COBRA: Toward Provably Efficient Semi-clairvoyant Scheduling in Data Analytics Systems

Xiaoda Zhang, Zhuzhong Qian, Sheng Zhang, Xiangbo Li, Xiaoliang Wang, Sanglu Lu
State Key Laboratory for Novel Software Technology, Nanjing University, China
Email: {zhangxiaoda, lxb}@dislab.nju.edu.cn, {qzz, sheng, waxili, sanglu}@nju.edu.cn

***Abstract*—Typical data analytics systems abstract jobs as directed acyclic graphs (DAGs). It is crucial to maximize throughput and speedup completions for DAG jobs in practice. Existing works propose clairvoyant schedulers optimizing these goals, however, they assume *complete* job information as a prior knowledge which limits their applicability. Instead, we remove the complete prior knowledge assumption and rely solely on a *partial* prior information, which is more practical. And we design a semi-clairvoyant task scheduler COBRA working within each job. COBRA adaptively adjusts its resource desires in a multiplicative-increase multiplicative-decrease (MIMD) manner according to nearly past resource utilizations and the current waiting tasks. On the other hand, COBRA seeks to satisfy task locality preferences by allowing each task to wait for some time that is bounded by a parameterized threshold. Surprisingly, even with the partial prior job information, we theoretically prove, COBRA, when working with the widely used fair job scheduler, is *O(1)*-competitive with respect to both makespan and average job response time. We experimentally validate that the performance promotion of COBRA in both real system deployment and trace-driven simulations.**

I. INTRODUCTION

Over the past decade, large data analytics systems have been commonly deployed by organizations like Facebook, Google, Microsoft, and Yahoo! to facilitate their applications such as machine learning and web indexing. Because these applications are usually used for real-time decision, it is crucial to simultaneously maximize job throughput and speedup job completions. Data analytics systems (DASs) including Hadoop [1], Dryad [2], and Spark [3] abstract jobs as DAGs. We use DAG to refer to a directed acyclic graph, where each vertex represents a task and edges encode input-output dependencies.

Resources in a DAS are scheduled in terms of containers (corresponding to some fixed/variable amount of memory and cores). A job shall first acquire some containers and then assign its tasks to the containers with respect to locality preferences. State-of-the-art works [4], [5], [6] optimize fairness, average job completion time or makespan, but they assume a *complete* prior knowledge, which includes precise running times and resource requirements for all tasks of the jobs at their submissions. Note some recurring jobs indeed possess predictable characteristics [7], [8]. However in a general case, cluster dynamics like task failures and speculations make it hard to know the job characteristics, which limits their applicability [2], [3], [9]. In addition, they employ *disposable* containers, *i.e.*, a job always returns the used container to the cluster once a task completes, which incurs nonnegligible rescheduling overheads for containers.

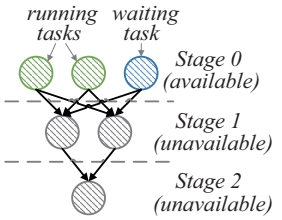


Fig. 1. Example of a running DAG job.

In this paper, we use a more practical assumption: only *partial* prior knowledge of a job is known, including fine-grained resource requirements and process times of tasks in *available* stages. In the example of Fig. 1, only the task information in *Stage 0* is currently known, while the task information in *Stage 1* and *Stage 2* is currently unknown because they have not been released yet. We consider that tasks in the same stage have identical characteristics, which conforms to the fact in practical systems as they perform the same computations on different partitions of the input. Furthermore, we break the fixed one-to-one mapping between tasks and containers, and allow multiple tasks to simultaneously run in the same container as long as the capacity of the container satisfies the tasks' requirements.

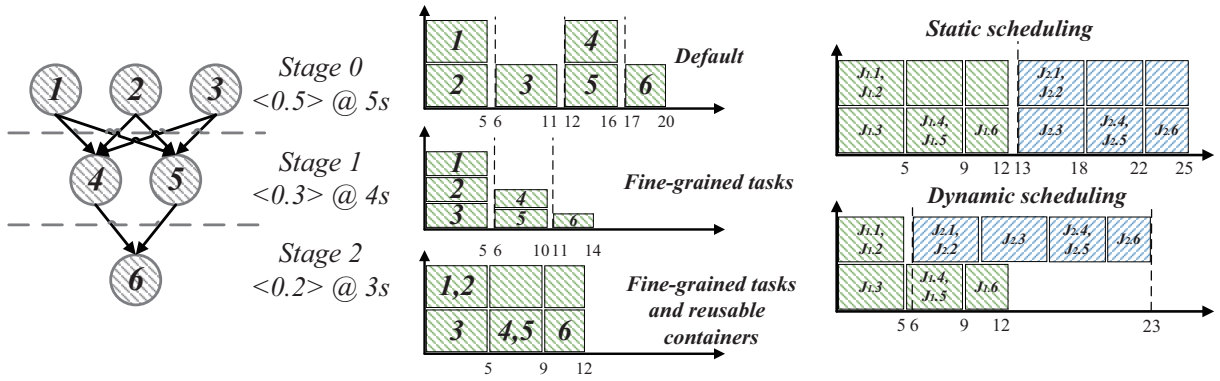
Inspired by adaptively scheduling instructions onto processors [10], [11], we devise a semi-clairvoyant[1] task scheduler COBRA[2] that works within each job, which is composed of two algorithms, `Af` and `Pdelay`. `Af` *either* dynamically requests more resources *or* proactively gives up allocated resources in a MIMD manner according to the utilization feedbacks, while `Pdelay` greedily satisfies task locality preferences by allowing each task to wait for some parameterized threshold time. In the scenario where multiple DAG jobs arrive and leave online, we prove that COBRA, when working with the widely used fair job scheduler [12], is $O(1)$-competitive with respect to both makespan and average job response time[3].

We implement this semi-clairvoyant COBRA, as well as the mechanism which monitoring resource utilizations of contain-

[1]Scheduling based on a partial prior knowledge of jobs is known as semi-clairvoyant scheduling.

[2]Cobras have limited vision but quick reaction to the changeable environment, which captures the key feature of our scheduler.

[3]The response time of a job is the duration time from its release to its completion.



(a) A DAG job with fine-grained resource requirements and processing times.

(b) An example showing fine-grained tasks and reusable containers help in speeding up a single job completion.

(c) An example showing dynamic resource scheduling helps in improving average job response time.

