Research Article

PAS: Performance-Aware Job Scheduling for Big Data Processing Systems

Yiren Li, 1,2 Tieke Li, 1 Pei Shen, 2 Liang Hao, 2 Jin Yang, 3 Zhengtong Zhang, 3 Junhao Chen, 3 and Liang Bao 3

1 School of Economics and Management, University of Science and Technology Beijing, Beijing 100083, China
2 HBSI Group Co., Ltd., Shijiazhuang 050023, China
3 School of Computer Science and Technology, Xidian University, Xi’an, Shaanxi 750001, China

Correspondence should be addressed to Jin Yang; yangjin@stu.xidian.edu.cn

Received 21 October 2021; Revised 20 February 2022; Accepted 7 March 2022; Published 12 April 2022

Copyright © 2022 Yiren Li et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Big data analytics has become increasingly vital in many modern enterprise applications such as user profiling and business process optimization. Today’s big data processing systems, such as Hadoop MapReduce, Spark, and Hive, treat big data applications as a batch of jobs for scheduling. Existing schedulers in production systems often maintain fair allocation without considering application performance and resource utilization simultaneously. It is challenging to perform job scheduling in big data systems to achieve both low turnaround time and high resource utilization due to the high complexity in data processing logics and the dynamic variation in workloads. In this article, we propose a performance-aware scheduler, referred to as PAS, which dynamically schedules big data jobs in Hadoop YARN and Spark and autonomously adjusts scheduling policies to improve application performance and resource utilization. Specifically, PAS schedules multiple concurrent jobs using different policies based on the predicted job completion time and employs a greedy approach and a one-step lookahead strategy to opportunistically maximize the average job performance while still maintaining a satisfactory level of resource utilization. We implement PAS in Hadoop YARN and evaluate its performance with HiBench, a well-known big data processing benchmark. Experimental results show that PAS reduces the average turnaround time by 25% and the makespan by 15% in comparison with four state-of-the-art schedulers.

1. Introduction

With the prosperity of big data analytics and artificial intelligence, big data processing systems (BDPSs) are playing critical roles in modern enterprises’ applications. The job scheduler continues to be a key component to a BDPS, in which diverse coexisting jobs from many users and applications contend for resources in a shared environment. As the data volume increases and the demand for analytics jobs surges, typical production BDPSs are frequently resource-constrained. Therefore, efficient resource management comes as the top priority for cluster schedulers [1]. Recently, due to the prevalence of “data analysis as a service” (DAAS), the BDPSs running on the public cloud are providing data analysis abilities for different users.

To balance interests of both users and service providers, performance (from user side) and resource efficiency (from provider side) need to be concerned simultaneously for the system design on those shared environments [2]. User-side performance is often measured using average turnaround time, which is the average time intervals from the time of each submission of their job to the time of its completion. Provider-side resource efficiency indicates the efficiency of the resource usage and it is usually measured with the resource utilization ratio and makespan of a set of jobs. These objectives are opposite, because of the conflict of interest between users and the service providers. Current production schedulers often settle for isolation guarantee as the primary objective and seek to maintain fair allocations at all time, which is neither necessary nor efficient [1].
Motivated by this intuition, in this article, we propose a performance-aware scheduling algorithm to opportunistically improve the average job performance for fast job completion for users while still achieving good enough resource efficiency for service providers. To this end, we develop a performance prediction method for estimating completion time of a data analytical job under different resource utilization conditions, formulate a job scheduling problem for big data processing systems (JS-BDPS) in order to minimize average turnaround time under a user-specified resource utilization constraint, and propose a performance-aware scheduling solution. Specifically, our work makes the following contributions to the field:

1. The method for performance prediction is based on generic computer and program models, which provide an accurate estimation of completion time for big data analytical jobs and make it directly applicable to different jobs running on various big data processing systems.

2. The formulation of the JS-BDPS problem and the performance-aware scheduling approach that employs a greedy and a one-step lookahead strategy to solve the JS-BDPS problem effectively and efficiently.

3. The performance prediction method is validated and justified by experimental results using a well-known big data benchmark on disparate computing nodes, and the performance superiority of the proposed PAS scheduling approach is illustrated by extensive simulation results in comparison with four state-of-the-art algorithms.

The rest of the article is organized as follows. We discuss the related work in Section 2. Section 3 formulates the JS-BDPS problem and discuss the objectives and constraints. In Section 5, we construct the regression model of a job for performance estimation and propose the performance-aware scheduling (PAS) policies to minimize the average turnaround time for a set of scheduling jobs. Section 6 presents the design and implementation of the PAS algorithm on top of Hadoop YARN and Spark. In Section 7, we describe the experimental setup and evaluates the performance model and scheduling algorithm using a well-known big data benchmark. We conclude with a discussion of our approach and a sketch of future work in Section 8.

2. Related Work

Job scheduling for BDPSs has received much attention from both industry and academia. From the perspective of scheduling goal, previous studies can be classified into two categories: performance-oriented and fairness-oriented approaches, as discussed next.

2.1. Performance-Oriented Scheduling. Maximizing resource utilization and minimizing makespan are two common goals for performance-oriented scheduling [2]. Yao et al. [3] proposed HaSTE to improve the resource utilization by using efficient task packing according to diverse resource requirements and dependencies between tasks. Their later work proposed OpERA [4] to leverage the knowledge of actual runtime resource utilizations as well as future resource availability for task assignments. Quasar [5] used classification techniques to find appropriate resource allocations to applications in order to fulfill their QoS requirements and maximize system resource utilization. Polo et al. [6] dynamically adjusted slots on each machine to maximize the cluster utilization. Verma et al. [7] allocated resource to jobs by using job profiles to estimate the requested resource that meets the deadline. Cheng et al. [8] proposed a deep reinforcement learning (DRL)-based job scheduler that dispatches the jobs in real time to deal with real-time workloads. Fan et al. [9] proposed an intelligent scheduling framework for different hardware resources and increasing diverse workloads in modern job scheduling. Zheng et al. [10] designed an online algorithm for SaaS providers to optimally purchase IaaS instances and schedule pleasingly parallel job.

