Justice: A Deadline-aware, Fair-share Resource Allocator for Implementing Multi-analytics

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 present *Justice*, a fair-share deadline-aware resource allocator for big data cluster managers. In resource constrained environments, where resource contention introduces significant execution delays, *Justice* outperforms the popular existing fair-share allocator that is implemented as part of Mesos and YARN. *Justice* uses deadline information supplied with each job and historical job execution logs to implement admission control. It automatically adapts to changing workload conditions to assign enough resources for each job to meet its deadline "just in time." We use trace-based simulation of production YARN workloads to evaluate *Justice* under different deadline formulations. We compare *Justice* to the existing fair-share allocation policy deployed on cluster managers like YARN and Mesos and find that in resource-constrained settings, *Justice* improves fairness, satisfies significantly more deadlines, and utilizes resources more efficiently.**

***Keywords*—resource-constrained clusters; deadlines; admission control; resource allocation; big data;**

I. Introduction

Scalable platforms such as Apache Hadoop [1] and Apache Spark [2] implement batch processing of distributed analytics applications, often using clusters (physical or virtual) as infrastructure. However, cluster administrators do not use space-sharing job schedulers (e.g. [16, 33, 34, 54]) to partition cluster resources for these platforms. Instead, many "big data" systems are designed to work with a cluster manager such as Mesos [24] or YARN [44], which divide cluster resources (processors and memory) at a more fine-grained level to facilitate effective resource sharing and utilization.

Cluster managers implement fair-share resource allocation [15, 20] via per-application negotiation. Each big data framework (e.g. Hadoop or Spark instance) negotiates with the cluster manager to receive resources and uses these resources to run the tasks of the associated submitted jobs (i.e. user applications). The cluster manager tracks the current allocation of each job and the available cluster resources and uses a fair-share algorithm to distribute resources to the frameworks as jobs arrive.

In this work, we investigate the efficacy of fair-share schedulers in cluster settings that are used increasingly by applications for the emerging "Internet-of-Things" (IoT) domain – distributed systems in which ordinary physical objects are equipped with Internet connectivity so that they may be accessed (in terms of their sensing capabilities) and actuated automatically. IoT applications combine data gathered via simple, low-power sensing devices with data analytics, to make data-driven inferences, predictions, and actuation decisions. Increasingly, IoT analytics are offloaded to more capable, "edge computing" systems that provide data aggregation, analyses, and decision support *near where data is collected* to provide low-latency (deadline-driven) actuation of edge devices and to reduce the bandwidth requirements of large-scale IoT deployments [14, 46]. Edge computing [19] (also termed Fog computing [5]) hypothesizes that these edge systems be small or medium-sized cluster systems capable of acting as an extension of more centralized infrastructure (i.e. as a kind of "content distribution network" for cloud-based analytics).

Edge analytics for IoT represents a shift in the requirements imposed on big-data frameworks and cluster managers, both of which were designed for very large-scale, resource-rich clusters and e-commerce applications (which co-locate data fusion, analysis, and actuation in the cloud). In this paper, we investigate the suitability of existing fair-share resource allocators given this shift in cluster or cloud settings where deadlines are necessary to provide low-latency actuation, and resource contention imposes significant overhead on job turnaround time.

We compare existing fair-share algorithms employed by Mesos [24] and YARN [44] to a new approach, called *Justice*, which uses deadline information for each job and historical workload analysis to improve deadline satisfaction and fairness. Rather than using fairness as the allocation criterion as is done for Mesos and YARN, *Justice* estimates the fraction of the requested resources that are necessary to complete each job just before its deadline expires. It makes this estimate when the job is submitted using the requested number of resources as the number necessary to complete the job as soon as possible. It then relaxes this number according to a running tabulation of an expansion factor that is computed from an on-line post-mortem analysis of all previous jobs run on the cluster. Because the expansion factor is computed across jobs (i.e. globally for the cluster) each analytics framework receives a "fair penalty" for its jobs, which results in a better fair-share, subject to the deadlines associated with each job. Further, *Justice* "risks" running some jobs with greater or fewer resources than it computes they need so that it can adapt its allocations automatically to changing workload characteristics.

We describe the *Justice* approach and compare it to the baseline allocator employed by Mesos and YARN, to simple intuitive extensions to this allocator, and to a job workload "oracle", which knows precisely (i.e. without estimation error) the minimum number of resources needed for each job to meet its deadline. For evaluation, our work uses large production workload traces from an industry partner that provides

commercial big-data services using YARN[1]. The original jobs in this trace were not resource constrained nor did they require completion according to individual deadlines. For these reasons we use a discrete-event, trace-driven simulation to represent how these workloads would execute with significantly fewer cluster resources under different deadline formulations found in related work [17, 32, 47, 48, 57].

