2016 IEEE 36th International Conference on Distributed Computing Systems

RUSH: A RobUst ScHeduler to Manage Uncertain Completion-Times in Shared Clouds

Zhe Huang[1], Bharath Balasubramanian[2], Michael Wang[1], Tian Lan[3], Mung Chiang[1], and Danny H.K. Tsang[4]
[1]Princeton University, NJ, USA,
[2]AT&T Labs Research, NJ, USA,
[3]George Washington University, DC, USA
[4]The Hong Kong University of Science & Technology, Sai Kung, Hong Kong
Email: {zheh, mwseven, chiangm}@princeton.edu, bharathb@research.att.com, tlan@gwu.edu, eetsang@ust.hk

***Abstract*—We address the problem of scheduling jobs with utilities that depend solely upon their completion-times in a shared cloud that imposes considerable uncertainty on the jobs' runtime. However, it is very hard to estimate the jobs' runtime in a shared cloud where jobs are often delayed due to reasons such as slow I/O performance and variations in memory availability. Unlike prior works, we acknowledge that runtime estimates are often erroneous and instead shift the burden of robustness to the job scheduler. Specifically, we present a scheduling problem that jointly accounts for: (i) job utilities specified as functions of their completion-time, and (ii) uncertainty in the jobs' runtime. Our proposed solution to this problem achieves lexicographic max-min fairness among the job utilities. We implement this as a robust scheduler, named RUSH, for YARN in Hadoop. Our experiments, using real-world data sets, illustrate RUSH's efficacy when compared with other commonly used schedulers.**

I. Introduction

With the rapid growth in demand for data analytics, shared cloud platforms such as Amazon EC2 and Microsoft's Azure are heavily utilized by various clients to run data processing jobs. This gives rise to challenges in providing individual job performance guarantees, especially regarding job completion-times. These guarantees are very important as data processing requirements become more stringent. Jobs with applications such as augmented reality and video streaming often require quick turn-around times with guaranteed service latency. It is crucial for the scheduler in the data processing platform (like Hadoop [1]) to allocate resources in a manner that acknowledges jobs' relative priorities and sensitivity with respect to completion-times.

The uncertainty in the jobs' runtimes, imposed by the shared infrastructure, is well recognized as a key challenge in designing job scheduling algorithms [2]–[4]. This uncertainty is due to factors such as job heterogeneity (some CPU-heavy while others I/O-heavy), scarcity of resources, and complex dependencies among tasks in the jobs. For example, in a typical Hadoop-MapReduce cluster, the aforementioned reasons for uncertainty may cause a task to run slower than expected, resulting in non-deterministic execution time. In this paper, we address this challenge by solving the following problem: **Given a set of jobs in a shared infrastructure, how do we allocate resources fairly across these jobs, through a systematic optimization framework taking into account: (i) job utilities specified as functions of their completion-time, (ii) uncertainty in the jobs' runtimes?**

The *de-facto* industry standards for schedulers, such as those used in Hadoop, Dryad or Spark [5], [6], are based on simple notions of proportional fair scheduling [7]–[9]. There have been prior works targeting the completion-times or deadlines of Hadoop jobs [3], [4]. These system models assume deterministic task runtimes that are proportional to the resource demand and must be estimated through benchmarking different applications' average task runtime. This approach does not address non-deterministic task runtimes in scheduling and is prone to performance degradation that is invariably caused by uncertainty due to the vagaries of shared infrastructure. Another line of works considers job scheduling with speculative execution to mitigate uncertainty due to task dependencies and data locality [2], [10]–[12]. However, none of them provide any formal guarantees regarding job completion times. On the other hand, in this paper, we focus on the design of an *optimal, robust, completion-time aware scheduler* for a computing cluster that runs a resource management framework such as Apache YARN [13]. In a setting where reliable estimates of jobs' runtime are not available, our RUSH scheduler calculates and allocates cluster resources to active jobs so that their completion time maximizes a lexicographical max-min fairness objective function. In the following paragraphs we provide high-level intuitions behind our design.

Runtime Estimation: The need for accurate runtime estimations to enable efficient job scheduling can be illustrated as follows. Without runtime estimation, one may fail to identify and preempt a low priority job with excessively long runtime. It may create a head-of-line blocking which may unnecessarily cause the subsequent jobs to miss their deadlines. However, job runtime estimation is not a trivial task due to the exceeding number of uncertain factors that may affect the jobs' runtime (e.g., hardware heterogeneity, bottlenecks in multiple-dimensional resources). A practical solution may adopt the black-box approach that learns the empirical runtime behaviors of jobs from historical observations. In our design, a distribution estimator (DE) module will be dynamically created and assigned to each of the jobs to continuously estimate the job's runtime probability distribution using customizable machine learning techniques. The goal of the DE module is to report

1063-6927/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDCS.2016.95

the runtime distributions to the scheduling algorithm from time to time with improving accuracies.

Robust Scheduling: For each submitted job, the corresponding client is only required to specify its relative job priority and utility function of the job's completion time. The scheduling algorithm calculates the approximated total workload according to the job's runtime distribution information reported and performs resource allocation accordingly in order to maximize the client-specified utilities regarding max-min fairness. The robustness in our approach comes from the fact that the proposed scheduling algorithm acknowledges that there exists runtime distribution estimation errors. Instead of directly using the estimates from the runtime distribution estimator modules, the scheduling problem is solved with the allowance that the actual distribution of the jobs' runtimes can diverge from this estimate within a certain threshold.

In summary, we make the following contributions in this paper:

- We frame a robust resource allocation/scheduling problem (RUSH) that accounts for: (i) job utilities as functions of the completion-time and (ii) the uncertainty in the jobs' runtime. The goal of scheduling is to achieve lexicographical max-min fairness among job utilities. We decouple the entire scheduling problem into two sub-problems, i.e., a worst case distribution estimation problem and a time-aware scheduling problem. These two components are part of the continuous runtime feedback cycle which allows the system to recalibrate scheduling decisions [Section II and III].
- We transform the two sub-problems into simpler equivalent problems and propose efficient algorithms to solve both. The proposed solution algorithms are simple and efficient enough to be implemented and integrated into the software stack of resource managers such as YARN [Section III].
- We implement our solution for RUSH in the YARN framework for Hadoop. The implementation is fast, lightweight and has no dependency on any third party optimization software. Since the implementation is directly interfaced with the YARN resource manager, the scheduler can be directly used by any computational cluster as a resource scheduler similar to the other YARN schedulers (e.g., the fair scheduler and the capacity scheduler) [Section IV].
- We examine the performance of our prototype using a Hadoop cluster hosted by our private cloud. The results show that the scheduling decisions made by RUSH are well protected by the built-in runtime estimation and robustness. Further, they illustrate that RUSH is able to perform completion-time aware scheduling to improve the jobs' utility, while requiring a small proportion of completed task samples for its runtime estimation. Finally, our experiments confirm that RUSH consumes a limited amount of CPU and RAM and causes very little overhead in terms of time [Section V].