Fig. 2. Illustrative examples showing how techniques can speed up job completions in both single and multiple job scenario.

ers, in Apache Spark [13] on YARN [14] system, and deploy it on a 28-node cluster. Both real system experiments on typical workloads and simulations on Google trace show our design is only within 10% performance loss compared to the state-of-the-art clairvoyant scheduler, but still achieves 35% reductions in makespan and average job response time compared to the existing implementations in the system.

Our main contributions in this paper are:

- We remove the complete prior job knowledge assumption in existing works. Instead, we rely on the fact that fine-grained resource requirements and processing times are known *only* for available stage tasks. We employ reusable containers, *i.e.*, a job could hold a container for executing multiple tasks before proactively returning it.
- We devise COBRA that works within each job and monitors the utilizations of its allocated containers. It can *adaptively* adjust its scheduling behavior according to the utilization feedbacks and waiting tasks. COBRA seeks to satisfy task locality preferences by allowing each task to wait for some time that is bounded by a *parameterized* threshold. Furthermore, we theoretically prove its performance efficiency.
- We implement COBRA in Apache Spark on YARN system, and use both deployment experiments and simulations to validate the performance promotion.

The rest of the paper is organized as follows. In Sec. II, we introduce the background and motivation of this work. We formulate the problem in Sec. III. The design of COBRA is presented in Sec. IV, followed by the competitive analysis in Sec. V. We describe the implementation of COBRA in Sec. VI and evaluate its performance in Sec. VII. We review the related work in Sec. VIII. Sec. IX concludes the paper.

II. BACKGROUND AND MOTIVATION

In this section, we first show the DAG job structure and resource scheduling hierarchy in a typical data analytics system. We then use illustrative examples to show how the techniques employed in this paper promote the job performance.

TABLE I
STATISTICS OF CONTAINER RESPONSE TIME IN OUR DEPLOYMENT.

Average	Median	95th percentile
996 ms	1286 ms	2711 ms

A. DAG job structure and resource scheduling hierarchy

In a typical DAS, a job is comprised of stages with dependencies, that is, a stage cannot start until its dependent stages complete. Tasks in the same stage perform the identical computations on different partitions of the input. In the example of Fig. 1, *Stage 1* should wait until all tasks in *Stage 0* complete. When a stage becomes available (*i.e.*, all prerequisite stages complete), the state of its tasks changes into waiting. In this paper, we consider the characteristics of waiting tasks are known (semi-clairvoyant), as contrary to the assumption of knowing all tasks' characteristics at a job submission (clairvoyant). In practice, it is usual that the current resources cannot satisfy all the waiting tasks in a single wave.

Resource scheduling in DASs is usually hierarchical. There is a monolithic job scheduler allocating containers to jobs according to their resource desires based on various principles [12], [15], [16]. While our task scheduler associated with each job is in charge of offering its resource desires to the job scheduler, and assigning its waiting tasks to the allocated containers, which will be illustrated in Sec. IV.

B. Illustrative examples for potential gains

Current job schedulers [12], [16] allocate resources in the terms of fixed-capacity container, and match tasks to containers in a one-to-one mapping. However, a task is usually far from fully utilizing the container resources. To fix this, existing clairvoyant schedulers like [4], [5], [6] introduce variable-capacity containers. But they still keep the mechanism of returning the used container once a task completes, resulting in the rescheduling overheads. We collect the container response time, which is the duration from a job sending a container request to the response of the request, across light load to

almost full load in our 28-node cluster. As shown in Table I, the overheads are nonnegligible. Choosing an alternative way, we maintain the containers with fixed-capacity, and allow multiple fine-grained tasks to *simultaneously* run in the same container. We prove in Sec. V that the *wasted* resources which are not utilized by tasks in a job, are bounded.

Importance of fine-grained resource requirements and reusable containers: We use the DAG job shown in Fig. 2(a) to illustrate the issues in scheduling DAGs. We assume that the (single type) fine-grained resource requirement of a task is normalized by the container capacity. Tasks in *Stage 0* have 0.5-unit requirement and 5-second processing time. Suppose there is a 2-unit environment, the default scheduler would split it into 2 containers, each with an unit capacity [14]. We assume the container response time is 1 second. In the example of Fig. 2(b) where the DAG job is released at time 0, the default scheduler schedules tasks as the figure shows and finishes the job at time 20. Tetris [4], which using variable-capacity containers for fine-grained tasks has more running tasks at the same time, and finishes the job at time 14. We allow fine-grained tasks and reusable containers, which further reduces the job completion time to 12 seconds.

Importance of dynamic resource scheduling: Suppose there are two identical DAG jobs as shown in Fig. 2(a): J_1 is released at time 0, and J_2 is released at time 5. In the 2-unit environment as before, static scheduling lets J_1 hold the two containers in its execution until J_1 completes [13], as shown in Fig. 2(c). The average job response time in static scheduling is 16 seconds. While in dynamic resource scheduling, J_1 gives up one container to the cluster at time 5, since waiting tasks ($J_1.4$ and $J_1.5$), whose characteristics are *known*, can be packed into one container. The returned container is in turn allocated to J_2 at time 6, leading to an early starting of J_2. The average job response time is 15 seconds in dynamic scheduling.

In summary, we can achieve higher job throughput by running more fine-grained tasks at the same time. Dynamic scheduling allows jobs to either timely give up resources or request more resources, and hence reduces average job response time.

III. Problem Formulation

In this section, we formulate the scheduling model and present the optimization goal of makespan and average job response time.

Suppose there is a set of jobs $\mathcal{J} = \{J_1, J_2, ..., J_{|\mathcal{J}|}\}$ to be scheduled on a set of fixed-capacity containers $\mathcal{P} = \{P_1, P_2, ..., P_{|\mathcal{P}|}\}$. These containers are different since they reside in different servers containing different input data for jobs. Time is discretized into ***scheduling periods*** of equal length L, where each period q includes the interval $[L \cdot q, L \cdot (q+1) - 1]$. L is a configurable system parameter.

We model a job J_i as a DAG. Each vertex of the DAG represents a task and each edge represents a dependency between the two tasks. Each task in a job prefers a unique subset of $\mathcal{P}$, as the containers in the subset store the input data for the task. For each task $t_{ij} \in J_i$, we denote by $t_{ij}.\boldsymbol{r}$ (a vector) to be the peak requirements for multiple type resources (*e.g.*, CPU, memory). For each resource type k, we assume $0 \leq t_{ij}.\boldsymbol{r}_k \leq 1$, normalized by the container capacity. We also assume $\max_k t_{ij}.\boldsymbol{r}_k \geq \theta$, where $\theta > 0$, *i.e.*, a task must consume some amount of resources. We associate $t_{ij}.p$ to be the processing time of task t_{ij}. Furthermore, the ***work*** of a job J_i is defined as $T_1(J_i) = \max_k \sum_{t_{ij} \in J_i} t_{ij}.\boldsymbol{r}_k \cdot t_{ij}.p$. The ***span*** $T_\infty(J_i)$ corresponds to the sum of task processing times on the critical path in J_i's dependency graph, *i.e.*, the shortest possible execution time of J_i. The release time $r(J_i)$ is the time at which the job J_i is submitted. A task is called in the *waiting* state when its predecessor tasks have all completed and itself has not been scheduled yet. Each job is handled by a dedicated task scheduler, which is oblivious to the further characteristics of the unfolding DAG.

