The Journal of Systems and Software 162 (2020) 110515

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Cost-efficient dynamic scheduling of big data applications in apache spark on cloud

Muhammed Tawfiqul Islam[a,*], Satish Narayana Srirama[a,b], Shanika Karunasekera[a], Rajkumar Buyya[a]

[a] Cloud Computing and Distributed Systems (CLOUDS) Laboratory School of Computing and Information Systems The University of Melbourne, Australia
[b] Mobile & Cloud Lab, Institute of Computer Science University of Tartu, Estonia

ARTICLE INFO

Article history:
Received 1 October 2019
Revised 17 November 2019
Accepted 23 December 2019
Available online 27 December 2019

Keywords:
Cloud
Apache spark
Scheduling
Cost-efficiency

ABSTRACT

Job scheduling is one of the most crucial components in managing resources, and efficient execution of big data applications. Specifically, scheduling jobs in a cloud-deployed cluster are challenging as the cloud offers different types of Virtual Machines (VMs) and jobs can be heterogeneous. The default big data processing framework schedulers fail to reduce the cost of VM usages in the cloud environment while satisfying the performance constraints of each job. The existing works in cluster scheduling mainly focus on improving job performance and do not leverage from VM types on the cloud to reduce cost. In this paper, we propose efficient scheduling algorithms that reduce the cost of resource usage in a cloud-deployed Apache Spark cluster. In addition, the proposed algorithms can also prioritise jobs based on their given deadlines. Besides, the proposed scheduling algorithms are online and adaptive to cluster changes. We have also implemented the proposed algorithms on top of Apache Mesos. Furthermore, we have performed extensive experiments on real datasets and compared to the existing schedulers to showcase the superiority of our proposed algorithms. The results indicate that our algorithms can reduce resource usage cost up to 34% under different workloads and improve job performance.

1. Introduction

Big Data processing has become crucial due to massive analytics demands in all the major business and scientific domains such as banking, fraud detection, healthcare, demand forecasting, and scientific explorations. Data processing frameworks such as Hadoop[1], Storm[2], and Spark (Zaharia et al., 2016) are the most common choice when it comes to big data processing. Large organisations generally run private compute clusters with one or more big data processing frameworks on top of it. As public cloud services can provide infrastructure, platform, and software for storing and computing of data, it is also becoming popular to deploy the big data processing clusters on public clouds. However, scheduling these big data jobs can be difficult in a cloud-deployed cluster since the jobs can be of different types such as CPU-intensive, memory-intensive, and network-intensive. Furthermore, jobs can also vary based on their resource demands to maintain a stable performance. Moreover, various types of Virtual Machines (VM) instances available on the cloud make it difficult to generate cost-effective scheduling plans. Therefore, in this paper, we propose efficient job scheduling algorithms that reduce the cost of using a cloud-deployed Apache Spark cluster while enhancing job performance.

To demonstrate the effectiveness of our scheduling algorithms, we have chosen Apache Spark as our target framework because it is a versatile, scalable and efficient big data processing framework and is rapidly replacing traditional Hadoop-based platforms used in the industry. Spark utilises in-memory caching to speed up the processing of applications. The resource requirements of a Spark job can be specified by using the number of required executors for that particular job, where each executor can be thought of as a process, having a fixed chunk of resources (e.g., CPU, memory and disk). However, different jobs can have varying executor size requirements depending on the type of workloads they are processing. Therefore, jobs exhibit different characteristics regarding resource dependability.

The default scheduling mechanism for Spark job scheduling is *First in First Out (FIFO)*[3], where each job is scheduled one after

* Corresponding author.
E-mail addresses: tawfiq.shobhon@gmail.com, muhammedi@student.unimelb.edu.au (M.T. Islam).

[1] http://hadoop.apache.org/.
[2] http://storm.apache.org/.
[3] https://spark.apache.org/docs/latest/job-scheduling.html#scheduling-across-applications.

https://doi.org/10.1016/j.jss.2019.110515
0164-1212/

another. If no resource limit is set, one job might consume all the resources in the cluster. On the other hand, if the user sets a limit on the required resources of a job, the remaining resources can be used to schedule the next job in the queue. In addition to the FIFO scheduler, a *Fair Scheduler* is also available to prevent resource contention among jobs. By default, both of these schedulers place the executors of a job in a round-robin fashion in all the VMs/worker nodes for load-balancing and performance improvement. However, when a cloud-deployed cluster is not fully loaded with jobs, round-robin executor placement leads to resource wastage in all the VM. Although Spark also has an option to consolidate the executor placements, the cluster manager does not consider the resource capacity and price of different cloud VM instance types, and thus fails to make cost-efficient placement decisions. Most of the existing scheduling techniques focus on Hadoop-based platforms (Kc and Anyanwu, 2010; Zaharia et al., 2009; Chen et al., 2010; Tian et al., 2009). Nevertheless, these mechanisms cannot be directly applied to Spark job scheduling as the architectural paradigm is different from in-memory computing frameworks. A very few works have been done to tackle the scheduling problem of in-memory computing-based frameworks like Apache Spark (Delimitrou and Kozyrakis, 2014; Sidhanta et al., 2016; Jyothi et al., 2016; Dimopoulos et al., 2017). However, most of these works assume the cluster setup to be homogeneous (there is only one type of VM instance for all the worker nodes) thus fail to make the scheduling technique cost-efficient from a cloud perspective.

As a motivating example, consider a cluster having 2 homogeneous VMs each having 8 CPU cores capacity. If a Spark job has 2 executors requirement with 2 cores for each, the total CPU cores requirement is 4. However, most of the existing strategies will use both the VMs to place these 2 executors which will lead to resource wastage and a higher VM usage cost. On the contrary, if a scheduler can consider the VM pricing model and different VM instance types in the cluster, executors from the jobs could be tightly packed in fewer cost-effective VMs. Thus, the instances with more resource capacity and higher price will be used only if there is a high load on the cluster. Therefore, in this paper, we formulate the scheduling problem of Spark jobs in a cloud-deployed cluster as a variant of the bin-packing problem. Here, our primary target is to reduce the cost of VM usage while maximising resource utilisation and improving job performance.

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

- We propose two job scheduling algorithms. The first algorithm is a greedy algorithm adapted from the Best-Fit-Decreasing (BFD) heuristic, and the second algorithm is based on Integer Linear Programming (ILP). Both of these algorithms can improve cost-efficiency of a cloud deployed Apache Spark cluster. Besides, our proposed algorithms also prioritise jobs based on their deadlines and enhance job performance for network-bound jobs.
- We develop a scheduling framework by utilising Apache Mesos (Hindman et al., 2011) cluster manager and this framework can be used to implement scheduling policies for any Mesos supported data processing frameworks in addition to Spark.
- We implement the proposed algorithms on top of the developed scheduling framework.
- We perform extensive experiments with real applications and workload traces under different scenarios to demonstrate the superiority of our proposed algorithms over the existing techniques.

The rest of the paper is organised as follows. In Section 2, we discuss the background of Apache Spark and Apache Mesos. In Section 3, we describe the existing works related to this paper. In Section 4, we show the motivating examples and formulate the scheduling problem. In Section 5, we demonstrate the implemented prototype system. In Section 6, we evaluate the performance of our proposed algorithms, show the sensitivity analysis of various system parameters and discuss the feasibility of our proposed algorithms. Section 7 concludes the paper and highlights future work.

2. Background

We use Apache Spark as the target big data processing framework and Apache Mesos as the cluster manager where we implement our scheduling policies. In this section, we briefly introduce the basic concepts, system architecture, resource provisioning and scheduling mechanisms in these two frameworks.

2.1. Apache spark

Apache Spark is one of the most prominent in-memory big data processing frameworks. It is a multi-purpose open-source platform with high scalability. Spark supports applications to be built with various programming languages like Java, Scala, R, Python etc. Besides, extensive and interactive analysis can be done using the available high-level APIs. Furthermore, a variety of input data sources like HDFS (Shvachko et al., 2010), HBase (George, 2011), Cassandra (Lakshman and Malik, 2010) etc. are supported by Spark. It outperforms traditional Hadoop-MapReduce based platform by conducting most of the computations in memory. In addition, results from the intermediate stages are cached in memory for faster data re-processing. Spark uses *Resilient Distributed Dataset (RDD)* (Zaharia et al., 2012) for data abstraction which is fault tolerant by nature. In contrast to HDFS, Spark does not implement replication. Spark keeps track of how a specific piece of data was calculated, so it can recalculate any lost RDD partitions if a node fails or is shutdown by a scheduler. A Spark cluster follows a Master-Worker model, where there should be at least one *Master* node and one or more *Worker* nodes. However, multiple master nodes can be used by leveraging ZooKeeper (Hunt et al., 2010). From a cloud perspective, each master/worker node can be deployed in a cloud *VM*. Spark has its default standalone cluster manager which is sufficient to deploy a production-grade cluster. Moreover, it also supports popular cluster managers like Hadoop Yarn (Vavilapalli et al., 2013), Apache Mesos (Hindman et al., 2011) etc.