II. System Model and Problem Formulation

The performance of job scheduling algorithms could suffer from many sources of uncertainty, including unpredictable system perturbations, non-deterministic execution time, random job interference and dynamic arrivals/departures [14]–[16]. In this paper, we propose a robust optimization framework called RUSH, that accounts for various uncertainties through a quantitative model and enables a robust scheduling framework for utility optimization under inaccurate and imperfect information. To measure the users' satisfaction regarding the completion-time of their jobs, a time-dependent utility function is introduced for each job. We assume that the utility functions are non-increasing, as satisfaction should never increase with delays in job completion-time. The proposed RUSH scheduler is designed to provide best scheduling service for cloud environments with the following conditions: **(1) processing batch jobs that consist of tasks that are not heavily correlated, (2) resources in a cloud are packed and apportioned among jobs using a homogeneous unit.** Following by the common practices of YARN, we refer to this homogeneous resource unit as a *container*[1]. Under the above conditions, RUSH's efficacy is the direct result of the fact that tasks runtime behaviors in each job can be estimated with sufficient accuracy as shown later in this paper. Some typical examples of these computing environments includes MapReduce-like systems such as Hadoop and Spark. We will focus on the robust scheduling problem for a given system state in this section and extend the results in Section IV to a dynamic setting where the robust scheduling problem is re-optimized constantly to rectify any previous mistake. Symbols used in our formulations and algorithms are summarized in Table I.

We consider a discrete time model with arbitrary fixed length time slots (e.g., 1 second). As acknowledged in prior works [3], [17], [18], the use of discrete time slots to approximate the continuous time of scheduling is necessary to enable tractable, efficient solutions. We assume that the cloud computing cluster has a capacity of C containers. Denote $\mathcal{N} = 1, ..., N$ to be a set of N user jobs. We introduce a decision variable $x_{i,t}$ as the number of containers assigned to job i at time t. We denote T_i as the completion-time of job i, which mathematically is defined as the time slot that job i receives its last container assignment before completion. Each job i is associated with an arbitrary non-increasing utility function $U_i(T_i)$. Please refer to Section IV for examples of utility function.

To model the uncertainty that impacts jobs' execution time in the cloud, we define a random demand variable v_i to quantify the total number of container time slots needed to finish all tasks of job i. For example, if job i requires 2 containers, each running for 10 time slots, it consumes a total of 20 container time slots. Required container time slots v_i is a random variable, which captures various sources of uncertainty and translates into a random completion-time of

[1]Please note that YARN allow containers with heterogeneous size, which is not considered in this paper.

TABLE I
TABLE OF NOTATION

Symbol	Definition	Symbol	Definition
$x_{i,t}$	number of container(s) assigned to job i at time t	T_i	completion-time of job i
v_i	total number of container time slots requested by job i	$U_i(t)$	utility function of job i
$\omega_i(v_i)$	a distribution of v_i that acts as a decision variable	$\phi_i(v_i)$	reference distribution of v_i
θ	minimum probability of any job i receiving more than v_i containers before T_i	δ_i	entropy threshold for job i
$\phi_{i,l}$	quantized PMF version of $\phi_i(v_i)$	$p_{i,j}$	quantized PMF version of $\omega_i(v_i)$
η_i	total resource demand of job i return by solving the WCDE problem	R_i	average container runtime for job i
C	cluster capacity in terms of number of containers	Δ	tolerance of the bisection algorithm

job i. More precisely, the necessary condition for job i to complete at time T_i is that it receives more than v_i container time before time T_i. Rather than considering the average v_i that is susceptible to uncertainty, or the worst case v_i that is overly conservative, we propose a robust scheduler that focuses on job completion-time with θ-th percentile. In particular, we require that the probability of job i being assigned enough resources and being completed by time T_i to be greater than θ. This robust design enables a range of solutions that offer different levels of robustness.

However, to use this model we need to find the exact distribution of v_i, which poses another challenge for our robust optimization. While the distribution could be numerically estimated from measurements obtained from a real system, such estimated distribution (denoted by $\phi_i(v_i)$) can be no more than an (often rough) approximation due to many forms of systemic uncertainties and time-varying dynamics. Using $\phi_i(v_i)$ directly to measure the θ-th percentile of v_i and T_i may introduce substantial inaccuracy that will propagate to the utility optimization problem. To tackle this issue, we propose another measure of robust optimization. We use $\phi_i(v_i)$ only as a reference distribution and consider a set of possible distributions, denoted as $\{\omega_i(v_i)\}$, that have a Kullback-Leibler (KL) distance (also known as relative entropy) δ_i from the reference point $\phi_i(v_i)$. We refer to δ_i as the "entropy threshold" of job i. A robust utility optimization problem is then formulated to maximize a guaranteed utility over any possible distribution in the set. This requires the robust optimization to be solved over a functional space. The intuition is to guarantee the scheduling performance for all the possible distributions in $\{\omega_i(v_i)\}$. If the exact distribution of v_i is no further than δ_i distance from $\phi_i(v_i)$ and in $\{\omega_i(v_i)\}$, the actual scheduling performance received by the users are captured and guaranteed even we do not know the exact distribution of v_i. However, this robust design significantly complicates the problem and makes it less traceable.

Our Robust Scheduling (RS) problem aims to maximize a lexicographical max-min fairness objective. Let $\mathbf{Y}$ and $\mathbf{W}$ be two N-dimensional vectors in $\mathbb{Z}^N$, and $\overrightarrow{\mathbf{Y}}$ and $\overrightarrow{\mathbf{W}}$ be the corresponding sorted vectors in non-decreasing order. $\mathbf{Y}$ is said to be lexicographically greater than $\mathbf{W}$, denoted by $\mathbf{Y} \succ \mathbf{W}$, if the first non-zero component of $\overrightarrow{\mathbf{Y}} - \overrightarrow{\mathbf{W}}$ is positive. Let $[U_1(T_1), U_2(T_2), ..., U_N(T_N)]$ be the vector of the utility values achieved by all the active jobs, the lexicographical max-min fairness objective is to produce a utility vector that is maximized according to the lexicographic order defined above. Intuitively, it means that RS problem ensures fairness by trying to first maximize the worst utility among jobs, then moving to maximize the second worst utility and so on. On the other hand, if assigning more resources does not help improve a particular job (e.g., an expired job with a minimal utility), the lexicographical max-min objective function will prefer to allocate resources to other jobs because doing so can improve their utility without lowering the utility of this job.

