulers that greedily occupy the resources allocated to them, even when they are not using them [9, 19, 48]. *CARBYNE* [19] attempts to address this issue by exploiting task-level resource requirements information and DAG dependencies. PYTHIA [11] addresses the same issue by introducing framework-independent admission control that resource allocators use to support dynamic fair-sharing of system resources. Similar to PYTHIA (and contrary to CARBYNE), *Justice* utilizes admission control without requiring job-repetitions and task-level information. Moreover, *Justice* adapts to changing cluster conditions to avoid over-provisioning and preserves fair-sharing in addition to satisfying deadlines.

**Performance Prediction**: To allocate the required resources and meet job deadlines, much related work focuses on exploiting historic [8, 22, 26, 28, 43, 44, 49, 51], and runtime [8, 20, 22, 23, 32, 43, 44, 50, 51] job information, while other research [8, 15, 25, 38, 41, 50] focuses on building job performance profiles and scalability models offline. Although,

effective in many situations, we show that approaches similar to these suffer when used under resource constrained settings.

Strategies that depend solely on repeated jobs, by definition, do not guarantee performance of ad-hoc queries. While approaches that use runtime models, sampling, simulations, and extensive monitoring, impose overheads and additional costs. Moreover, trace analysis in this paper and other research [15] shows that some production workloads have small ratio of repeated jobs and these jobs have often large execution times dispersion. Therefore, approaches based on past executions might not have the required mass of similar jobs over a short period of time in order to predict with high statistical confidence. Furthermore, the vast number of jobs have very short computation times [6, 15, 30, 31, 33]. Thus, approaches that adapt their initial allocation after a job has already started might be ineffective. Lastly, most of these approaches require task-level information, for the specific framework they target, either Hadoop [20, 22, 23, 25, 28, 32, 38, 43, 44, 49, 50, 51] or Spark [34, 45]. For this reason, they cannot be integrated into resource managers as *Justice* can.

## 7 Conclusions

We present *Justice*, a fair-share and deadline-aware resource allocator with admission control for multi-analytic resource negotiators such as Mesos and YARN. *Justice* uses historical job statistics and deadline information to automatically adapt its resource allocation and admission control to achieve fairness and satisfy deadlines even when resource availability is highly constrained or contended for (e.g. as in private cloud and/or edge and fog cloud settings). We evaluate *Justice* using trace-based simulation of two production YARN workloads under different resource constraints and deadline formulations. We compare *Justice* to the existing fair-share allocator that ships with Mesos and YARN and find that *Justice* is able to achieve better fairness, deadline satisfaction, and resource utilization for the settings we investigate.

This work is funded in part by NSF (CNS-1564157, CCF-1539586, CNS-1218808, CNS-0905237, ACI-0751315), NIH (1R01EB014877-01), ONR NEEC (N00174-16-C-0020), Huawei, and the CEC (PON-14-304).

## References

[1] *Apache Hadoop*. http://hadoop.apache.org/ [Online; accessed 2-January-2017].

[2] *Apache Spark*. http://spark.apache.org/ [Online; accessed 2-January-2017].

[3] M. Babaioff et al. ERA: A Framework for Economic Resource Allocation for the Cloud. In: *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee. 2017, pp. 635–642.

[4] A. Bhattacharya et al. Hierarchical scheduling for diverse datacenter workloads. In: *ACM SoCC*. 2013.

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

[6] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In: *VLDB* 5.12 (2012), pp. 1802–1813.

[7] 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.

[8] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In: *ACM SIGPLAN Notices* 49.4 (2014), pp. 127–144.

[9] 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.

[10] 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.

[11] 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.

[12] K. Doka et al. Mix'n'Match Multi-Engine Analytics. In: *IEEE International Conference on Big Data. IEEE*. 2016.

[13] A. R. Elias et al. Where is The Bear?–Automating Wildlife Image Processing Using IoT and Edge Cloud Systems. In: *ACM Conference on IoT Design and Implementation*. ACM. 2017.

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

[15] 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.

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

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

[18] I. Gog et al. Musketeer: all for one, one for all in data processing systems. In: *European Conference on Computer Systems*. 2015.

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

[20] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In: *VLDB* 4.11 (2011), pp. 1111–1122.