Our results show that *Justice* performs similarly to the oracle in terms of fairness and deadline satisfaction, and significantly better than the baseline Mesos and YARN allocator. In addition, *Justice* achieves greater productivity and significantly better utilization than its counter-parts when resources are severely constrained. We next describe *Justice* and its implementation. We then overview our empirical methodology (Section III) and the characteristics of the workload we evaluate (Section IV). We present our results in Section V. In Sections VI and VII, we discuss related work and conclude.

II. JUSTICE

Justice is a fair-share preserving and deadline-aware resource allocator with admission control for resource negotiators such as Mesos [24] and YARN [44] that manages batch applications with deadlines for resource-constrained, shared clusters. *Justice* employs a black-box, framework-agnostic prediction technique (based on measurements of historical job execution times) to estimate the minimum number of CPUs (parallelism) that a job requires to meet its deadline.

Prior work has shown that most fair-share allocators (possibly designed for the "infinite" resources available in a cloud) fail to preserve fairness when resources are limited [11, 22, 53]. This shortfall is due, in part, to their greedy allocation (a result of their inability to predict future demand) and lack of corrective mechanisms (ex: job preemption or dropping). Instead, *Justice* improves upon these techniques by proactively adapting to future demand and cluster conditions through its resource allocation and admission control mechanisms.

Existing fair-share allocators are deadline-insensitive. They assume that a job submitted by a user has value to that user regardless of how large the turn-around time may be. For *Justice*, we assume that because cluster resources may be scarce, each job is submitted with a "maximum runtime" parameter that tells the resource negotiator when the completion of each job is no longer valuable. Henceforth, we term this parameter the "deadline" for the job. The only assumption we make about this user-specified deadline is that the deadline is feasible, i.e., an optimal allocation in an empty cluster is sufficient to complete the job before its deadline.

Note that current resource negotiators such as Mesos and YARN do not include a provision for specifying job maximum runtime. Instead, many cluster administrators statically split their clusters with the use of a capacity scheduler [6], or require users to reserve resources in advance [8, 43] to create differentiated service classes with respect to turn-around time. However, such approaches are inefficient and impractical in resource-constrained clusters, as they further limit peak cluster capacity. In contrast, *Justice* incorporates deadline information to drive its resource allocation, admission-control, and job-dropping decisions.

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

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: **end function**

A. Resource Allocation

To determine how many CPUs to allocate to a new job, *Justice* uses execution time data logged for 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 "just-in-time" (represented as the `deadlineCPUs` variable in Algorithm 1). *Justice* assumes that this minimum required capacity utilizes perfect parallelism (speedup per CPU) and that the number of tasks for a job (the division of its input size and the HDFS block size) is the maximum parallelization possible for the job. 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 cluster capacity (`cluster_capacity`).

To bootstrap the system, *Justice* admits all jobs regardless of deadline, i.e., it allocates `requestedTasks` CPUs to the jobs. 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 by exceeding its deadline, *Justice* invokes the pseudocode function `TRACK_JOB` shown in Algorithm 1.

`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 cluster manager 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 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 result-

ing 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.

Then, to compute the cpu allocation fraction (alocCPUFrac) for each newly submitted job, *Justice* takes the average of the LastCPUFrac and either MinCPUFrac or MaxCPUFrac, depending on whether the last completed job met or violated its deadline respectively. In other words, consecutive successes make *Justice* more aggressive and it allocates smaller resource fractions (alocCPUFrac converges to MinCPUFrac) while deadline violations make *Justice* more conservative and it increases the allocated fraction to prevent future violations (alocCPUFrac converges to MaxCPUFrac).

Justice also utilizes a Kalman filter mechanism to correct inaccuracies of its initial estimations. Every time a job completes its execution, *Justice* tracks the estimation error; the divergence of the given allocation fraction from the ideal minimum fraction (deadlineCPUs). To correct the allocation estimations, *Justice* calculates a weighted average of the historical errors. It can be configured to assign the same weights to all past errors or use exponential smoothing and "trust" more recent errors. Lastly, a validation mechanism ensures that the corrected fraction is still between allowable limits (the fraction should not be less than the minimum observed MinCPUFrac or greater than 1).