Denote $\mathcal{T}$ to be the set of time slots considered in the time horizon. With the model defined above, the RS problem can be mathematically defined as follows.

$$\text{(RS)}$$

$$\underset{\mathbf{X},\ \{\omega_i(v_i)\}}{\text{lex-max}} \quad [U_1(T_1), U_2(T_2), ..., U_N(T_N)]$$

$$s.t.$$

$$T_i = \max\{t | x_{i,t} > 0, \forall t \in \mathcal{T}\}, \qquad \forall i \in \mathcal{N} \quad (1)$$

$$\sum_{i \in \mathcal{N}} x_{i,t} \leq C, \qquad \forall t \in \mathcal{T} \quad (2)$$

$$\min_{\omega_i(v_i)} \ Pr(v_i \leq \sum_{t \in \mathcal{T}} x_{i,t} | v_i \sim \omega_i(v_i)) \geq \theta, \forall i \in \mathcal{N} \quad (3)$$

$$\int_0^\infty \omega_i(v_i) dv_i = 1, \qquad \forall i \in \mathcal{N} \quad (4)$$

$$\int_0^\infty \left[\ln \frac{\omega_i(v_i)}{\phi_i(v_i)}\right] \omega_i(v_i) dv_i \leq \delta_i, \qquad \forall i \in \mathcal{N} \quad (5)$$

$$x_{i,t} \in \mathbb{Z}^+, \qquad \forall i \in \mathcal{N}, \forall t \in \mathcal{T} \quad (6)$$

$$\omega_i(v_i) \in \{f : \mathbb{Z}^+ \to [0,1]\}, \qquad \forall i \in \mathcal{N}. \quad (7)$$

Constraint (1) defines the job's completion-time. Constraint (2) is the capacity constraint. The minimization over different distribution functions in (3) guarantees that the probability of job i receiving enough resources to complete at T_i is larger than θ even for the worst case distribution (and therefore for all distributions within a δ-distance defined by (4) and (5)). Note that the summation in (3) is equivalently over all $t \in \mathcal{T}$, because constraint (1) ensures that $x_{i,t} = 0$ for any $t > T_i$. Constraint (6) demands that the assignment is performed in the unit of container time slot. Constraint (7) means that function $\omega(v_i)$ participates in the optimization as a decision variable.

Solving the RS problem consists of solving two main sub-problems: (i) a time-aware scheduling (TAS) sub-problem for assigning containers to jobs, associated with constraints (1 - 3), and (ii) a worst case distribution estimation (WCDE) sub-problem associated with constraints (3 - 5). These two sub-

problems are coupled through constraint (3). In the next section, we present our solution method that transforms constraint (3) and decomposes the RS problem into the WCDE and TAS sub-problems.

III. Solution Methods for Robust Scheduling

In this section, we propose a robust, completion-time aware scheduler that collectively maximizes the lexicographical max-min utility of all jobs. The scheduling algorithm takes into account both (i) random runtime time due to system uncertainty and (ii) unpredictable distribution perturbation due to estimation inaccuracy. First, we transform the problem of finding the worst case distribution in (3) into a problem that can be solved efficiently with the bisection method. Second, we develop an onion peeling algorithm that performs robust scheduling with respect to the worst case distribution. It guarantees the optimal completion-time of all jobs that maximizes the lexicographical max-min objective. Finally, we address a practical challenge that each job requires continuous and uninterrupted container assignment for atomic execution. A mapping algorithm with provable optimality is developed to assign continuous container time slots according to the optimal completion-time obtained. **Due to the limited space, the proofs are presented in the technical report [19]**.

A. Solving for the Worst-case Distribution

To solve the RS problem, locating the worst case distribution function $\omega_i(v_i)$ for job i in the minimization (3) is the key. We refer to it as the worst case distribution estimation (WCDE) sub-problem. We show that it is possible to transform the WCDE problem and decouple it among different jobs. Let $\Omega_i^{-1}(x)$ be the inverse CDF of $\omega_i(v_i)$. Due to monotonicity, constraint (3) can be rewritten as $\sum_{t\in\mathcal{T}} x_{i,t} \geq \max \Omega_i^{-1}(\theta)$. This implies that the processes of locating the worst case distribution function among all the jobs can be decoupled, and each job only has to compute the value of $\max \Omega_i^{-1}(\theta)$ satisfying constraints (4) and (5). To facilitate the computation, we replace the continuous PDF $\omega_i(v_i)$ with a discrete PMF through quantization into a finite number of bins in a range of $[0, \tau_{max}]$. Let $p_{i,l}$ be the probability that v_i falls in bin l according to distribution $\omega_i(v_i)$. We further define $\phi_{i,l}$ to be the quantized PMF of the reference distribution $\phi_i(v_i)$. The WCDE problem can be transformed as follows.

$$
\begin{aligned}
&\text{(WCDE)}\\
&\max_{p_{i,l}} \quad \Omega_i^{-1}(\theta)\\
s.t.\quad &\\
&\sum_{l=0}^{\tau_{max}} \left[\ln \frac{p_{i,l}}{\phi_{i,l}}\right] p_{i,l} \leq \delta_i, \qquad (8)\\
&\sum_{l=0}^{\tau_{max}} p_{i,l} = 1,\\
&p_{i,l} \in \mathbb{R}^+, \qquad \forall l \in [0, \tau_{max}] \cap \mathbb{Z}^+.
\end{aligned}
$$

In the new WCDE problem, the objective $\Omega_i^{-1}(\theta)$ depends on the underlying probability distribution $p_{i,l}$. We leverage the monotonicity of $\Omega_i^{-1}(\theta)$ to develop an efficient bisection search method, which in each iteration considers a target objective value L and evaluates its feasibility. The bisection search is summarized in Algorithm 2. To evaluate the feasibility of a given L in the bisection search, we notice that $\Omega_i^{-1}(\theta) \geq L$ implies $\theta \geq \Omega_i(L) = \sum_{l=0}^{L} p_{i,l}$ because Ω_i is a CDF. An objective value L is feasible only if there exists a distribution $p_{i,l}$ satisfying $\theta \geq \sum_{l=0}^{L} p_{i,l}$ as well as the relative entropy constraint (8). This can be easily verified by solving a Relative Entropy Minimization (REM) problem as follows:

$$
\begin{aligned}
&\text{(REM)}\\
&\min_{p_{i,l}} \quad \sum_{l=0}^{\tau_{max}} \left[\ln \frac{p_{i,l}}{\phi_{i,l}}\right] p_{i,l}\\
s.t.\quad &\\
&\sum_{l=0}^{\tau_{max}} p_{i,l} = 1, \qquad (9)\\
&\sum_{l=0}^{L} p_{i,l} \leq \theta, \qquad (10)\\
&p_{i,l} \in \mathbb{R}^+, \qquad \forall l \in \{0, ..., \tau_{max}\}.
\end{aligned}
$$

If the problem returns an optimal value smaller than δ_i, we know that the corresponding target objective L must be feasible for the WCDE problem.