To minimize the makespan of a set of independent MapReduce jobs, Verma et al. [11] introduced a heuristic scheduling algorithm. Huang et al. [12] tried to optimize the makespan by estimating the completion time of jobs that are prone to error. Hou et al. [13] proposed a deadline-aware scheduling algorithm to reduce the average job execution time by checking the percentage of tasks and allocating resources. Wang et al. [14] proposed a workflow-based scheduling algorithm to satisfy the budget and the deadline constraints. Khan et al. [15] applied linear regression method to perform runtime estimation for Hadoop jobs. Lim et al. [16] proposed CP-Scheduler to estimate task execution time and handle MapReduce jobs with deadlines. Shao et al. [17] proposed an energy-aware greedy algorithm (EAGA) for fine-grained task placement to minimize the energy consumption and job execution time. Chen et al. [18] observed that with demand elasticity, a job requires significantly less amount of resources, only at the cost of a moderate performance penalty. Amer et al. [19] tackled the multi-objective scheduling problem and presents a modified Harris hawks optimizer (HHO) for multi-objective scheduling problem. Khan et al. [20] proposed a task scheduling method based on a hybrid optimization algorithm, which effectively schedules jobs with the least amount of waiting time. Meyer et al. [21] proposed a machine learning-driven classification scheme for dynamic interference-aware resource scheduling in cloud computing environments. They presented a classification approach to better represents the workload variations for resource scheduling. Chen et al. [22] considered the heterogeneous characteristics of data centers and modeled energy consumption based on the frequency and kernel number of the virtual machine CPU.

2.2. Fairness-Oriented Scheduling. Fairness is another important factor for a scheduling framework. Matei et al. [23] proposed a delay scheduling policy to improve the performance of Fair Scheduler by increasing the data locality of Hadoop. Dominant resource fairness (DRF) [24] is the first work to generalize the max-min fairness to multiple resource
types on Hadoop YARN. Wang et al. [25] extended the DRF algorithm for the heterogeneous environment. Many production schedulers, such as Hadoop’s Fair Scheduler [26], Quincy [27], Mesos [28], and Choosy [29], support max-min fairness or its extensions. Liu et al. [30] presented a resource allocation mechanism to enable fair sharing multiple types of resource among multiple tenants. Huang et al. [31] calculated the approximated total workload according to the job’s runtime distribution and performed resource allocation accordingly in order to maximize the client-specified utilities regarding max-min fairness. Wang et al. [32] corrected the monopolizing behavior of long reduce tasks from large jobs and dynamically balanced the execution of different jobs for fair and fast completion.

Some other studies consider the trade-off between performance and fairness simultaneously. In [33], a general meta-scheduler to leverage existing schedulers in Hadoop YARN to implement the efficiency-fairness trade-off was proposed. Wang et al. [34] utilized many metrics to efficiently balance the performance and the fairness, as well as to reduce the makespan of MapReduce tasks. Tang et al. [35] presented DynamicMR, a dynamic Hadoop slot allocation (DHSA) framework aiming to improve the performance of MapReduce workloads while maintaining the fairness. Pastorelli et al. [36] presented HFSP, a size-based scheduler with aging to implement fairness and near-optimal system response times on Hadoop. Niu et al. [37] presented an adaptive scheduler called Gemini, which adaptively decides the proper scheduling policy according to the running workload, in order to achieve better performance as well as fairness.

3. Problem Statement

This article focuses on the job scheduling problem for big data processing systems (short for JS-BDPS problem). Generally, the architecture of scheduling frameworks of BDPSs can be centralized, centralized two-level, distributed two-level, or shared-state [2]. This article focuses on the centralized two-level (CTL) architecture because it is easy to implement and can generate an optimal or near-optimal scheduling plan. Many practical scheduling frameworks, such as Yarn [38], Mesos [28], Fuxi [39], Teris [40], and Corral [41], have adopted the CTL architecture. In a typical scheduling framework with the CTL architecture, the scheduler is responsible for allocating resources to the various running jobs subject to constraints of capacities, queues, etc.

The goal of JS-BDPS problem is to find an optimal scheduling policy that minimize the average turnaround time (ATAT) of the submitted jobs within a predefined time period, given a set of jobs, and the underlying runtime environment. The reason we choose the turnaround time metric is because it can tell how long a job can take to finish execution since it arrives at the scheduler, which characterizes a most important capability of a scheduler. Other measurements such as energy, cost, makespan, and load balancing are integrated by turnaround time in some ways [2].

Notably, the JS-BDPS problem has the following components:

3.1. Job. A job represents a big data processing application running on a specific BDPS. We model it as a 4-tuple \( j = \langle f, r, t^w, t^e \rangle \), where \( f \) denotes the processing logic of \( j \); \( r \) represents the required resource for running \( j \); \( t^w \) records the waiting time, that is the accumulated queuing time for scheduling, of \( j \); and \( t^e \) indicates the execution time of \( j \) red and assumes the jobs obtain the same priority. Once submitted, jobs need to be queued for a while until they are scheduled. Thereafter, the scheduler assigns the necessary resource \( r \) to \( j \) and \( j \) will be continuously executed in the BDPS until completion.

3.2. Resource. A cluster of BDPS often contains different types of resource, that is, CPU, memory, and network bandwidth, that a job needs for executing, and a job usually need to describe the necessary resource it required to initiate its execution. For example, a MapReduce job running on a YARN scheduler has to declare the number of CPU core and the memory size when submitting. Without loss of generality, we model the resource requirement of a job \( j \) as a 2-tuple \( r = \langle c, m \rangle \), where \( c \) is the number of CPU cores and \( m \) denotes the memory size, and \( R = \langle \langle C, M \rangle \rangle \) is used to indicate the total resource held by a BDPS cluster.

3.3. Turnaround Time. Turnaround time (TAT) is an important metric in evaluating the scheduling algorithms from the users’ perspective [2]. For a specific job \( j_i \), its turnaround time \( TAT_i \) is the time interval from the time of \( j_i \)’s submission to the time of the \( j_i \)’s completion, that is \( TAT_i = t^w_i + t^e_i \).

3.4. Resource Utilization Ratio. The resource utilization ratio (RUR) is an important performance metric that is used to measure the efficiency of a BDPS cluster from the service providers’ perspective [33]. Given any time point \( t \), the RUR of a BDPS cluster is defined as a 2-tuple:

\[
RUR = \frac{R_{\text{used}}}{R} = \left\langle \frac{C_{\text{used}}}{C}, \frac{M_{\text{used}}}{M} \right\rangle,
\]

where \( R_{\text{used}} \) is the already used resource of the cluster at \( t \) and \( R \) is the total resource of the cluster.

In summary, given a set of \( n \) jobs \( J = \{ j_1, j_2, \ldots, j_n \} \) submitted to the scheduler, the JS-BDPS problem can be stated as follows:

\[
\text{min } ATAT = \frac{1}{n} \sum_{i=1}^{n} TAT_i,
\]

subject to \( L \leq RUR^t \leq U, \forall t \).

where equation (2) states that the goal of JS-BDPS problem is to find the optimal job scheduling sequence that minimizes the average turnaround time (ATAT) of \( J \). The constraint (2) is that at any time point \( t \), the RUR of any solution must be
within $[L, U]$, the resource utilization bound predefined by the BDPS.