When a Spark job/application is launched in a cluster, the *Driver* program of that job creates one or more executors in the worker nodes. *Executor* is a process of an application that holds a fixed chunk of resources (CPU cores, memory, and disk) and all the executors from the same job have identical resource requirements. Tasks are run in parallel in multiple threads inside each executor which lives during the entire duration of that job. As all the jobs have an independent set of executors, jobs are isolated, and each job's driver program can create its own set of executors and schedule tasks in them.

Resource allocation in a Spark cluster can be done in three ways: (1) Default: the user does not set any limits on the required resources for a job, and it uses all the resources of the entire cluster. Therefore, only one job can run in the cluster at a time and even if that job only requires a small chunk of resources, all the resources are allocated to it; (2) Static: if a user sets a limit on the required resources for a job, only that amount of resources will be allocated for that job, and any remaining resources can be assigned to any future job. Therefore, in this mode, it is possible to run multiple applications in the cluster and (3) Dynamic: resources are allocated similarly as the static allocation mechanism, but if any resource (CPU core only) is not utilised, it could be released to the cluster so that any other application can use it. Besides, this resource can be taken back from the cluster in future if needed by the original job.

By default, Spark supports FIFO scheduling across jobs. Therefore, jobs wait in a FIFO queue and run one after another. A new job is scheduled whenever any resources are available to create any executor for the next job. Besides, Spark also has a FAIR scheduler, which was modelled after the Hadoop Fair Scheduler[4]. Here, jobs can be grouped into pools, and different scheduling options can be set for each pool. For example, weight determines the priority of a job pool. By default, each pool has a weight 1, but if any pool is assigned 2 as the weight, it will get twice the resources than other pools. Within each job pool, jobs are scheduled in a FIFO fashion. Each pool also has a minimum share (minShare) of resources in the cluster, and a cluster manager only assigns more resources to a highly weighted pool once all the pools have met their minimum share of resources. By default, Spark spreads the executors from the same job into multiple workers for load balancing. In addition, the standalone cluster manager can also consolidate executors into fewer worker nodes (by greedily using the current worker node to place as many executors as possible). However, Spark assumes that all the worker nodes are homogeneous (same resource capacity), and it also does not consider the price of using a worker node (if it is deployed on cloud VM).

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. It resides between the application and the OS layer and makes it easier to deploy and manage large-scale clusters. In Mesos, jobs/applications are called frameworks and multiple applications from different data processing frameworks like Spark, Strom, and Hadoop can run in parallel in the cluster. Therefore, Mesos can be used to share a pool of heterogeneous nodes among multiple frameworks efficiently. Mesos utilises modern kernel features by using *cgroups* in Linux and *zones* in Solaris to provide isolation of CPU, memory, file system etc.

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. If a framework's scheduler accepts a resource offer from Mesos, the resources specified by that offer can be used to launch any computing tasks. Mesos also provides flexible Scheduler[5] HTTP APIs which can be used to write custom user-defined scheduling policies on top of any big data processing platform. Besides, it provides Operator[6] HTTP APIs to control the resource provisioning and scheduling of the whole cluster. Mesos supports dynamic resource reservation; 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 scheduling framework on top of Mesos to implement our proposed scheduling algorithms.

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

[5] http://mesos.apache.org/documentation/latest/scheduler-http-api/.

[6] http://mesos.apache.org/documentation/latest/operator-http-api/.

3. Related work

Most of the data processing frameworks like Hadoop, Spark schedule jobs in a FIFO manner and distributes the tasks/executors from each job in a distributed round-robin fashion. To avoid resource contention FAIR scheduler was introduced for fair distribution of cluster resources among the jobs. In Mesos, scheduling is done by the Dominant Resource Fairness (DRF) (Ghodsi et al., 2011) scheduling algorithm, which identifies the dominant resource type (CPU/memory) of each job. Then it offers resources to each job in such a way that overall use of cluster resources is well-balanced.

There has been a significant amount of research in the area of cluster scheduling. However, most of these schedulers focused Hadoop-MapReduce based clusters. Kc and Anyanwu (2010) addressed the problems of Hadoop FIFO scheduler by introducing a deadline constraint scheduler that prioritises map/reduce tasks from each job based on their deadline. LATE (Zaharia et al., 2009) is a delay scheduler that targets to improve job throughput and response times by considering data locality into the scheduler in a multi-user MapReduce cluster. However, it treats the cluster setup to be homogeneous thus performs poorly in heterogeneous environments. SAMR (Chen et al., 2010) proposed a self-adaptive scheduling algorithm that classifies the performance of jobs from the historical data. It also identifies slow nodes dynamically and creates backup tasks so that MapReduce jobs will have a better performance in a heterogeneous environment. Tian et al. (2009) considered job heterogeneity and proposed a triple-queue scheduler to keep the CPU and I/O bound applications isolated to improve the overall cluster performance. However, all of these works are focused on Hadoop-MapReduce performance modelling and scheduling and cannot be applied to an in-memory data processing framework like Spark.

As a platform like Spark has many configuration parameters, it is hard to set the appropriate resource requirement for a job. Wang et al. (2016) tried to fine-tune Spark configuration parameters to improve the overall system performance. Gounaris et al. (2017) investigated the problem of resource wastage that happens when a Spark application consumes all the nodes in a cluster. Gibilisco et al. (2016) built multiple polynomial regression models on the application profile data and selects the best model to predict application execution time with unknown input data or cluster configuration. Wang and Khan (2015) tried to model application performance in DAG-based in-memory analytics platforms. Here, the execution times from multiple stages of a job are collected and then used to predict the execution time. Islam et al. (2017) focused on fine-grained resource allocation for Spark jobs with deadline guarantee. However, these works can only be applied to predict job-specific resource demands under homogeneous cluster environments.

There are a very few cluster schedulers (Soualhia et al., 2017) that support Spark jobs focusing on performance improvement and cost saving. Quasar (Delimitrou and Kozyrakis, 2014) is a cluster management system that minimises resource utilisation of a cluster while meeting user-provided application performance goals. It uses efficient classification techniques to find the impacts of resources on an application's performance. Then it uses this information for resource allocation and scheduling. It also dynamically adjusts resources for each application by monitoring resource usage. Morpheus (Jyothi et al., 2016) estimates job performance from historical data using performance graphs. Then it performs a packed placement of containers where it places a job that results in the minimal cluster resource usage cost. Moreover, Morpheus dynamically re-provisions failed jobs to improve overall cluster performance. Justice (Dimopoulos et al., 2017) is a fair share resource allocator that uses deadline information of each job and historical job execution logs in an admission control. It automatically adapts

Table 1
Related Work.

Features	Related Work					Our Work
	DRF (Ghodsi et al., 2011)	Quasar (Delimitrou and Kozyrakis, 2014)	Morpheus (Jyothi et al., 2016)	Justice (Dimopoulos et al., 2017)	OptEx (Sidhanta et al., 2016)	
Frameworks	✗	✗	✗	✗	✗	√
VM types	✗	✗	✗	✗	✗	√
Job types	√	√	√	√	√	√
Cost-efficient	✗	✗	✗	√	√	√
Performance	√	√	√	√	√	√
Self-adaptive	✗	√	√	√	✗	√
Deadline	✗	√	√	√	√	√

to workload changes and provides sufficient resources to each job so that it meets deadlines just in time. OptEx (Sidhanta et al., 2016) models the performance of Spark jobs from application profiles. Then the performance model is used to schedule a cost-efficient cluster by deploying each job as a service in the minimal set of nodes required to satisfy its deadline.

The problems with most of the cluster schedulers are that they do not consider executor-level job placement. All of them only select the total number of resources or nodes needed for each job while making any scheduling decision. However, our scheduler takes advantage of VM heterogeneity (different types of VM instances) and uses smaller VMs for executor placement to minimise the overall resource usage cost of the whole cluster. Besides, most of the cluster schedulers use the round-robin placement of executors in the VMs while we consolidate the executors to use less number of VMs. Therefore, it minimises inter-node communications for network-bound jobs thus improves the performance. A comparison of our approach with the existing works is illustrated in Table 1. It can be observed that our proposed solution considers multiple VM types in the scheduling algorithm. Moreover, we also provide a scheduling framework to incorporate new scheduling policies.

Currently, commercial cloud service providers such as AWS and Windows Azure provide clusters and big data analytics services on the Cloud. For example, Apache Spark on Amazon EMR[7] and Azure HDInsight[8]. Besides job scheduling, there are many other ways to reduce costs in a commercial cloud computing platform. For example, EC2 spot instances and reserved instances have many features[9]. Commercial cloud service providers optimise instance usage costs from their side by turning off idle instances. Our proposed approach complements these solutions by tight packing of executors in fewer instances so that those instances can be turned off. Hence, even if all the nodes are Spot instances, our approach is still cost-efficient as we use minimal number of instances as compared to the default Spark scheduler. While the commercial cloud service providers work on the VM instance level, our approach works on the executor level scheduling which is more fine-grained. Therefore, for the most cost-benefit, job scheduling from user-side also plays a vital role and while used in conjunction with commercial cloud providers' instance features, significant performance improvement and cost reduction can be achieved. Lastly, our approach can also be used for a local cluster which is deployed with on-premise physical resources.