Algorithm 1 Adaptive feedback algorithm (`Af`)

Input: $d(J_i, q-1)$, $a(J_i, q-1)$, $u(J_i, q-1)$
Output: $d(J_i, q)$

```
if q = 1 then
    d(J_i, q) ← 1
else if u(J_i, q−1) < δ and no waiting tasks then
    d(J_i, q) ← d(J_i, q−1) / ρ                 //inefficient
else if d(J_i, q−1) > a(J_i, q−1) then
    d(J_i, q) ← d(J_i, q−1)                     //efficient and deprived
else
    d(J_i, q) ← d(J_i, q−1) · ρ                 //efficient and satisfied
end if
return d(J_i, q)
```

Definition 1. *The **makespan** of a job set $\mathcal{J}$ is the time taken to complete all the jobs in $\mathcal{J}$, that is,* $\mathrm{T}(\mathcal{J}) = \max_{J_i \in \mathcal{J}} \mathrm{T}(J_i)$, *where* $\mathrm{T}(J_i)$ *is the completion time of job J_i.*

Definition 2. *The **response time** of a job J_i is the duration between its release time $r(J_i)$ and its completion time* $\mathrm{T}(J_i)$, *that is* $\mathrm{T}(J_i) - r(J_i)$.

Definition 3. *The **average response time** of a job set $\mathcal{J}$ is given by* $\frac{1}{|\mathcal{J}|} \sum_{J_i \in \mathcal{J}} (\mathrm{T}(J_i) - r(J_i))$.

The job scheduler and task scheduler interact as follows. The job scheduler reallocates resources between scheduling periods. At the end of period $q-1$, the task scheduler of job J_i determines its desire $d(J_i, q)$, which is the number of containers J_i wants for period q. Collecting the desires from all running jobs, the job scheduler decides allocation $a(J_i, q)$ for each job J_i (with $a(J_i, q) \leq d(J_i, q)$). Once a job is allocated containers, the task scheduler further schedules its tasks. And the allocation does not change during the period.

Given a job set $\mathcal{J}$ and a cluster with container set $\mathcal{P}$, we seek for a combination of a job scheduler, and a task scheduler within each job, which minimizes makespan and average response time of $\mathcal{J}$, while satisfying the task locality preferences. We propose Cobra as the task scheduler (Sec. IV), and show that the combination of the existing fair job scheduler and Cobra achieves our goal (Sec. V).

Algorithm 2 Parameterized delay scheduling (`Pdelay`)

Input: J_i, n: container, δ: uti. threshold, τ: a parameter
Output: $tlist$: task list

1: For each $t_{ij} \in J_i$, increase $t_{ij}.wait$ by the time since last event *UPDATE*; $cont \leftarrow$ true
2: **while** $n.\boldsymbol{free} > 0$ **and** $cont$ **do**
3: $cont \leftarrow$ false;
4: **if** J_i has a node-local task t_{ij} on n **and** $n.\boldsymbol{free} \geq t_{ij}.\boldsymbol{r}$ **then**
5: $tlist$.add(t_{ij}); $n.\boldsymbol{free}$ −= $t_{ij}.\boldsymbol{r}$; $cont \leftarrow$ true;
6: **else if** J_i has a rack-local task t_{ik} on n **and** $n.\boldsymbol{free} \geq t_{ij}.\boldsymbol{r}$ **and** $t_{ik}.wait \geq \tau \cdot t_{ik}.p$ **then**
7: $tlist$.add(t_{ik}); $n.\boldsymbol{free}$ −= $t_{ik}.\boldsymbol{r}$; $cont \leftarrow$ true;
8: **else if** J_i has a task t_{il} with $t_{il}.wait \geq 2\tau \cdot t_{il}.p$ **and** $n.\boldsymbol{free} \geq 1 - \delta$ **then**
9: $tlist$.add(t_{il}); $n.\boldsymbol{free}$ −= $t_{il}.\boldsymbol{r}$; $cont \leftarrow$ true;
10: **end if**
11: **end while**
12: **return** $tlist$

IV. Our Approach – Cobra

This section presents Cobra, which is comprised by two algorithms, Adaptive feedback algorithm (`Af`) and Parameterized delay scheduling (`Pdelay`). `Af` interacts with the job scheduler to request or release resources, while `Pdelay` is in charge of assigning tasks to the allocated containers.

`Af` (Algorithm 1) determines the desire for next period $d(J_i, q)$ based on its last period desire $d(J_i, q-1)$, the last period allocation $a(J_i, q-1)$, the last period resource utilization $u(J_i, q-1)$ and waiting tasks. $u(J_i, q-1)$ is defined as $\max_k \boldsymbol{u}(J_i, q-1)_k$, in which $\boldsymbol{u}(J_i, q-1)_k$ corresponds to the average resource utilization for resource type k in period $q-1$, and can be measured by the monitoring mechanism.

Consistent with [10], we classify the period $q-1$ as *satisfied* versus *deprived*. `Af` compares the job's allocation $a(J_i, q-1)$ with its desire $d(J_i, q-1)$. The period is satisfied if $a(J_i, q-1) = d(J_i, q-1)$, as the job J_i acquires as many containers as it requests from the job scheduler. Otherwise, $a(J_i, q-1) < d(J_i, q-1)$, the period is deprived. The classification of a period as *efficient* versus *inefficient* is more involved than that in [10]. `Af` uses a parameter δ as well as the waiting task information. The period is inefficient if the utilization $u(J_i, q-1) < \delta$ *and* there is no waiting task in period $q-1$. Otherwise the period is efficient.

If the period is inefficient, `Af` decreases the desire by a factor ρ. If the period is efficient but deprived, it means that the job efficiently used the resources it was allocated, but `Af` had requested more containers than the job actually received from the job scheduler. It maintains the same desire in period q. If the period is efficient and satisfied, the job efficiently used the resources that `Af` requests. `Af` assumes that the job can use more containers and increases its desire by a factor ρ. In all three cases, `Af` allows `Pdelay` to assign multiple tasks to execute in a container (container reusing).

