SLA-Based Scheduling of Spark Jobs in Hybrid Cloud Computing Environments

Muhammed Tawfiqul Islam, *Member, IEEE*, Huaming Wu, *Member, IEEE*, Shanika Karunasekera, *Member, IEEE*, and Rajkumar Buyya, *Fellow, IEEE*

Abstract—Big data frameworks such as Apache Spark is becoming prominent to perform large-scale data analytics jobs in various domains. However, due to limited resource availability, the local or on-premise computing resources are often not sufficient to run these jobs. Therefore, public cloud resources can be hired on a pay-per-use basis from the cloud service providers to deploy a Spark cluster entirely on the cloud. Nevertheless, using only cloud resources can be costly. Hence, both local and cloud resources nowadays are used together to deploy a hybrid cloud computing cluster. However, scheduling jobs in a cluster deployed on hybrid clouds is challenging in the presence of various Service-Level Agreement (SLA) demands such as cost minimization and job deadline guarantee. Most of the existing works either consider a public or a locally deployed cluster and mainly focus on improving job performance in the cluster. In this article, we propose efficient scheduling algorithms that leverage from different VM instance pricing in a hybrid cloud deployed cluster to optimize the Virtual Machine (VM) usage cost for both local and cloud resources and maximize the job deadline met percentage. We have conducted extensive simulation-based experiments to compare our proposed algorithms with the baseline approaches. In addition, we have developed a prototype system on top of Apache Mesos cluster manager and performed real experiments to evaluate the applicability of our proposed approaches in a real platform with benchmark applications. The results show that our proposed algorithms are highly scalable and reduce the cost of VM usage of a hybrid cluster for up to 20 percent.

Index Terms—Spark, hybrid cloud, cluster-scheduling, SLA, big data applications, deadline, cost-minimization

1 Introduction

Analysing data at a massive scale is becoming crucial due to the availability of huge data in various domains such as scientific research, social media, business. Several prominent big data processing platforms such as Hadoop [1], Spark [2] and Storm [3] are used to analyze this enormous volume of data. A big data processing platform can be deployed in local-premises using computing resources owned by a company. Besides, as cloud service providers offer flexible, scalable, and affordable computing resources on-demand, it is also becoming popular to deploy a big data processing cluster in the cloud. Although most of the deployments of a big data computing cluster are either local, or on the cloud, many organizations are also using a hybrid setup where both local and cloud resources are used together to form the cluster.[1] However, it is challenging to schedule jobs in a cluster deployed on hybrid clouds while ensuring the SLA parameters such as monetary cost minimization, and deadline. In this paper, we propose scheduling algorithms that can satisfy the SLA requirements of the jobs in a big data processing cluster deployed in a hybrid-cloud.

We have chosen Apache Spark as our target big data processing platform as it is vastly replacing traditional Hadoop-based platforms. Spark can utilize memory to store intermediate results to speed up the processing. Moreover, it is more scalable than other platforms and more suitable for running complex analytics jobs. Spark programs can be written in many high-level programming languages, and it also supports diverse data sources such as HDFS [4], Hbase [5], Cassandra[6] and Amazon S3.[2] The data abstraction of Spark is called Resilient Distributed Dataset (RDD) [7], which is fault-tolerant. When a Spark job is launched, it creates one or more executors that use a fixed chunk of resources in any cluster nodes. These executors are used by a job to run multiple tasks in parallel at different stages of the data processing pipeline to work on various partitions of the dataset.

The default scheduler of Spark is FIFO[3], which schedules the jobs on a first-come-first-serve basis. The executors from a job are distributed in different nodes in a round-robin fashion for balancing the cluster load and improve performance. In addition, it can also consolidate the core usages and minimize the total number of nodes used in the cluster. However, if the nodes (VMs) are deployed in a public cloud,

1. [Online]. Available: https://www.marketsandmarkets.com/PressReleases/hybrid-cloud.asp

2. [Online]. Available: https://aws.amazon.com/s3/

3. [Online]. Available: https://spark.apache.org/docs/latest/job-scheduling.html

• *Muhammed Tawfiqul Islam, Shanika Karunasekera, and Rajkumar Buyya are with the Cloud Computing and Distributed Systems (CLOUDS) Lab, School of Computing and Information Systems, The University of Melbourne, Parkville, VIC 3010, Australia. E-mail: {tawfiqul.islam, karus, rbuyya}@unimelb.edu.au.*
• *Huaming Wu is with the Center for Applied Mathematics, Tianjin University, Tianjin 300072, China. E-mail: whming@tju.edu.cn.*

Manuscript received 25 May 2020; revised 30 Mar. 2021; accepted 18 Apr. 2021. Date of publication 27 Apr. 2021; date of current version 7 Apr. 2022. (Corresponding author: Muhammed Tawfiqul Islam.) Recommended for acceptance by S. Helal. Digital Object Identifier no. 10.1109/TC.2021.3075625

distributing the executors across different VMs can be costly as most of the VMs will be always turned on. In addition, there will be free resources in these VMs in an off-peak period when not many jobs are running in the cluster at the same time. Furthermore, if a hybrid cloud setup is considered, challenges within inter-cluster scheduling exist which include: design issues for federated multi-cluster, latency issues between different regional sub-clusters, and locality of the data. There are numerous works on inter-cluster schedulers [8], [9], which focus on addressing these challenges from a performance standpoint. However, these schedulers do not consider the VM usage cost of the Spark cluster deployed in a hybrid cloud setup. In this paper, we complement these works and address two key objectives for hybrid cloud scheduling: cost-minimization and deadline violation reduction. We propose scheduling algorithms that work on the cluster-scheduling level, and utilize the pricing of different VM instance types in a hybrid cloud to effectively handle the following challenges:

- Performing cluster-level scheduling to make fine-grained decisions for executor placements on a hybrid cloud environment.
- Minimizing the deadline violations for the jobs in the cluster.
- Minimizing the monetary cost of using the Virtual Machines (VMs) of the whole cluster.

In summary, our work makes the following key *contributions*:

- We formulate an optimization problem for SLA-based scheduling of Spark jobs in a hybrid cloud.
- We propose two job scheduling algorithms. The first algorithm is a modified version of the First-Fit (FF) heuristic for solving bin packing problems. The second algorithm uses a greedy approach to iteratively find the cost-optimal placement for each executor of a job. Both algorithms can improve the cost-efficiency of a hybrid Apache Spark cluster.
- We develop an event-based simulator in Java that can be used to simulate, test, and compare different job scheduling policies.
- We implement both of the proposed algorithms on top of Apache Mesos [10] cluster manager with separate extendable modules. Therefore, the implemented system is pluggable to Mesos and can be easily deployed in a hybrid cloud setup.
- We conduct extensive experiments in both simulated and real environments. Furthermore, we use real applications and workload traces under different scenarios to showcase the superiority of our proposed algorithms over the existing approaches.

The rest of the paper is organized as follows. In Section 2, we present the background to different frameworks and also the architectural considerations for a hybrid cloud deployment. In Section 3, we discuss the existing works related to this paper. In Section 4, we show the system model and formulate the scheduling problem. In Section 5, we present the proposed algorithms. In Section 6, we show the simulation experiment setup, baseline algorithms and experimental results for simulation-based experiments. In Section 7, we showcase the implemented prototype system in real platforms, discuss the benchmark applications and real experimental cluster setup, and demonstrate the feasibility of the proposed algorithms with performance evaluation from real experimental results. Section 8 concludes the paper and highlights future work.

2 Background

2.1 Apache Spark

As compared to the disk-based MapReduce tasks of a typical Hadoop system, Apache Spark allows most of the computations to be performed in memory and provides better performance for some applications such as iterative algorithms. The intermediate results are written to the disk only when they cannot be fitted into the memory. Spark uses *Resilient Distributed Datasets (RDD)* to hold data in a fault-tolerant way. Each job/application is divided into multiple sets of tasks called stages which are inter-dependant. All these stages form a directed acyclic graph (DAG) and each stage is executed one after another. In a typical Apache Spark cluster, applications are submitted through a cluster manager to run in the cluster. Spark supports *Apache Mesos*, or *Hadoop Yarn*, or *Kubernetes* as cluster managers to allocate resources among applications. In addition, its own default *Standalone* cluster manager is also sufficient to handle a production cluster. All these cluster managers support both static and dynamic allocation of resources.

Workers are the physical/compute nodes of an Apache Spark cluster where one or more application processes can be created depending on the resource capacity. In cloud deployments, one or more worker nodes can be created inside each Virtual Machine (VM). A Spark cluster can have one or more worker nodes but there is only a single *Master* node that is responsible for managing the worker nodes. Each application in Spark has a *SparkContext* object in its main program (also called the *Driver Program*) which creates and maintains *Executor* processes on worker nodes. An application uses its own set of executors to run tasks in parallel, in multiple threads and to keep data in memory and storage. In addition, these executors live for the whole duration of that application. All the executors of the same application must be identical in size. Hence, they will have the same amount of resources (CPU cores, memory, disk). There are two benefits of isolating applications from each other. First, a driver program can independently schedule its own tasks in the acquired executors. Second, each worker can have multiple executors from different applications running in their own JVM processes.

2.2 Apache Mesos

Apache Mesos is considered to be a data-center level cluster manager due to its capability of efficient resource isolation and sharing across distributed applications. In Mesos, jobs/applications are called frameworks and multiple applications from different data processing frameworks like Spark, Storm, and Hadoop can run in parallel in the cluster. Mesos introduces a novel two-level scheduling paradigm where it decides a possible resource provisioning scheme according to the weight, quota or role of a framework and offers resources to it. The framework's scheduler is responsible for either rejecting or accepting those resources offered by Mesos according to its scheduling policies. Mesos provides HTTP[4] APIs to control the resource provisioning and scheduling of the whole cluster. Mesos supports dynamic resource reservations, thus resources can be dynamically reserved in a set of nodes by using the APIs and then a job/framework can be scheduled only on those resources. When a job is completed, resources can be taken back and reserved for any future job. It is a significant feature of Mesos as any external scheduler implemented on top of Mesos can have robust control over the cluster resources. Furthermore, the external scheduler can perform fine-grained resource allocation for a job in any set of nodes with any resource requirement settings. Lastly, various policies can be incorporated into an external scheduler without modifying the targeted big data processing platform or Mesos itself; so the scheduler can be extended to work with other big data processing platforms. For the benefits mentioned above, we have built a prototype system on top of Mesos to implement our proposed scheduling algorithms. The proposed scheduling algorithms can be plugged to work with other modern cluster managers, such as Kubernetes, which also supports fine-grained resource allocation for containers (e.g., pods from Kubernetes terms). Kubernetes provides a scheduling framework, which adds new plugin APIs on top of the default scheduler to implement new scheduling features. Thus, by utilizing the plugin APIs, pods can be allocated to a specific node by following a new scheduling policy.