4. Cost-efficient job scheduling

In this section, we explain the motivations of this work, the problem formulation, the proposed job scheduler and the executor placement algorithms and the complexity of the proposed algorithms.

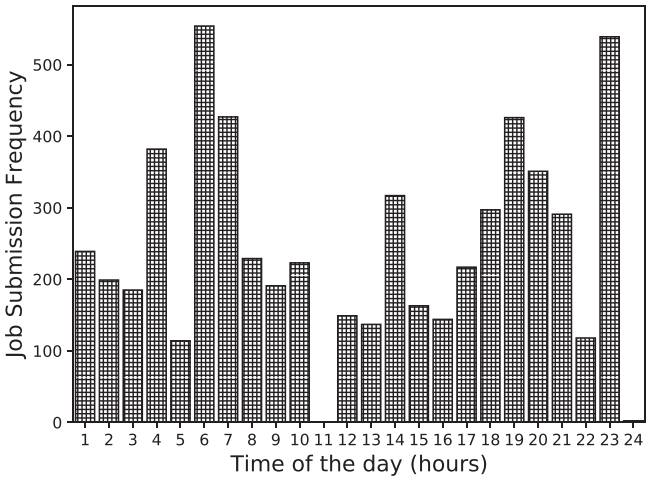


Fig. 1. Job submission frequencies in a single day (Facebook Hadoop Workload Trace-2009).

4.1. Motivation

The utilisation of resources in a big data cluster varies at different times of the day. For example, Fig. 1 depicts the job submission frequencies at different hours in a particular day from a Facebook Hadoop workload trace[10]. There are several hours in a day when the job submission rate is lower than usual. Therefore, if a big data processing cluster is deployed in the public cloud, it would be costly to keep all the VMs turned on as the cluster might not be fully utilised. However, the bill of using a VM is charged as pay-per-use basis and most of the cloud providers per-second billing period[11]. Hence, if a VM is not used to schedule any jobs, it can be turned off to reduce the monetary cost of the cluster. The turned off VMs can be turned on again in future depending on the overall resource demands in the cluster.

Cloud service providers offer different types of VMs which have different pricing model. In general a small VM with lower resource capacity is cheaper than a large VM with high resource capacity[12]. Therefore, if a cluster is deployed with different types of VM instances, smaller VMs can be used in the low-load period of the cluster to save cost whereas the bigger VMs can be utilised only in the high-load period.

[7] https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark.html.
[8] https://azure.microsoft.com/en-au/services/hdinsight/.
[9] https://aws.amazon.com/emr/features/.
[10] https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.
[11] https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes.
[12] https://aws.amazon.com/ec2/pricing/on-demand/.

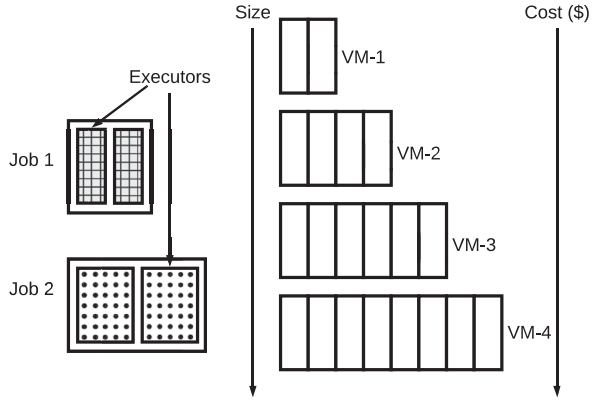


Fig. 2. An example cluster with different types of Jobs and VMs.

Most of the cluster schedulers place the executors from each job in a distributed (round-robin) fashion in the VMs which has the following problems:

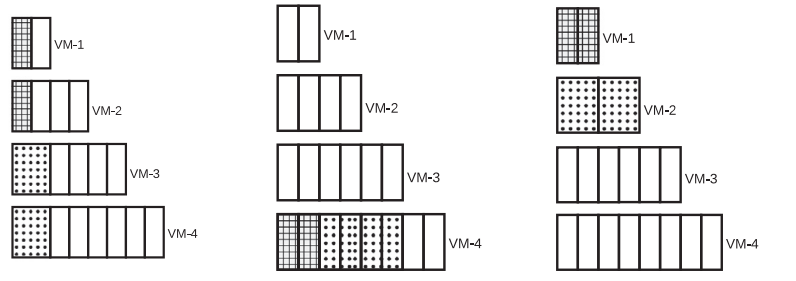
- VMs are under-utilised, and resources are wasted in all the VMs. This problem leads to a higher cost of using the whole cluster as most of the VMs are turned on at all times.
- In the cloud, different types of VM instances are available to use as the worker nodes and using only a single type of VM to compose a cluster might not be cost-effective. For example, a cluster has only one type of VM (16 CPU core, 64GB memory). If at a light-load hour only a single job is submitted (2 CPU core, 2GB memory), even using one VM would be costly. Using only small VM instances to compose a cluster would also fail as executors from different Spark jobs might have different size (resource requirement), so executors with high resource requirement will not fit in smaller VMs.
- For network-bound jobs, performance is reduced due to increased network transfers among the executors due to the distributed placement of executors in different VMs.

The consolidated executor placement option of Spark can not save cost as it does not consider the prices of different workers (VMs), and may choose the biggest VM to consolidate executors. Fig. 2 shows an example scheduling scenario where two jobs (with different resource demand) are submitted to a cluster composed of four VMs (with different resource capacity). For simplicity, let us assume that the executors from all the jobs require only one type of resource (e.g., CPU cores). The total number of slots in each VM represents its resource capacity. Similarly, the width of each executor of a job represents its resource demand. Therefore, in our example, each executor from job-1 requires 1 CPU core, and each executor from job-2 requires 2 CPU cores. VM-1, VM-2, VM-3, and VM-4 have a resource capacity of 2, 4, 6 and 8 CPU cores, respectively. In addition, the cost of using each VM is equivalent to its size, hence VM-1 is the cheapest VM whereas VM-4 is the costliest VM. Fig. 3a-3c depicts some of the possible executor placement strategies. Fig. 3a shows a distributed executor placement strategy (round-robin) which is used by most of the scheduling policies. In this placement, all the VMs are used but under-utilised. Therefore, this placement will lead to the highest VM usage cost. An alternative strategy which can be used in Spark to consolidate executors can be seen in Fig. 3b. However, as the cluster manager is unaware of the VM instance pricing or resource capacity, if it chooses to place job-1 in VM-4, job-2 will also be placed in VM-4 to consolidate executors from both jobs in fewer VMs. Even though Spark's executor consolidation strategy provides a better VM usage cost than the round-robin strategy, it can be further improved as shown in Fig. 3c. Here, when job-1 first arrives it is placed in the cheapest VM (VM-1) where the executors of the current job fits properly. Then, job-2 is placed into the 2nd cheapest VM (VM-2), as VM-1 is already used. This strategy provides the cheapest VM cost usage even though executors are consolidated in more than one VM.

4.2. Problem formulation

In an Apache Spark cluster, the resource requirements of the executors from the job are same. In addition, each worker node (VM) has a set of available resources (e.g., CPU cores, memory) which can be used to place executors from any job if the resource requirements are met. Therefore, for each submitted job in the cluster, the main problem is to find the placement of all its executors to one or more available VMs. Besides, resource capacity in each VM must not be exceeded while placing one or more executors in that VM during the scheduling process. As the compact assignment of executors leads to cost reduction due to fewer VM usages, we model the scheduling problem as a variant of the bin-packing problem. Table 2 shows the notations we use to formulate the problem.

We consider the resource requirement of an executor in two dimensions – CPU cores and memory. Therefore, each executor of a job can be treated as an item with multi-dimensional volumes that needs to be placed to a particular VM (bin) in the scheduling process. 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 ($i \in \xi$). There are K types of VM available each with a two-dimensional resource capacity (CPU, Mem) and incurs a fixed cost P_k, if used. The problem is to select VMs and place all the executors into these VMs such that the total cost is minimised and the resource constraints are met.



(a) Distributed executor placement (b) Packed executor placement in 1 VM (c) Packed executor placement in 2 VMs

Fig. 3. Different Executor Placement Strategies.

Table 2
Definition of Symbols.