4. Discussion

4.1. Performance-Aware Job Scheduling. The key idea of our performance-aware job scheduling (PAS) approach is to train a performance prediction model for each job according to their historical profiles and repeatedly assign resource to the job(s) with the lowest cost gain using the trained model. PAS also applies a heuristic one-step lookahead strategy to find potentially good scheduling policy.

4.2. Predicting Job Completion Time. A BDPS can support many data analytical jobs running on it simultaneously. We observe that the completion time of a specific job $j$ varies under different resource utilization ratio ($RUR$) of the BDPS. Suppose the completion time when executing it exclusively, that is $RUR_j = 0$, on a BDPS. If $RUR$ increases to $RUR_j > 0$, by adding more jobs to the BDPS, at each time slot $\Delta t$ there are statistically $(1 - RUR_j) \cdot \Delta t$ share of processing time can be used for executing $j$. Furthermore, when $RUR_j$ continues to grow, the swapping and scheduling time among different jobs becomes significantly large [42]. Based on these observations and the job complexity estimation method proposed in [43], we model the job completion time $t$ and the resource utilization ratio ($RUR$) as a power-law function:

$$t = a \cdot RUR^b + c, \quad (3)$$

where $a$, $b$, and $c$ are regression constants.

The prediction model indicates that when resource utilization ratio grows, the completion time of a job grows in a power number. Such conclusion has been well verified in our experiments.

4.3. Scheduling Gain and ATAT. To schedule a set of jobs dynamically, PAS has to make decision periodically. More specifically, at each time point $t$, it needs to decide whether to schedule a set of jobs $J$ from the queue (for decreasing the $t_w$ and $t_e$ of all jobs in $J$), or to do nothing (for decreasing the execution time of all running jobs). To lucubrate the scheduling process, we discrete the time into many small and equal slots, that is $\Delta t$, and denote the $k$th time interval as $t^k = [t_0 + (k-1)\Delta t, t_0 + k\Delta t]$, where $t_0$ represents the start time of scheduling. Note that the discretized time slot $\Delta t$ is a hyperparameter of PAS that is configurable by the administrator of the BDPS, and it defines the scheduling frequency of the PAS.

Based on the discretized time slot, we can define the scheduling gain to quantify the gain by scheduling a set of jobs:

**Definition 1.** Scheduling gain at $t^k$, denoted by $G^k$, is defined as the decrease of the waiting time for scheduling a subset of submitted jobs ($J^s_k$) minus the increase of the execution time for the running jobs ($J^r_k$) caused by the newly scheduled and added jobs ($J^a_k$). $G^k$ can be formulated by:

$$G^k = \Delta t \cdot \left[ |J^s_k| + \sum_{j \in J^r_k \cup J^a_k} \frac{T(j, R^0_{used})}{T(j, R^k_{used})} - \sum_{j \in J^s_k} \frac{T(j, R^0_{used})}{T(j, R^k_{used})} \right]$$

where $R^0_{used}$, $R^{k-1}_{used}$, and $R^k_{used}$ denote the used resource at $t^0$, $t^{k-1}$, and $t^k$, respectively; $R^k_{used} = R^{k-1}_{used} + \sum_{j \in J^a_k} (r_t)$.

Based on the definition of scheduling gain, we can define the relationship between $G^k$ and ATAT:

**Theorem 1.** Given a time period with $N$ time slots for running a set of jobs $J$, the average value of accumulated scheduling gain (AASG), that is $\sum_{k=1}^N G^k/|J|$, equals to the average turnaround time (ATAT) of $J$.

**Proof.** By definition, the ATAT can be formulated as:

$$\text{ATAT} = \frac{\sum_{i=1}^N t^w_i + t^e_i}{|J|} \quad (5)$$

where the average value of accumulated scheduling gain is defined as:

$$\text{AASG} = \frac{\sum_{k=1}^N G^k}{|J|} \quad (6)$$

Comparing equations (5) and (7), we have

$$\sum_{i \in J^w} t^w_i = \sum_{k=1}^N \Delta t \cdot |J^w_k| \quad \text{(8)}$$

$$\sum_{i \in J^e} t^e_i = \sum_{k=1}^N \Delta t \cdot \frac{T(j, R^0_{used})}{T(j, R^k_{used})} \quad \text{(9)}$$

Finally, we have AASG = ATAT. \qed

We can see from Theorem 1 that maximizing AASG equals to minimizing ATAT, one of the ultimate goal of our JS-BDPS problem, as defined in equation (2). In order to maximize AASG, we apply the greedy strategy in this article: at any time slot $t^k \in N$, we try to find the scheduling job set $J^s_k$ that can maximize $G^k$.

4.4. Small Job First Policy. To minimize the ATAT while achieving a better resource efficiency, the PAS algorithm tries to utilize as much resource as possible for running more
jobs at any time slot $r^k$ on the one hand, and on the other hand it needs to compare the $G^k$ values under different scheduling job sets and choose the scheduling sets with the best value of $G^k$. Given a fixed resource utilization ratio $RUR^k$ at $r^k$ and a set of jobs $J^k$ that needs to be scheduled, all possible number of scheduling job sets equals to $2^{|J^k|}$ at the worst case. To reduce the searching space of this process, we propose a small job first policy, defined as follows:

**Theorem 2.** Given any two valid scheduling job sets $J^k_1$ and $J^k_2$ if $|J^k_1| \geq |J^k_2|$ and $\sum_{j \in J^k_1} (r_j) = \sum_{j \in J^k_2} (r_j)$, then we have $G^k_1 \geq G^k_2$.

**Proof.** According to the Definition 1, the scheduling gain $G^k$ over $J^k_1$ is defined as:

$$G^k_{J^k_1} - G^k_{J^k_2} = \Delta t \cdot \left[ \left| \frac{|J^k_1|}{|J^k_2|} \right| - 1 \right] + \Delta \left( \sum_{j \in J^k_1 \cup J^k_2} \frac{T(j, R^k_{\text{used}})}{T(j, R^k_{\text{used}})} - \sum_{j \in J^k_1 \cup J^k_2} \frac{T(j, R^k_{\text{used}})}{T(j, R^k_{\text{used}})} \right)$$

Because $\sum_{j \in J^k_1} (r_j) = \sum_{j \in J^k_2} (r_j)$, we have $R^k_{\text{used}} = R^k_{\text{used}}$. Hence, we substitute the second subtraction part of equation (12) with equation (14) and draw the final conclusion that the scheduling gain $G^k$ over $J^k_1$ is greater than or equal to the scheduling gain $G^k_2$ over $J^k_2$.

**Theorem 2.** indicates that given a fixed resource $r$ we can use for scheduling at any time slot $r^k$, the scheduler should always schedule as many jobs as it can to minimize the ATAT. In another word, the small jobs with less resource requirements should be scheduled first, we call this strategy the small job first (SmJF) policy.