TABLE II
Explanations of notations

Notation	Explanation
$d(J_i, q)$	J_i's desire for period q
$a(J_i, q)$	J_i's allocation for period q
$u(J_i, q)$	J_i's resource utilization in period q
δ	the utilization threshold parameter
ρ	the resource adjustment parameter
τ	the task waiting time parameter

`Pdelay` (Algorithm 2) is based on framework of the original delay scheduling algorithm [17]. When a container updates its status, the algorithm adds the waiting time for each waiting task of a job J_i since the last event *UPDATE* happened (Line 1), followed by the task assignment procedure. Delay scheduling sets the waiting time thresholds for tasks as an invariant, while we modify the threshold for each task to be linearly dependent of its processing time (which is *known*), under the intuition of that "long" tasks are tolerant for waiting for longer time to acquire their preferred resources.

`Pdelay` operates as follows: It first checks whether there is a node-local task waiting, which means the free container is on the same server as the task prefers. Assigning the task to its preferred server which containing its input data helps in reducing data transmission over the network. We use $n.\boldsymbol{free}$ (a vector) to denote the free resources on container n. We consider $n.\boldsymbol{free} \geq t_{ij}.\boldsymbol{r}$ if and only if $n.\boldsymbol{free}_k \geq t_{ij}.\boldsymbol{r}_k$ for each resource type k. Secondly, the algorithm would check whether there is a rack-local task for the n, as the container shares the same rack as the task's preferred server. If the task has waited for more than the threshold time ($\tau \cdot t_{ij}.p$), and the container has enough free resources, we assign the task to the container. Finally, when a task has waited for long enough time ($2\tau \cdot t_{il}.p$), and $n.\boldsymbol{free} \geq 1-\delta$ ($n.\boldsymbol{free}_k \geq 1-\delta$ for every k), we always allow the task could be assigned if possible. When $n.\boldsymbol{free} \geq 1-\delta$, the utilized resource of the container n $< \delta$ for each resource type k. We assume $t_{il}.\boldsymbol{r}_k + \delta \leq 1$, for each i, l, k, as the upper bound for task resource requirement.

Please refer to Table II for the involved notations in our algorithms and their explanations.

V. Analysis of Cobra + Fair Job Scheduler

In this section, we show that Cobra when working with the fair job scheduler, is $O(1)$-competitive with respect to makespan. Towards this end, we first prove that the satisfied steps[4] in a job J_i execution course are bounded. And then, we show the deprived steps of a job are also bounded by proving that the allocated resources in its execution course is $O(1) \cdot T_1(J_i)$. We omit the proof of performance efficiency for average response time for space limitation. We leave the proof of theorem 1 in the full version of this paper.

Theorem 1. *For a job J_i with work $T_1(J_i)$ and span $T_\infty(J_i)$ on a cluster of $\mathcal{P}$ containers,* Cobra *produces at most*

[4]A step is equivalent to a time tick in a scheduling period. If a scheduling period is satisfied/deprived, all L steps in it are satisfied/deprived.

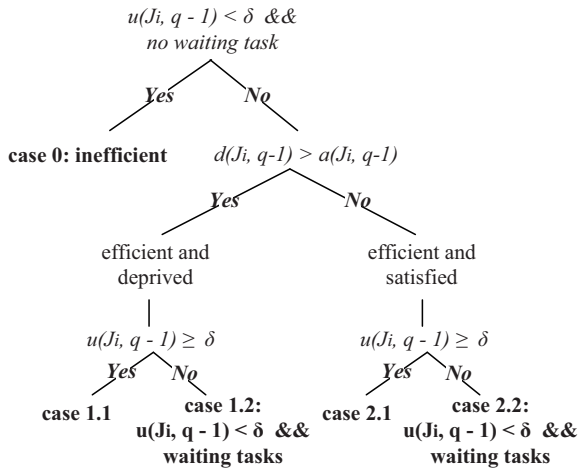


Fig. 3. Case decomposition in Af.

$\frac{2T_\infty(J_i)}{1-\delta} + L\log_\rho|\mathcal{P}| + L$ *steps in inefficient, plus efficient and satisfied periods.*

Corollary 2. COBRA *produces at most* $\frac{2T_\infty(J_i)}{1-\delta} + L\log_\rho|\mathcal{P}| + L$ *satisfied steps.*

Proof. Since an inefficient period could be either satisfied or deprived, the sum of satisfied steps from both inefficient and efficient periods is naturally at most $\frac{2T_\infty(J_i)}{1-\delta} + L\log_\rho|\mathcal{P}| + L$ by theorem 1. □

Next, we prove the wasted resources by a job J_i in its execution is bounded under the schedule of COBRA. We distinguish the detailed cases in Af as shown in Fig. 3. We first give a bound of the wasted resources in case 0 + case 1.1 + case 2.1, then we bound the wasted resources in case 1.2 + case 2.2, and finally the total wasted resources bound is the sum of them.

Lemma 3. *For a job J_i with work $T_1(J_i)$,* COBRA *wastes at most $\frac{1+\rho-\delta}{\delta}T_1(J_i)$ amount of resources in case 0 + case 1.1 + case 2.1, where ρ is the resource adjustment parameter, and δ is the utilization threshold parameter.*

Proof. In case 0, we know the utilization $u(J_i, q-1) < \delta$, while in case 1.1 and case 2.1, the utilization $u(J_i, q-1) \geq \delta$.

We use a potential-function argument, which is based on the desire for period q. We define the potential before period q as:

$$\Phi(q) = \frac{1+\rho-\delta}{\delta}T_1^q(J_i) + \frac{\rho L}{\rho-1}d(J_i, q),$$

where $T_1^q(J_i)$ is the remaining work of job J_i before period q. Hence, the initial potential is

$$\begin{aligned}\Phi(1) &= \frac{1+\rho-\delta}{\delta}T_1(J_i) + \frac{\rho L}{\rho-1}d(J_i, 1)\\ &= \frac{1+\rho-\delta}{\delta}T_1(J_i) + \frac{\rho L}{\rho-1},\end{aligned}$$

since $d(J_i, 1) = 1$. Suppose the job would complete at Q period, the the final potential is

$$\begin{aligned}\Phi(Q+1) &= \frac{1+\rho-\delta}{\delta}T_1^{Q+1}(J_i) + \frac{\rho L}{\rho-1}d(J_i, Q+1)\\ &\geq \frac{\rho L}{\rho-1},\end{aligned}$$

since $T_1^{Q+1}(J_i) = 0$ and $d(J_i, Q+1) \geq 1$. Thus, the total decrease in potential is $\Phi(1) - \Phi(Q+1) \leq \frac{1+\rho-\delta}{\delta}T_1(J_i)$. The the decrease in potential during any period q is

$$\begin{aligned}\Delta\Phi(q) &= \Phi(q) - \Phi(q+1)\\ &= \frac{1+\rho-\delta}{\delta}(T_1^q(J_i) - T_1^{q+1}(J_i))\\ &+ \frac{\rho L}{\rho-1}(d(J_i, q) - d(J_i, q+1)).\end{aligned}$$

We will show in case 0, case 1.1, case 2.1, respectively, if the wasted resources in period q is $w_q = (1-u(J_i,q))a(J_i,q)L$, then the potential decreases at least by w_q.