Symbol	Definition
job	The current job to be scheduled
E	Total executors required for job
ξ	The index set of all the executors of job, $\xi = \{1, 2, 3, \ldots, E\}$
Ψ	The index set of all the VM types, $\Psi = 1, 2, \ldots, K$
m_k	An upper-bound on the number of type k VMs
δ_k	The index set for each type k VM; $\delta_k = \{1, 2, \ldots, m_k\}, k \in \Psi$
P_k	Price of using a VM of type k
ω_{jk}^{cpu}	Available CPU in the jth VM of type k, $j \in \delta_k$, $k \in \Psi$
ω_{jk}^{mem}	Available Memory in the jth VM of type k, $j \in \delta_k$, $k \in \Psi$
τ^{cpu}	CPU demand of any executor of job
τ^{mem}	Memory demand of any executor of job
RA_{jk}	Resource Availability metric of the jth VM of type k
RD_{job}	Resource Demand metric for job

The optimisation problem is:

$$\text{Minimise: } Cost = \sum_{k\in\Psi} P_k \left(\sum_{j\in\delta_k} y_{jk} \right) \tag{1}$$

$$\sum_{k\in\Psi} \sum_{j\in\delta_k} x_{ijk} = 1 \qquad \forall i \in \xi \tag{2}$$

$$\sum_{i\in\xi} (x_{ijk} * \tau^{cpu}) \leq \omega_{jk}^{cpu} * y_{jk} \qquad \forall k \in \Psi, j \in \delta_k \tag{3}$$

$$\sum_{i\in\xi} (x_{ijk} * \tau^{mem}) \leq \omega_{jk}^{mem} * y_{jk} \qquad \forall k \in \Psi, j \in \delta_k$$

$$x_{ijk},\ y_{jk} \in \{0, 1\}, \qquad \forall i \in \xi, k \in \Psi, j \in \delta_k \tag{4}$$

Cost Minimisation: As shown in Eq. 1, our objective is to minimise the cost of using the whole cluster while scheduling any job. The total cost is modelled as the aggregated cost of using all the VMs. The binary decision variable y_{jk} is used which controls whether VM j of type k is used or not.

$$y_{jk} = \begin{cases} 1 & \text{if the } j\text{th VM of type } k \text{ is used;} \\ 0 & \text{otherwise.} \end{cases}$$

Executor Placement Constraint: An executor can be placed only in one of the VMs and this placement constraint is denoted in Eq. 2. The binary decision variable x_{ijk} is used which controls whether executor i is placed on VM j of type k.

$$x_{ijk} = \begin{cases} 1 & \text{if executor } i \text{ is placed in } j\text{th VM of type } k; \\ 0 & \text{otherwise.} \end{cases}$$

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. The resource constraints for CPU cores and memory are shown in Eq. 3 and 4, respectively.

Bin packing is a combinatorial optimisation problem and has proved to be NP-Hard (Coffman Jr. et al., 2013). The above optimisation problem is an Integer Linear Programming (ILP) formulation of the multi-dimensional bin packing problem. When the scheduler has to schedule a job, the ILP model can be constructed by using the current job's resource demand and cluster resource availability. Then, it can be solved by exact methods such as Simplex (Nelder and Mead, 1965), Branch and Bound (Ross and Soland, 1975) to find the most cost-effective executor placement for that job. However, constructing the ILP dynamically before scheduling each job can be time-consuming. Especially, if the problem size goes bigger (large cluster, or jobs with many executors), the ILP might not be feasible as it requires exponential time to solve. In this case, efficient heuristic methods can be used for faster executor placement.

4.3. Job scheduler

The proposed job scheduler exhibits the following characteristics:

- The scheduler is online, that means it has no prior knowledge of job arrival and dynamically schedules jobs upon arrival.
- The scheduler prioritises jobs based on their deadline.
- The scheduler tries to minimise the cost of VM usage while placing the executors of a job.

Before discussing the scheduling algorithm, we introduce the important concepts used to design the scheduler.

Resource Unification Thresholds (RUT): As we have two types of resources (e.g., CPU and memory), the resource capacity of a VM and resource demand of a job cannot be represented with only one type of resource. Therefore, to holistically unify multiple types of resources, we introduce *RUT* and use it as a system parameter. Each of the thresholds acts as a weight for a single resource type, and the summation of these threshold values is 1 (Eq. 5). In our case, α is the threshold associated with CPU and β is the threshold associated with memory. Note that, this is a generalised unification which can be extended to multiple resource types depending on the system needs. A detailed discussion on how to assign Resource Unification Threshold (RUT) values is provided in Section 6.6.

Resource Availability (RA_{jk}): It is a metric that represents the resource availability of a VM in the unified form. Eq. 6 and Eq. 7 shows the formula to compute the total amount of CPU and memory in the cluster, respectively. We use the formula shown in Eq. 8 to calculate RA_{jk} of a VM. Here, the currently available amount from each resource type is converted to the percentage of resource w.r.t the total cluster resource (of the same type) and then multiplied to the corresponding RUT. Then, the total resource capacity is found by summing these values.

$$\alpha + \beta = 1 \tag{5}$$

$$CPU_{total} = \sum_{k\in\Psi} \sum_{j\in\delta_k} \omega_{jk}^{cpu} \tag{6}$$

$$MEM_{total} = \sum_{k\in\Psi} \sum_{j\in\delta_k} \omega_{jk}^{mem} \tag{7}$$

$$RA_{jk} = \frac{\omega_{jk}^{cpu}}{CPU_{total}} * \alpha + \frac{\omega_{jk}^{mem}}{MEM_{total}} * \beta \tag{8}$$

$$RD_{job} = \left(\frac{\tau^{cpu}}{CPU_{total}} * \alpha + \frac{\tau^{mem}}{MEM_{total}} * \beta \right) * E \tag{9}$$

Resource Demand (RD_{job}): It is a metric that represents the resource demand of a job in the unified form. We first find the resource demand of one executor, then multiply it to the total executors to find the RD_{job} as shown in Eq. 9.

JobBuffer, JobQueue and DeadlineJobQueue: We use a *JobBuffer* to hold all the incoming jobs that are submitted to the scheduler. Moreover, two priority queues: *JobQueue* and *DeadlineJobQueue* are used to keep regular and deadline-constrained jobs, respectively. In *JobQueue*, jobs are kept sorted in descending order of their resource demand (RD_{job}). Jobs are kept sorted based on the Earliest Deadline First (EDF) strategy in the *DeadlineJobQueue*. The scheduler can transfer jobs from the *JobBuffer* to the priority queues at any time.

Algorithm 1 shows the policy used by the proposed scheduler. When the scheduler starts, at first it fetches deadline-constrained jobs from the *JobBuffer* (line 3). As *DeadlineJobQueue* is kept sorted based on EDF, if a newly added deadline-constrained job has a tighter deadline than the already awaiting jobs, it will be extracted

Algorithm 1: Algorithm for the Job Scheduler.

```
   Input: JobBuffer, JobQueue, DeadlineJobQueue
 1 while SchedulerTerminationSignal ≠ true do
 2     while true do
 3         FetchDeadlineJobs(JobBuffer)
 4         if DeadlineJobQueue = ϕ then
 5             break
 6         end
 7         Job = ExtractJob(DeadlineJobQueue)
 8         if PlaceExecutor(Job) is successful then
 9             LaunchJob(Job, PlacementList)
10         end
11     end
12     while true do
13         FetchRegularJobs(JobBuffer)
14         if DeadlineJobQueue ≠ ϕ then
15             break
16         end
17         Job = ExtractJob(JobQueue)
18         if PlaceExecutor(Job) is successful then
19             LaunchJob(Job, PlacementList)
20         end
21     end
22 end
```

from the queue to be scheduled before any other jobs (line 7). If the *PlaceExecutor*() procedure returns success in finding VMs to place the executors, the job will be launched in the cluster (lines 8–9). The scheduler is not preemptive, so when a job is scheduled (whether it is a regular or a deadline-constrained job), it will not be killed or suspended. Therefore, while any deadline-constrained jobs are waiting and the cluster does not have sufficient resources to execute them (*PlaceExecutor*() procedure returns failure), the scheduler does not fetch any regular jobs until all the deadline-constrained jobs are scheduled. If there are no deadline-constrained jobs to schedule (line 4), only then the scheduler fetches regular jobs (line 13). Otherwise, it keeps trying to place executors for deadline-constrained jobs.

Before scheduling any regular jobs, the scheduler always checks whether any new deadline-constrained job has arrived. If so, it goes back to schedule those jobs (line 14–15). Otherwise, it starts scheduling regular jobs (lines 17–19). In some cases, it might be difficult to place a regular job with huge resource demand (as the *JobQueue* is kept sorted in decreasing order of resource demand for jobs). In these cases, the scheduler skips the current job and tries to schedule the next job from the *JobQueue*.

4.4. Executor placement

We propose two algorithms for cost-effective executor placements for any job in the cluster. The first algorithm constructs the Integer Linear Programming (ILP) model as shown in Section 4.2 and tries to solve the ILP problem to find the most cost-effective executor placement for the current job. The second algorithm uses a greedy approach which is a modified version of the Best Fit Decreasing (BFD) heuristic to solve bin packing problems. Both of these algorithms can be used as the *PlaceExecutor*() procedure of Algorithm 1.