### 4.5. One-Step Lookahead

After using SmJF policy, we can still obtain many scheduling job sets having the same size and the same $G^k$ value. In this section, we propose a one-step lookahead (OSLA) policy that can find the scheduling job set(s) to achieve the possibly smaller ATAT value in the future, which is defined as follows:

**Definition 2.** One-step lookahead (OSLA) policy: Suppose the further scheduling policy is shortest job first (SJF) after $r^k$. According to the assumption, we have $|J^k_1| = |J^k_2|$, $G^k_{J^k_1} = G^k_{J^k_2}$. Let the following scheduling sequence for $J^k_1$ is $S_1$, $G^k_{J^k_1} = G^k_{J^k_2}$.

Similarly, the $G^k_{J^k_1}$ over $J^k_2$ is defined as:

$$G^k_{J^k_2} = \Delta t \cdot \left[ \left| \frac{|J^k_2|}{|J^k_1|} \right| - 1 \right] + \Delta \left( \sum_{j \in J^k_1 \cup J^k_2} \frac{T(j, R^k_{\text{used}})}{T(j, R^k_{\text{used}})} - \sum_{j \in J^k_1 \cup J^k_2} \frac{T(j, R^k_{\text{used}})}{T(j, R^k_{\text{used}})} \right)$$

Thus, $G^k_{J^k_1} - G^k_{J^k_2}$ equals:

$$G^k_{J^k_1} - G^k_{J^k_2} = \Delta t \cdot \left[ \left| \frac{|J^k_1|}{|J^k_2|} \right| - 1 \right] + \Delta \left( \sum_{j \in J^k_1 \cup J^k_2} \frac{T(j, R^k_{\text{used}})}{T(j, R^k_{\text{used}})} - \sum_{j \in J^k_1 \cup J^k_2} \frac{T(j, R^k_{\text{used}})}{T(j, R^k_{\text{used}})} \right)$$

and for $J^k_1$ is $S_2$, the average turnaround time under $S_1$ is $ATAT_{S_1}$ and under $S_2$ is $ATAT_{S_2}$, if $ATAT_{S_1} \leq ATAT_{S_2}$, we should choose $J^k_1$ as our scheduling job set.

Definition 2 indicates that at any time slot $r^k$, if two candidate job sets $J^k_1$ and $J^k_2$ have the same size and the same $G^k$ value, our scheduler should lookahead for one step and apply a simple shortest job first (SJF) policy $J^k_1$ and $J^k_2$ to generate two scheduling sequences $S_1$ and $S_2$, respectively. Thereafter, we simulatively schedule $J^k_1$ under $S_1$ and $J^k_2$ under $S_2$ separately and estimate the $ATAT_{S_1}$ and $ATAT_{S_2}$. Finally, we choose the job set with the smaller $ATAT$ value. The reason why we use the simple OSLA policy here is because the following scheduling process is dynamic and too complex to predict—an approximate and greedy policy works well under this circumstance.

### 5. Algorithm Design and Implementation

Based on the small job first policy (SmJF) and the one-step lookahead (OSLA) policy, we can implement our PAS algorithm. The detailed process is specified in Algorithm 1.

PAS initially sets the available resource $R'$ to upper bound $B$ (line 1) and starts to try different resource utilization values decreasingly (line 3 and 5). At each iteration, it applies the SmJF algorithm to select job sets according to $R'$ (line 4). After trying all possible $R'$, we choose the candidate scheduling job set(s) with the best scheduling gain value (line 7) and apply the OSLA algorithm to find the final scheduling job set $J^k$ (line 8).

#### 5.1. Small Job First Algorithm

As Theorem 2 shows, given a fixed resource utilization value $R'$, we should greedily choose jobs with smaller resource demand first to decrease the waiting time for these jobs without severely increasing the
time ATAT job first (SJF) policy (line 6). Once \( S_r \) contains jobs with the same tries to add all jobs in the bin \( S_{JF} \) jobs in to their resource demands compare the updated resource utilization \( R_c \) other jobs in algorithm is described in Algorithm 2.

Algorithm 1: PAS algorithm: \( \text{PAS} (J^k, R^k_{\text{used}}, [L,U]) \).

```
Require: \( J^k \): the job set waiting for scheduling at \( t^k \); \( R^k_{\text{used}} \): the used resource at \( t^k \); \([L,U]\): the resource boundary.
Ensure: \( J^k_s \): scheduling job set at \( t^k \).
(1) \( R' \leftarrow U \);
(2) \( J' \leftarrow \emptyset \);
(3) while \( R' \geq R^k_{\text{used}} \) do
(4) \( J' \leftarrow J' \cup \text{SmJF}(J^k, R') \);
(5) \( R' \leftarrow R' - \epsilon \);
(6) end while
(7) \( J^k_s \leftarrow \arg \max (\text{SG}(J')) ;
(8) \( J^k_s \leftarrow \text{OSLA}(J^k_s, J^k_{\text{max}}, R^k_{\text{used}}, U) \); return \( J^k_s \);
```

running time for the already running jobs. The \( \text{SmJF} \) algorithm is described in Algorithm 2.

\( \text{SmJF} \) algorithm starts by sorting the jobs in \( J^k \) according to their resource demands \( r \) in the increasing order to generate a bin list \( B = \{B_1, B_2, \ldots, B_l \} \), in which each bin contains jobs with the same \( r \) (line 3). At each iteration step \( i \), \( \text{SmJF} \) tries to add all jobs in the bin \( B_1 \) first (line 5) and compares the updated resource utilization \( R_c \) with the target value \( R' \) (line 6). If \( R_c \) is still smaller than or equal to \( R' \), we can now safely add all jobs in \( B_1 \) to the scheduling job set \( J^k_s \) (line 7) and move to the next iteration. If \( R_c \) is already greater than \( R' \), we have to release the resource possessed by jobs in \( B_1 \) first and recalculate the desirable number of jobs \( n \) (line 9). We should then enumerate all possible subsets of \( B_1 \) containing \( n \) jobs (line 10). The algorithm ends after adding subsets of some \( B_i \) in \( B \) to \( J^k_s \) (line 11), or finally adding all jobs in \( J^k \) to \( J^k_s \) (line 13).

5.2. One-Step Lookahead Algorithm. Given each candidate scheduling set \( J_i \) in \( J^k_s \), the OSLA algorithm first appends all jobs in \( J_i \) to the scheduling sequence \( S \) (line 2–5). Thereafter, other jobs in \( J^k_s \) are appended to \( S \) according to the shortest job first (SJF) policy (line 6). Once \( S \) is complete, we call the simulation algorithm to estimate the average turnaround time ATAT of \( S \) starting with \( J_i \) (line 7). Finally, the candidate scheduling set \( J_i \) with the smallest ATAT values are chosen and returned (line 9). The whole process is shown in Algorithm 3.