After alocCPUFrac is calculated, corrected, and validated as described above, *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 cluster capacity, whichever is smaller) as the number of CPUs to assign to the job for execution. If this number of CPUs is not available, it enqueues the job. *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), recomputes the CPU allocation of each enqueued job and drops any enqueued jobs with infeasible deadlines.

B. Admission Control

After estimating the resources that jobs need to meet their deadlines, *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 cluster 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 cluster 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, *Justice* gives priority to jobs with a small number of tasks and greatest time-to-deadline. 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 initiates execution. Once a job run commences, its CPU allocation does not change.

III. Experimental Methodology

We compare *Justice* to the fair-share allocator that currently ships with the open-source Mesos [24] and YARN [44] using trace-based simulation. Our system is based on Simpy [40] and replicates the execution behavior of industry-provided production traces of big data workloads (cf Section IV).

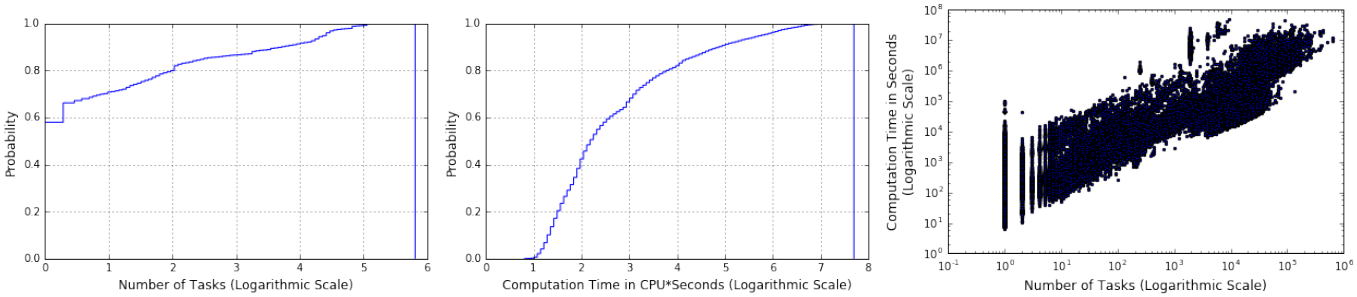
The current Mesos and YARN fair-share allocator does not take into account the notion of deadline. When making allocation decisions, it (tacitly) assumes that each job will use the resources allocated to it indefinitely and that there is no limit on the turn-around time a job's owner is willing to tolerate. 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, 18, 20, 41, 50] without admission control. 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



(a) CDFs of number of tasks per job (b) CDFs of computation time per job (c) Job computation time vs number of tasks

Fig. 1: **Workload Characteristics:** Number of tasks per job, computation time per job, and computation time relative to jobs size (in number of tasks). Small jobs are large in number but consume a very small proportion of trace computation time.

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 II, 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 in the cluster. 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.

A. Deadline Types

We evaluate the robustness of our approach by running experiments using deadline formulations from prior work [17, 32, 47, 48, 57] 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 trace). 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 [32, 57]). 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 [17]). In this study, 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 [47, 48]); we refer to these deadlines as *Aria1x3x* and *Aria2x4x*, respectively.

CPUs	Jobs	Comp. Time (Hours)	1-Task Pct	1-Task Time Pct
9345	159194	8585673	58%	0.1%

TABLE I: Trace Summary. Columns are 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.

IV. Workload Characterization

To evaluate *Justice*, we use a 3-month trace from production Hadoop deployments executing over different YARN clusters. The trace was recently donated to the *Justice* effort by an industry partner on condition of anonymity. The 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 not 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 (vcores) as part of future work. *Justice* uses the number of map tasks (as `requestedTasks` in Algorithm 1) and map time (as `compTime` in Algorithm 1).

Table I summarizes the job characteristics of the trace. The table shows the peak cluster capacity (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. There are 159194 jobs submitted

and the peak observed capacity (maximum number of CPUs in use) is 9345[2].

The table also shows that even though there are many single-task jobs, they consume a small percentage of the total computation time. 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 logarithmic scale. Approximately 60% of the jobs have a single task and 70% of the jobs have fewer than 10 tasks. Only 13% of the jobs have more than 1000 tasks. Also, the vast majority of jobs have short computation times. Approximately 70% of jobs have computation time that is less than 1000 CPU*seconds, i.e. 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). 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% of the total computation time. 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 this real-world trace such jobs are small in number. 18% of the jobs repeat more than once and 12% 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 [17].