4.4.1. ILP-Based executor placement:

Algorithm 2 shows the ILP-based executor placement approach. At first, the cluster status is updated to obtain the latest resource availability of each VM. After this step, the optimisation target, executor placement constraints, and resource capacity constraints are

Algorithm 2: ILP-based Executor Placement Algorithm.

```
   Input: Job, the current job to be scheduled
   Output: PlacementList, a list of VMs where the executors of
           Job will be placed
 1 Procedure PlaceExecutor(Job)
 2     PlacementList ← ϕ
 3     Update Cluster Resource Availability
 4     Generate Optimisation target (Eq. 1)
 5     Generate Executor Placement Constraints (Eq. 2)
 6     Generate Resource Capacity Constraints (Eq. 3,4)
 7     Solve ILP Problem
 8     if ILP is solved then
 9         return PlacementList
10     end
11     return Failure
12 end
```

dynamically generated by using the current cluster resource availability and the resource demand for the executors of the current job. Then the constructed ILP problem is solved (by an ILP solver). If a feasible solution is found, the *PlacementList* is returned which contains the chosen VMs where the executors can be created. Otherwise, if the modelled problem is not solvable, a failure is returned. Note that, when the constraints of resource availability are generated before scheduling each job, the VMs which are already used by other jobs will be set ($y_{jk} = 1$) so that the cost of using that machine will be taken into account in the optimisation target. Therefore, if there are any free resources available in the used VMs, the ILP solver will automatically try to fit as many executors as possible in those VMs before using any new VM to optimise cost.

4.4.2. BFD Heuristic-based executor placement:

To find the VMs where a job's executors can be placed, our proposed scheduler also uses a greedy algorithm. Algorithm 3 shows

Algorithm 3: BFD Heuristic-based Executor Placement Algorithm.

```
   Input: Job, the current job to be scheduled
   Output: PlacementList, a list of VMs where the executors of
           Job will be placed
 1 Procedure PlaceExecutor(Job)
 2     PlacementList ← ϕ
 3     Sort(VMList)
 4     forall the VM ∈ VMList do
 5         while Placement of an executor in VM satisfies the
           constraints (Eq.3,4) do
 6             Update Resource Availability in VM
 7             PlacementList.add(VM)
 8             if PlacementList.size = E then
 9                 return PlacementList
10             end
11         end
12     end
13     if Cluster has unused VM(s) then
14         Turn on the smallest VM_new that satisfies the
           constraints (Eq.3,4)
15         VMList ← VMList ∪ VM_new
16         goto step 3
17     end
18     return Failure
19 end
```

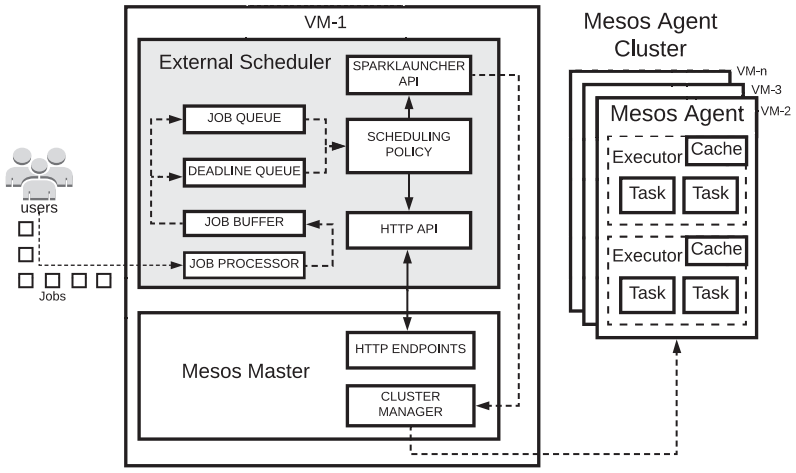


Fig. 4. The implementation of the prototype system on top of Apache Mesos.

the procedure *PlaceExecutor*() which can be used to find the executor placement of any job. At first, the *VMList* (a list of used VMs in the cluster) is sorted based on an ascending order of Resource Availability (RA_{jk}) of the VMs (line 3). Then, it iterates all the VMs (line 4) and checks whether the current VM's resource availability satisfies an executor's resource demand (line 5). If so, it updates the resource availability of that VM (line 6) and adds this VM to a list called *PlacementList* (line 7). Instead of looking at the next VM, the current VM is greedily used to place as many executors as possible so that we have a tight packing of the executors and use a fewer number of VMs in the cluster. If this procedure finds placements for all the executors of a given job, it returns the *PlacementList* (lines 8–9). If the VMs in *VMList* are not sufficient to place all the executors, and the cluster has unused VM(s) (line 13), the smallest VM that satisfies the resource constraints will be turned on (line 14) and added to the *VMList* (line 15). Then the placement finding steps will be repeated (line 16). Otherwise, if the cluster does not have sufficient resources to place all the executors of the current job, a failure will be returned (line 18).

4.5. Complexity analysis

To calculate the worst-case time complexity of Algorithm 1, we first assume that, p and r is the total number of deadline-constrained and regular jobs, respectively that need to be scheduled. If the total number of VM in the cluster is m, the time required to sort the *VMList* is $mlog(m)$. If an exact algorithm is used to solve the ILP model built in Algorithm 2, the worst-case time complexity is $O(2^n)$ where n is the maximum number of slots available for placing executors across all the VMs. However, the worst-case time complexity of the BFD-based greedy approach shown in Algorithm 3 is $O(me)$, where e is the maximum number of possible executors for any job. Therefore, if ILP-based executor placement is used, the worst-case time complexity of Algorithm 1 is, $O((2^n mlog(m))(p+r))$. Thus, it might require exponential time to complete the scheduling process for ILP based approach. In contrast, for the BFD-based executor placement, Algorithm 1 has a polynomial worst-case time complexity of $O((m^2 log(m))(p+r))$.

5. System design and implementation

We design a scheduler on top of the Mesos cluster manager instead of modifying the native Spark scheduler to implement our scheduling algorithms. The benefit of keeping a separate module for the scheduler without extending the existing framework is two-fold. First, it can be extended to work with any other data processing frameworks supported by Mesos. Second, it can be used as a generic scheduling framework so that new policies can be incorporated into the scheduler. The prototype scheduler can be treated as an external scheduler in the system architecture as depicted in Fig. 4. The implementation of the prototype system is open-source[13] so that it can be used or extended by the research community.

The external scheduler can be installed in any VM, but in our case, we plugged it in the Mesos master node and ran it as a separate application alongside with the Mesos master process. Users submit jobs to the external scheduler and depending on the scheduling policy, the scheduler provisions resources in the cluster and launch any job with the help of Mesos master. In the architectural diagram shown in Fig. 4, dashed lines represent job submission or executor creation flow where solid lines represent the control flows of the scheduler. As discussed previously in the algorithm section, there are three data structures to keep the jobs in the scheduler: Job buffer (to hold the incoming jobs), deadline queue (to hold deadline-constrained jobs), and job queue (to hold regular jobs). When the scheduler decides to schedule a job in the cluster, at first, it uses the Mesos HTTP APIs and sends JSON formatted request messages to Mesos master HTTP API endpoints to dynamically reserve resources. After getting the acknowledgment of successful resource reservation by the Mesos master, it launches that job through the Mesos cluster manager by using the SparkLauncher APIs. At this stage, the driver program of the launched Spark job takes control and creates executor(s) in one or more VMs by using the reserved resources only. At any point of the scheduling process, if a VM is unused and no jobs are currently reserved on it for any future jobs to be scheduled, it is turned off by the scheduler to save resource usage cost. Additionally, the scheduler can also turn on one or more VMs if the currently available resources in the active VMs is not sufficient to schedule new jobs.

We have implemented this pluggable external scheduling framework in Java. We have used SCPSolver[14] API with LPSolve Solver Pack[15] library to solve the proposed ILP-based executor placement model in the scheduler. To implement the automatic VM turn on/off mechanism from the scheduling process, we have developed a module by using OpenStack Boto3[16] library.

[13] https://github.com/tawfiqul-islam/SLA-Scheduler.
[14] http://scpsolver.org/.
[15] http://lpsolve.sourceforge.net/5.5/.
[16] https://boto3.readthedocs.io/en/latest/.

Table 3
Experimental cluster details.

Instance Type	CPU Cores	Memory (GB)	Pricing (AWS)	Quantity
m1.large	4	16	$0.24/h	6
m1.xlarge	8	32	$0.48/h	5
m2.xlarge	12	48	$0.72/h	3

However, this module can be easily extended to support any other cloud service providers by using their APIs. The scheduler also uses Mesos scheduler HTTP API and operator HTTP API to control the resource provisioning in the cluster. The Mesos master accepts messages in JSON format while communicating through the HTTP APIs. Therefore, java-json[17] API was used to construct/parse JSON formatted messages. Furthermore, SparkLauncher[18] API was used to automate Spark job submission from the scheduler. The scheduler accepts job submission requests from the users through a job processor interface that listens on a configurable TCP port. Job submission requests to the scheduler should be constructed in JSON format with some simple fields. In a job submission request, the users have to specify the details of a job having the following fields: job-id, input-path, output-path, application-path, application-main-class, resource requirement (CPU cores, memory in GB and total-executors) and an optional application argument (e.g., iteration).