Given a possible job scheduling sequence \( S \), the simulation algorithm estimates the ATAT of \( S \) by greedily choosing jobs from the beginning of the \( S \), putting them into the running job set, and updating their waiting and execution time. The detailed working process is shown in Algorithm 4.

5.3. Timeout Mechanism. One potential issue of our PAS algorithm is that the long-running job with large resource demand will suffer from the starvation problem. To deal with it, we set a timeout value, that is the maximal waiting deadline, to each job. Once the waiting time of a job exceeds the timeout, it will be executed first without any delay.

We can set the timeout value as the makespan of the current scheduling job set \( J^k_s \); this ensures that the long-running job(s) with large resource demand has the opportunity to execute after all other jobs in \( J^k_s \) complete their execution. The timeout value for each job can be set at the first time in the simulation algorithm, in which the job scheduling sequence \( S \) is given and the makespan value is able to estimate.

5.4. Implementation on Hadoop YARN. We incorporate \( \text{PAS} \) into Hadoop YARN (2.6.0) by implementing an independent \( \text{PAS algorithm} \) module and adding a \text{PAS scheduler} plugin to YARN. The implementation detail is shown in Figure 1.

As shown in Figure 1, \( \text{PAS} \) consists of two modules. The first module is called \( \text{PAS algorithm} \), which is implemented by Python 3.5. At each time slot \( t_k \), \( \text{PAS algorithm} \) first receives the necessary scheduling information, for example \( R^k_{\text{used}}, J_k, \text{etc.} \), from the \( \text{PAS scheduler} \), and then calculate the suggested scheduling job sets by calling the proposed \( \text{PAS algorithm} \) with the scheduling information. The communication between \( \text{PAS algorithm} \) and \( \text{PAS scheduler} \) is implemented by TCP protocol.

We have implemented the \( \text{PAS scheduler} \) as a customized plugin of YARN. \( \text{PAS scheduler} \) focuses on sending the scheduling information periodically to the \( \text{PAS algorithm} \) and allocating resource for the scheduling job set returned by the \( \text{PAS algorithm} \).

More specifically, at each time slot \( t^k \), \( \text{PAS scheduler} \) needs to determine whether there still has more resource to allocate in comparison with \( U \), the upper bound of resource utilization:

(i) If so, \( \text{PAS scheduler} \) sends the information including current used resource \( R^k_{\text{used}} \) and the job set waiting for scheduling \( J_k \) to the \( \text{PAS algorithm} \) and requests for the scheduling job set. Once received the \( J^k_s \), \( \text{PAS scheduler} \) will attach all the jobs in \( J^k_s \) to the execution queue of YARN.

(ii) If not, \( \text{PAS scheduler} \) will do nothing and wait for the next time slot.

6. Experiments

6.1. Experimental Setup. We have implemented our approach and conducted extensive experiments under different testing scenarios. The source code and dataset can be found in an anonymous website: https://github.com/anon4review/pas. In
Require: $J^k$: the job set waiting for scheduling at $r^k$; $R^k_{used}$: the used resource at $r^k$; $R'$: the target resource utilization after scheduling.
Ensure: All possible scheduling job sets $J^k_s$
(1) $J^k_s \leftarrow \emptyset$;
(2) $R^k \leftarrow R^k_{used}$;
(3) Sort the jobs in $J^k$ by their resource demands $r$ in the increasing order to form a bin list $B = \{B_1, B_2, \ldots, B_l\}$, in which a bin contains jobs with the same $r$;
(4) for $i = 1$ to $l$ do
(5) $R^k \leftarrow R^k + |B_i| \cdot r_i$;
(6) if $R^k \leq R'$ then
(7) $J^k_s \leftarrow J^k_s \cup \{B_i\}$;
(8) else
(9) $n \leftarrow \lfloor R' - R^k - |B_i| \cdot r_i/r_j \rfloor$;
(10) $J' \leftarrow$ all subset from $B_i$ containing $n$ jobs;
(11) $J^k_s \leftarrow J^k_s \cup J'$;
(12) end if
(13) end for return $J^k_s$;

Algorithm 2: Small job first algorithm: SmJF($J^k$, $R^k_{used}$, $R'$).

Require: $J^k$: the job set waiting for scheduling at $r^k$; $J^k_c$: a group of candidate scheduling sets; $R^k_{used}$: the used resource at $r^k$; $U$: the upper bound of resource utilization.
Ensure: $J^k_s$: the scheduling set.
(1) Ordered job scheduling sequence $S \leftarrow \emptyset$;
(2) for $J_i \in J^k_c$ do
(3) for $j_i \in J_i$ do
(4) $S \leftarrow S \cup j_i$;
(5) end for
(6) Append jobs in $J^k - J_i$ to $S$ according to the shortest job first (SJF) policy;
(7) ATAT$\leftarrow$Simulation($J^k$, $J_i$, $R^k_{used}$, $S$, $U$);
(8) end for
(9) $J^k_s \leftarrow \arg\min_{J_i \in J^k_c} (\text{ATAT}_i)$; return $J^k_s$;

Algorithm 3: One-step lookahead algorithm: OSLA($J^k$, $J^k_c$, $R^k_{used}$, $U$).

Require: $J^k$: the job set waiting for scheduling at $r^k$; $J_i$: a group of candidate scheduling sets; $R^k_{used}$: the used resource at $r^k$; $S$ the scheduling sequence under SJF policy, $U$: the upper bound of resource utilization.
Ensure: ATAT for $J^k$ under the scheduling sequence $S$.
(1) $S' \leftarrow S$;
(2) The running job set: $J_r \leftarrow \emptyset$;
(3) The current used resource: $R \leftarrow R^k_{used}$;
(4) The turnaround time for a job $j$: TAT$\leftarrow 0$;
(5) while $S' \neq \emptyset$ do
(6) for $j_i \in S'$ do
(7) if $(R + r_i) \leq U$ then
(8) $S'.\text{remove}(j_i)$;
(9) $J_r.\text{append}(j_i)$;
(10) $R \leftarrow R + r_i$;
(11) for $j_p \in J_r$ do
(12) $t_p.\text{remain} \leftarrow t_p.\text{remain} \cdot T(j_p, R)/T(j_p, R - r_i)$;
(13) end for
(14) else
(15) $j_{fast} \leftarrow \text{arg min}_{j_i \in J_r} (t_i.\text{remain})$;
(16) for $j_i \in J_r$ do
(17) $t_i.\text{remain} \leftarrow t_i.\text{remain} - t_{fast}.\text{remain}$;
(18) end for

Algorithm 4: Continued.
In this section, we first describe our experiment setup and then report the experimental results to prove the efficiency and effectiveness of the proposed approach.