In case 0 (inefficient), $w_q = (1-u(J_i,q))a(J_i,q)L \leq a(J_i,q)L \leq d(J_i,q)L$, where the last inequality follows from the fact that the allocation is never greater than the corresponding desire. After an inefficient period, Af sets $d(J_i, q+1) = d(J_i, q)/\rho$. Thus, the decrease in potential is

$$\begin{aligned}\Delta\Phi(q) &\geq \frac{\rho L}{\rho-1}(d(J_i, q) - d(J_i, q+1))\\ &= \frac{\rho L}{\rho-1}(d(J_i, q) - \frac{d(J_i, q)}{\rho})\\ &= d(J_i, q)L\\ &\geq w_q,\end{aligned}$$

where the first inequality follows from the fact the remaining work does not increase in period q, $T_1^q(J_i) - T_1^{q+1}(J_i) \geq 0$.

In case 1.1 (efficient and deprived), we have $u(J_i, q) \geq \delta$, thus, $w_q = (1-u(J_i,q))a(J_i,q)L \leq (1-\delta)a(J_i,q)L$, and the desire for period $q+1$ is set to be $d(J_i, q+1) = d(J_i, q)$. Thus, the decrease in potential is

$$\begin{aligned}\Delta\Phi(q) &= \frac{1+\rho-\delta}{\delta}(T_1^q(J_i) - T_1^{q+1}(J_i))\\ &\geq \frac{1+\rho-\delta}{\delta}\delta a(J_i, q)L\\ &= (1+\rho-\delta)a(J_i, q)L\\ &> (1-\delta)a(J_i, q)L\\ &\geq w_q,\end{aligned}$$

where the first inequality follows from that the remaining work for J_i in period q decreases exactly $u(J_i,q)a(J_i,q)L \geq \delta a(J_i,q)L$.

In case 2.1 (efficient and satisfied), $u(J_i, q) \geq \delta$, thus, $w_q = (1-u(J_i,q))a(J_i,q)L \leq (1-\delta)a(J_i,q)L$, and the desire for

period $q+1$ is set to be $d(J_i, q+1) = d(J_i, q) \cdot \rho$. Thus, the decrease in potential is

$$\begin{aligned}\Delta\Phi(q) &\geq \frac{1+\rho-\delta}{\delta}\delta a(J_i,q)L \\ &+ \frac{\rho L}{\rho-1}(d(J_i,q) - d(J_i,q+1)) \\ &= (1+\rho-\delta)a(J_i,q)L - \rho d(J_i,q)L \\ &= (1-\delta)a(J_i,q)L \\ &\geq w_q,\end{aligned}$$

where the last equality follows from the fact that $a(J_i, q) = d(J_i, q)$ in the satisfied period.

In the all three cases, once job J_i wastes w_q resources in period q, the potential decreases at least by w_q. Thus, the overall waste during the three cases of J_i's execution is at most $\frac{1+\rho-\delta}{\delta}T_1(J_i)$, establishing the lemma. □

Lemma 4. *For a job J_i with work $T_1(J_i)$,* COBRA *wastes at most $\frac{2\tau}{\theta}T_1(J_i)$ amount of resources in case 1.2 + case 2.2, where τ is the task waiting time parameter, and θ is the resource requirement lower bound of tasks.*

Proof. In case 1.2 and case 2.2, the resource utilization is lower than δ, but the wasted resources by each task is bounded due to our `Pdelay` algorithm.

Specifically, the situation with most wasted resources occurs when the container has 0 utilization, and the task waits for longest time. In `Pdelay`, the task t_{ij} in J_i waits for no more than $2\tau \cdot t_{ij}.p$ time, and a container with 0 utilization would have at most $\frac{1}{\theta}$ number of t_{ij} simultaneously running. Hence, the wasted resources for t_{ij} is (a vector) $\boldsymbol{w}_{ij} \leq 2\tau t_{ij}.p \cdot \frac{1}{\theta} t_{ij}.\boldsymbol{r}$ (with $\boldsymbol{w}_{ij}^k \leq 2\tau t_{ij}.p \cdot \frac{1}{\theta} t_{ij}.\boldsymbol{r}_k$ for each resource type k). The maximum wasted resources among all resource types by J_i is $\max_k \sum_{t_{ij}\in J_i} \boldsymbol{w}_{ij}^k$, which is $\leq \frac{2\tau}{\theta}\max_k \sum_{t_{ij}\in J_i} t_{ij}.p \cdot t_{ij}.\boldsymbol{r}_k = \frac{2\tau}{\theta}T_1(J_i)$, establishing the lemma. □

Theorem 5. *For a job J_i with work $T_1(J_i)$,* COBRA *wastes at most $(\frac{1+\rho-\delta}{\delta} + \frac{2\tau}{\theta})T_1(J_i)$ amount of resources in its execution course.*

Proof. Combine lemma 3 and lemma 4. □

To prove our task scheduler COBRA guarantees good performance for jobs, we settle the job scheduler as the fair scheduler [12], perhaps the most widely used job scheduler in both industry and academia. The fair scheduler operates in a max-min fairness manner among running jobs [18]. Once there is a free resource, the fair scheduler always allocates it to the job which currently occupies the fewest fraction of the cluster resources, unless the job's requests have been satisfied. We will prove COBRA + fair scheduler guarantees bounded performance of a set of jobs.

Suppose there is a set of jobs $\mathcal{J}$ and a cluster of containers $\mathcal{P}$. Recall that $\mathrm{T}(\mathcal{J})$ is the makespan produced by our algorithms. Let $\mathrm{T}^*(\mathcal{J})$ denote the makespan of the set $\mathcal{J}$ and $\mathcal{P}$ scheduled by an optimal scheduler. A deterministic algorithm is c-competitive if there is a constant b such that $\mathrm{T}(\mathcal{J}) \leq c \cdot \mathrm{T}^*(\mathcal{J}) + b$ for any job set $\mathcal{J}$.