6. Performance evaluation

In this section, we first provide the experimental setup details which includes the cluster resource configurations, benchmark applications, and baseline schedulers. Then we show the evaluations of the proposed algorithms in terms of cost, job performance, deadline violations, and scheduling overhead. Moreover, we also provide a sensitivity analysis of the system parameters and discuss the applicability of the proposed algorithms.

6.1. Experimental setup

6.1.1. Cluster configuration:

We have used Nectar Cloud[19], 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 with their similar pricing in Amazon AWS (Sydney, Australia) is shown in Table 3. In summary, our experimental cluster has 14 VMs with a total CPU (cores) of 100 and memory of 400GB. 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 external scheduler was plugged into the Mesos master node. Spark supports different input sources as mentioned before, and the users can select which data sources they want to use. However, HDFS is the most prominent distributed storage service as it is highly scalable, and provides fault-tolerance through replication. Generally, HDFS keeps replica of a storage block in 3 datanodes. Hence, if any of these datanodes (VMs) are turned-off to save cost, HDFS will automatically create replicas on the available VMs. However, a storage block might be lost if all the 3 datanodes where its replicas reside are turned off. Therefore, in this special case, the VM turn on/off module should be modified to allow HDFS to create replicas before shutting down all the datanodes. For the simplicity of the current system implementation to test our proposed approach, we have mounted a 1TB volume in the master node and created a *Network File System (NFS)* to share this storage space with all the Mesos agents. As the NFS server is running on the master node which will not be turned off, the current implementation does not need to consider about data loss due to VM turn off. In addition, the performance overhead due to fetching the input data from the NFS server is negligible as it is only done once at the beginning of the jobs execution, and all the intermediate results are stored in each VMs local storage which is managed by Spark. For providing fault-tolerance, we plan to extend our implementation to work with HDFS in the future. We have used Bash scripting to automate the cluster setup process so that a large-scale deployment can also be conducted through these scripts. Furthermore, an existing cluster can also be scaled up if more VMs are provisioned from the Cloud service provider.

6.1.2. Benchmarking applications:

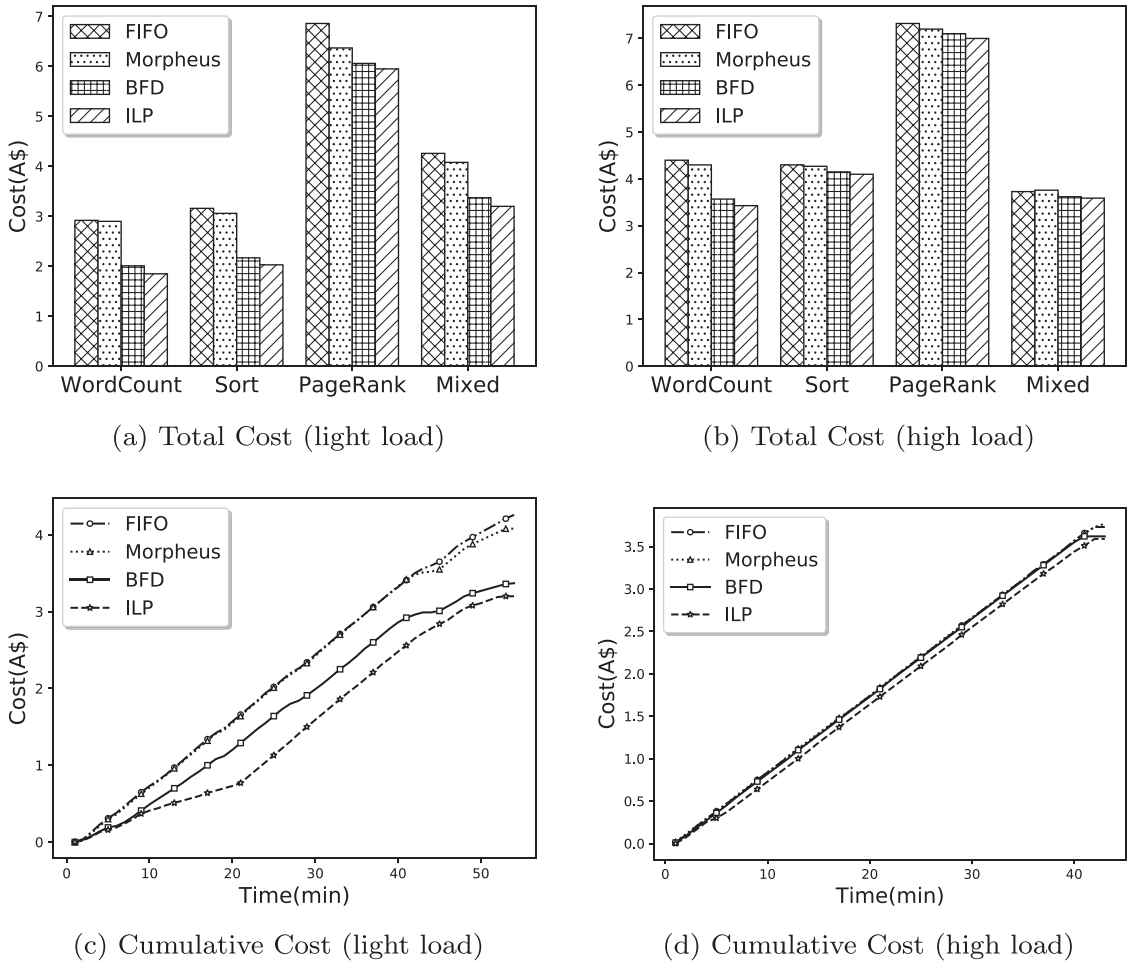
We have used BigDataBench (Wang et al., 2014), a big data benchmarking suite to evaluate the performance of our proposed algorithms. We have chosen three different types of applications from BigDataBench, namely WordCount (compute-intensive), Sort (memory-intensive) and PageRank (network/shuffle-intensive). Each application was used to generate a workload where each job in a workload has varying input size ranging from 1GB to 20GB (for WordCount and Sort) or iterations ranging from 5 to 15 (for PageRank). To generate a heterogeneous workload, we have randomly mixed the previously mentioned different types of applications. We have extracted the job arrival times from two different hours of a particular day from the Facebook Hadoop workload trace[20]. From a high-load hour, 100 jobs are used, and from a light-load hour, 50 jobs are used. The arrival rate of jobs in the high-load hour is higher than the light-load hour. Therefore, in the high-load hour, most of the resources are overwhelmed with jobs while in the light-load hour, the cluster is slightly under-utilised. The job profiles are collected by first submitting each job to run independently (without any interference from other jobs) in the cluster. Then the job completion time is averaged from multiple runs (5 for each job). While generating a workload, each job's average completion time is used as a hard deadline.

6.1.3. Baseline schedulers:

The problem with most of the cluster schedulers for Spark jobs is that they do not consider executor-level job placement. Most of these approaches only select the total number of resources or nodes (VMs) needed for each job while making any scheduling decisions. However, our approach works on a fine-grained level by incorporating executor placements in job scheduling. Therefore, the existing works can not be directly compared with our proposed approach. The following schedulers are compared with our proposed scheduling algorithms:

- FIFO: The default FIFO scheduler of Apache Spark deployed on top of Apache Mesos. It schedules jobs on a first come first serve basis. We have used the consolidation option of the scheduler so that it tries to pack executors in fewer VMs instead of distributing executors on a round-robin fashion. As most of the existing scheduling algorithms use this default approach for executor placement, and it is also the common choice of a user with Spark jobs, we chose this scheduler to be one of the baselines.
- Morpheus (Jyothi et al., 2016): We have adapted the executor placement policy of Morpheus. In this policy, *lowcost* packing is used for executor placement. Depending on the current cluster

[17] http://www.oracle.com/technetwork/articles/java/json-1973242.html.
[18] https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/launcher/package-summary.html.
[19] https://nectar.org.au/research-cloud/.
[20] https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.



(a) Total Cost (light load)

(b) Total Cost (high load)

(c) Cumulative Cost (light load)

(d) Cumulative Cost (high load)

Fig. 5. Cost comparison between the scheduling algorithms under different workload types.

load, this policy finds the scarce resource demand (e.g., memory or CPU cores) of each job (Eq. 10). Then jobs are sorted in increasing order of their scarce resource demands. Therefore, resources in the cluster are well-balanced throughout the scheduling process so that more jobs can be executed in the long run. As Morpheus also uses a packing based approach for executor placement, we chose it as a baseline.

$$c_{job} = Max\left(\frac{CPU_{load} + CPU_{job}}{CPU_{total}}, \frac{MEM_{load} + MEM_{job}}{MEM_{total}}\right) \quad (10)$$

Note that, Spark dynamic resource allocation feature was turned on for both the baseline and the proposed scheduling algorithms.