6.1.1. Running Environment. We use Hadoop YARN (2.6.0) on Spark (1.6.0) and conduct our experiments on a local cluster of five physical servers. Each server is equipped with two 8-core Intel Xeon E5-2650 v2 2.6 GHz processors, 256 GiB RAM, 1.5 TB disk, and running CentOS 6.0, Java 1.7.0_55 and Python 3.5. All of servers are connected via a high-speed 1.5 Gbps LAN. To avoid interference and comply with the actual deployment, we run the YARN on Spark, the workload generators, and the PAS algorithm on different physical servers at each experiment.

6.1.2. Baseline Algorithms. To evaluate the performance of PAS, we compare it with four state-of-the-art algorithms, namely FIFO [38], AHP [44], SJF [45], and DRF [24]. We provide a brief description for each algorithm as follows and compare the scheduling goals for the five algorithms in Table 1:

- **FIFO** sorts all jobs in the order of submission (first in, first out), and it is the default scheduler of the YARN.
- **AHP** is an improvement in priority-based job scheduling algorithm in cloud computing, which is based on multiple criteria and multiple attribute decision-making model.
- **SJF** sorts all jobs in the order of execution time, the shortest will be sorted first.
- **DRF** is an extension of classic fair scheduling [26] for multiple types of resource. DRF determines CPU and memory resource shares based on the availability of those resources and the job requirements.

In PAS algorithm, we set the lower and upper bounds of CPU and memory utilization ratios to [0.5, 0.9] and [0.75, 1], respectively.

6.1.3. Workloads. In our experiment, we use HiBench, a well-known big data benchmark, to generate Spark workloads. More specifically, we choose 10 different workloads.

```
(18) TAT_i ← TAT_i + t_{fast}.remain;
(19) t_{remain} ← t_{remain} · T(f_j, R - r_{fast})/T(f_j, R);
(20) end for
(21) TAT_i ← TAT_i + t_{fast}.remain;
(22) R ← R - r_{fast};
(23) f_j.remove(f_{fast});
(24) end if
(25) end for
(26) end while
(27) ATAT ← 1/|J^k| · ∑_{j_i ∈ J^k} (TAT_i); return ATAT;
```

**Algorithm 4:** Job scheduling simulation algorithm: Simulation ($f^k$, $J^k$, $R^k_{used}$, $S$, $U$).

![Figure 1: PAS implementation on Hadoop YARN.](image-url)
with three micro benchmarks—SparkPi (SP), WordCount (WC), and Sort (ST)—and seven machine learning workloads—support vector machine (SVM), Bayesian classification (BC), K-means clustering (KM), gradient boosting trees (GBT), linear regression (LR), principal component analysis (PCA), and random forest (RF). For each workload, we have two different scales of datasets and three different two-dimensional resource demands (e.g., \(2C, 4GB\)) means two CPU cores and 4 GB memory), as shown in Table 2.

We construct six groups of jobs for scheduling with different job numbers, that is 15, 30, 45, 60, 75, and 90, by randomly choosing these jobs from the candidate workloads listed in Table 2. For each testing group and a scheduling algorithm, we conduct 10 independent runs and record the results separately.

### 6.1.4. Evaluation Metrics

We consider three well-known metrics in our experiments for performance evaluation, namely average turnaround time (ATAT), resource utilization ratio (RUR), and makespan (MS).

**Average turnaround time (ATAT).** ATAT can measure the performance of a scheduler from the user’s perspective. The detailed definition of ATAT can be found in Section 3. Here we define the ATAT improvement in an algorithm over the baseline algorithm in comparison as:

\[
\text{Imp}_{\text{baseline}} = \frac{\text{ATAT}_{\text{baseline}} - \text{ATAT}}{\text{ATAT}_{\text{baseline}}} \times 100\% ,
\]

where ATAT\text{baseline} is the baseline ATAT, and ATAT\text{baseline} is that of the algorithm being evaluated.

**Resource utilization Ratio (RUR).** RUR measures the resource efficiency of a BDPS from the service providers’ perspective. The detailed definition of RUR can also be found in Section 3.

**Makespan (MS).** Makespan defines the time difference between the start and finish of a sequence of jobs. It measures the resource efficiency of a BDPS from the service providers’ perspective [2].

### 6.2. Experimental Results

#### 6.2.1. Estimation of Job Completion Time under Different RUR

We choose three workloads, that is a CPU-intensive workload (SP), a memory-intensive workload (WC), and a machine learning workload (SVM), to verify the prediction model for job completion time. We first run each of these three workloads on the experimental environment with a series of RUR. For each workload, we perform regressions on the estimated expressions, that is equation (3), and provide in Table 3 the results of the root mean square error (RMSE), normalized root mean square error (NRMSE), and R-square (\(R^2\)), which clearly indicate that a high level of goodness-of-fit is achieved in these regressions.

For a better illustration, we plot the CPU and memory utilization ratio for running a group of 45 jobs under five scheduling algorithms in Figures 4 and 5, respectively. We can see from Figure 4 that the CPU utilization ratios under PAS algorithm (red line) are more stable over the time than others: the CPU utilization ratios are between 50% and 90% (our predefined boundary) during the 69.04% of time slots. Figure 5 shows that the difference of memory utilization ratios among these algorithms is not significative, but the memory utilization ratios under PAS can be still kept in between 75% and 100% during 76.19% of time slots. Note that the resource utilization jitter exists under all scheduling algorithms because there might be a delay between the release of resource for already finished jobs and the allocation of resource for a set of newly added jobs.

Table 1: Scheduling goals for the five algorithms.

| Algorithm | Resource | Performance | Fairness | Priority |
|-----------|----------|-------------|----------|----------|
| FIFO      |          | ✓           |          | ✓        |
| AHP       | ✓        |             |          | ✓        |
| SJF       | ✓        |             |          |          |
| DRF       | ✓        |             |          |          |
| PAS       | ✓        | ✓           |          |          |

6.2.2. ATAT. Table 4 shows the ATAT (ms) values of five different scheduling algorithms. As expected, our algorithm outperforms the default FIFO scheduler of YARN by 25.3% to 42.0%. Furthermore, PAS outperforms all other three algorithms: 13.6%–36.4% improvement over AHP, 10.2%–28.8% improvement over SJF, and 4.1%–12.1% improvement over DRF.

Figure 3 shows the boxplots of the ATAT performance under six groups of job sets and five algorithms over 10 independent runs. In these boxplots, the bottom and top of the box are the first and third quartiles, the bands inside the boxes represent robust estimates of the uncertainty about the medians for box-to-box comparison. The ends of the whiskers represent possible alternative values and the symbol “+” denotes outliers. As shown by the ATAT boxplots, PAS has lower ATAT values than other four scheduling algorithms on all of the six job sets, which indicates the superiority and the robustness of our PAS algorithm.