Theorem 6. *A set of jobs $\mathcal{J}$ scheduled by* COBRA + *fair job scheduler finish within $(\frac{2}{1-\delta} + \frac{1+\rho}{\delta} + \frac{2\tau}{\theta})\mathrm{T}^*(\mathcal{J}) + L\log_\rho|\mathcal{P}| + 2L$ steps.*

Proof. Let J_i denote the last job completed among jobs in $\mathcal{J}$. Let $S(J_i)$ denote the set of satisfied steps, and $D(J_i)$ denote the set of deprived steps. Suppose that J_i is released at period q, and begins to be scheduled at period $q+1$, that is, $qL \leq r(J_i) \leq (q+1)L$. Therefore, $\mathrm{T}(\mathcal{J}) \leq r(J_i) + L + |S(J_i)| + |D(J_i)|$.

By corollary 2, $S(J_i) \leq \frac{2T_\infty(J_i)}{1-\delta} + L\log_\rho|\mathcal{P}| + L$. Next, we bound $|D(J_i)|$.

For each $t \in D(J_i)$, we claim that all containers have been allocated to jobs by fair scheduler, otherwise the remaining unallocated containers would have been distributed among the deprived jobs. Thus, $\sum_{t\in D(J_i)}\sum_{J_k\in\mathcal{J}} a(J_k, t) = |\mathcal{P}||D(J_i)|$. The allocation for each job is either used by its tasks, or wasted, hence, the allocation for each job is bounded by its work $T_1(J_i)$ and waste $(\frac{1+\rho-\delta}{\delta} + \frac{2\tau}{\theta})T_1(J_i)$, where the latter term is by theorem 5. The total allocation of all jobs is $\sum_{J_i\in\mathcal{J}}(1 + \frac{1+\rho-\delta}{\delta} + \frac{2\tau}{\theta})T_1(J_i) = (\frac{1+\rho}{\delta} + \frac{2\tau}{\theta})T_1(\mathcal{J})$, where $T_1(\mathcal{J})$ is the total work of job set $\mathcal{J}$. Due to $|\mathcal{P}||D(J_i)| \leq (\frac{1+\rho}{\delta} + \frac{2\tau}{\theta})T_1(\mathcal{J})$, we have $|D(J_i)| \leq (\frac{1+\rho}{\delta} + \frac{2\tau}{\theta})\frac{T_1(\mathcal{J})}{|\mathcal{P}|}$.

Therefore, $\mathrm{T}(\mathcal{J}) \leq r(J_i) + L + |S(J_i)| + |D(J_i)| \leq r(J_i) + L + \frac{2T_\infty(J_i)}{1-\delta} + L\log_\rho|\mathcal{P}| + L + (\frac{1+\rho}{\delta} + \frac{2\tau}{\theta})\frac{T_1(\mathcal{J})}{|\mathcal{P}|} \leq \frac{2}{1-\delta}\max_{J_k\in\mathcal{J}}(T_\infty(J_k) + r(J_k)) + (\frac{1+\rho}{\delta} + \frac{2\tau}{\theta})\frac{T_1(\mathcal{J})}{|\mathcal{P}|} + L\log_\rho|\mathcal{P}| + 2L$.

Since there are two lower bounds on $\mathrm{T}^*(\mathcal{J})$, by [19], $\mathrm{T}^*(\mathcal{J}) \geq \max_{J_k\in\mathcal{J}}(T_\infty(J_k) + r(J_k))$, and $\mathrm{T}^*(\mathcal{J}) \geq \frac{T_1(\mathcal{J})}{|\mathcal{P}|}$, we have

$$\mathrm{T}(\mathcal{J}) \leq (\frac{2}{1-\delta} + \frac{1+\rho}{\delta} + \frac{2\tau}{\theta})\mathrm{T}^*(\mathcal{J}) + L\log_\rho|\mathcal{P}| + 2L.$$

□

Since the scheduling parameters including δ, ρ, τ, θ, L, and the number of containers in cluster $|\mathcal{P}|$, are constants once the system is well configured. Therefore, COBRA + fair scheduler is $O(1)$-competitive with respect to makespan. Finally, we state this scheduling combination is provably efficient with respect to average job response time for batched jobs.

Theorem 7. COBRA + *fair job scheduler is O(1)-competitive with respect to average job response time, when scheduling a set of batched jobs $\mathcal{J}$.*

VI. IMPLEMENTATION OF COBRA IN A REAL SYSTEM

In this section, we first describe how to estimate task characteristics and available container resources, and then how to implement COBRA design in a real system.

We implement our design in Apache Spark [13] on YARN [14] system, as shown in Fig. 4. Apache YARN is a cluster resource management platform, which breakdowns cluster scheduling into three parts. A node manager (NM) runs on every node in the cluster, responsible for managing local containers and reporting node liveness. A job manager (acting as a task scheduler) running on some node, holds the job

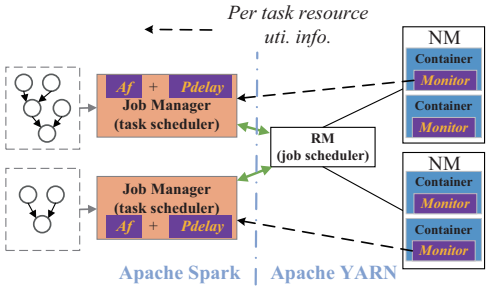
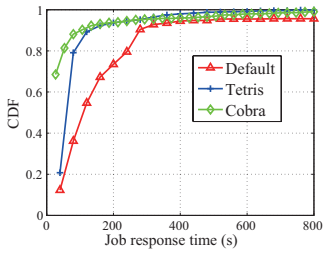


Fig. 4. Typical architecture of a data analytics system and the modifications to add COBRA and monitor process (in purple rectangles).



(a) CDF of job response time.

	Average JRT	Makespan
Default	187 s	2884 s
Tetris	108 s	1475 s
Cobra	121 s	1730 s

(b) Average job response time and makespan.

Fig. 5. Simulation results on the production trace.

TABLE III
INPUT SIZES FOR FOUR WORKLOADS.