2.3 Scheduling Levels

From the above discussion, we can observe that there are two levels of scheduling in the cluster. These are (1) Cluster Level: decision to select an appropriate VM to create an executor for a Spark job. From the cluster manager perspective, a container can be created and allocated with a fixed set of resources and then this container can be assigned to a job's executor. (2) Application Level: The Spark application driver process is responsible for scheduling tasks in the provisioned executors for a job. This scheduler should consider the locality of the data to improve the performance of a job. In this paper, we work on the cluster level to decide in which VM each executor of a job should be created so that we can optimize the overall cluster usage cost. In addition, we also consider the deadline constraint to prioritize jobs with tight deadlines. As our proposed approaches work on a higher level, they can be applied to the Hadoop jobs as well. For example, a cluster manager such as Mesos supports jobs from different types of frameworks such as Hadoop and Spark. Thus, the proposed scheduling algorithms can be extended to support Hadoop jobs, where each Mesos container should be provisioned for a map or a reduced task.

2.4 Hybrid Cloud Deployment

There are different architectural considerations regarding the deployment of a hybrid cloud. For example, in a true multi-cluster setup, all the executors of a job should be placed in the same cluster. However, in a multi-cluster federation, there is a central point of control and the same job's executors can be distributed across multiple clusters. The latter approach may result in locality and latency issues, as the executors from the same job have to communicate over different regional boundaries. However, in a multi-cluster federation, there is only one cluster (although over multiple regions) from the job's perspective. In addition, there is more room for cost-efficiency as it is possible to squeeze out the free resources in the cheapest VMs across multiple clusters. Thus, in this paper, we choose a federated multi-cluster setup, where a central Mesos cluster manager is responsible to manage all the VMs across two different regions. The Mesos cluster manager is deployed in the local region to work as the central point of control. In addition, the external scheduler and other resource reservation modules are also run locally for faster communication with the cluster manager. Although there can be latency and performance issues caused by this setting, we try to capture these issues in the system model by considering the increase in job completion times caused by these issues.

3 Related Work

The default framework scheduler for Spark is FIFO, which places the executors of a job in a round-robin fashion to balance the load in the cluster and improve performance. In addition, it can also consolidate the core usage to minimize the total nodes used in the cluster. However, it does not consider the pricing of VM instances in either a single or a hybrid cluster setup. Fair[5] and DRF [11] based schedulers can be used to improve the fairness among multiple jobs in a cluster, but they do not improve the cost-efficiency of the cluster.

There is some existing research for SLA-based job scheduling, which only focuses on Hadoop MapReduce-based jobs. Hwang *et al.* [12] proposed a resource provisioning model that can minimize the VM cost for deadline-constrained MapReduce applications in cloud. Mashayekhy *et al.* [13] proposed a greedy algorithm that finds the assignments of the map and the reduce tasks in machine slots to minimize the energy consumption of a Hadoop cluster. Nayak *et al.* [14] proposed a negotiation-based adaptive scheduler for scheduling Hadoop jobs in cloud. Cheng *et al.* [15] have considered future resource availability to improve job performance and reduce job deadline violations. Zeng *et al.* [16] proposed a greedy algorithm that reduces the monetary cost of using the public cloud while satisfying job deadlines. ChEsS [17] is a Pareto-based job-to-cluster assignment framework for cost-effective job scheduling across multiple MapReduce clusters. However, most of these works either consider a single cluster setup or tries to improve job performance. Moreover, these approaches are applicable to Hadoop jobs only as the architecture paradigm of Hadoop is different from Spark.

There are a few works that tried to improve different aspects of scheduling for Spark-based jobs. Sparrow [18] tried to improve the performance of the default Spark scheduling by using a decentralized, randomized sampling-

4. [Online]. Available: http://mesos.apache.org/documentation/latest/operator-http-api/

5. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

based scheduler. Wu *et al.* [19] proposed a framework that provides the capability to perform large-scale data analytics across multiple-clusters. Maroulis *et al.* [20] provided an energy-efficient scheduler that uses the DVFS technique to tune the CPU frequencies for the workloads to reduce energy consumption. However, as our main target is cost-effectiveness, this approach can not be applied to our problem. Li *et al.* [21] also provided an energy-efficient scheduler. However, it does not consider cost as an objective. In addition, the algorithm assumes each job has the same executor size, which is equal to the total resource capacity of a VM. However, in reality, each job can have different resource requirements, and the VM instance size can also vary. Liu *et al.* [22] proposed a hierarchical multi-cluster big data framework for Apache Spark, which only focuses on improving job performance when the cluster is deployed in a hybrid-cloud. However, they do not consider any cost-efficiency in these clusters, and job deadlines. Sidhanta *et al.* [23] provided a mathematical model to estimate job completion times of a Spark job given its input size, iteration, and job type. In addition, they provide an optimal cluster composition technique that utilizes the default FIFO scheduler. However, this work does not consider different VM pricing in a hybrid-cloud setup. In addition, it is assumed that each job has the same executor size, which is the total resource capacity of a VM. However, we model the executor sizes at a more fine-grained level, so that multiple executors from one or more jobs can be co-located inside a single VM.

MCTE [24] is a cloud task scheduling strategy to minimize the task completion time and execution cost for the smart grid cloud. However, this work did not consider a hybrid cloud setup and cost minimization as an objective. AsQ [25] is also a task scheduling algorithm that places the task in either local or cloud VMs. Peláez *et al.* [26] introduced the problem of managing virtual machines and scheduling jobs in a cost-efficient way while meeting the deadlines. In the bag of tasks model, the tasks are independent of each other so the run-time of an individual task does not depend on whether another task from the same bag is placed in the cloud or local VM. However, in our work, we focus on the cluster-level scheduling where an executor runs one or more interdependent tasks that follows a DAG model.

If a cluster is deployed in a hybrid cloud, some of the VMs reside in the local premises/region and the rest of the VMs are hired from a cloud service provider. Thus, the cloud portion of the cluster can be considered to be in a different region. Therefore, challenges within inter-cluster scheduling exist which include: choosing a proper federated multi-cluster setup that determines how the clusters should be managed, increased latency between different executors deployed in different regions, and locality of the data required for a job. There are numerous works on inter-cluster schedulers, e.g., Yarn Federation,[6] Kubernetes Federation,[7] Medea [8] and Hyrda [9], which focused on addressing these challenges with objectives to improve the overall performance of the production cluster. Because for multiple regional clusters, it is more critical to focus on performance improvement and load-balancing. However, if a hybrid cloud setup is created with the use of public cloud VMs, minimizing cluster resource usage cost should be a key objective, along with maintaining an acceptable performance for the applications.

In summary, most of the existing approaches focus mainly on performance improvement. In addition, they do not consider a fine-grained level of executor placement while scheduling jobs. In contrast, our approach guarantees to launch a job on its required resources, tries to minimize deadline violations, can handle different sizes of executors of jobs and different VM instance sizes, and can reduce the overall cost of VM usage of a hybrid cloud deployed cluster by utilizing different pricing.

TABLE 1
Definition of Symbols

Symbol	Definition
J	The current job to be scheduled
E	Total executors required for J
ξ	The index set of all the executors of J, $\xi = \{1, 2, 3, \ldots, E\}$
T_C^L	Profiled completion time for J for local-only placement of executors
T_C^H	Profiled completion time for J for hybrid placement of executors
T_C	Estimated completion time for J
T_D	Deadline for J
T_A	Arrival time for J
T_S	Start time for J
T_W	$T_S - T_A$, waiting time for J
M	The total number of local VMs
N	The total number of cloud VMs
δ^L	The index set for all the local VMs; $\delta^L = \{1, 2, \ldots, M\}$
δ^C	The index set for all the cloud VMs; $\delta^C = \{1, 2, \ldots, N\}$
P_j^L	The Price for a local VM; $j \in \delta^L$
P_j^C	The Price for a cloud VM; $j \in \delta^C$
C_j^L	Available CPU in a local VM, $j \in \delta^L$
C_j^C	Available CPU in a cloud VM, $j \in \delta^C$
M_j^L	Currently available Memory in a local VM, $j \in \delta^L$
M_j^C	Currently available Memory in a cloud VM, $j \in \delta^C$
C_i^τ	CPU demand of any executor of J, $i \in \xi$
M_i^τ	Memory demand of any executor of J, $i \in \xi$
t_j^L	Remaining active time for a VM before placing executor(s) of J, $j \in \delta^L$
t_j^C	Remaining active time for a VM before placing executor(s) of J, $j \in \delta^C$
Δt_j^L	Change in remaining active time after executor(s) of J is placed, $j \in \delta^L$
Δt_j^C	Change in remaining active time after executor(s) of J is placed, $j \in \delta^C$

4 SLA-based Job Scheduling

In this section, we describe the hybrid cloud model and formulate the problem of dynamic job scheduling between local VMs and cloud VMs. Major notations and descriptions presented in this paper are listed in Table 1.