6.2. Evaluation of cost efficiency

In this evaluation, we show the applicability of our proposed scheduling algorithms to different types of applications while reducing the cost of using a big data cluster. To calculate the total cost incurred by a scheduler, we save the status of a VM (whether it was turned on or off) in each second. Lastly, all the per-second costs ($cost_i$, cost incurred in ith second, $i = 1, 2, 3, \ldots T$; T=total makespan of the scheduler) incurred by a scheduler is calculated by using Eq. 1. Then all these per-second costs are summed for the whole makespan of the scheduling process as shown in Eqn. 11 to find the $Total_{cost}$.

$$Total_{cost} = \sum_{i \in T} Cost_i \quad (11)$$

Fig. 5 depicts cost comparison between the scheduling algorithms under different workload types. The bar charts in Fig. 5a and Fig. 5b show the total cost incurred by different scheduling algorithms in the light-load and high-load hour, respectively. As our proposed scheduling algorithms use bin packing to consolidate the executors to a minimal set of VMs, the cost is reduced significantly as compared to other schedulers. In general, the ILP-based scheduling algorithms incur slightly lower cost than the BFD-based scheduling algorithm in all the scenarios as it can find the cost-effective executor placement for a job. Moreover, Morpheus performs slightly better than FIFO to lower the cost, because it prioritises jobs in such a way that cluster resources are well-balanced to execute more jobs in the overall scheduling process.

As shown in Fig. 5a, both BFD-based and ILP-based scheduling algorithms exhibit significant cost reductions during the light-load hour. As compared to baseline scheduling algorithms, BFD and ILP reduce the cluster usage cost by at least 30% and 34%, respectively for WordCount and Sort applications. For PageRank application, BFD and ILP reduce the resource usage cost by at least 12% as compared to FIFO. Moreover, BFD and ILP reduce the resource usage cost by at least 5% as compared to Morpheus. As our proposed scheduling algorithms try to place the executors from the same job in fewer nodes (VMs), most of the shuffle operations happen intra-node thus improving job performance which results in overall cost reduction for network-bound applications. In the case of the mixed workload, BFD and ILP reduce the resource usage cost by 21% and 25%, respectively as compared to FIFO. Furthermore, BFD and ILP reduce the resource usage cost by 17% and 22%, respectively as

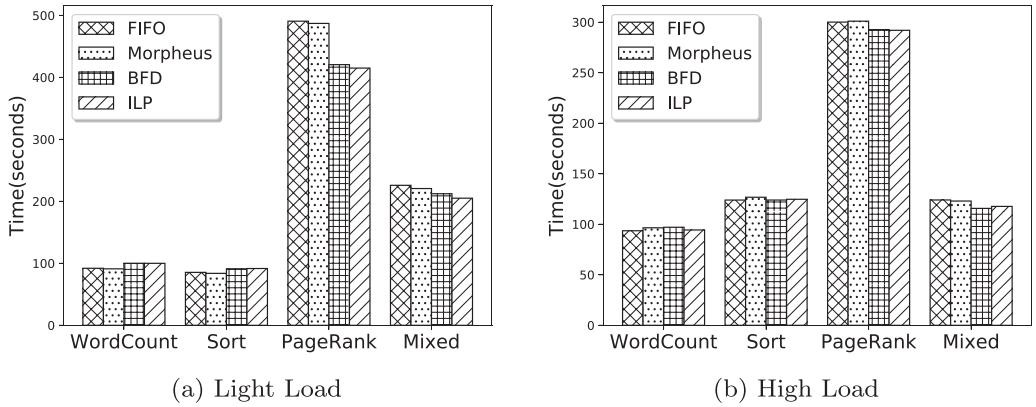


Fig. 6. Comparison between the scheduling algorithms regarding average job completion times under different workload types.

compared to Morpheus. In the case of the high-load hour as shown in Fig. 5b, the cost reduction is smaller than the light-load period as the cluster is over-utilised. In this scenario, BFD and ILP show about 5–20% of cost reduction in different workloads.

Fig. 5 c and Fig. 5d represents the cumulative VM cost by different scheduling algorithms during the whole scheduling process for the mixed workload in the light load and high load hours, respectively. It can be observed that in the high-load hour, the cumulative cost graph of all the scheduling algorithms look similar as it is not possible to reduce the cost significantly of an over-utilised cluster. However, in the light-load hour, the cost savings can be observed to increase over time for both BFD and ILP.

6.3. Evaluation of job performance

Fig. 6 a and 6 b report the average job completion times for different scheduling algorithms in light-load and high-load hours, respectively. It can be observed that for WordCount and Sort applications, sometimes FIFO and Morpheus perform slightly better than our proposed algorithms. As our algorithms use fewer VMs to place all the executors, these VMs are stressed as both CPU cores, and memory resources are used at full capacity. However, it is negligible as compared to the total resource cost usage by the baseline schedulers. On the contrary, network-bound applications such as PageRank reduces the performance of both FIFO and Morpheus due to excessive network communications during the shuffle periods. Therefore, both BFD and ILP outperform the baseline algorithms in case of PageRank and mixed applications. As all the algorithms perform similarly for CPU/memory intensive applications, performance benefits in mixed workload mainly depend on the proportion of network-intensive applications. In the high-load hour, the cluster is overloaded with jobs so it might not be possible to consolidate the executors from the same job in fewer VMs. Therefore, the performance benefits can be observed to be higher in the light-load hour than the high-load hour for the mixed and PageRank applications. In the light-load hour, our proposed algorithms improve job completion time for at least 14% and 5% for PageRank and mixed applications, respectively. In the high-load hour, our algorithms improve job completion time for at least 3% and 5% for PageRank and mixed applications, respectively.

6.4. Evaluation of deadline violation

In this evaluation, we compare the percentage of deadline violations of different scheduling algorithms. This performance metric (θ^d) is found by using Eq. 12 where θ_m and θ_s is the number of missed and satisfied deadlines by a scheduler, respectively.

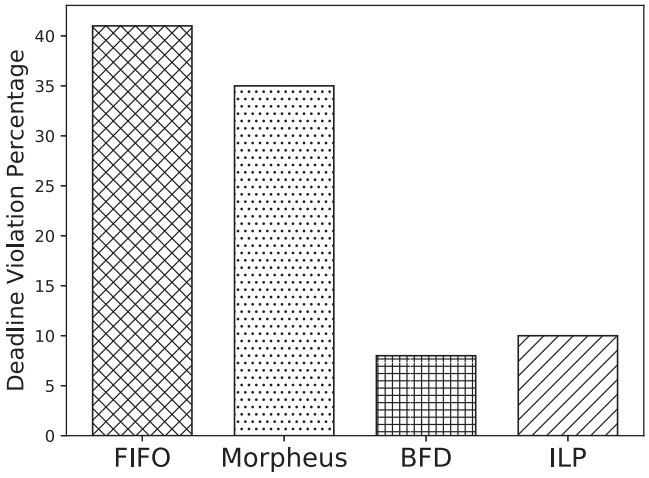


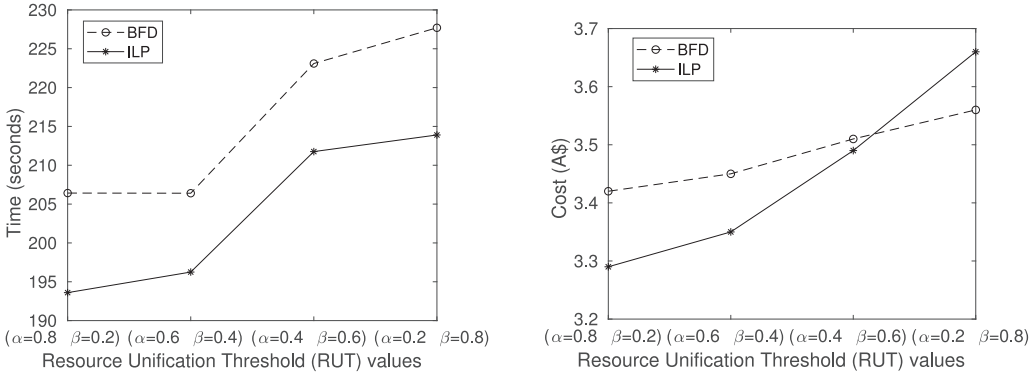
Fig. 7. Comparison of deadline violations by different scheduling algorithms.

$$\theta^d = \frac{\theta_m}{\theta_s} \times 100\% \tag{12}$$

Both FIFO and Morpheus do not consider deadline-constrained jobs. In FIFO, a high priority job with the earliest deadline has to wait in the scheduling queue if it is submitted after one or more non-priority jobs. It will be scheduled only after executing all the previously arrived jobs. Morpheus determines the job priority by itself, where a job which results in the most balanced distribution of resources in the cluster (if that job is scheduled) will have the highest priority. However, in reality, top priority deadline-constrained jobs might not provide balanced resource distributions upon placement. Therefore, other non-priority jobs will be executed before these jobs. Both BFD and ILP use a simple Earliest Deadline First (EDF) strategy. Thus, all the jobs are kept sorted according to their deadlines, and the job with the earliest deadline is scheduled first. Fig. 7 depicts the deadline violation percentage of different schedulers. For this experiment, we have executed a heterogeneous mix (different application types) of priority (strict deadline) and non-priority jobs to measure the deadline violations by each scheduler. For FIFO and Morpheus, deadline violation occurred for 41% and 35% of jobs, respectively. However, both BFD and ILP were able to meet the deadlines for most of the jobs and have deadline violation percentage of only 8% and 12%, respectively. ILP has slightly higher deadline violation than the BFD because sometimes it takes a significant time to find the most

Table 4
Comparison of average scheduling delays (unit: seconds) of different scheduling algorithms.