V. Results

We evaluate *Justice* using the production trace for different resource-constrained cluster capacities (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 III.

A. Fairness Evaluation

We use Jain's fairness index [28] 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 and for two resource-constrained cluster sizes; very constrained cluster with 2250 CPUs (left graph) and moderately constrained cluster with 4500 CPUs (right graph). The results show that in resource constrained settings, fair-shair allocation policies generate substantially lower fairness indices compared to *Justice*.

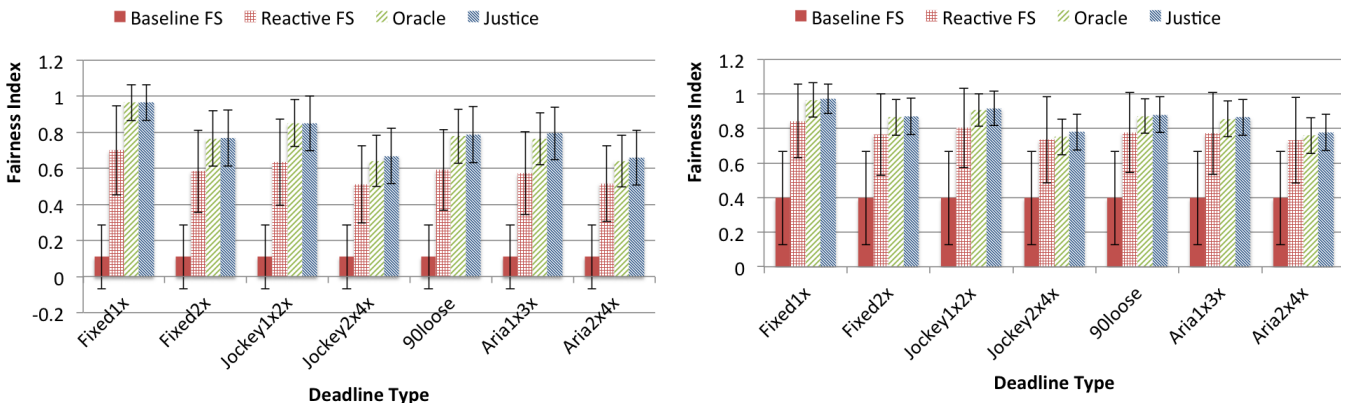
In resource constrained clusters, when CPU demands exceed available cluster resources, fair-share mechanisms can violate fairness. This occurs because these mechanisms do not anticipate the arrival of future workload. Thus jobs that require large fractions of the total resource pool get their full allocations, causing jobs that arrive later to block or to be under-served [11]. Moreover, jobs that are waiting in queue may miss their deadlines while waiting (i.e. receive an A_i value of zero) or receive an under allocation once they are released.

Note that adding the ability to simply drop jobs that have missed their deadlines does not alleviate the fairness problem entirely. The Reactive FS policy (described in Section III) achieves better fairness than the Baseline fair-share scheduler, but does not achieve the same levels as *Justice*. 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* overlap, we also conducted a Welch's t-test [51] for all deadline-types and cluster sizes. 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

[2]We have tested *Justice* on a second trace that contains more than 1 million job entries from the same industry partner. The distribution properties of job sizes are remarkably similar to the trace we have chosen to use. However, because many of the jobs in this larger trace repeat (creating more autocorrelation in the job series), we believe that the smaller trace is a greater challenge for *Justice*'s predictive mechanisms. The results for this larger trace are, indeed, better than the results we present in this paper. We have omitted them for brevity.



(a) Fairness Index with 2250 CPUs

(b) Fairness Index with 4500 CPUs

Fig. 2: **Fairness Evaluation:** Average of Jain's fairness index (and 0.95 error bars) with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph). Experiments denoted as Fixed have deadlines multiples of 1 and 2. Experiments denoted as Jockey have deadline 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 deadline multiples drawn from uniformly distributed intervals [1, 3] and [2, 4]

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 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.