4.1 System Model

When a hybrid cloud setup is considered, both local and cloud VM instances can be chosen to be identical in resource

6. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/Federation.html
7. [Online]. Available: https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution/

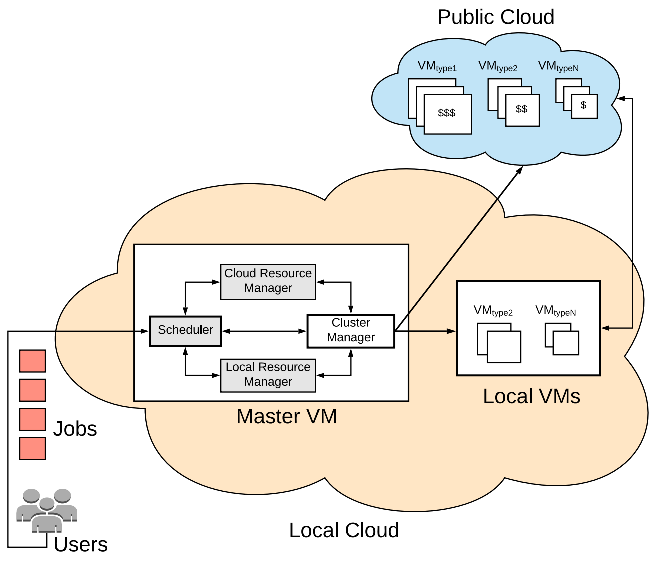


Fig. 1. Proposed Hybrid Cloud Model. The resource managers (cloud and local) are controlled by the scheduler to create executors of job in VMs, turn on/off VMs, and monitor the cluster states.

capacity. However, when the target objective is to reduce cost, having a setup with different types of VM instances is more cost-effective, because jobs with fewer resource requirements can be fitted into small VMs to optimize cost. In addition, if the local part of the cluster is made with commodity resources, it is not possible to create similar VM instances with a set of heterogeneous physical hosts. Therefore, to tackle the scheduling problem more efficiently, the scheduler has to consider different VM instance sizes (as depicted in Fig. 1) to optimize cost. We consider a federated multi-cluster deployment where the cluster manager is the central point of control. The cluster manager controls both the local and the cloud VMs. The resource managers track the resource availability of the cluster and dynamically feed the updated status of the cluster to the scheduler. Thus, the scheduler has to match the resource requirement of the jobs with the resource availability in the cluster while trying to meet the target objectives. In our implemented prototype, we deploy the external scheduler, both of the resource managers, and the cluster manager in the local cloud.

In an Apache Spark cluster, each job consists of a set of executors with the same resource requirement. Furthermore, each VM/worker node has a set of available resources (e.g., CPU and memory) which can be used to place executors. However, executors from different jobs can have different sizes. For example, suppose the CPU and memory requirements of an executor of job-1 are 2 cores and 4GB, respectively. Thus, if job-1 has 5 executors, all the executors must follow this resource requirement (e.g., 2 cores and 4GB memory). However, job-2 can have different resource requirements for its executors. For example, 4 cores, and 8GB of memory for each executor, which is different from the size of the executors from job-1.

For each submitted job in the cluster, the main problem is to find the mapping of all its executors to one or more available VMs. Besides, the combined resource requirements of all the placed executors in a VM are bound by its resource capacity. Therefore, resource constraints in each VM must be met while making any scheduling decisions. We consider a multi-tenant case where multiple jobs from different users can run on the cluster at the same time. Thus, if one or more executors from different jobs are placed in the same VM, then the resource capacity constraints of that VM must be satisfied by considering all the different sizes of executors from multiple jobs. This problem can be simplified by tracking the resource availability of VMs dynamically. Thus, the resource availability of the VMs can be presented to the scheduler, instead of the resource capacity. Initially, the resource capacity and resource availability of a VM will be the same. Although the resource capacity of a VM is always fixed, the resource availability of a VM will be reduced over time if one or more executors from one or more jobs are placed in it. In addition, if one or more jobs complete execution that had executor(s) in this VM, then the resource availability of the VM will be increased.

We consider the resource requirement of an executor in two dimensions – CPU cores and memory. Suppose, we are given a job with E executors where each executor has CPU and memory requirements of τ_i^{cpu} and τ_i^{mem}, respectively. Furthermore, each job has a deadline that needs to be met by the scheduler. After handling all the constraints, the scheduler should try to reduce the resource (or VM) usage cost of the cluster. In our case, we have a hybrid cloud setup where some VMs are located in local-premises and some VMs are hired from the cloud on a pay-per-use basis. We assume that if the resource requirements are met, the performance of all the executors from the same job is similar regardless of whether they are placed on local or cloud VMs.

Suppose J is the current job to be scheduled in the cluster. If one or more previous jobs are still running in the cluster, the scheduler has to make a decision on whether to utilize the spare resources on the already active VMs to place one or more executors of J, or turn on new local/cloud VMs. Therefore, to make a cost-optimal scheduling decision for each job, the scheduler should use a combination of both local/cloud VMs.

In our proposed model, the scheduler uses a queue which follows the EDF (earliest deadline first) order of jobs, to reduce deadline violations. The scheduler iterates over each job, dynamically observes the latest cluster resource availability, and makes scheduling decisions to place the executors for that job. For simplicity, we present the model on a per-job basis, which means the model represents what the scheduler observes for making decisions for the next job in the scheduling queue. In the following Sections 4.2 and 4.3, we model the cost and resource constraints for both local and cloud VMs. Then in Section 4.4, we combine the resource models and constraints to formulate the scheduling problem.

4.2 Local Resource Model

Definition 1. ***(Local VM Set):*** *Consider a set* $\delta^L = \{1, 2, \cdots, M\}$*, where* M *is the total number of local VMs,* $1 \leq j \leq M$ *is the jth VM deployed locally.*

The expression of local cost for the current job, J *is derived as follows:*

$$Cost^L = \sum_{j \in \delta^L} x_j \times P_j^L \times \Delta t_j^L, \tag{1}$$

where P_j^L is the unit price for a local VM and we define a binary decision variable x_j to indicate whether a local VM is active or not, i.e.,

$$x_j = \begin{cases} 1 & \text{if } \sum_{i\in\xi} u_{ij} > 0; \\ 0 & \text{otherwise} \end{cases}, \tag{2}$$

where we define a binary decision variable u_{ij} to indicate whether executor i is placed in a local VM j or not, i.e., $\forall j \in \delta^L$, we have

$$u_{ij} = \begin{cases} 1 & \text{if executor } i \text{ is placed in the local VM } j; \\ 0 & \text{otherwise} \end{cases}, \tag{3}$$

Δt_j^L is the change in the remaining active time for a local VM j if any executor of J is placed in it, which is calculated by:

$$\Delta t_j^L = \begin{cases} T_C - t_j^L & \text{if } T_C > t_j^L; \\ 0 & \text{otherwise} \end{cases}, \tag{4}$$

Here, t_j^L is the remaining active time for a local VM before placing any executor of the current job. T_C is the estimated completion time of the current job, J. We assume that the T_C can be provided for each job, which is generally measured from the job profile information. Now, the executors of the job can be placed only on the local VMs, or in a hybrid manner where both local and cloud VMs can be used. However, if any cloud VMs are used for executor placements, T_C will be higher than local-only placements, due to local to Cloud data transmissions and network latency. Suppose, the job is profiled in both settings, and T_C^L indicates the profiled completion time for the job for local-only placement. In addition, T_C^H indicates the profiled job completion time for a hybrid setting. Thus, for the local model, T_C can be defined as:

$$T_C = \begin{cases} T_C^L & \text{if } \sum u_{i,j} == E \;\; \forall i \in \xi, \forall j \in \delta^L; \\ T_C^H & \text{otherwise} \end{cases}. \tag{5}$$

Here, E is the total number of executors required by the job, so if the summation of all the local placements equals E, it indicates that the job will be running entirely in the local VMs.

Furthermore, the total resource demands of all the executors placed in a VM should not exceed the total resource capacity of that VM. Note that, this can be done simply if the current resource availability of the VM is checked against the resource demands of executor(s) of the current job. Suppose, C_i^τ and M_i^τ are the CPU and memory resource demands for each executor of the current job, respectively. Thus, the resource constraints for local VMs must be satisfied as follows:

$$\sum_{i\in\xi}(u_{ij} \times C_i^\tau) \le x_j \times C_j^L, \qquad \forall j \in \delta^L, \tag{6}$$

$$\sum_{i\in\xi}(u_{ij} \times M_i^\tau) \le x_j \times M_j^L, \qquad \forall j \in \delta^L. \tag{7}$$

where C_j^L and M_j^L are the currently available CPU and memory resources in the local VM j, respectively. Therefore, the scheduler can choose to place one or more executors from the current job in the same VM if the current resource availability permits.

4.3 Cloud Resource Model

Definition 2. ***(Cloud VM Set):*** *Consider a set $\delta^C = \{1, 2, \cdots, N\}$, where N is the total number of cloud VMs, $1 \le j \le N$ is the jth VM deployed on the cloud.*

Similarly, the expression of cloud cost for the current job, J is derived as follows:

$$Cost^C = \sum_{j\in\delta^C} y_j \times P_j^C \times \Delta t_j^C, \tag{8}$$

where P_j^C is the unit price for a cloud VM; we define a binary decision variable y_j to indicate whether a cloud VM is active or not, i.e.,

$$y_j = \begin{cases} 1 & \text{if } \sum_{i\in\xi} v_{ij} > 0; \\ 0 & \text{otherwise} \end{cases}, \tag{9}$$

where we define a binary decision variable v_{ij} to indicate whether executor i is placed in a cloud VM j or not, i.e., $\forall j \in \delta^C$, we have

$$v_{ij} = \begin{cases} 1 & \text{if executor } i \text{ is placed in the cloud VM } j; \\ 0 & \text{otherwise} \end{cases}, \tag{10}$$

Δt_j^C is the change in the remaining active time for a cloud VM if any executor of the current job is placed in it, which is calculated by

$$\Delta t_j^C = \begin{cases} T_C^H - t_j^C & \text{if } T_C^H > t_j^C; \\ 0 & \text{otherwise} \end{cases}, \tag{11}$$

where T_C^H is the estimated completion time of the current job, when one or more cloud VMs are used; and t_j^C is the remaining active time for a cloud VM before placing any executor of the current job. Further, the resource constraints for cloud VMs must be satisfied as follows:

$$\sum_{i\in\xi}(v_{ij} \times C_i^\tau) \le y_j \times C_j^C, \qquad \forall j \in \delta^C \tag{12}$$

$$\sum_{i\in\xi}(v_{ij} \times M_i^\tau) \le y_j \times M_j^C, \qquad \forall j \in \delta^C. \tag{13}$$

On the one hand, because the total number of the local VMs might be limited, we can use the cloud VMs for computing. Therefore, we can assume that $M < N$. On the other hand, however, the usage cost of the VMs in local VMs is usually lower than that on the cloud; hence we can assume that $P_j^L < P_j^C$. Therefore, similar VM instances deployed in local-premises cost lower than cloud VM instances.

4.4 Problem Formulation

Based on the system model, we now formulate the job scheduling problem to minimize the cost of using the whole cluster while scheduling the current job. The total cost is modeled as the aggregated cost of using all the VMs from both local and cloud.

Executor Placement Constraint. An executor can be placed only in one of the VMs and this placement constraint is denoted as:

$$\sum_{j\in\delta^L} u_{ij} + \sum_{j\in\delta^C} v_{ij} = 1, \qquad \forall i \in \xi. \tag{14}$$

Resource Capacity Constraints. The total resource demands of all the executors placed in a VM should not exceed the total resource capacity of that VM. These constraints are described in (6), (7), (12) and (13).