6.2.3. RUR. For a better illustration, we plot the CPU and memory utilization ratio for running a group of 45 jobs under five scheduling algorithms in Figures 4 and 5, respectively. We can see from Figure 4 that the CPU utilization ratios under PAS algorithm (red line) are more stable over the time than others: the CPU utilization ratios are between 50% and 90% (our predefined boundary) during the 69.04% of time slots. Figure 5 shows that the difference of memory utilization ratios among these algorithms is not significative, but the memory utilization ratios under PAS can be still kept in between 75% and 100% during 76.19% of time slots. Note that the resource utilization jitter exists under all scheduling algorithms because there might be a delay between the release of resource for already finished jobs and the allocation of resource for a set of newly added jobs.
Finally, Figure 6 shows the makespan values for six groups of workloads under five scheduling algorithms. We can see from Figure 6 that PAS outperforms all other three algorithms in term of makespan, with the exception of DRF algorithm. The reason why DRF has a good performance in makespan is probably because it allows schedulers to take into account the heterogeneous resource demands, leading to both fairer allocation of resources and higher utilization [24].

### Table 2: Ten different candidate workloads in our experiment.

| #  | Job name | Scale       | Resource demand                   |
|----|----------|-------------|-----------------------------------|
| 1  | SP       | Tiny, large | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 2  | WC       | Tiny, large | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 3  | ST       | Tiny, large | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 4  | SVM      | Tiny, small | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 5  | BC       | Tiny, small | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 6  | KM       | Tiny, small | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 7  | GBT      | Tiny, small | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 8  | LR       | Tiny, small | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 9  | PCA      | Tiny, small | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |
| 10 | RF       | Tiny, large | (2C, 4GB), (3C, 6GB), (4C, 8GB)   |

### Table 3: Regression errors in performance estimation.

| Job name | RMSE  | NRMSE (%) | $R^2$   |
|----------|-------|-----------|---------|
| SP       | 0.7013| 2.06      | 0.9730  |
| WC       | 2.5911| 6.18      | 0.7496  |
| SVM      | 1.7258| 2.55      | 0.9692  |

![Figure 2: Boxplots of ATAT values under five algorithms over 10 independent runs.](image)

### Table 4: ATAT performance of five algorithms.

| #  | PAS    | AHP    | SJF    | FIFO   | DRF    |
|----|--------|--------|--------|--------|--------|
| 15 | 66234.4| 88424.8| 93058.13| 103481.3| 73318  |
| 30 | 85147.7| 106808.8| 94857.4| 125558.4| 91477.4667|
| 45 | 109096.1| 129421.9| 130781.3| 174098.5| 118586.6|
| 60 | 138780.8| 205562.2| 161795.8| 174463.3| 135836.6|
| 75 | 130243.3| 150723 | 177455.8| 210194.8| 136140.944|
| 90 | 123840.5| 194835 | 147746.5| 210194.8| 136140.944|
Figure 3: Boxplots of ATAT values under six groups of job sets and five algorithms over 10 independent runs.

Figure 4: CPU trends w.r.t. the time slots.
7. Conclusion and Future Work

In this article, we propose PAS algorithm to optimize average turnaround time and resource efficiency simultaneously. PAS scheduler constructs a performance prediction model for an accurate estimation of completion time of big data analytical jobs, it then dynamically schedules multiple jobs concurrently using different policies based on the prediction model and employs a greedy and a one-step lookahead strategy to opportunistically improve the average job performance for fast job completion while still achieving good enough resource efficiency.

It is of our future work to refine our prediction model by supporting the automatic selection of the appropriate parameters for any jobs given the hardware and software settings of a BDPS. For practical applications, we will take the priority of jobs and fairness into consideration. We will also investigate the performance dynamics of BDPSs and design better scheduling approaches to account for such dynamics in our algorithm design.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] C. Chen, W. Wang, and Bo Li, “Performance-aware fair scheduling: exploiting demand elasticity of data analytics jobs,” in Proceedings of the IEEE INFOCOM 2018-IEEE Conference on Computer Communications, pp. 504–512, IEEE, Honolulu, HI, USA, April 2018.

[2] J. Liu, E. Pacitti, and P. Valduriez, “A survey of scheduling frameworks in big data systems,” International Journal of Cloud Computing, vol. 7, no. 2, pp. 103–128, 2018.

[3] Yi Yao, J. Wang, B. Sheng, J. Lin, and N. Mi, “Haste: hadoop yarn scheduling based on task-dependency and resource-demand,” in Proceedings of the 2014 IEEE 7th International Conference on Cloud Computing, pp. 184–191, IEEE, Anchorage, AK, USA, June 2014.

[4] Yi Yao, H. Gao, J. Wang, N. Mi, and B. Sheng, “Opera: opportunistic and efficient resource allocation in hadoop yarn by harnessing idle resources,” in Proceedings of the 2016 25th International Conference on Computer Communication and Networks (ICCCN), pp. 1–9, IEEE, Waikoloa, HI, USA, August 2016.
[5] C. Delimitrou and C. Kozyrakis, “Quasar: resource-efficient and QoS-aware cluster management,” in ACM SIGARCH - Computer Architecture News, vol. 42, no. 1, pp. 127–144, ACM, 2014.

[6] J. Polo, C. Castillo, D. Carrera et al., “Resource-aware adaptive scheduling for mapreduce clusters,” in Proceedings of the 12th International Middleware Conference, pp. 180–199, Lisbon, Portugal, December 2011.

[7] A. Verma, L. Cherkasova, and R. H. Campbell, “Aria: automatic resource inference and allocation for mapreduce environments,” in Proceedings of the 8th ACM international conference on Autonomic computing, pp. 235–244, ACM, Karlsruhe Germany, June 2011.

[8] F. Cheng, Y. Huang, B. Tanpure, P. Sawalani, L. Cheng, and C. Liu, “Cost-aware job scheduling for cloud instances using deep reinforcement learning,” Cluster Computing, vol. 25, no. 1, pp. 619–631, 2022.

[9] Y. Fan, “Job scheduling in high performance computing,” Distributed, Parallel, and Cluster Computing, vol. 18, 2021.

[10] B. Zheng, Li Pan, and S. Liu, “Market-oriented online bi-objective service scheduling for pleasingly parallel jobs with variable resources in cloud environments,” Journal of Systems and Software, vol. 176, Article ID 110934, 2021.

[11] A. Verma, L. Cherkasova, and R. H. Campbell, “Two sides of a coin: optimizing the scheduling of mapreduce jobs to minimize their makespan and improve cluster performance,” in Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 11–18, IEEE, Washington, DC, USA, August 2012.