B. Deadline Satisfaction

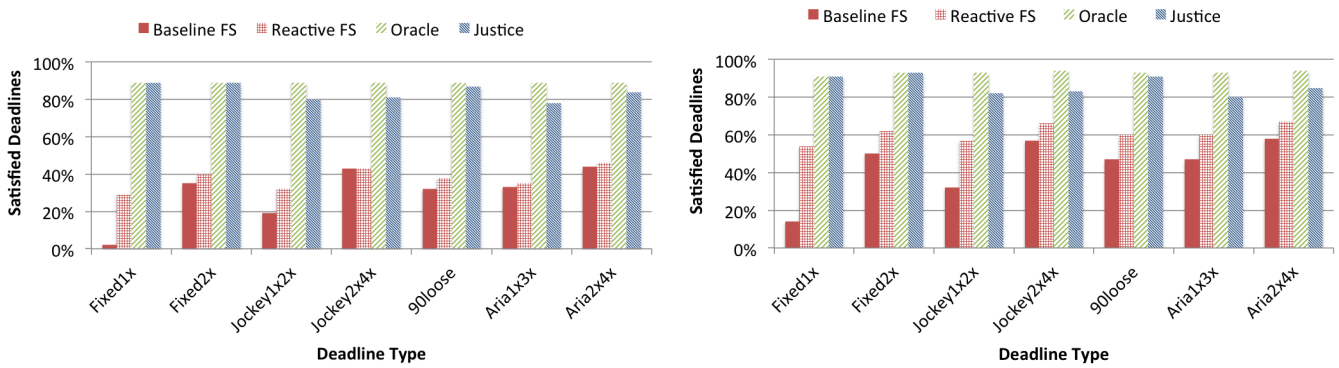
We next evaluate how well the allocators perform in terms of deadline satisfaction. Our goal with this set of experiments is to verify that *Justice* is not simply achieving fairness by dropping a large fraction of jobs – so that those that remain receive a fair allocation.

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 3 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 use of greedy allocation. They allocate as many resources as are available until they run out of resources 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 by their deadline, 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."



(a) Satisfied Deadlines with 2250 CPUs

(b) Satisfied Deadlines with 4500 CPUs

Fig. 3: **Deadline Satisfaction:** Satisfied Deadlines Ratio (SDR) with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph) for different deadline types.

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 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.

C. 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 4 and Figure 5 present PTR and WTR, respectively, for different allocation policies and deadline type for two constrained clusters. 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 cluster. *Justice* outperforms fair-share policies because it proactively drops jobs with violated deadlines and jobs that it predicts are likely to miss their deadline.

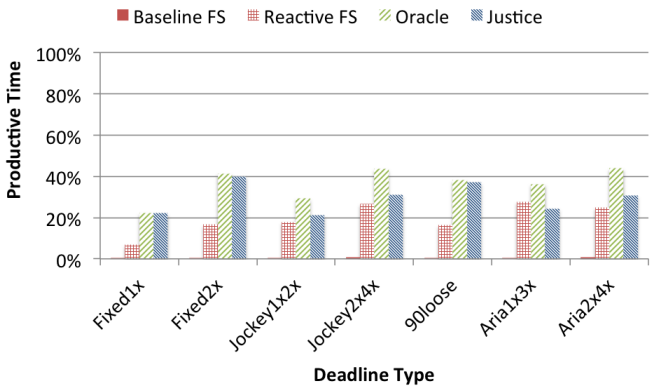
Our experiments also show that the more constrained or utilized the cluster is, the better *Justice* performs in terms of PTR and WTR, relative to the other allocators we consider. Baseline FS fails to satisfy deadlines of bigger jobs because it shares a very limited resources equally between bigger 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 II), as soon as they become infeasible.

When the cluster 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 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 3 is 44% and 33% higher than reactive FS for Jockey1x2x and Aria1x3x respectively).

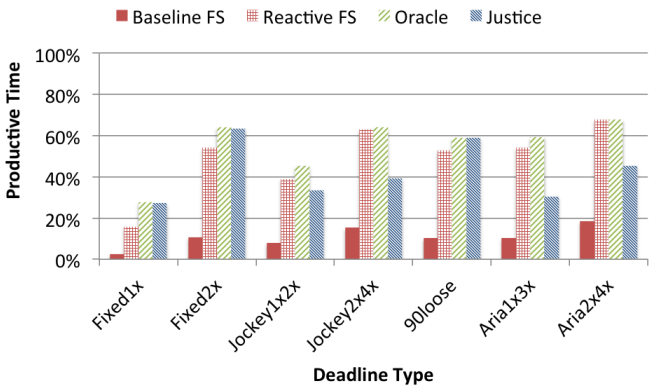
Specifically, as resource scarcity is reduced for a fixed workload, large jobs that are admitted by the Baseline FS and Reactive FS allocators stand a better chance of getting the "extra" resources necessary to complete, and thus, add to the PTR compared to *Justice*, which might have excluded them due 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.