Job Deadline Constraint. If the job deadline is considered, whether a job fails to complete before the given deadline can be predicted by using:

$$T_C < T_D - T_W, \tag{15}$$

where $T_W = T_S - T_A$ is the waiting time for the current job to be scheduled. Note that, if the executors are not placed entirely in the local VMs, then T_C will be set to T_C^H in the local resource model.

On the one hand, if the job deadlines are not considered in the scheduling algorithm, a job that is predicted to fail will be scheduled, only to waste resources which could be used by any future job to successfully complete before their deadlines. On the other hand, if a job is predicted to violate its deadline, it can be discarded without passing to the scheduling algorithms. Thus, more resources will be freed to ensure that more jobs can be successfully finished before the deadline. In the experiment section, we show the impact on deadline violations by the scheduling algorithms for both cases.

Therefore, the job scheduling problem can be formulated as Cost-Min:

$$\begin{aligned} \min : & \; Cost^{total} = Cost^L + Cost^C, \\ s.t. \; : & \; (6), (7), (12), (13), (14), (15). \end{aligned} \tag{16}$$

The above problem is mixed-integer linear programming (MILP) [27] and non-convex [28], generally known as NP-hard problem [29]. The computational complexity will significantly increase due to the binary variables.

5 Proposed Job Scheduling Algorithms

We try to maximize the deadline met percentage by two ways: (1) by following an Earliest Deadline First (EDF) order to schedule jobs, so that if multiple jobs are waiting to be scheduled at the same time, jobs with tighter deadlines will have higher priority, and (2) before passing the job specifications to the scheduler, we utilize a job's completion time estimate (T_C) to check whether the job has a chance of violating the deadline. If so, we remove this job from the queue and do not schedule it. In this way, we keep some resources free in the cluster for future jobs to increase the overall deadline met numbers. The job queue is maintained externally from the scheduling algorithm, along with the cluster resource availability. Both the job queue and the cluster states are updated dynamically. Only the current job's specification and the cluster states are passed to a scheduling algorithm to make placement decisions. In this way, we reduce the overhead on the scheduling algorithm. If it is estimated that the job will be completed before the deadline, it is passed to the scheduler to make cost-effective executor placement decisions. We propose two algorithms to solve the scheduling problem. The first algorithm is a modified version of the First Fit (FF) heuristic algorithm for the bin packing optimization problem. The second algorithm has a greedy approach and iteratively places all the executors of a job in the most cost-optimal position.

5.1 First Fit (FF) Heuristic-Based Algorithm

In the bin packing problem, items of different volumes must be packed into a finite number of bins or containers each of a fixed given volume in a way that minimizes the number of bins used. In our case, we have a similar problem where the executors can be considered as the items which need to be packed into a finite number of VMs (bins). Thus, the scheduling problem formulated in Section 4.4 can be thought of as a two-dimensional (2D) vector bin packing problem, where each of the VM is a bin having two dimensions, i.e., CPU cores and memory. Each executor from a job has a fixed resource requirement in these two dimensions; thus, an executor can be thought of as an item. Therefore, the objective is to minimize the total number of bins (VMs) used to pack (place) a given set of items (executors) for each job. Algorithm 1 shows the modified version of the First Fit (FF) heuristic [30] algorithm, which can be used for executor placement in the scheduling process.

Algorithm 1. First Fit (FF) Heuristic Algorithm

```
Input: Job {$E, \xi, C_i^\tau, M_i^\tau, T_C$}: The current job to be scheduled, ActiveVMList: The list of all the active VMs (includes both cloud and local VMs)
Output: PlacementList, a list of VMs where the executors of Job will be placed
1: Procedure (FF(Job, ActiveVMList))
2:   PlacementList ← $\phi$
3:   forall vm ∈ ActiveVMList do
4:     while Placement of an executor in vm satisfies all the resource constraints do
5:       Update(vm)
6:       PlacementList.add(vm)
7:       if PlacementList.size = E then
8:         return PlacementList
9:       end
10:    end
11:  end
12:  if Cluster has unused VM(s) then
13:    Turn on the cheapest $vm_{new}$ that satisfies all the resource constraints of an executor
14:    ActiveVMList ← ActiveVMList ∪ $vm_{new}$
15:    goto step 3
16:  end
17:  return Failure
18: end Procedure
```

The input to this algorithm is the specification of the current job ($E, \xi, C_i^\tau, M_i^\tau, T_C$) to be scheduled, and a list of currently active VMs (either in local or cloud) in the cluster. The output is the *PlacementList*, which is a list of VMs where the executors of the current job should be placed. For each active VM, the algorithm first checks whether the placement of an executor of the current job will satisfy the resource constraints (lines 3-4). If so, the resource capacity of the current VM is updated (line 5), and the current VM is added to the $PlacementList$. The algorithm tries to place as many executors as possible in the same VM if the resource requirements are met. Otherwise, it tries the next active VM. If the total number of added VMs to the $PlacementList$ reaches the total required number of executors for the current job, the algorithm returns with the placement list. If the currently active VMs are not sufficient to place any executor, then the cheapest VM is turned on (if available) and is added to the active VM list (lines 12-14). Then, steps 3-10 are repeated again. If the cluster does not have sufficient resources to place all the executors of the current job, the algorithm returns failure (line 17).

5.2 Greedy Iterative Optimization (GIO) Algorithm

The aforementioned MILP problem can be solved in polynomial time if the problem is relaxed from a per-job basis (finding the most cost-optimal placements of all the executors of the current job) to a per-executor basis (only find the most cost-effective placement of one executor from the current job). Although solving the relaxed problem will provide near-optimal results as compared to the original problem, it can be solved in polynomial time. We propose a greedy iterative optimization (GIO) algorithm, which utilizes the pricing model of different VM instances and the estimated completion time of each job to find cost-efficient executor placement (on a per-executor basis).

Suppose, the executor(s) from one or more jobs are running in a vm (deployed either in the cloud or in the local part of the cluster). Let T_{vm} be the active remaining time of the vm. If any executor of the current job J is placed in vm, the additional active remaining time of vm due to this placement is ΔT_{vm}, which can be found in:

$$\Delta T_{vm} = \max(0,\, T_C - T_{vm}). \tag{17}$$

Now, if the cluster has sufficient local resources to place all the executors from the current job, then T_C can be set to T_C^L, otherwise it can be set to T_C^H (Eq. (5)). Hence, we can calculate the cost incurred by placing an executor of J in vm by using:

$$Cost_e^J = \Delta T_{vm} \times P_{vm}. \tag{18}$$

Here, if vm is deployed locally, then $P_{vm} = P_j^L$ ($j \in \delta^L$). Otherwise, if vm is deployed on cloud, then $P_{vm} = P_j^C$ ($j \in \delta^C$). Suppose, vm is already in use and has some free resources to place one or more executors for the current job. If placing the new job's executor(s) in it does not make it run longer than before (if $T_C \leq T_{vm}$), or only makes it run further for a short period of time ($T_C - T_{vm}$ approaches 0), we can save cost by placing the current job's executor(s) in it.

Algorithm 2 shows the proposed GIO algorithm. The input to this algorithm is the current job to be scheduled, and a list of all the local VMs, and the list of all the cloud VMs. The output is the $PlacementList$, which is a list of VMs where the executors of the current job should be placed. At first, we check whether the current local resource availability is sufficient to place all the executors of the current job (line 3). If yes, we only utilize the local VMs (line 4), otherwise all the VMs (line 5). Then, the $VMList$ is sorted in an increasing order of $Cost_e^J$ values (line 10). If the resource constraints are met, then the current vm is greedily used to place as many executors as possible (lines 12-14). If the currently chosen vm was inactive, it is turned on (lines 15-16). The steps for executor placement are repeated until all the executors of the current job are placed (lines 18-19). If the cluster does not have sufficient resources to place all the executors for the current job, a failure is returned (line 23).

Algorithm 2. Greedy Iterative Optimization (GIO) Algorithm

```
Input: Job $\{E, \xi, C_i^\tau, M_i^\tau, T_C\}$: The current job to be scheduled, $LocalVMList$: The list of all the local VMs, $CloudVMList$: The list of all the Cloud VMs
Output: $PlacementList$, a list of VMs where the executors of $Job$ will be placed
1:  Procedure ($GIO(Job, LocalVMList, CloudVMList)$)
2:    $VMList \leftarrow \phi$
3:    if $LocalAvailability(Job, LocalVMList) == true$ then
4:      $VMList \leftarrow LocalVMList$
5:    end
6:    else
7:      $VMList \leftarrow LocalVMList \cup CloudVMList$
8:    end
9:    $PlacementList \leftarrow \phi$
10:   $Sort(VMList)$ //Sort the VMs in an increasing order of $Cost_e^J$ (Eq. (18))
11:   forall $vm \in VMList$ do
12:     while Placement of an executor in $vm$ satisfies all the resource constraints do
13:       $Update(vm)$
14:       $PlacementList.add(vm)$
15:       if $vm$ was unused then
16:         Turn on $vm$
17:       end
18:       if $PlacementList.size == E$ then
19:         return $PlacementList$
20:       end
21:     end
22:   end
23:   return $Failure$
24: end Procedure
```

Note that, for both FF and GIO algorithms, if there are not enough resources for the current job (a failure is returned by the algorithms), the scheduler will wait until more resources are freed so that it can schedule the current job.