[12] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. K. Tsang, “Need for speed: cora scheduler for optimizing completion-times in the cloud,” in Proceedings of the 2015 IEEE Conference on Computer Communications (INFOCOM), pp. 891–899, IEEE, Hong Kong, China, April 2015.

[13] X. Hou, T. K. Ashwin Kumar, J. P. Thomas, and H. Liu, “Dynamic deadline-constraint scheduler for hadoop yarn,” in Proceedings of the 2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), pp. 1–8, IEEE, San Francisco, CA, USA, August 2017.

[14] Y. Wang and W. Shi, “Budget-driven scheduling algorithms for batches of mapreduce jobs in heterogeneous clouds,” IEEE Transactions on Cloud Computing, vol. 2, no. 3, pp. 306–319, 2014.

[15] M. Khan, Y. Jin, M. Li, X. Yang, and C. Jiang, “Hadoop performance modeling for job estimation and resource provisioning,” IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 2, pp. 441–454, 2015.

[16] N. Lim, S. Majumdar, and P. Ashwood-Smith, “A constraint programming based hadoop scheduler for handling mapreduce jobs with deadlines on clouds,” in Proceedings of the 6th ACM/SPEC international conference on performance engineering, pp. 111–122, ACM, Austin TX USA, January 2015.

[17] Y. Shao, C. Li, J. Gu, J. Zhang, and Y. Luo, “Efficient jobs scheduling approach for big data applications,” Computers & Industrial Engineering, vol. 117, pp. 249–261, 2018.

[18] K. Chen and L. Huang, “Timely-throughput optimal scheduling with prediction,” IEEE/ACM Transactions on Networking, vol. 26, no. 6, pp. 2457–2470, 2018.

[19] A. Dina, A. Gamal, Z. Ibrahim, and A. N. Aida, “Elite learning Harris hawks optimizer for multi-objective task scheduling in cloud computing,” The Journal of Supercomputing, vol. 78, no. 2, pp. 2793–2818, 2022.

[20] M. Sha Alam Khan and R. Santhosh, “Task scheduling in cloud computing using hybrid optimization algorithm,” Soft Computing, 2021.

[21] V. Meyer, D. F. Kirchoff, L. Matheus, D. Silva, A. Cesar, and F. De Rose, “ML-driven classification scheme for dynamic interference-aware resource scheduling in cloud infrastructures,” Journal of Systems Architecture, vol. 116, Article ID 102064, 2021.

[22] R. Chen, X. Chen, and C. Yang, “Using a task dependency job-scheduling method to make energy savings in a cloud computing environment,” The Journal of Supercomputing, vol. 78, 2021.

[23] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, 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, Paris, France, April 2010.

[24] G. Ali, M. Zaharia, H. Benjamins, and A. Konwinski, “Dominant resource fairness: fair allocation of multiple resource types,” Usenix Security Symposium, vol. 11, p. 24, 2011.

[25] W. Wang, B. Li, and B. Liang, “Dominant resource fairness in cloud computing systems with heterogeneous servers,” in Proceedings of the IEEE INFOCOM 2014-IEEE Conference on Computer Communications, pp. 583–591, IEEE, Toronto, ON, Canada, April 2014.

[26] The Apache Software Foundation Hadoop: Fair Scheduler, https://hadoop.apache.org/docs/r3.3.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html/

[27] 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, Helena, MT, USA, October 2009.

[28] H. Benjamins, A. Konwinski, M. Zaharia et al., “Mesos: a platform for fine-grained resource sharing in the data center,” NSDI, vol. 11, p. 22, 2011.

[29] G. Ali, M. Zaharia, S. Scott, and I. Stoica, “Choosy: max-min fair sharing for datacenter jobs with constraints,” in Proceedings of the 8th ACM European Conference on Computer Systems, pp. 365–378, ACM, Czech Praha, April 2013.

[30] H. Liu and B. He, “Reciprocal resource fairness: towards cooperative multiple-resource fair sharing in iaas clouds,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 970–981, IEEE Press, New Orleans, LA, USA, November 2014.

[31] Z. Huang, B. Balasubramian, M. Wang, T. Lan, M. Chiang, and D. H. K. Tsang, “Rush: a robust scheduler to manage uncertain completion-times in shared clouds,” in Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pp. 242–251, IEEE, Nara, Japan, June 2016.

[32] Y. Wang, J. Tan, W. Yu, L. Zhang, and X. Meng, “Preemptive reducetask scheduling for fair and fast job completion,” in Proceedings of the 10th International Conference on Autonomic Computing (\(\text{ICA}C\)) 13), pp. 279–289, San Jose, CA, USA, June 2013.

[33] Z. Niu, S. Tang, and B. He, “An adaptive efficiency-fairness meta-scheduler for data-intensive computing,” IEEE Transactions on Services Computing, vol. 12, 2016.
[34] Q. Wang and X. Huang, "Pft: a performance-fairness scheduler on hadoop yarn," in *Proceedings of the 2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pp. 76–80, IEEE, Beijing, China, August 2016.

[35] S. Tang, B. S. Lee, and B. He, "Dynamicmr: a dynamic slot allocation optimization framework for mapreduce clusters," *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 333–347, 2014.

[36] M. Pastorelli, A. Barbuzzi, D. Carra, M. Dell’Amico, and P. Michiardi, "Hfsp: size-based scheduling for hadoop," in *Proceedings of the 2013 IEEE International Conference on Big Data*, pp. 51–59, IEEE, Silicon Valley, CA, USA, October 2013.

[37] Z. Niu, S. Tang, and B. He, "Gemini: an adaptive performance-fairness scheduler for data-intensive cluster computing," in *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 66–73, IEEE, Vancouver, BC, Canada, November 2015.

[38] V. Kumar Vavilapalli, A. C. Murthy, C. Douglas et al., "Apache hadoop yarn: yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, Santa Clara, CA, USA, October 2013.

[39] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A fault-tolerant resource management and job scheduling system at internet scale," in *Proceedings of the VLDB Endowment. 40th International Conference on Very Large Data Bases*, vol. 7, no. 13, pp. 1393–1404, Hangzhou, China, September 2014.

[40] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM - Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2015.

[41] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs," in *ACM SIGCOMM - Computer Communication Review* vol. 45, no. 4, pp. 407–420, ACM, 2015.

[42] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI’16)*, pp. 363–378, Santa Clara, CA, USA, March 2016.

[43] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 395–404, ACM, Dubrovnik, Croatia, September 2007.

[44] J. P. Swachil and U. R. Bhoi, "Improved priority based job scheduling algorithm in cloud computing using iterative method," in *Proceedings of the 2014 4th International Conference on Advances in Computing and Communications*, pp. 199–202, IEEE, Cochin, India, August 2014.

[45] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling—a status report," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 1–16, Springer, New York, NY, USA, June 2004.