D. Cluster Utilization

The final set of experiments that we perform investigate how allocation policies for resource constrained clusters impact cluster 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 $busy$

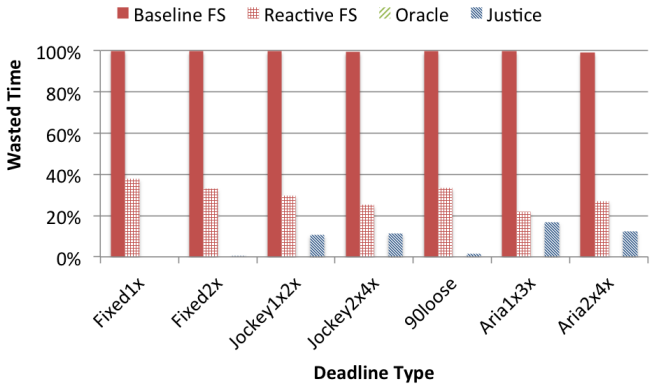


(a) Productive Time with 2250 CPUs

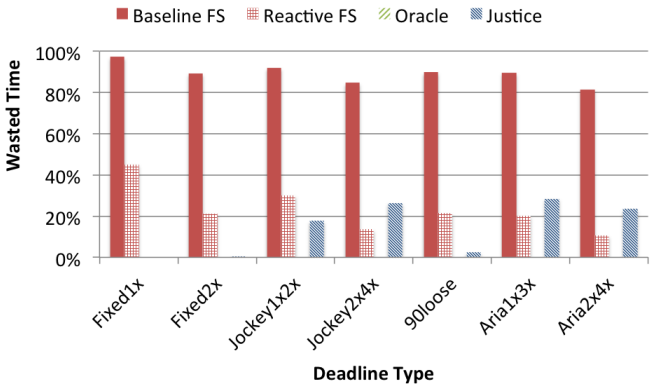


(b) Productive Time with 4500 CPUs

Fig. 4: **Productivity:** Productive Time Ratio (PTR) with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph) for different deadline types.



(a) Wasted Time with 2250 CPUs



(b) Wasted Time with 4500 CPUs

Fig. 5: **Resource Waste:** Wasted Time Ratio (WTR) with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph) for different deadline types. Lower is better.

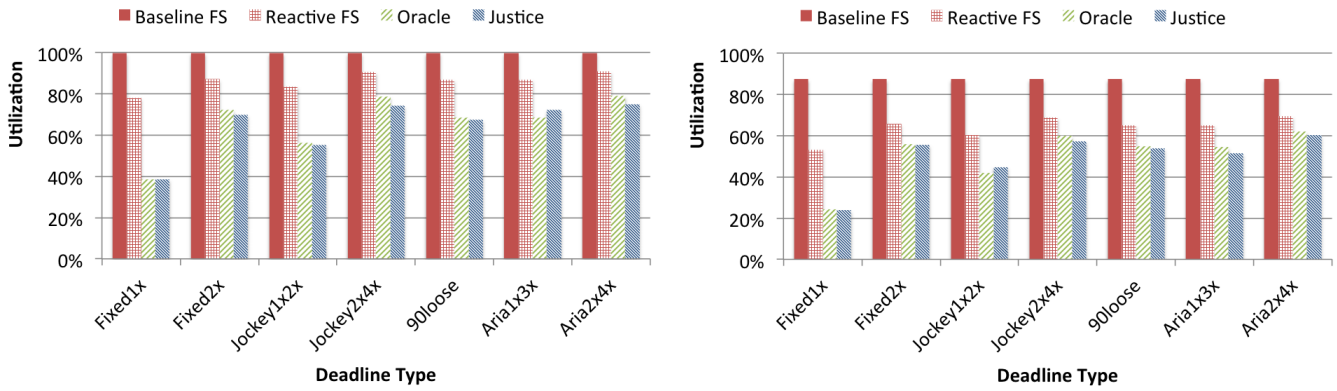
is the total busy time and $idle$ is the total idle time across a workload.

Figure 6 shows the cluster utilization for 2250 CPU (left graph) and 4500 CPU (right graph) cluster sizes, for the different deadline types that we consider. The results in the left graph (2250 CPU) 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 4a and 5a respectively) exhibits higher PTR and lower WTR. So *Justice* enables more productive work with less waste and less cluster utilization. One might assume that the utilization difference is due to less productivity or more overhead but as these results show, the lower utilization is simply because *Justice* does not need the capacity to meet a greater fraction of deadlines (cf Figure 3a) while achieving greater fairness (cf Figure 2a).