5.3 Complexity Analysis

To calculate the worst-case time complexity of the proposed algorithms, we first assume that the total number of VMs in the cluster is m, which includes both cloud and local VMs. In the worst-case scenario, for every executor, the scheduler has to iterate through each and every VM to find its placement. Hence, if the current job's total number of executor requirements is e, the worst-case time complexity of Algorithm 1 is O(me). For Algorithm 2, the time required to check the local resource availability is m. In addition, the

TABLE 2
Simulation Cluster Details

Instance Type	CPU Cores	Memory (GB)	Quantity (small-scale)	Quantity (large-scale)
m1.large	4	16	Local=1; Cloud=2	Local=10; Cloud=50
m1.xlarge	8	32	Local=1; Cloud=2	Local=10; Cloud=50
m2.xlarge	12	48	Local=1; Cloud=2	Local=10; Cloud=50

TABLE 3
VM Instance Pricing Models

	Pricing Model 1		Pricing Model 2		Pricing Model 3		Pricing Model 4		Pricing Model (Real)	
Instance Type	Price (Cloud)	Price (Local)	Price (Cloud)	Price (Local)	Price (Cloud)	Price (Local)	Price (Cloud)	Price (Local)	Price (Cloud)	Price (Local)
m1.large	\$0.004/s	\$0.001/s	\$0.002/s	\$0.001/s	\$0.002/s	\$0/s	\$0.002/s	\$0.002/s	\$0.24/h	\$0.12/h
m1.xlarge	\$0.008/s	\$0.002/s	\$0.004/s	\$0.002/s	\$0.004/s	\$0/s	\$0.004/s	\$0.004/s	\$0.48/h	\$0.24/h
m2.xlarge	\$0.012/s	\$0.003/s	\$0.006/s	\$0.003/s	\$0.006/s	\$0/s	\$0.006/s	\$0.006/s	\$0.72/h	\$0.36/h

time required to sort the $VMList$ (which may contain all the m VMs in worst-case) is $\mathrm{mlog(m)}$. Therefore, the worst-case time complexity of Algorithm 2 is $\mathrm{O(m + mlog(m) + me)}$.

6 Performance Evaluation - Simulation

We have used both simulation and real experiments to compare our proposed scheduling algorithms with the baseline algorithms. In this section, we discuss the experimental setup for simulation experiments, baseline scheduling algorithms used to compare our proposed algorithms, and the results from the simulation experiments.

6.1 Simulation Setup

Table 2 shows the simulation cluster details. We have used three types of VMs, each having different resource capacities. We have designed the clusters for both small-scale and large-scale experiments. Generally, we have more resources on the cloud than the local part of the cluster. Therefore, the small-scale cluster contains 3 VMs from each type of VM instance, where 1 VM is considered to be deployed locally, and 2 VMs are considered to be deployed on cloud. For the large-scale experiment, 60 VMs from each type of VM instance are used, where 10 VMs are considered to be deployed locally, and 50 VMs are considered to be deployed on cloud.

The cloud VM pricing model is based on the time-based pricing where a cloud service provider offers different VM instances to their customers. For a hybrid cloud setting, the public part of the cluster can be set up by using VMs from any cloud service provider, so the prices may vary. However, when optimizing cost for the whole cluster, we need to differentiate between the price of a local or cloud VM for a similar instance type. Thus, we have designed different 'pricing models', where each model indicates how close or far the price is for the same instance type in the local and the cloud part of the cluster. As shown in Table 3, the price of the same instance type in cloud is four times higher than local in pricing model 1, but only two times higher in pricing model 2. In pricing model 3, the price of using any local instance is 0. Lastly, in pricing model 4, the price of using the same type of instance is equal regardless of whether the instance is in cloud or locally deployed.

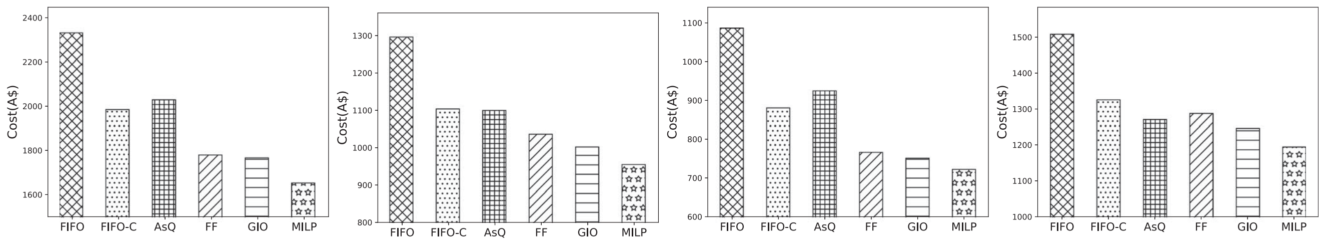
The job arrival times are generated from a Poisson distribution. We have designed our experiment to simulate both a high-load and a light-load period of the cluster. A Poisson mean of 5 and 100 is used to generate the job arrival rates for the high-load and light-load periods, respectively. These mean values for Poisson distribution are chosen to reflect job arrival rates in real clusters in both low-load and high-load periods, which is observed in Facebook Hadoop workload trace.[8] The estimated job completion time for each job is generated using an exponential distribution with lambda ($\lambda = 0.01$). In addition, if a scheduler places the executors in a hybrid setting, where one or more executors are placed in the cloud VMs, then the simulation environment dynamically increases the job completion times by 30 percent. This is due to the fact that inter-cluster latency between executors and data locality issues will cause performance degradation for the jobs. A relaxed deadline for each job is generated by adding the job's estimated completion time with a threshold value (1000 seconds for the light-load period, 5000 seconds for the high-load period). All the resource requirements for each job are generated randomly within a range of 1-6 (for CPU cores), 1-10 (for memory in GB), and 1-8 (for total executors). All the simulation experiments are repeated 5 times to accommodate the randomness while calculating the statistics.

We have implemented an event-based simulator in Java to simulate the job scheduling in a hybrid cloud setup. We have implemented the proposed and baseline algorithms in this simulator to evaluate and compare them regarding different aspects. The simulator is open-source[9], and can be used to simulate new scheduling policies.

6.2 Baseline Schedulers

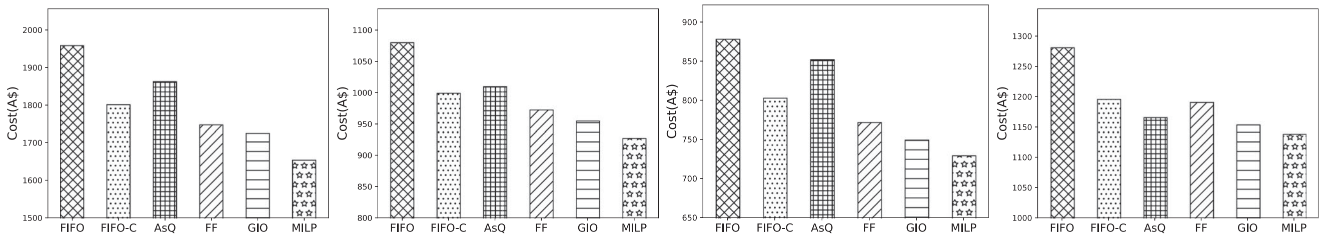
- *First in First out (FIFO):* It is used as a default scheduler in many big data processing frameworks including Apache Spark. Here, the executors of a job are placed in a round-robin fashion. However, as this

8. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository
9. [Online]. Available: https://github.com/tawfiqul-islam/RM-Simulator



(a) Pricing Model 1 (b) Pricing Model 2 (c) Pricing Model 3 (d) Pricing Model 4

Fig. 2. Cost comparison between the scheduling algorithms under different VM instance pricing models in a lightly loaded cluster (the lower the better). The scheduling delay is omitted to show how close the schedulers perform to the MILP solution regarding true cost calculation.



(a) Pricing Model 1 (b) Pricing Model 2 (c) Pricing Model 3 (d) Pricing Model 4

Fig. 3. Cost comparison between the scheduling algorithms under different VM instance pricing models in a highly loaded cluster (the lower the better). The scheduling delay is omitted to show how close the schedulers perform to the MILP solution regarding true cost calculation.

default FIFO scheduler does not consider pricing models or different instance types in the hybrid cloud, resources are wasted if the cluster is not fully loaded with jobs.

- *First in First out Consolidate (FIFO-C):* Another round-robin approach used by the Spark scheduler to minimize the total number of VMs used. Note that, it works by packing executors on the already running VMs to avoid choosing the unused VMs.
- *AsQ [25]:* This scheduler addresses the task scheduling problem in hybrid cloud and has similar objectives as our work. AsQ considers the deadline constraint and tries to minimize the cost of the public cloud by maximizing the utilization of the private cloud. In addition, to avoid network latency issues between the public and private cloud, AsQ places the tasks for a job either in a local-only or in a cloud-only manner.
- *Mixed-Integer Linear Programming (MILP):* We have designed a MILP-based scheduler that generates the optimal cost-efficient placements for all the executors of each job. We have used SCP Solver API[10] to solve the MILP problem in this scheduler. SCP solver uses a revised branch-and-cut [31] based approach for solving the MILP problem. However, the solver can take a significantly long time to solve the scheduling problem if the problem size is big (large cluster with many VMs, or jobs with many executors).

6.3 Simulation Results

In this subsection, we demonstrate the results from the simulation experiments with both small-scale and large-scale setups. However, as the MILP-based algorithm is not scalable and becomes infeasible when the problem size goes bigger, it is excluded from the large-scale simulation experiments. The small-scale experiment is used to compare the proposed algorithms with the baseline algorithms regarding cost-efficiency, scheduling overhead, and deadline violation. Furthermore, the large-scale setup is used to show the scalability of the proposed algorithms.

6.3.1 Evaluation of Cost Efficiency

In this evaluation, we have measured the cost of using the whole cluster to calculate the cost incurred by a specific scheduling algorithm. We save the turn-on or turn-off status of every VM in each second. Then we use one of the pricing models to calculate the cost incurred by using each VM during the whole scheduling process. Lastly, the total cost is calculated by summing up the cost of all the VMs. Note that, the MILP-based algorithm sometimes take exponential time to complete. Therefore, for fair cost comparison and to show how close the proposed schedulers performed to the MILP-based algorithm, the increased amount of VM usage cost due to the scheduling overhead is excluded. Figs. 2 and 3 depict the comparison of cost between different scheduling algorithms under different pricing models in both lightly loaded and highly loaded clusters, respectively. It can be observed that, under any pricing models, the proposed FF and GIO scheduling algorithms significantly reduce the cost usage of the cluster than the default FIFO and FIFO-C scheduling algorithms. The GIO scheduling algorithm can reduce the cost by up to 25 percent, whereas the FF scheduling algorithm can reduce the cost up to 15 percent than the FIFO and FIFO-C algorithms. Although FIFO-C utilizes a round-robin approach, it tries to do so in the active VMs only. Thus, this approach reduces the cost as compared to the naive FIFO. The AsQ algorithm only places the executors from the same job either in a local-only or cloud-only fashion. However, the proposed FF and GIO algorithms utilize both cloud and local VMs, thus, can reduce the cost further. The FF algorithm