Now it only remains to solve the REM problem. It is easy to see that the REM problem minimizes a convex function over linear constraints. The problem can be solved in closed-form using its KKT conditions. Let μ and ν be two Lagrangian multipliers for constraints (9) and (10), respectively. We can directly obtain the optimal solution from the KKT conditions:

$$P_{i,l} = \frac{\phi_{i,l}}{e^{1+\mu+\nu}},\ \forall l \leq L \quad \text{and} \quad P_{i,l} = \frac{\phi_{i,l}}{e^{1+\mu}}, \quad \text{otherwise.} \qquad (11)$$

Equation (11) implies that the optimal solution to REM problem partitions $\{P_{i,l}\}$ into 2 groups (for $l \leq L$ and $l > L$) and assigns to each group a normalized version of $\phi_{i,l}$. To determine the Lagrangian multipliers, we can use the fact that constraint (9) must hold and that $\nu = 0$ if constraint (10) is a strict inequality due to the slackness condition. Therefore, we propose a closed-form method as shown in Algorithm 1 to determine the appropriate normalization factors, $1/e^{1+\mu+\nu}$ and $1/e^{1+\mu}$, based on these constraints. It has a time complexity of $O(1)$ and is guaranteed to find a feasible solution to the KKT conditions, thus an optimal solution for the REM problem.

Theorem 1. *The closed-form method solves the REM problem optimally.*

Using Algorithm 1, we can evaluate the feasibility of any given L in the bisection search. To initialize the bisection method, we can pick the largest possible utility value among all jobs as an upper bound and use a feasible solution $p_{i,l} = \phi_i(v_i)$ to obtain a lower bound in the WCDE problem.

Algorithm 1 Closed-Form Method

Input: Target objective value L
Calculate $p_{i,l} = \phi_{i,l} / \sum_{l=0}^{T_{max}} \phi_{i,l}$ for all l
if $p_{i,l}$ violates (10) **then**
  Calculate $p_{i,l} = \frac{\theta \phi_{i,l}}{\sum_{l=0}^{L} \phi_{i,l}}$ for $l \in [0, L]$
  Calculate $p_{i,l} = \frac{(1-\theta) \phi_{i,l}}{\sum_{l=L+1}^{T_{max}} \phi_{i,l}}$ for $l \in [L+1, T_{max}]$
end if
return $p_{i,l}$

Algorithm 2 WCDE Bisection Search

Initiate $L_f = \Phi_i^{-1}(\theta)$ and $L_i = \max_{i,x}(U_i(x))$.
while $L_i - L_f > 1$ **do**
  Select $L = \lfloor \frac{1}{2}(L_i + L_f) \rfloor$
  Calculate $p_{i,l}^*$ using Algorithm 1 with input L
  Set $L_f = L$ if $\sum_{l=0}^{\tau_{max}} \left[\ln \frac{p_{i,l}^*}{\phi_{i,l}} \right] p_{i,l}^* \leq \delta_i$
  Else, set $L_i = L$
end while

B. Solving the Robust Scheduling Problem

By solving the WCDE problem, the optimal value of $\max \Omega_i^{-1}(\theta)$ is obtained for each job. To simplify the presentation, we use η_i to represents this value for job i. Therefore, the Time-Aware Scheduling (TAS) problem now becomes a deterministic problem with deadline constraints:

$$\begin{aligned}
&\text{(TAS)} \\
&\underset{\mathbf{x}}{\text{lex-max}} \quad [U_1(T_1), U_2(T_2), ..., U_N(T_N)] \\
s.t. \\
&T_i = \max\{t | x_{i,t} > 0, \forall t \in \mathcal{T}\}, && \forall i \in \mathcal{N} \\
&\sum_{i \in \mathcal{N}} x_{i,t} \leq C, && \forall t \in \mathcal{T} \\
&\sum_{t \in \mathcal{T}} x_{i,t} \geq \eta_i, && \forall i \in \mathcal{N} \\
&x_{i,t} \in \mathbb{Z}^+, && \forall i \in \mathcal{N}, \forall t \in \mathcal{T}.
\end{aligned}$$

In our previous work [3] we have shown that the above TAS problem can be transformed and efficiently solved using linear programming techniques (e.g., simplex method). However, considering that multiple decision variables are introduced for each job at each time slot in the linear programming solution, the large number of decision variables introduced by a large number of jobs may deem the performance of linear programming method unsatisfactory. We propose a novel solution using techniques that we refer to as onion peeling and continuous time slot mapping which allow very fast convergence even as the size of the problem becomes large.

To solve the TAS problem, we use the onion peeling method to determine the optimal target completion-time for jobs one by one. Then the continuous time slot mapping method will produce a detailed container assignment. Note that the lexicographical max-min objective is maximizing the minimum utility of jobs "layer" by "layer". In each layer, a job that reaches the maximum possible utility is identified and will not participate in the optimization in the next layer. Without loss of generality, we now consider the first layer problem in which the optimization is simply maximizing the minimum utility among all the jobs (i.e., max-min problem). Because the jobs' utility functions are non-increasing as completion-time grows, and also because all other constraints are convex, the max-min problem can be again solved using bisection method. Denote $U_i^{-1}(\cdot)$ to be the inverse of job i's utility function. Given any target max-min objective value L, each job i has to finish before time $U_i^{-1}(L)$ in order for all jobs to collectively achieve a max-min utility no less than L. Therefore, in each iteration of the bisection algorithm, to test whether a target utility value L is feasible or not, one just need to examine whether a set of completion-time $U_i^{-1}(L)$ is feasible for jobs $i = 1, \ldots, N$. To simplify the representation, without loss of generality, we sort the jobs according to their $U_i^{-1}(L)$ values. We further define $\mathcal{N}_k$ to be the set of first k jobs. To test feasibility of $U_i^{-1}(L)$, we only need to examine whether there exist enough resources (i.e., container time slots) in the system to support each desired completion-time $U_i^{-1}(L)$, i.e.,

$$\sum_{i \in \mathcal{N}_k} \eta_i \leq C \cdot U_k^{-1}(L), \forall k \in 1, 2, ...N \tag{12}$$

Theorem 2. *When condition (12) is satisfied, there exists a feasible resource scheduling $\{x_{i,t}\}$ to finish each job i before time $U_k^{-1}(L)$, thus attaining a utility value at least L.*

With Theorem 2, the design of the bisection algorithm is straightforward. Moreover, for each infeasible L value during the process of bisection search, there exists at least one infeasible job that cannot meet its completion-time in (12). The infeasible jobs corresponding to the last infeasible L before the bisection search terminates are clearly bottlenecks for further utility maximization. In other words, these jobs have already achieved their optimal utility in this layer and cannot be further improved. Therefore, only the remaining jobs should proceed to the next layer in our lexicographical max-min optimization, while (12) is examined with consideration of resource allocated previously. We repeat this process to identify the completion-time of bottleneck jobs one by one, and hence the name "peeling an onion". Let Δ be the tolerance of the bisection algorithm. Pseudocode shown in Algorithm 3 summarizes the onion peeling algorithm.