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 that correspond to the experiments shown in Figure 6a for the Aria1x3x deadline type. Figure 7 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 trace duration with probabilities 98% and 76% respectively.

We next consider the workload characteristics of the trace that we study (Section IV). We have shown (Figure 1b) that



(a) Utilization with 2250 CPUs

(b) Utilization with 4500 CPUs

Fig. 6: **Cluster Utilization:** Utilization with constrained cluster capacity of 2250 CPUs (left graph) and moderately constrained capacity of 4500 CPUs (right graph) for different deadline types.

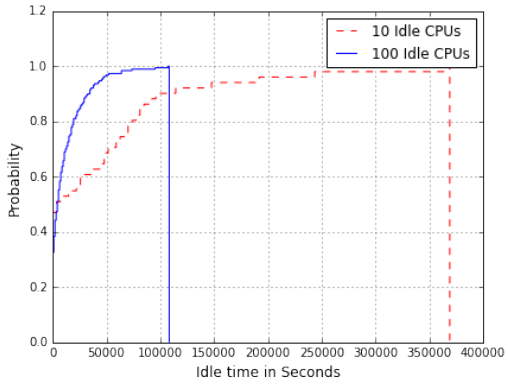


Fig. 7: CDFs of idles times of 10 and 100 CPU groups.

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 cluster capacity for sufficient durations to as to admit significant additional workload. That is, if the trace 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.

VI. Related Work

Sharing on Multi-tenant Resource Allocators: Cluster managers like Mesos [24] and YARN [44] enable the sharing of cluster resources by multiple data processing frameworks. Recent research [13, 21, 39] builds on this sharing, to allow users to run jobs without knowledge of the 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, 6, 15, 20]. However, under resource-constrained cluster settings, the fair-shair policies [15, 20] fail to preserve fairness (Section V-A). Also, all the sharing policies in these works are deadline-agnostic. To meet deadlines, administrators add cluster resources, use a capacity scheduler [6], or require users to reserve resources in advance [3, 8, 43]. Such solutions are costly, inefficient, or impractical, especially 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 [11, 22, 53]. *CARBYNE* [22] attempts to address this issue by exploiting task-level resource requirements information and DAG dependencies. It also uses prior runs of recurring jobs to estimate task demands and durations. Then, it intervenes both at the higher level, on the resource allocator, and internally, on framework task schedulers. It withholds a fraction of job resources from jobs that do not use them while maintaining similar completion times. PYTHIA [12] is addressing the same issue by introducing framework-independent admission control that resource allocators can use to support dynamic fair-sharing of the cluster and deadline driven workflows for either Hadoop or Spark jobs, without requiring task-level information or depending on recurring jobs. 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 resource over-provisioning and preserves fair-sharing on top of satisfying deadlines.

Performance Prediction: In order to allocate the required resources and meet job deadlines, much related work focuses on exploiting historic [9, 25, 30, 31, 47, 48, 55, 57], and runtime [9, 23, 25, 26, 36, 47, 48, 56, 57] job information, while other research [9, 17, 29, 42, 45, 56] 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 [17] shows that some production clusters 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 [7, 17, 32, 35, 37]. 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 [23, 25, 26, 29, 31, 36, 42, 47, 48, 55, 56, 57] or Spark [38, 49]. For this reason, they cannot be used on top of resource managers like Mesos.

Justice in contrast, does not depend on job repetitions and can therefore target clusters with more diverse workloads; it does not impose overheads to perform extensive runtime monitoring or use job sampling and offline simulations to predict performance. *Justice* is also framework-independent because it does not require modeling of the different stages of any particular big data framework. Lastly, unlike all these approaches, *Justice* focuses on satisfying job deadlines in addition to preserving fair-sharing across jobs utilizing the cluster.

Admission Control: Admission control has been suggested as a solution for SaaS providers to effectively utilize their clusters and meet Service Level Agreements (SLAs) [27, 52], to provide map-reduce-as-a-service [10], and to resolve blocking caused by greedy YARN allocations [53]. *Justice* is similar in that but targets multi-analytics, resource-constrained clusters. We design *Justice* for use by resource managers for deadline-driven big data workloads, to be framework and task independent.