10. [Online]. Available: http://scpsolver.org/

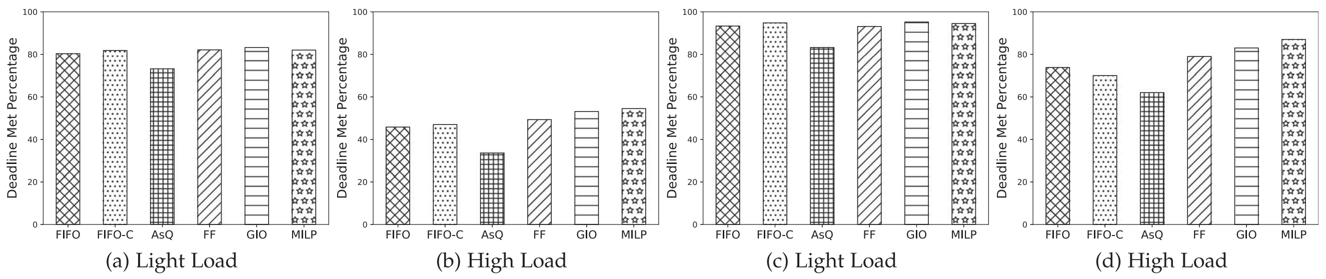


Fig. 4. Comparison of deadline met percentage between the scheduling algorithms in both high-load and light-load periods of the cluster (the higher the better). (a) and (b) shows the result when the deadline constraint is not considered. (c) and (d) shows the result when the deadline constraint is considered, and the jobs which are predicted to fail are removed from the job queue.

starts the cheapest VM when the current set of VMs do not have sufficient resource capacity to schedule a new job. When placing executors, it does not consider VM prices and job runtimes in VMs, but selects the first available VM which satisfies the resource constraints. However, as the GIO algorithm takes the job duration and pricing models into consideration, it always performs slightly better than the FF. In addition, it considers network latency and data transmission issues into consideration, and only goes for a hybrid placement if there are not sufficient local resources available. However, even for hybrid placement, it uses the spare resources from both local and cloud VMs to reduce cost significantly. As the MILP algorithm solves the scheduling problem optimally before placing the executors of each job, it provides the most cost-efficient solution. However, both FF and GIO algorithm reduces the cost significantly and operates very close to the ILP solution. Both algorithms only incur 8-10 percent more cost than the ILP algorithm under different pricing models in both lightly loaded and highly loaded clusters.

6.3.2 Evaluation of Job Deadline

This evaluation is done by taking the percentage of jobs that finish before the given deadline. We have done experimentation in two cases. In the first case, we have recorded the deadline met percentage when all the algorithms do not use the deadline as a constraint. In the second case, all the algorithms consider the deadline as a constraint, and if it can be predicted from the job estimation time that a job is going to fail to meet its deadline, that job is not scheduled. The reason to conduct experiments in both cases is to observe the effects of freeing up resources from the failed jobs (estimated), which creates more room for future jobs so that they can meet the deadline.

Figs. 4a and 4b depict the deadline met percentage by all the scheduling algorithms in light load and high load clusters, respectively, when the deadline is not used as a constraint. The deadline met percentage is lower in case of high load scenarios as the cluster is over-utilized, and there is a shortage of resources which causes many jobs to violate the deadline. For the light load case, the deadline met percentage is higher as there are more resources to accommodate the jobs whenever they arrive. In both cases, the MILP algorithm performs the best as it creates the least amount of resource fragments by tightly packing the executors. However, the FIFO algorithm distributively places executors that create many resource fragments in the cluster, which causes resource scarcity and more deadline violations. The FIFO-C algorithm performs slightly better than FIFO due to the consolidated approach. However, the AsQ algorithm chooses either local-only or cloud-only mode for placement. Thus, when the cluster is overloaded with many jobs at the same time, there is an increase in deadline violations due to resource scarcity. Both the proposed algorithms perform closely to the MILP-based algorithm where the GIO and FF algorithms are behind in the deadline met percentage by 5 and 8 percent, respectively. Figs. 4c and 4d exhibit the deadline met percentage by all the scheduling algorithms in both light load and high load clusters when the deadline constraint is used. It can be observed that, as many predicted to be failed jobs are not scheduled in the cluster, the overall deadline met percentage improved significantly for all the scheduling algorithms. The MILP-based algorithm performs the best in this case as well, followed by the GIO and FF algorithm, while the AsQ performs the worst.

6.3.3 Evaluation of Scheduling Delay

The scheduling delay is the time an algorithm takes to make scheduling decisions for all the executors of a job. We have measured it by measuring the time it takes from calling a particular scheduling algorithm up to the return from the scheduling algorithm with all the executor placement decisions for a job. The average scheduling delay for an algorithm is calculated by taking the average of the scheduling delays for all the jobs scheduled by that algorithm.

Table 4 shows the average scheduling delay by each algorithm in the small-scale setup. As the FIFO and FIFO-C algorithms follow a round-robin approach while placing the executors, the decision time is the shortest. Thus these algorithms have the lowest scheduling overheads. AsQ, FF and GIO are heuristic-based approaches, so these algorithms also showcase low scheduling delays which are

TABLE 4
Average Scheduling Delay (Small-Scale)

Algorithm	Average Scheduling Delay
FIFO	0.18 μs
FIFO-C	0.20 μs
AsQ	0.31 μs
FirstFit	0.28 μs
GIO	0.40 μs
ILP	1.85 s

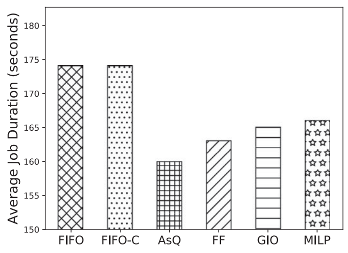


Fig. 5. Comparison of average job duration between the scheduling algorithms (the lower the better).

closed to the native schedulers (FIFO and FIFO-C). However, the MILP-based solution takes as long as 10-minutes in the worst-case even in the small-scale cluster setup and has an average scheduling delay of 1.85 s. Therefore, even though this algorithm can find the optimal cost-efficient executor placements, it is not scalable. Thus, it is only applicable to small-scale clusters.

6.3.4 Evaluation of Job Performance

We evaluate the job performance for the scheduling algorithms by measuring the average job duration for each scheduling algorithm during the whole scheduling process. As shown in Fig. 5, AsQ algorithm provides the lowest average job duration as it places the executors from the same job in the same regional boundary (either local or cloud). However, FIFO and FIFO-C algorithms always distribute the executors, so most of the placements are hybrid which causes the simulation environment to penalize these decisions to simulate the latency issues caused by federated scheduling. FF, GIO, and MILP algorithms have a slightly higher average job duration than the AsQ algorithm. However, due to the tight packing of executors, sometimes these algorithms also place executors within a single region, thus the performance overhead negligible if compared with the FIFO and FIFO-C.

6.3.5 Evaluation of Scalability

We have performed simulation on a large-scale setup where the cluster has 60 VMs (10 local VMs and 50 cloud VMs). We simulated the scheduling of 10,000 jobs in one whole day. As the MILP-based algorithm is not scalable, we only conducted the experiments with FIFO, FIFO-C, AsQ, FF, and GIO algorithms.

Fig. 6 shows the cost comparison results between the scheduling algorithms in both light load and high load scenarios for the large-scale experiment. It can be seen that both FF and GIO outperform the default FIFO and FIFO-C by a significant margin and reduce the cost up to 80 percent. The AsQ algorithm also tries to find cost-efficient placements in local-only or cloud-only settings. However, as our approaches leverage the hybrid setting to squeeze out spare resources in all the VMs across the cluster, the FF and GIO algorithms reduce the cost up to 15 percent as compared to the AsQ. Note that, for the small-scale setup, AsQ algorithm performed poorly as compared to the FIFO-C, this is due to the fact that there is limited resource availability in a small cluster, so local-only or cloud-only mode of placement is heavily punished in a higher load. However, for the large-scale setup, both the local and cloud portions of the cluster have sufficient resources, thus the AsQ outperforms the FIFO-C.

Table 5 presents the average scheduling delay for all the algorithms. It can be observed that even for a large-scale setup with many jobs, all the algorithms have a scheduling overhead at μs level thus making all of them extremely scalable.

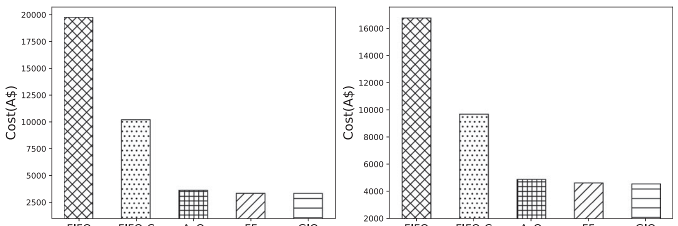


Fig. 6. Cost comparison in large-scale simulation (the lower the better). FIFO incurs a very high cost as round-robin placements of executors lead to many active VMs simultaneously.

TABLE 5
Average Scheduling Delay (Large-Scale)

Algorithm	Average Scheduling Delay
FIFO	0.20 μs
FIFO-C	0.24 μs
AsQ	0.0.47 μs
FirstFit	0.33 μs
GIO	0.83 μs

7 Performance Evaluation - Real Experiments

To show the applicability of the proposed algorithms in a real scenario and to validate the results from the simulation experiments, we have conducted real experiments on a Mesos cluster. This section presents the implemented system, experimental setup, benchmark applications and experimental results regarding different aspects of job scheduling.