Workload	Input sizes		
	small	medium	large
Wordcount	200 MB	1 GB	N/A
TPC-H	N/A	1 GB	10 GB
Iterative machine learning	170 MB	1 GB	N/A
Pagerank	150 MB	1 GB	N/A

context, asks resources from the cluster, and does resource allocations for its tasks. A cluster-wide resource manager (RM) receives requests from job managers and offers the allocations to the jobs. We implement per-job task scheduler COBRA in Apache Spark, through which DAG jobs could be submitted to run on YARN. Compatible with the typical two-level scheduling model, COBRA can also be implemented in computation frameworks MPI [20] and Apache Tez [21].

A. Estimating task characteristics and available resources

We estimate the dynamic resource availability on each container by adding a resource monitor process. The monitor process reads resource usages (*e.g.*, CPU, memory) from OS counters and reports these to COBRA. COBRA and its per-container monitors interact in an asynchronous manner to avoid overheads.

Based on the fact that tasks in a stage have similar resource requirements, we estimate the requirements using the measured statistics from the first few executions of tasks in a stage. We continue to refine these estimations as more tasks have been measured. We estimate task processing time as the average processing time of all finished tasks in the same stage. For the first few tasks in a stage, we over-estimate their task requirements to avoid slowing down tasks, and we set their waiting time thresholds (in `Pdelay`) to be fixed (say 5 secs).

B. COBRA implementation

We modify the original implementation of delay scheduling in Spark to take τ as a parameter read from the configuration file. When assigning tasks, `Pdelay` uses the thresholds depending on τ and task processing times.

Since resource allocation in YARN is performed in an incremental manner, we approximate `Af` as follows. We continuously (per second) measure the container utilizations in a job J_i in a period q of length L, and calculate the average at the end of the period. We acquire the desire number of containers for the next period $d(J_i, q{+}1)$ by `Af`. If $d(J_i, q{+}1) \geq d(J_i, q)$, we directly update the desire and let COBRA push this new desire to the job scheduler.

When $d(J_i, q{+}1) < d(J_i, q)$, the problem is involved, since we should proactively give up a few containers to the cluster. *Which* containers should be killed, and *when* the kill should be performed? We add controls when tasks complete. If the desire for next period is less than the current number of running containers, we aggressively kill the containers which firstly become free, until reaching the desire.

VII. EVALUATION

We evaluate COBRA using our prototype implementation on a 28-node cluster. To understand performance at larger scale, we do trace-driven simulations using production cluster traces.

A. Setup

Cluster: In our 28-node cluster, each has 2 cores, 4GB memory, 25GB disk, a 1 Gbps NIC and runs Ubuntu 14.04.

Baselines: We compare COBRA to existing work Tetris [4], which allows variable-capacity containers and assumes a complete prior knowledge of job information. Compared to clairvoyant Tetris, we show COBRA is within limited performance loss while doing semi-clairvoyant scheduling. Further, we compare COBRA to state-of-the-art scheduling algorithms implemented in Spark 2.0 and Hadoop 2.6. The static scheduling allows a job to request the default number of containers, which holds the containers in its execution until the job completes. The dynamic scheduling in Spark allows jobs to request new containers in their executions, only according to the waiting task information. Compared to it, we show the benefits of using container utilization feedbacks.

Workloads: We use workloads for our evaluation including *Wordcount*, *TPC-H benchmark*, *Iterative machine learning* and *Pagerank*. For each workload, the variation in input sizes is based on real workloads from Yahoo! and Facebook, in scale with our 28-node cluster, as shown in Table III. For the job distribution, we set 46%, 40% and 14% of jobs are with small, medium and large input sizes respectively, which also conforms to realistic job distribution [22].

Trace-driven simulations: To evaluate COBRA at larger scale. We build a trace-driven simulator that replays logs from

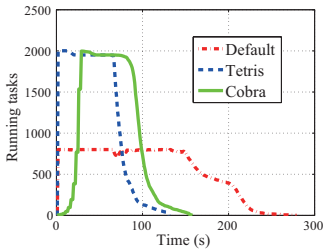
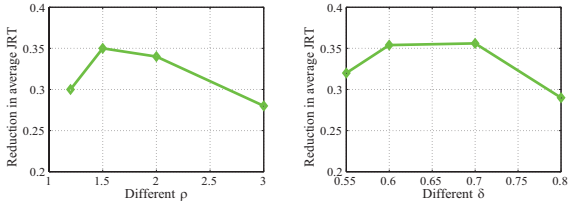
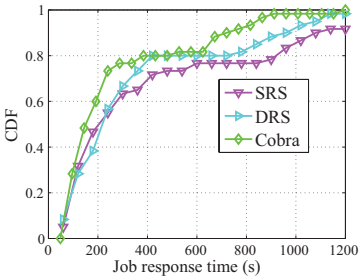


Fig. 6. Number of running tasks when scheduling large jobs.



(a) Varying ρ (b) Varying δ

Fig. 7. Reduction in average job response time in different parameter settings.



(a) CDF of job response time.

	Average JRT	Makespan
SRS	418 s	1935 s
DRS	342 s	1287 s
Cobra	261 s	1187 s

(b) Average job response time and makespan.

Fig. 8. Deployment results on typical workloads.

TABLE IV
OVERHEADS OF COBRA IN DEPLOYMENT.

	Average	Standard deviation
(NM) monitoring mechanism	33.28 ms	22.49 ms
`Af`	3.37 ms	3.97 ms
Uti. transmission delay	63.53 ms	39.11 ms

Google's production clusters [23]. We mimic these aspects from the original trace: job arrival time, durations of tasks, resource requirements of tasks, and the locations of inputs. We use the cluster with 100 machines, each with 8 cores, 16GB memory, 50GB disk, and a 1 Gbps NIC.

Metrics: Our primary metrics of interests are improvements in makespan and job response time. We compute the percentage improvement (or reduction).

B. Trace-driven simulations

In this subsection, we describe the results of different schedulers in our simulation setup. Unless otherwise specified, results use default parameters ($\rho = 1.5$, $\delta = 0.7$, and $L = 5$ secs). And we will measure the sensitivity to parameters.

Fig. 5 compares the Default scheduling strategy (fixed–capacity and disposable containers), and clairvoyant scheduler Tetris. Compared to Tetris, COBRA speedups short jobs, due to the container reusing mechanism and allowing more tasks to run in a single container. Although Tetris has the complete prior knowledge of job information and mimics SRPT strategy, COBRA is within 7% performance loss in terms of average job response time compared to it. But COBRA still improves average job response time by 35% when compared to the Default scheduler (shown in Fig. 5(b)). In terms of makespan, our design improves it by about 40% and still approximates Tetris's performance (within 9%).