C. Mapping the Solution to a Practical Assignment

We now address another important constraint that arises from resource scheduling in real systems, i.e., resource allocation of any single task has to be continuous and uninterrupted. Unlike the assumption of infinitely-divisible workload made in many prior works, this constraint means that when a task receives a container, this assignment has to be continuous before the container is released. Therefore, the workload of a task carried out by a container cannot be further divided.

Algorithm 3 Onion Peeling Algorithm

Initiate $\mathcal{N} = \{1, 2, ...N\}$ and $G_t = 0, \forall t \in \mathcal{T}$
Set $L_f = \min_{i,x}(U_i(x))$
while $\mathcal{N}$ is not empty **do**
  Set $L_i = \max_{i,x}(U_i(x)|i \in \mathcal{N})$.
  while $L_i - L_f > \Delta$ **do**
    Select $L = \frac{1}{2}(L_i + L_f)$
    if $\sum_{i \in \mathcal{N}_k} \eta_i + G_{U_k^{-1}(L)} \leq CU_k^{-1}(L),\ \forall k \in \mathcal{N}$ **then**
      Set $L_f = L$
    else
      Set $L_i = L$
    end if
  end while
  Locate the bottleneck job i
  Assign $T_i = U_i^{-1}(L_f)$ as its target completion-time
  Set $G_t = G_t + \eta_i,\ \forall t \geq T_i$
  Remove i from $\mathcal{N}$
end while

After solving the utility maximization to get a set of optimal target completion-time, we create a practical container assignment $\{x_{i,t}\}\ \forall i, t$ that achieves the completion-time and satisfies the continuity requirement. Let R_i denote the average container runtime for job i, reported by the distribution estimator. Job i should be assigned consecutive time slots in an integer multiple of R_i, in order for each task to continuously occupy a container until it finishes. Let T_i be the target completion-time obtained by the onion peeling method. Due to the continuity constraint, a practical container assignment may further extend the job's completion-time beyond T_i. However, the following theorem shows that it is possible to find a practical solution satisfying the continuity constraint while achieving a completion-time very close to T_i.

Theorem 3. *There exists a continuous container assignment scheme that achieves a job completion-time no later than $T_i + R_i$ for all the jobs.*

Theorem 3 shows that the actual achievable job completion-time under the continuity constraint is upper bounded by $T_i + R_i$. The extra delay is no more than R_i, the average container holding time by a single task, which is relatively small compared to the overall runtime of the job. More importantly, we can compensate for this extra time by reducing the time budget of each job in the onion peeling method by R_i. This allows the target completion-time produced by the onion peeling method to be implementable in practice.

To summarize, first, jobs are ordered by their target completion-time obtained by the onion peeling method. We maintain C number of queues. When performing assignment for a job i, the operation always starts from the first queue. The total workload of η_i is assigned to the current queue in the unit of R_i until the current queue occupation is larger than T_i. After the assignment, if a job has residual workload, the next queue is used to continue the process. We repeat this process until all the jobs are assigned. Please note that this approach may stretch the workload of a job beyond the target job completion-time. The condition that we switch to another queue guarantees that the jobs' completion-time is upper bounded by $T_i + R_i$. The above continuous time slot mapping is summarized as Algorithm 4.

Algorithm 4 Continuous Time Slot Mapping

Input: Target completion-time $\{T_1, ...T_N\}$
Input: Estimated workload $\{\eta_1, ...\eta_N\}$
Input: Average container runtime $\{R_1, ...R_N\}$
Initiate occupation of queues $O_k = 0, \forall k \in \{1, ...C\}$
for every job $i \in \mathcal{N}$ **do**
  Select the first queue, set $k = 1$
  while $\eta_i \neq 0$ **do**
    Assign $A = \min(\lfloor \frac{T_i - O_k}{R_i} \rfloor, \eta_i)$ to queue k
    Update $\eta_i = \eta_i - A$ and $O_k = O_k + A$
    Select the next queue, set $k = k + 1$
  end while
end for

IV. RUSH-YARN Architecture and Implementation

In this section, we describe the architecture and implementation of our robust scheduling algorithm as a scheduler in the YARN Hadoop framework. The RUSH scheduler has three main components: (i) a job configuration interface, (ii) a distribution estimator (DE) unit for each job and (iii) a container assignment (CA) unit. A new job's requirements are submitted through the configuration user interface and the CA unit is triggered whenever there is an empty container in the system. When the CA unit is triggered, it obtains the estimated total workload for all the jobs from the corresponding DE units, calculates and assigns the containers to each of them based on our algorithms. The CA unit repeats this in a feedback cycle as containers are freed, tasks finish and job utilities are updated. We illustrate the RUSH-YARN architecture in Figure 1 and explain it in further detail in the following paragraphs.

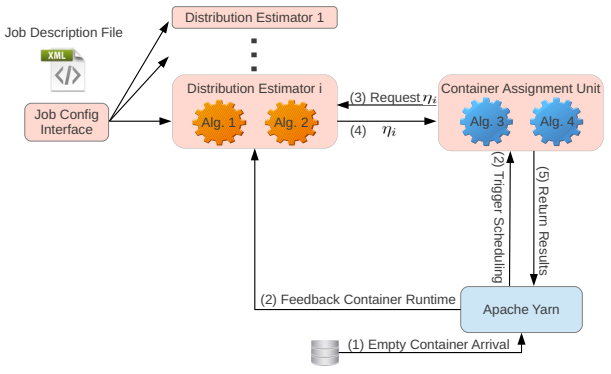

Fig. 1. RUSH-YARN architecture describing the feedback cycle of estimation, recalculation of container assignment and YARN integration for actual allocation of containers to jobs.

Job Configuration User Interface: When a job is submitted to the YARN Hadoop cluster, an XML file with its requirements such as time budget B, priority value W and utility value sensitivity β is submitted through this interface. Currently, we provides three utility classes: (1) piece-wise linear; (2) sigmoid; and (3) constant. We encourage the users to submit their own utility classes that best describe their quality of service requirements. Given a completion-time T, the linear utility class produces a utility value of $\max(\beta(B-T)+W, 0)$. This utility class represents completion-time sensitive jobs whose utility decays linearly along with a further completion-time. The sigmoid class has a more versatile utility function of $W/(1+e^{\beta(B-T)})$, where the sensitivity coefficient β can determine if a job is time-sensitive or time-critical. It determines how steep the utility drops when the job's completion-time exceeds the time budget.