7.1 System Implementation

We have developed a prototype system to evaluate the performance of the proposed job scheduling algorithms in a real hybrid cloud setup. Fig. 7 shows the architecture of the system. To implement any scheduling policy, the capability of placing an executor in any VM is needed. Apache Mesos [10] cluster manager provides this functionality by dynamic resource reservations, where any type of resource (e.g., CPU cores or memory) can be reserved in any VM so that only the desired executor can run with the reserved resources. Mesos provides HTTP APIs[11] to control dynamic resource reservation of a cluster. Therefore, a scheduler can dynamically place executors in any VM during the scheduling process. As we have a hybrid cluster comprising of both local and

11. [Online]. Available: http://mesos.apache.org/documentation/latest/operator-http-api/

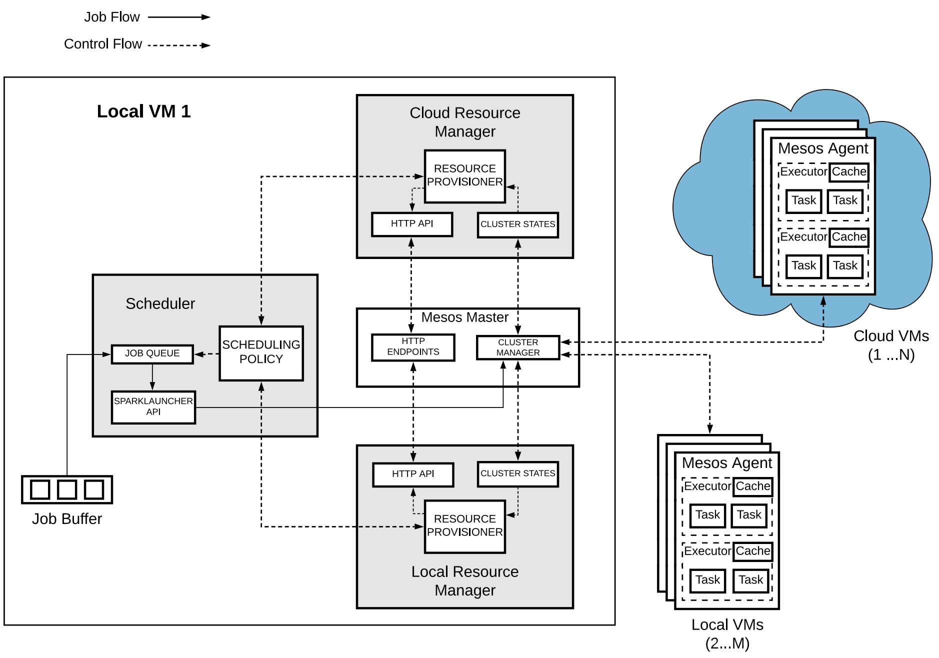


Fig. 7. System Architecture. The resource managers communicate with the Mesos cluster manager through the REST APIs. The Mesos master is deployed in the local part of the cluster (local VM-1).

cloud VMs, a Mesos cluster can be set up using these VMs, where each VM works as a Mesos agent. Here, each Spark executor runs inside a Mesos container in a Mesos agent.

As shown in the system architecture, we have implemented three additional modules (grey boxes) that work in collaboration with the Mesos master. All these modules are deployed into a local VM along with the Mesos master, which works as a central point of control for both the local and cloud VMs. Thus, from a job's perspective, there is a single cluster. However, the local and cloud VMs are deployed in different regions to exhibit a true hybrid cloud setup. There are two resource managers in the implemented system - Cloud and Local; for managing the VMs. Each resource manager can communicate with the Mesos master using the HTTP APIs for performing resource provisioning. Furthermore, resource managers can fetch cluster states (e.g., job and resource status) from the Mesos master. The scheduling module controls the resource manager modules to perform resource provisioning for any executor. Moreover, it can also instruct the resource managers to turn on/off any VM. When the resources are reserved for all the executors of a job, the scheduling module can directly launch a Spark job in the cluster by using the SparkLauncher API.[12] The developed modules are not extended from the default Spark's framework scheduler. Therefore, it is pluggable to the Mesos cluster manager and can be extended to work with any other Mesos-supported big data frameworks. We have implemented our proposed and baseline scheduling algorithms in the scheduler module. Java programming language was used to implement the proposed modules and scheduling algorithms. OpenStack Boto API[13] was used to automate the VM turn on/off mechanisms. The developed pluggable modules and the scheduling algorithms are open source[14] and can be used to implement and test new scheduling policies.

7.2 Experimental Setup

We have used Nectar Cloud,[15] a national cloud computing infrastructure for research in Australia to deploy a Mesos cluster. It is a cluster consisting of three different types of VM instances. The detailed VM configurations and quantity used from each type is the same as the small-scale setup shown in Table 2. However, the pricing model is different from the simulation pricing models. As shown in Table 3 (Pricing Model (Real)), the pricing of the real cloud instances is similar to the VM instance pricing in Amazon AWS (Sydney, Australia). Also, the price of a locally deployed instance is set to be half of the same instance price deployed in cloud. We set up a true hybrid cluster by deploying the VMs in two different regions: Melbourne and Tasmania. We have used the VMs deployed in Melbourne as the local VMs, and the VMs deployed in Tasmania as the cloud VMs. The end-to-end delay between VMs within the same regional boundary is approximately 10ms, whereas, the end-to-end delay between VMs from different regional boundaries is approximately 40ms. In addition, we have performed *iperf* testing to measure the bandwidth between the VMs. Within the same regional boundary, the bandwidth between the VMs is approximately 2Gbps, whereas, the bandwidth between two VMs from different regional boundaries is around 600Mbps. We store the input dataset in the local part of the cluster with an NFS server. Thus, the worker nodes (VMs) can mount the input data from the server and only access the portion of the data which they need to process.

12. [Online]. Available: https://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/launcher/package-summary.html
13. [Online]. Available: https://pypi.org/project/boto/
14. [Online]. Available: https://github.com/tawfiqul-islam/Hybrid-Cloud-Scheduler
15. [Online]. Available: https://nectar.org.au/research-cloud/

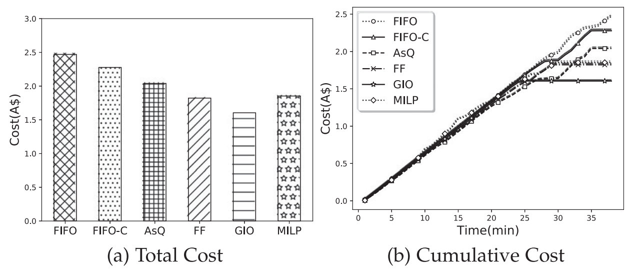


Fig. 8. Cost Comparison between different scheduling algorithms (the lower the better). (a) shows the total cost incurred over a scheduling period and (b) shows the cumulative cost incurred over time.

Our experimental cluster has 10 VMs with a total of 76 CPU (cores) and 304GB of memory. In each VM, we have installed Apache Mesos (version 1.4.0) and Apache Spark (version 2.3.1). One *m1.large* type VM instance was used as the Mesos master while all the remaining VMs were used as Mesos Agents. The Mesos master node is deployed locally in the Melbourne region. The implemented scheduler and resource manager modules were plugged into the Mesos master node.

7.3 Benchmarking Applications

We have used BigDataBench [32] benchmarking suite for the real experiments. We have taken three types of applications from this benchmark, which are: WordCount (compute-intensive), Sort (memory-intensive), and PageRank (network-intensive). We have randomly mixed all these three applications mentioned above to generate the workload. The job arrival times from the Facebook Hadoop workload trace[16] are extracted for an hour. Collecting job profiles to estimate the completion times is a well-known mechanism. In our experiments, each job is profiled in the real cluster for 10 times, and the average job completion time is taken to use as the estimated job completion time (T_C). These estimated job completion times are used in the problem model by the proposed scheduling algorithms to make scheduling decisions. However, to determine the schedulers' performance regarding cost optimization in the real experiment, we measure both the job completion time and the use of VM resources in real-time during the scheduling process for rigorous performance evaluation. The active time remaining for either a cloud or local VM (Δt_j^L for local and Δt_j^C for cloud) can be calculated by using the job completion time estimates (T_C) for the jobs which have one or more executors placed in a particular VM. The maximum estimated completion time is taken among these jobs and is subtracted from the current clock time to get an estimate on a VM's active remaining time.

7.4 Real Experiment Results

We have evaluated the proposed algorithm regarding cost efficiency, job deadline, and average job completion time. For these experiments, we have used *Pricing Model (Real)* as shown in Table 3 for the VM pricing, which is similar to the Amazon AWS pricing scheme for the cloud instances. The price of the same instance type deployed locally is considered to be half of the cloud instance price.

16. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository

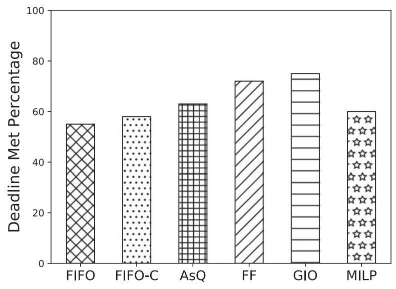


Fig. 9. Comparison of deadline met percentage between the scheduling algorithms (the higher the better).

7.4.1 Evaluation of Cost Efficiency

In this evaluation, we show the cost efficiency of different scheduling algorithms in the real experimental setup. Both the total cost and the cumulative cost is collected while running 100 jobs (mix of WordCount, Sort, and PageRank) for one hour. Fig. 8a exhibits the total cost incurred and Fig. 8b shows the cumulative cost incurred by different scheduling algorithms. It can be observed that the default FIFO and FIFO-C algorithms have the highest VM usage cost which increases linearly over time. However, the MILP and the proposed FF and GIO algorithms reduce the cost significantly as they utilize the pricing model of VMs and uses the cheaper VMs for executor placement. Although the AsQ algorithm utilizes the pricing model, it restricts executor placements to local or cloud-only. Thus, in a peak load where the cluster does not have sufficient resources, AsQ algorithm fails to utilize the spare resources in VMs by avoiding hybrid placement. The MILP algorithm finds the most cost-optimal placement of executors for each job, due to the scheduling overhead of MILP (computational complexity in some cases), the GIO algorithm performs slightly better and provides a lower cost. Furthermore, the MILP algorithm is only applicable to a small cluster as it is not scalable due to the exponential increase in decision-making for a large cluster.

7.4.2 Evaluation of Job Deadline

In this evaluation, we compare the deadline met percentage from different scheduling algorithms. Fig. 9 shows the comparison of deadline met percentage between the scheduling algorithms. As the FIFO and FIFO-c algorithms do not consider the EDF strategy, they have higher deadline violations as compared to the other algorithms. Although AsQ gives a better deadline met percentage than the default algorithms, it shows a lower deadline met percentage in a peak load, as jobs have to wait longer for a local-only or a cloud-only placement. The proposed FF and GIO algorithms show a higher deadline met percentage due to tight packing of executors and utilizing spare resources in the hybrid setting. Although hybrid placement increases job duration, the jobs do not have to wait longer as the algorithms schedule the jobs as soon as the combined resources (in both local and cloud VMs) are sufficient to place all the executors. MILP algorithm solves the executor placement in the most cost-efficient way. However, in many cases, it takes a lot of time to find a solution (high scheduling delay), which causes jobs to wait longer and violate deadlines.