To understand in a bit more details, we choose several large jobs which have more than 2000 tasks, and submit them at the time 0. Fig. 6 shows the example run. First, we see that COBRA has almost the same number of running tasks as Tetris at peak, which is much more than the Default scheduler. Second, due to our partial prior knowledge of job information, COBRA *exponentially* increases the number of containers, as well as the number of running tasks, which leads to the quickly approaching the peak requirements of the jobs.

We evaluate COBRA in different conditions. We calculate the average job response time reduction compared to Default scheduler. In Fig. 7(a), we set $\delta = 0.7$, and use different ρ. The lager value of ρ, the more aggressively the job requests new containers. Both in this trace and our deployment experiments, we find setting $\rho = 1.5$ is a good choice. Setting ρ as 3 will speed up the corresponding job but can delay other jobs since it takes resources away from them. Analogously, setting smaller δ will incur more wasted resources, delaying other job completions. While larger δ will delay the corresponding job, since the job would request new containers only when the utilization is larger than δ, delaying its own waiting tasks.

C. Deployment results

In this subsection, we use the workloads stated before, and set the job submission time following an exponential distribution with mean interval as 60 secs. We compare COBRA with Static Resource Scheduler (SRS) and Dynamic Resource Scheduler (DRS) [13]. SRS requests 2 (default) containers for each job, and lets jobs hold the allocations in their entire executions. DRS allows jobs to request new containers in their executions *solely* according to waiting task information.

Fig. 8(a) shows the CDF of job response time in different scheduling algorithms. In terms of average job response time, COBRA achieves 37% reduction compared to the SRS, which conforms to the reduction in simulations. COBRA also improves about 38% with respect to makespan (Fig. 8(b)).

We evaluate the overheads of COBRA design in deployment. Compared to the default implementation in YARN, we add the monitoring mechanism in each container process, which has moderate overhead as shown in Table IV. For the `Af` overhead, it just maintains the update operation and incurs negligible costs. As `Pdelay` has the similar computations

as the original delay scheduling, its additional overheads can also be omitted. As each COBRA continuously (per second) collects its container utilizations, we find the average delay of the utilization message transmissions is 63.53 ms across different system loads.

In summary, COBRA improves makespan and average job response time by about 35% in both our deployment and trace-driven simulations, and has limited (within 10%) performance loss when compared to the existing clairvoyant scheduler. The gains accrue from allowing fine-grained tasks to share the same containers and dynamically managing containers based on utilization feedbacks. We evaluate the moderate overheads of COBRA in deployment.

VIII. RELATED WORK

The techniques in this work are related to the following.

Scheduling techniques in DASs: Data-locality is a primary goal when scheduling tasks in recent works. Delay scheduling [17] and Quincy [24] try to improve the locality of individual tasks (*e.g.*, Map tasks in MapReduce) by scheduling them close to their input data. ShuffleWatcher [25] tries to improve the locality of the shuffle by scheduling both Map and Reduce tasks on the same set of racks. Tetris [4] ensures better packing of tasks at machines, aiming at reducing fragmentation of clusters. Corral [7] jointly places data and compute, achieving improved locality for all stages of a job.

Fairness vs. Cluster efficiency vs. Job performance: DRF [15] allocates multiple resources in a manner that is pareto-efficient, strategy-proof, envy-free and incentive sharing. Carbyne [6] allows jobs to yield fractions of their resources and reschedules these leftover resources to improve job level performance and cluster utilization. Graphene [5] constructs a schedule for a DAG by placing tasks out-of-order on to a virtual resource-time space, and uses an online heuristic to enforce the desired schedules and simultaneously manages other concerns including cluster efficiency and fairness.

Adaptive scheduling: A-Greedy [10] and GRAD [11] use adaptive feedbacks when scheduling DAG jobs to achieve bounded performance. In their works, the processors are identical to tasks and tasks are with unit processing time. While in our context, tasks may have arbitrary processing times, and tasks have distinct locality preferences. The `Af` algorithm use the waiting task information, which is absent in these works.

IX. CONCLUSION

DAG jobs are a common scheduling abstraction in data analytics systems. However, we find that the assumptions in existing works do not hold in the general case. Our task scheduler COBRA is a provably efficient online solution. The key contributions are: 1) COBRA adaptively requests new resources or gives up allocated resources according to both resource utilization and waiting task information; 2) it greedily assigns tasks to containers based on task locality preference and processing time information. We theoretically prove COBRA's performance efficiency in combination with the fair job scheduler. We experimentally validate COBRA substantially improves the scheduling of DAG jobs in both deployment and trace-driven simulations.

ACKNOWLEDGMENTS

This work is partially supported by the National Key R&D Program of China under Grant No. 2017YFB1001801; National Natural Science Foundation of China under Grant No. 61472181, 61502224, 63170028; Jiangsu Natural Science Foundation under Grant No. BK20151392. And this work is also partially supported by Collaborative Innovation Center of Novel Software Technology and Industrialization. We thank Yibo Jin for his helpful discussion about the algorithm design, and Haisheng Tan for providing us Google trace information.

REFERENCES

[1] "Apache Hadoop," http://hadoop.apache.org/.
[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys 2007*.
[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI 2012*.
[4] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *SIGCOMM 2014*.
[5] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and dependency-aware scheduling for data-parallel clusters," in *OSDI 2016*.
[6] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *OSDI 2016*.
[7] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *SIGCOMM 2015*.
[8] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *EuroSys 2012*.
[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, Jan. 2008.
[10] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, "Adaptive scheduling with parallelism feedback," in *PPoPP 2006*.
[11] Y. He, W. J. Hsu, and C. E. Leiserson, "Provably efficient online non-clairvoyant adaptive scheduling," in *IPDPS 2007*.
[12] "Apache YARN–Fair Scheduler," http://tinyurl.com/j9vzsl9.
[13] "Apache Spark," http://spark.apache.org/.
[14] "Apache YARN," http://tinyurl.com/zyy8kbc.
[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *NSDI 2011*.
[16] "Apache YARN–Capacity Scheduler," http://tinyurl.com/j739ojm.
[17] 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 *EuroSys 2010*.
[18] "Max-min fairness," https://en.wikipedia.org/wiki/Max-min_fairness.
[19] T. Brecht, X. Deng, and N. Gu, "Competitive dynamic multiprocessor allocation for parallel applications," *Parallel Processing Letters*, vol. 07, no. 01, pp. 89–100, 1997.
[20] "MPI," http://www.mpich.org/.
[21] "Apache Tez," http://tez.apache.org/.
[22] 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 *SoCC 2013*.
[23] "Google cluster-data," https://github.com/google/cluster-data/.
[24] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *SOSP 2009*.
[25] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *USENIX ATC 2014*.