Distribution Estimator (DE): The main task of the DE unit is to estimate the reference total workload distribution $\phi(v_i)$ and produce η_i. Algorithm 2 is efficient with a time complexity being logarithmic in the number of bins that the estimator chooses to describe the PMF of the reference distribution. Currently, we provide two distribution estimator classes: (i) The mean time estimator, that reports an impulse distribution at the bin equals to the multiple of the mean container runtime and the number of pending tasks. (ii) The Gaussian estimator, that learns the sample mean and variance from the runtime statistical data received from YARN. Based on the central limit theorem, the estimator returns a Gaussian distribution with sampled mean and sampled variance multiplied by the number of pending tasks.

Container Assignment (CA): The CA unit is responsible for solving the TAS problem using the onion peeling and the continuous time slot mapping algorithms to produce scheduling decisions. When at least one container is empty in the system, the CA unit first obtains the new workload estimates, η_i, for each job from the DE unit. The onion peeling algorithm is then used to estimate the job completion-time, following which, the continuous time slot mapping method is used to produce the actual container assignment. However, only the container assignment for the next time slot will be used since every time a scheduling event is triggered, the feedback cycle is repeated and a new assignment is calculated. When the decision for the next time slot is obtained, it is compared with the number of containers occupied by each job, and the CA unit selects a job that has the largest difference between the new and old assignments to assign the available container.

YARN Integration: We integrate RUSH with the YARN (version 2.6) by interfacing the CA unit described above with YARN resource manager (code available in [19]). This allows the job-container assignment prescribed by CA to be applied by the resource manager. Since RUSH is integrated with the resource manager of YARN, RUSH obtains first hand statistic information of all the jobs and resources. This direct integration with YARN, bypassing any third party software makes our prototype light-weight and stable. Figure 2 shows the enhanced HTTP management interface for the RUSH scheduler that is able to provide a projected completion-time for all the jobs. More importantly, when a job cannot be finished without its utility dropping to zero, the interface highlights this job (e.g., the red row in Figure 2), thereby alerting the user to reconfigure job requirements by submitting a new job configuration XML file.

Fig. 2. RUSH-YARN enhanced HTTP interface that shows jobs' target completion-time and highlights impossible jobs in red (zoom in for details).

V. Experimental Results

In this section, we present results evaluating RUSH on a YARN Hadoop cluster with real-world data sets. To examine the performance of RUSH, we construct a v2.6 Hadoop cluster with 48 vCPU cores and 48 GB of RAM. The Hadoop cluster is built using six virtual machines hosted on heterogeneous servers including two Dell R320 servers with Intel E5-2470v2 CPU at 2.7GHz, two Dell T320 servers with Intel E5-2470 CPU at 2.3GHz, and two Dell Optiplex desktop computers with Intel i5-3470 CPU at 3.2GHz.

A. Robustness in the Distribution Estimation

Robustness in scheduling is a fundamental part of our work. In this section, we present results evaluating the DE unit described previously. First, we recap the definitions of δ_i and θ. The DE unit estimates the worst-case workload distribution for each job i that is at most δ_i KL distance away from the reference distribution. A larger δ_i allows the DE unit more margin in selecting the worst-case distribution and makes the scheduling decisions more conservative. The θ-th percentile requires the scheduler to allocate more than v_i among of container time slots with a probability higher than θ. We try to answer two questions: (i) how many task runtime samples does the DE unit need to produce meaningful distribution estimation results? (ii) how confident are we in these results?

We construct a Hadoop job with 100 map tasks and 1 reduce task. Each of these MapReduce tasks lasts for a time that is randomly generated using a Gaussian distribution with a mean of 60 seconds and a standard deviation of 20. Since we know the distribution of the task execution times *a priori*, we can evaluate the efficacy of our distribution estimator in comparison with this ground truth. This job is submitted to the Hadoop cluster repeatedly for 100 times. During each run, the Gaussian distribution estimator is used to produce the reference distribution. In Figure 3, we plot the probability of the η_i value produced by the DE unit being larger than

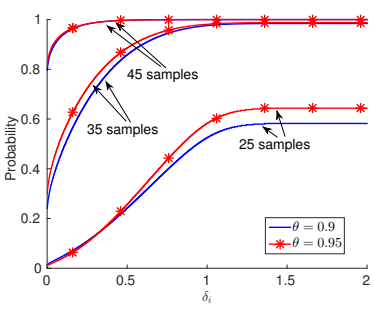

Fig. 3. The probability of the estimated total demand being larger than the random demand posed by job i. To meet the θ-th percentile constraint, the DE unit needs more than 35 runtime samples and an entropy threshold value larger than 0.7. **The DE unit is able to produce an accurate distribution estimation after 35% of the tasks are completed.**

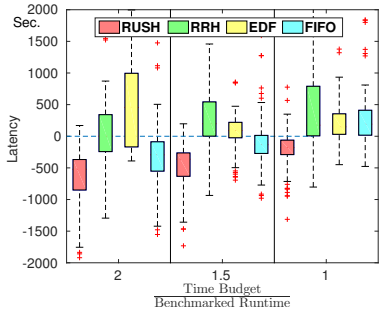

Fig. 4. Latency of jobs with different time budgets. The third quartile bar of the RUSH boxes is always lower than 0. It indicates that 75% of the jobs are able to finish before their time budgets expire. **When the time budgets are reduced, RUSH sacrifices a small number of jobs to maintain the fairness among the majority of the jobs.**

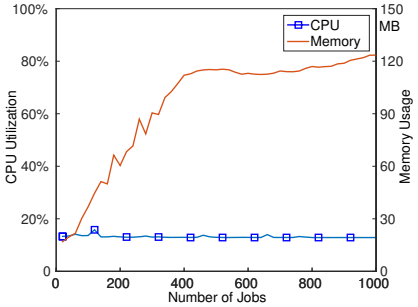

Fig. 5. CPU and memory consumption of RUSH. The constantly low CPU usage shows that the algorithm complexity is very limited. The memory consumption even for 1000 jobs is less than 130 MB. **The results show that RUSH is very lightweight.**

the random workload v_i. We can evaluate this probability because the distribution of v_i is known. When the DE unit only has 25 samples, the produced η_i fails to cover v_i with a probability of θ no matter how conservative δ_i is. This is because the DE unit does not have enough samples to construct an accurate distribution estimation. However, when we have more than 35 samples, a δ_i larger than 0.7 achieves the goal. This implies that among the 101 MapReduce tasks, the DE unit is able to construct an accurate enough reference distribution after the first 35 tasks are completed. The RUSH scheduler has the rest of the 101 MapReduce tasks to correct any scheduling mistakes made previously. More samples also allow the scheduling decision to be less conservative by selecting a smaller entropy threshold δ_i value.