VII. Conclusions and Future Work

We present *Justice*, a fair-share and deadline-aware resource allocator with admission control for multi-analytic cluster managers like Mesos and YARN. *Justice* uses historical job statistics and deadline information to automatically adapt its resource allocation and admission control mechanisms to changing workload conditions. By doing so, it is able to estimate the minimum number of resources to allocate to a job to meet its deadline "just-in-time". Thus, it utilizes resources efficiently and satisfies job deadlines while preserving fair-share.

We evaluate *Justice* using trace-based simulation of large production YARN workloads in resource-constrained settings and under different deadline formulations. We compare *Justice* to the existing fair-share allocator that ships with Mesos and YARN and find that *Justice* outperforms it significantly in terms of fairness, deadline satisfaction and efficient resource usage. Unlike, fair-share allocators that, when resources are limited, are unable to adapt their decisions and as a result, violate fairness and waste significant resources for jobs that miss their deadlines, *Justice* monitors the changing cluster conditions and applies admission control to minimize resource waste while preserving resource allocation fairness and meeting more deadlines.

Justice is a practical solution that can work on top of existing open-source resource managers like Mesos and YARN. Its predictions do not depend on the internal structure of the processing engines running on top of the resource manager (e.g., Hadoop, Spark) nor it interacts with them in any way. Therefore, it is easy to maintain as it does not need to adapt to the fast evolution of the processing engines but only to the API changes of the resource manager. Moreover, *Justice* does not rely on job repetitions which make it suitable even for generic workloads that include ad-hoc queries. *Justice* is ideal for the constrained IoT analytics settings, not only because it is optimized to avoid wasting resources for infeasible jobs but also because it does not perform offline simulations, sampling, or extensive online monitoring that would require more computing resources and additional overheads. Lastly, *Justice* does not add complexity, as it only requires minimal information (a deadline and the input size) from the resource manager and the user.

As future work, we are extending *Justice* to support more diverse workloads. We are interested in applications with differing deadline criticality, including hard and soft deadlines, but also applications without deadlines. We also plan to consider infeasible deadlines, i.e., when the cluster cannot meet the deadline the user specified even under ideal allocation, to find ways to incentivize users to assign realistic deadlines and to penalize them when their deadlines are very tight. Moreover, we plan to evaluate our mechanism taking into consideration memory use in addition to CPU, as well as to understand the performance of *Justice* for different types of applications (e.g., CPU Vs I/O bound). Currently, *Justice* does not depend on job repetitions and works well for generalized workloads like the ones we used in this paper. However, for workloads where recurrent jobs constitute a significant amount, exploiting job repetitions patterns might further improve allocation efficiency. Lastly, we plan to evaluate *Justice* with IoT analytics workloads and deploy it in real IoT analytics clusters so we can derive insights from more field-specific characteristics.

VIII. Acknowledgements

This work is funded in part by NSF (CCF-1539586, CNS-1218808, CNS-0905237, ACI-0751315), NIH (1R01EB014877-01), ONR NEEC (N00174-16-C-0020), Huawei Technologies, and the California Energy Commission (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] F. Bonomi et al. Fog computing and its role in the internet of things. In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16.

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

[7] 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.

[8] 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.

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

[10] J. Dhok, N. Maheshwari, and V. Varma. Learning based opportunistic admission control algorithm for MapReduce as a service. In: *India software engineering conference*. ACM. 2010, pp. 153–160.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] H. Feng, V. Misra, and D. Rubenstein. PBS: a unified priority-based scheduler. In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 35. 1. ACM. 2007, pp. 203–214.

[17] 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.

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

[19] P. Garcia Lopez et al. Edge-centric computing: Vision and challenges. In: *ACM SIGCOMM Computer Communication Review* 45.5 (2015), pp. 37–42.

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

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

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

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

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

[25] 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.

[26] 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.

[27] M. Islam et al. Towards provision of quality of service guarantees in job scheduling. In: *IEEE International Conference Cluster Computing*. IEEE. 2004.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] *Moab Workload Manager*. https://en.wikipedia.org/wiki/Moab_Cluster_Suite.

[34] *Open Lava Workload Manager*. http://www.openlava.org/.

[35] 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.

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

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

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

[39] 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.

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

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

[42] 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.

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

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

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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).

[51] *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.

[52] L. Wu, S. K. Garg, and R. Buyya. SLA-based admission control for a Software-as-a-Service provider in Cloud computing environments. In: *Journal of Computer and System Sciences* 78.5 (2012), pp. 1280–1299.

[53] 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.

[54] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[55] 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.

[56] 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.

[57] 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.