[21] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In: *NSDI*. 2011, pp. 22–22.

[22] M. Hu et al. Deadline-Oriented Task Scheduling for MapReduce Environments. In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2015, pp. 359–372.

[23] 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.

[24] 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.

[25] V. Jalaparti et al. Bridging the tenant-provider gap in cloud services. In: *ACM Symposium on Cloud Computing*. 2012.

[26] S. A. Jyothi et al. Morpheus: towards automated SLOs for enterprise clusters. In: *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*. 2016, p. 117.

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

[28] 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.

[29] 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.

[30] 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.

[31] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In: *ACM SIGMOD International Conference on Management of data*. 2009, pp. 165–178.

[32] J. Polo et al. Performance-driven Task Co-Scheduling for MapReduce Environments. In: *IEEE Network Operations and Management Symposium*. 2010, pp. 373–380.

[33] K. Ren et al. Hadoop's Adolescence; A Comparative Workloads Analysis from Three Research Clusters. In: *SC Companion*. 2012.

[34] S. Sidhanta, W. Golab, and S. Mukhopadhyay. OptEx: A Deadline-Aware Cost Optimization Model for Spark. In: *arXiv preprint arXiv:1603.07936* (2016).

[35] A. Simitsis et al. Optimizing analytic data flows for multiple execution engines. In: *ACM SIGMOD International Conference on Management of Data*. 2012, pp. 829–840.

[36] *Simpy*. https://simpy.readthedocs.io/en/latest/.

[37] J. Tan et al. Multi-resource fair sharing for multiclass workflows. In: *ACM SIGMETRICS Performance Evaluation Review* 42.4 (2015).

[38] F. Tian and K. Chen. Towards optimal resource provisioning for running mapreduce programs in public clouds. In: *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE. 2011, pp. 155–162.

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

[40] V. K. Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In: *ACM Symposium on Cloud Computing*. 2013.

[41] 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.

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

[43] 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.

[44] A. Verma et al. Deadline-based workload management for MapReduce environments: Pieces of the performance puzzle. In: *2012 IEEE Network Operations and Management Symposium*. IEEE. 2012, pp. 900–905.

[45] 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.

[46] W. Wang, B. Liang, and B. Li. Multi-resource fair allocation in heterogeneous cloud computing systems. In: *IEEE Trans. Parallel Distrib. Syst.* 26.10 (2015).

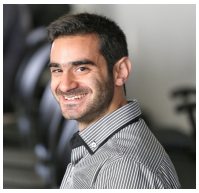
[47] *Welch's T-Test*. https://en.wikipedia.org/wiki/Welch's_t-test [Online; accessed 22-July-2017]. URL: https://en.wikipedia.org/wiki/Welch's_t-test.

[48] 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.

[49] 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.

[50] 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.

[51] Z. Zhang et al. Automated profiling and resource management of pig programs for meeting service level objectives. In: *ACM International Conference on Autonomic computing*. 2012, pp. 53–62.

**Stratos Dimopoulos** Stratos Dimopoulos holds a Ph.D. degree in CS from UC Santa Barbara. His research interests are broadly in the area of big data processing systems, distributed systems and cloud computing. Before joining UCSB, he got his BS degree in Informatics and Telecommunications from the National University of Athens and his MsC in Computer Science from the Athens University of Economics and Business.

**Chandra Krintz** Chandra Krintz is a Professor of Computer Science (CS) at UC Santa Barbara and Chief Scientist at AppScale Systems Inc. Chandra holds M.S./Ph.D. degrees in CS from UC San Diego. Chandra's research interests include programming systems, cloud and big data computing, and the Internet of Things (IoT). Chandra has supervised and mentored over 60 students and has led several educational and outreach programs that introduce young people to computer science.

**Rich Wolski** Dr. Rich Wolski is a Professor of Computer Science at the UC Santa Barbara, and co-founder of Eucalyptus Systems Inc. Having received his M.S. and Ph.D. degrees from UC Davis (while a research scientist at Lawrence Livermore National Laboratory) he has also held positions at the UC San Diego, and the University of Tennessee, the San Diego Supercomputer Center and Lawrence Berkeley National Laboratory. Rich has led several national scale research efforts in the area of distributed systems, and is the progenitor of the Eucalyptus open source cloud project.