B. Time-awareness and the Utility Performance

In this section, we present results comparing the timeliness and utilities achieved by RUSH with the following commonly used scheduling techniques that are time-aware: (i) **FIFO** scheduling, in which jobs are scheduled according to the order of their arrival time. (ii) **Earliest-deadline-first** (EDF) scheduling, in which jobs are scheduled according to the order of their time budget. (iii) **Risk-reward-heuristic** (RRH) scheduling [20], in which scheduling decisions are made based on the future utility gain and opportunity cost of reallocating resources. The FIFO scheduler, being the default scheduler used in most Hadoop implementations, is an important point of comparison. The EDF scheduling is the optimal policy for meeting the jobs' deadlines when jobs are queued in a preemptive queue [21]. However, it does not consider the completion-time sensitivity of jobs. To the best of our knowledge, the RRH scheduler is the only job scheduler that considers the different utility sensitivities of the jobs' completion-time. Other non-time-aware schedulers such as the fair scheduler [8] are not considered because these schedulers do not capture the jobs' utility value according to their completion time.

To closely emulate a real-world Hadoop cluster, 100 jobs are created from an equal mix of eight heterogeneous Hadoop job templates (Movie Classification, Histogram of Movies, Histogram of Ratings, InvertedIndex, SelfJoin, SequenceCount, WordCount and Terabyte Data Sorting) with multiple real-world data sets from the PUMA benchmark suite [22]. These jobs are submitted to the cluster according to a Poisson arrival process with mean arrivial time of 130 seconds. To randomize the runtime of the jobs, each job is assigned with a data set whose size is randomly selected between 1 and 10 GB. The Gaussian distribution estimator and the Sigmoid job utility class are used. The job's priority W is randomly selected from 1 to 5. 20% of the jobs are time critical jobs whose utility drops rapidly if the final runtime pass a target completion-time. 60% of the jobs are time sensitive jobs whose utility drops gradually. The rest of the 20% jobs are time insensitive. The runtime of each job is benchmarked with all the resources available in the cluster. We repeat our experiments for cases in which the time budget of jobs is varied from 2, 1.5 and 1 times of the benchmarked runtime. The scheduling for these experiments become more challenging because the deadlines of the jobs are made tighter.

We define latency as the difference between the actual job runtime and the time budget. Figure 4 shows the box-plot statistic information on the latency for jobs that are completion-time sensitive and critical. As we reduce the time budget from 2.5 times to 1 time the benchmarked time, the scheduling problem becomes harder and the performance of all the schedulers worsens. Please note that RUSH is able to finish the majority of the completion-time sensitive and critical jobs even when the jobs' time budget is equal to the benchmarked time. This is because RUSH is able to delay the execution of the completion-time insensitive jobs (not shown in the figure) and a very limited number of completion-time sensitive jobs. EDF and FIFO fail to finish the completion-time sensitive and critical jobs before their time budgets are depleted because they do not capture the completion-time sensitivities among jobs and spend unnecessary resources on executing completion-time insensitive jobs. The fact that EDF and FIFO only execute one job at a time creates head-of-line blocking which also prevent completion-time sensitive

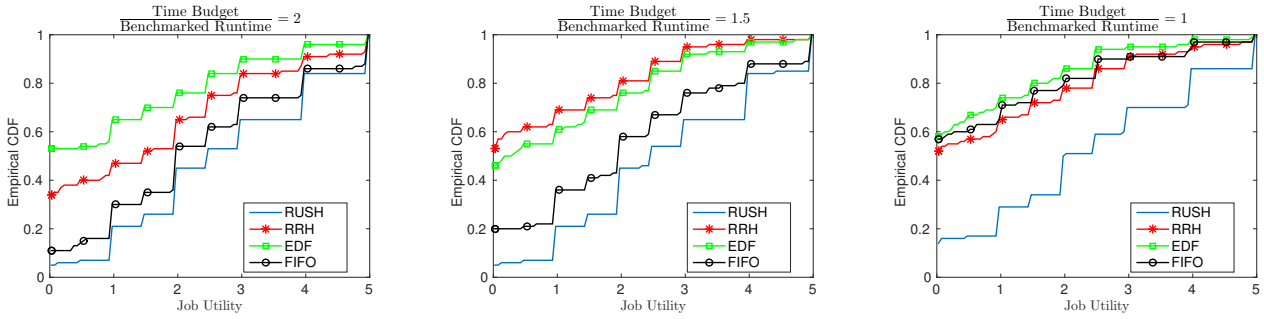

Fig. 6. Distributions of 100 jobs' utilities with different jobs' time budget values. **They demonstrate that RUSH in general achieves much better utility values for the jobs because of the lexicographic max-min fairness.**

and critical jobs from finishing before their deadline. RRH is not able to complete the completion-time sensitive jobs because the design of RRH favors heavily the completion-time critical jobs. As a result, a few completion-time critical jobs are completed much earlier than their corresponding deadline (shown as the lower outliers in the RRH box) at the cost of the completion-time sensitive jobs.

Finally, we illustrate that RUSH is able to achieve much better utility values with its robust scheduling design and lexicographic max-min fairness when compared to the other schedulers. Figure 6 shows the CDF of the jobs' utilities achieved by the examined schedulers. Since the lexicographic max-min fairness achieved by the RUSH scheduler not only maximizes the minimum utility, but also iteratively maximizes the next lowest utility and so forth, the majority of the jobs receive a higher utility value. We show three graphs each of which with a different ratio of the time budget to benchmarked runtime. As this ratio gets smaller, scheduling becomes harder. RUSH consistently outperforms all other schedulers. This is reflected by RUSH shifting the CDF curves of jobs' utilities to the right hand side for all the three graphs, an effect that is more pronounced as the ratio of time budget to benchmarked runtime decreases. Please note that the other schedulers fail to finish a considerable number of jobs before their deadlines, resulting in those jobs receiving zero utility. For example, in the third graph, more than 50% of the jobs for the other schedulers receive zero utility. RUSH successfully minimizes the number of jobs which receive zero utility because the lexicographic max-min fairness always tries its best to complete jobs before their deadlines.

C. Resource Consumption and Execution Time

In this experiment, we measure the resource consumption and execution time of RUSH hosted by a virtual machine with 8 vCPU cores and 8 GB of memory. Multiple WordCount jobs with random configurations are submitted to create scheduling events that have simultaneous jobs varied from 20 to 1000. Each experiment is repeated 1000 times. Figure 5 shows that RUSH consumes only 15% CPU cycles and less than 130 MB of memory on average. The average algorithm runtime increases linearly from 0.32 second to 7.34 seconds. These results confirm that RUSH is both light-weight and efficient.