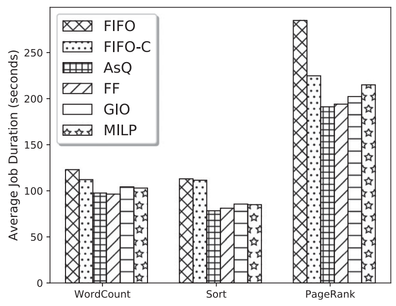


Fig. 10. Comparison of average job duration between the scheduling algorithms for different types of jobs (the lower the better).

7.4.3 *Effects on Job Performance*

Although it is possible to minimize the cost of using a hybrid cluster by packing more executors in fewer nodes, it causes some performance overhead for CPU/memory-bound jobs. However, when the executors from the same job are distributed over multiple regional VMs, the job completion time increases due to network latency and data transmission delays. As shown in Fig. 10, the default FIFO and FIFO-C algorithms always distribute the executors, so most of the placements are hybrid which causes a high average job duration. Network-bound jobs (PageRank) suffer the most, where a lot of network communications take place. The AsQ algorithm provides the lowest average job duration for different types of jobs, as the data transmissions between executors only occur within the same regional boundary. Although the FF, GIO, and MILP algorithms utilize hybrid placement to reduce cost, they have a slightly higher average job duration than the AsQ algorithm. However, due to the tight packing of executors, sometimes these algorithms also place executors in a single region, thus the performance overhead is not as extreme as the FIFO and FIFO-C. Nevertheless, this slight performance degradation is negligible as compared to the cost-saving in the hybrid cluster.

8 Conclusion and Future Work

In this paper, we have formulated the SLA-based Spark job scheduling problem in a hybrid cloud as an optimization problem. We have proposed two greedy heuristics-based algorithms to solve the scheduling problem. Besides, we have implemented the proposed algorithms on top of Apache Mesos to show the applicability in real environments. We have compared the proposed approaches in both simulated and real experiments to show the superiority of them over the baseline approaches. The results show that our proposed algorithms can significantly reduce VM usage costs in a hybrid cloud. Although there are performance overheads due to data transmission delays caused by hybrid placements, it is negligible as compared to the cost-saving benefits. Moreover, the proposed approaches are highly scalable and have low scheduling overhead, which is similar to the native Spark schedulers.

This paper focuses more on the user's perspective, and when a user submits a Spark job, they do not provide network or disk as resource constraints. Thus, we work on a higher level where we consider resource capacity/demand constraints which are required at the executor creation stage. However, we try to capture the network transmission issues by considering the job duration increase in the problem model. In the future, we plan to investigate more on the performance impacts caused by hybrid placements. In addition, we plan to investigate the trade-offs between cost-efficiency and job performance. A more sophisticated model needs to be devised, which can consider both objectives together to generate efficient job schedules. In addition, we would like to explore deeper into the effects of VM turn-on/off mechanisms on job performance and cost-efficiency. We also plan to incorporate the proposed scheduling algorithms in modern container orchestration systems such as Kubernetes. As Fog computing and Edge computing are becoming increasingly popular, we plan to extend the scheduling algorithms to work with a multi-tier Fog-Edge-Cloud deployed cluster.

Acknowledgments

This work was supported in part by a project funding from the Australian Research Council and in part by the National Natural Science Foundation of China under Grant 61801325.

References

[1] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. 4th ACM Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.
[2] M. Zaharia *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.
[3] A. Toshniwal *et al.*, "Storm at twitter," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156.
[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. 26th IEEE Symp. Mass Storage Syst. Technol.)*, 2010, pp. 1–10.
[5] L. George, *HBase: The definitive guide: random access to your planet-size data*. Newton, MA, USA: O'Reilly Media, 2011.
[6] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[7] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 15–28.
[8] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: Scheduling of long running applications in shared production clusters," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–13.
[9] C. Curino *et al.*, "Hydra: A federated resource manager for data-center scale analytics," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, Feb. 2019, pp. 177–191.
[10] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.
[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness : Fair allocation of multiple resource types maps reduces," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[12] E. Hwang and K. H. Kim, "Minimizing cost of virtual machines for deadline-constrained MapReduce applications in the cloud," in *Proc. 2012 ACM/IEEE 13th Int. Conf. Grid Comput.*, Sep. 2012, pp. 130–138.
[13] L. Mashayekhy, M. M. Nejad, D. Grosu, Q. Zhang, and W. Shi, "Energy-aware scheduling of MapReduce jobs for big data applications," *IEEE Trans. Parallel. Distrib. Syst.*, vol. 26, no. 10, pp. 2720–2733, Oct. 2015.
[14] D. Nayak, V. S. Martha, D. Threm, S. Ramaswamy, S. Prince, and G. Fatimberger, "Adaptive scheduling in the cloud - SLA for Hadoop job scheduling," in *Proc. Sci. Inf. Conf.*, 2015, pp. 832–837.
[15] D. Cheng, J. Rao, C. Jiang, and X. Zhou, "Resource and deadline-aware job scheduling in dynamic Hadoop clusters," in *Proc. IEEE 29th Int. Parallel Distrib. Process. Symp.*, 2015, pp. 956–965.
[16] X. Zeng, S. Garg, Z. Wen, P. Strazdins, L. Wang, and R. Ranjan, "SLA-Aware scheduling of map-reduce applications on public clouds," in *Proc. 18th IEEE Int. Conf. High Perform. Comput. Commun.*, 2016, pp. 655–662.

[17] N. Zacheilas and V. Kalogeraki, “ChEsS: Cost-effective scheduling across multiple heterogeneous mapreduce clusters,” in *Proc. IEEE Int. Confe. Autonomic Computi.*, 2016, pp. 65–74.
[18] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, “Sparrow : Distributed , low latency scheduling,” in *Proc. ACM Symp. Operating Syst. Princ.*, 2013, pp. 69–84.
[19] D. Wu, S. Sakr, L. Zhu, and H. Wu, “Towards big data analytics across multiple clusters,” in *Proc. 17th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2017, pp. 218–227.
[20] S. Maroulis, N. Zacheilas, and V. Kalogeraki, “A framework for efficient energy scheduling of spark workloads,” in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 2614–2615,.
[21] H. Li, H. Wang, S. Fang, Y. Zou, and W. Tian, “An energy-aware scheduling algorithm for big data applications in spark,” *Cluster Comput.*, vol. 23, no. 2, pp. 593–609, Jun. 2019.
[22] Z. Liu, H. Zhang, and L. Wang, “Hierarchical spark: A multi-cluster big data computing framework,” in *Proc. IEEE 10th Int. Conf. Cloud Comput*, Jun. 2017, pp. 90–97.
[23] S. Sidhanta, W. Golab, and S. Mukhopadhyay, “Deadline-aware cost optimization for spark,” *IEEE Trans. Big Data*, vol. 7, no. 1, pp. 115–127, Mar. 2021.
[24] H. Zhang, J. Shi, B. Deng, G. Jia, G. Han, and L. Shu, “MCTE: Minimizes task completion time and execution cost to optimize scheduling performance for smart grid cloud,” *IEEE Access*, vol. 7, pp. 134793–134803, 2019.
[25] W.-J. Wang, Y.-S. Chang, W.-T. Lo, and Y.-K. Lee, “Adaptive scheduling for parallel tasks with QoS satisfaction for hybrid cloud environments,” *J. Supercomputing*, vol. 66, no. 2, pp. 783–811, Feb. 2013.
[26] V. Peláez, A. Campos, D. F. García, and J. Entrialgo, “Online scheduling of deadline-constrained bag-of-task workloads on hybrid clouds,” *Concurrency Comput: Prac. Exp.*, vol. 30, no. 19, Oct. 2018, Art. no. e4639,.
[27] A. M. Maia, Y. Ghamri-Doudane , D. Vieira, and M. F. de Castro, “Optimized placement of scalable IoT services in edge computing,” in *Proc. IFIP/IEEE Symp. Integr. Netw. Serv. Manag.*, 2019, pp. 189–197.
[28] S. BurerandA. N. Letchford,“Non-convex mixed-integer nonlinear programming: A survey,” *Surveys Operations Res. Manage. Sci.*, vol. 17, no. 2, pp. 97–106, 2012.
[29] X. Wang, J. Wang, X. Wang, and X. Chen, “Energy and delay tradeoff for application offloading in mobile cloud computing,” *IEEE Syst. J.*, vol. 11, no. 2, pp. 858–867, Jun. 2017.
[30] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, and D. Vigo, “Bin packing approximation algorithms: Survey and classification,” in *Handbook of Combinatorial Optimization*. New York, NY, USA: Springer, 2013, pp. 455–531.
[31] L. Caccetta, “Branch and cut methods for mixed integer linear programming problems,” in *Applied Optimization*. Boston, MA, USA: Springer, 2000, pp. 21–44.
[32] L. Wang *et al.*, “Bigdatabench: A big data benchmark suite from internet services,” in *Proc. 20th IEEE Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 488–499.

Muhammed Tawfiqul Islam (Member, IEEE) received the BS and MS degrees in computer science from the University of Dhaka, in 2010 and 2012, respectively. He is currently working toward the PhD degree with Cloud Computing and Distributed Systems Laboratory, University of Melbourne, Australia. His research interests include big data, cloud computing, stream computing, and AI.

Huaming Wu (Member, IEEE) received the BE and MS degrees in electrical engineering from the Harbin Institute of Technology, China, in 2009 and 2011, respectively, and the PhD degree with the highest honor in computer science from Freie Universität Berlin, Germany, in 2015. He is currently an associate professor with the Center for Applied Mathematics, Tianjin University. His research interests include mobile cloud computing, edge computing, fog computing, Internet of Things, and deep learning.

Shanika Karunasekera (Member, IEEE) received the BSc degree in electronics and telecommunications engineering from the University of Moratuwa, Moratuwa, Sri Lanka, in 1990 and the PhD degree in electrical engineering from the University of Cambridge, Cambridge, U.K., in 1995. She is currently a professor with the School of Computing and Information Systems, University of Melbourne, Australia. Her current research interests include distributed computing, mobile computing, and social media analytics.

Rajkumar Buyya (Fellow, IEEE) is currently a Redmond Barry Distinguished professor and the director with the Cloud Computing and Distributed Systems Laboratory, University of Melbourne, Australia. He has authored more than 725 publications and seven text books including, *Mastering Cloud Computing* published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese, and international markets, respectively. Software technologies for Grid and Cloud computing developed under his leadership have gained rapid acceptance and are in use at several academic institutions and commercial enterprises in 50 countries around the world.