VI. Related Work

Scheduling is a fundamental problem for Hadoop-like systems that share resources among jobs [23]. Task schedulers in data-processing systems like Hadoop are simple (e.g., FIFO, Fair Scheduler [7], [8], Capacity Scheduler [9]) and usually ignore completion-time. Multiple recent works in this area [24], [25] have addressed job deadlines. For example, Quincy [24] prevents jobs from missing their deadline by enforcing that a job which runs for t seconds is given exclusive access to a cluster no more than Jt seconds when there are J jobs concurrently executing on that cluster. Similarly, [25] evicts low priority jobs in order to minimize the chance of high priority jobs missing the deadline. The above completion-time aware approaches have very specific heuristic rules that have no notion of optimality or overall fairness. There also exists prior works [2], [10]–[12], [26], [27] that perform task scheduling in data processing clusters with the considerations of dependencies among tasks such as data locality and network-awareness. However, these approaches require the scheduler to know the dependencies and their effects on the jobs' completion-time. Further, they do not make any formal guarantees regarding jobs' completion-time.

Real-time and high performance computing systems do model the completion-time awareness using individual job's utility [20], [28]–[30], with different job deadline types (hard, soft etc.). For example, the authors of [20] model the jobs' utility using a piece-wise linear function and scheduling decisions are made based on the future utility gain and opportunity cost of reallocating resources. Unlike RUSH, these approaches do not provide any guarantees on optimality. They do not handle a wide range of utility functions and being heuristics, and they are not robust in terms of scheduling. While [3] optimally schedules for completion-time through job utilities, as mentioned in the introduction, it relies on benchmarking for job runtime estimates which is prone to error. In contrast, our main focus is on robustness in scheduling for jobs' completion-time.

While robust scheduling with uncertainty is a well studied problem [14]–[16], these proposals are not suitable for practical systems because their solutions are not specifically designed for Hadoop-like systems. The complexity of their

solutions make it unclear how they can be applied to optimize the completion-time in compute clusters.

In our solution, the workload and runtime distribution is captured by the very efficient distribution estimator. There is a body of work in recent years that focuses on the online estimation of job's runtime. For example, [4], [31] both apply linear regression method to perform runtime estimation for Hadoop jobs. These techniques can be implemented as distribution estimation classes and integrated into our system.

VII. Conclusion

Allocating resources among the jobs to maximize the total utility in a fair manner is a challenging problem. This is mainly because of the inherent unpredictability in a shared infrastructure that necessitates scheduling algorithms that need to be re-calibrated multiple times during the life-time of a job, while taking into account the job and cluster heterogeneity. We address these challenges and present RUSH, which performs robust, completion-time aware scheduling. Our scheduler is well integrated with the Hadoop YARN framework and is ready for wide-deployment. The experimental results indicate that RUSH can perform completion-time aware scheduling in a fair, efficient and optimal manner. To further improve the robustness of the scheduler, we plan to include the estimation of task failure probability in our future work.

References

[1] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[2] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[3] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. Tsang, "Need for speed: Cora scheduler for optimizing completion-times in the cloud," in *2015 IEEE Conference on Computer Communications (INFOCOM)*.

[4] A. Verma, L. Cherkasova, and R. Campbell, "Resource provisioning framework for mapreduce jobs with performance goals," in *Middleware 2011* (F. Kon and A.-M. Kermarrec, eds.), vol. 7049 of *Lecture Notes in Computer Science*, pp. 165–186, Springer Berlin Heidelberg, 2011.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pp. 59–72, 2007.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[7] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, pp. 265–278, ACM, 2010.

[8] "Hadoop Fair Scheduler." https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.

[9] "Hadoop Capacity Scheduler." https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html.

[10] X. Zhang, Y. Feng, S. Feng, J. Fan, and Z. Ming, "An effective data locality aware task scheduling method for mapreduce framework in heterogeneous environments," in *2011 International Conference on Cloud and Service Computing (CSC)*, pp. 235–242, Dec 2011.

[11] E. Arslan, M. Shekhar, and T. Kosar, "Locality and network-aware reduce task scheduling for data-intensive applications," in *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*, pp. 17–24, IEEE Press, 2014.

[12] T.-Y. Chen, H.-W. Wei, M.-F. Wei, Y.-J. Chen, T. sheng Hsu, and W.-K. Shih, "Lasa: A locality-aware scheduling algorithm for hadoop-mapreduce resource assignment," in *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pp. 342–346, May 2013.

[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.

[14] C. Artigues, R. Leus, and F. Talla Nobibon, "Robust optimization for resource-constrained project scheduling with uncertain activity durations," *Flexible Services and Manufacturing Journal*, vol. 25, no. 1-2, pp. 175–205, 2013.

[15] W. Herroelen and R. Leus, "Robust and reactive project scheduling: a review and classification of procedures," *International Journal of Production Research*, vol. 42, no. 8, pp. 1599–1620, 2004.

[16] L. Blni and D. C. Marinescu, "Robust scheduling of metaprograms," *Journal of Scheduling*, vol. 5, no. 5, pp. 395–412, 2002.

[17] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "A throughput optimal algorithm for map task scheduling in mapreduce with data locality," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 33–42, Apr. 2013.

[18] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," in *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, pp. 1609–1617, IEEE, 2013.

[19] "Rush Technical Report." https://goo.gl/QLAt2q.

[20] D. E. Irwin, L. E. Grit, and J. S. Chase, "Balancing risk and reward in a market-based task service," in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04, (Washington, DC, USA), pp. 160–169, 2004.

[21] J. Kleinberg and É. Tardos, *Algorithm Design*. Alternative Etext Formats, Pearson/Addison-Wesley, 2006.

[22] "Puma: Purdue Mapreduce benchmarks suite." https://sites.google.com/site/farazahmad/pumabenchmarks.

[23] B. Rao and D. L.S.S.Reddy, "Article: Survey on improved scheduling in hadoop mapreduce in cloud environments," *International Journal of Computer Applications*, vol. 34, pp. 29–33, November 2011. Published by Foundation of Computer Science, New York, USA.

[24] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 261–276, ACM, 2009.

[25] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 6:1–6:17, ACM, 2013.

[26] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri.," in *OSDI*, vol. 10, p. 24, 2010.

[27] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu, "Mimp: deadline and interference aware scheduling of hadoop virtual machines," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pp. 394–403, IEEE, 2014.

[28] B. Ravindran, E. D. Jensen, and P. Li, "On recent advances in time/utility function real-time scheduling and resource management," *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, vol. 0, pp. 55–60, 2005.

[29] L. D. Briceno, B. Khemka, H. J. Siegel, A. A. Maciejewski, C. Groer, G. Koenig, G. Okonski, and S. Poole, "Time utility functions for modeling and evaluating resource allocations in a heterogeneous computing system," *2013 IEEE International Symposium on Parallel and Distributed Processing*, vol. 0, pp. 7–19, 2011.

[30] A. Auyoung, L. Grit, J. Wiener, and J. Wilkes, "Service contracts and aggregate utility functions," in *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, pp. 119–131, 2006.

[31] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang, "Hadoop performance modeling for job estimation and resource provisioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, pp. 1–1, 2015.