Schedulers	Light-load				High-load			
	WC	Sort	PR	Mixed	WC	Sort	PR	Mixed
FIFO	0.002	0.004	0.002	0.004	0.003	0.003	0.003	0.004
Morpheus	0.004	0.004	0.003	0.005	0.005	0.004	0.003	0.004
BFD-based	0.006	0.005	0.005	0.004	0.005	0.004	0.004	0.005
ILP-based	3.31	3	0.75	1.92	0.73	2.63	0.65	1.3



(a) Average Job Completion Time (b) Cost

Fig. 8. Effects of Resource Unification Threshold (RUT) values on average job completion time and cost.

cost-effective placement by this approach which causes deadline misses.

6.5. Evaluation of scheduling overhead

In this evaluation, we compare the scheduling delays caused by different scheduling algorithms. It is found by measuring the time it takes to find the executor placements of a job. Table 4 records the average scheduling delays by different scheduling algorithms under different workload types in both high-load and light-load hours. It can be observed that the native FIFO is the fastest among all the schedulers with scheduling delays averaging only from 2ms to 4ms. Both Morpheus and BFD are also fast as their average scheduling delay varies in the range from 3ms to 5ms and 4ms to 6ms, respectively. In contrast, as the ILP tries to find the most cost-effective executor placement for each job, in some cases it might require exponential time to complete. The results also indicate the same as the average scheduling delay varied from 0.65 seconds to up to 3.31 seconds for ILP. Although most of the jobs had a scheduling delay within 1 second, for the ILP, the average is higher as for some jobs it took about 3–4 minutes. The higher scheduling delay of ILP-based scheduling algorithm might cause some deadline misses. It can also be observed in Fig. 7 that, ILP-based scheduling algorithm has a slightly higher deadline miss percentage than the BFD-based algorithm. However, this performance degradation is negligible as compared to the baseline scheduling algorithms. Furthermore, for regular jobs or periodic jobs (e.g., long-running data analytics) that do not have strict deadlines, using the ILP-based scheduling algorithm is preferred as it can provide better cost reduction in the long run.

6.6. Effects of resource unification thresholds (RUT)

RUT is a system parameter, and we have performed a sensitivity analysis to demonstrate the effects of it on both cluster usage cost and job performance. In our experimental cluster, we have two types of resources (e.g., CPU cores and memory). Resource unification thresholds (RUT) play a vital role in the scheduling process by acting as a weight while combining these two types of resources to determine the resource capacity of the VMs or the resource demand of the jobs. We have associated α as the RUT for CPU cores and β as the RUT for memory. The proper balance between RUT values depends on both the VM instance types and the workload types. Fig. 8 represents the effects of different RUT values on both average job completion time (Fig. 8a) and resource usage cost (Fig. 8b). This analysis was done by running both BFD and ILP-based scheduling algorithms with the mixed workload. It can be observed from the figure that, decreasing the α value and increasing the β value tends to increase both average job completion time and resource usage cost in our experimental cluster. As using $\alpha = 0.8$ and $\beta = 0.2$ gives us both lower cost and job completion time, we use these RUT values in our experiments.

RUT values can also be tuned to give more priority to specific VMs or jobs. For example, if a cluster has more memory-bound jobs, to prefer VMs which have more memory to fit these jobs correctly, the β value can be increased, and α value can be decreased so that VMs which have high memory capacity/availability are preferred in the scheduling process. Similarly, jobs can also be prioritised based on their demand on a particular resource-type by adjusting the corresponding RUT values.

6.7. Discussion

The proposed scheduling algorithms can be applied to optimise the cost of using a cloud-deployed Apache Spark cluster. Our performance evaluation results show that the BFD heuristic-based approach performs very close to the ILP-based approach in all the cases. However, the ILP-based approach might have significant scheduling delays for a large cluster (many VMs). Therefore, in this case, we recommended using the BFD-based scheduling algorithm as it gives similar results with a small scheduling overhead identical to the native FIFO. Another approach could be using both algorithms and using a time-constraint in the ILP. If the ILP can be solved within the time-constraint, the executor placements found by this approach will be used. Otherwise, the solution from the BFD-based approach will be used.

The proposed approach can also be used with HDFS. As HDFS generally creates replicas in 3 datanodes (VMs), if all these 3 VMs are selected to be turned off in the scheduling process to save cost, a storage block which was only saved in these 3 VMs will be lost. To mitigate this issue, it is not required to modify the scheduling algorithms. However, the VM turn on/off module should be modified for allowing HDFS to create replicas before shutting down a VM (datanode).

7. Conclusions and future work

Scheduling is a challenging task in big data processing clusters deployed on the cloud. It gets even harder in the presence of different types of VMs and job heterogeneity. Most of the existing schedulers only target on improving job performance. In this paper, we have used bin packing to formulate the scheduling problem and proposed two dynamic scheduling algorithms that enhance job performance and minimise resource usage cost. We have built a prototype system on top of Apache Mesos which can be extended to incorporate new scheduling policies. Therefore, this system can be used as a scheduling framework. We have demonstrated the outcomes of our extensive experiments on real datasets to prove the applicability of the proposed algorithms under various workload types.

Moreover, we have compared our algorithms with the existing baseline schedulers. The results suggest that our proposed scheduling algorithms reduce resource usage cost up to 34% in a cloud-deployed Apache Spark cluster. Furthermore, both network-bound and mixed jobs gain performance benefits (up to 14%) from tighter packing of executors in fewer VMs. We have also done the sensitivity analysis of the system parameter and discussed the effects of it on both cost and job performance. Lastly, we have discussed the feasibility of the proposed approach.

In the future, we plan to extend the proposed scheduling algorithms by incorporating some essential SLA requirements, such as budget, and job inter-dependency. Furthermore, we would like to combine the performance prediction/modelling of jobs with the schedulers to dynamically determine the resource requirements of jobs while satisfying SLA and performance constraints. Although Spark jobs do not specify network as a resource constraint, when co-locating multiple jobs in the same VM, network and I/O usage should also be considered. To achieve this, we want to extend our work so that these constraints can be added to the optimisation problem. As major cloud service providers such as AWS is offering Arm-based instances[21] that consume less-power and inexpensive, user-centric cost optimisation techniques should be benefited if arm-based instances are used. Although arm-based instances are inexpensive, x86-based instances still outperform arm-based instances if compared regarding instance performance. Therefore, in the future we plan to investigate the trade-offs between arm-based and x86-based instances regarding cost and performance.

CRediT authorship contribution statement

Muhammed Tawfiqul Islam: Conceptualization, Methodology, Software, Formal analysis, Validation, Investigation, Data curation, Writing - original draft, Visualization. **Satish Narayana Srirama:** Conceptualization, Methodology, Formal analysis, Writing - review & editing. **Shanika Karunasekera:** Conceptualization, Formal analysis, Writing - review & editing, Supervision. **Rajkumar Buyya:** Conceptualization, Resources, Writing - review & editing, Supervision, Project administration, Funding acquisition.

[21] https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors/.

Acknowledgements

We would like to thank Dr. Xunyun Liu, Prof. Vlado Stankovski, Mr. Shashikant Ilager, Dr. Maria Rodriguez, Dr. Adel Nadjaran Toosi and Dr. Minxian Xu for providing their valuable suggestions to improve this work. This work is partially supported by a research grant from the Australian Research Council (ARC).

References

Chen, Q., Zhang, D., Guo, M., Deng, Q., Guo, S., 2010. Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In: Proceedings of the 10th IEEE International Conference on Computer and Information Technology.

Coffman Jr., E.G., Csirik, J., Galambos, G., Martello, S., Vigo, D., 2013. Bin Packing Approximation Algorithms: Survey and Classification. Springer New York, New York, NY, pp. 455–531.

Delimitrou, C., Kozyrakis, C., 2014. Quasar: Resource-efficient and qos-aware cluster management. In: Proceedings of the 19th International Conference on Architectural support for programming languages and operating systems (ASPLOS).

Dimopoulos, S., Krintz, C., Wolski, R., 2017. Justice: A deadline-aware, fair-share resource allocator for implementing multi-analytics. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER).

George, L., 2011. HBase: the definitive guide: random access to your planet-size data. O'Reilly Media, Inc..

Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I., 2011. Dominant resource fairness: Fair allocation of multiple resource types. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI).

Gibilisco, G.P., Li, M., Zhang, L., Ardagna, D., 2016. Stage aware performance modeling of dag based in memory analytic platforms. In: Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD).

Gounaris, A., Kougka, G., Tous, R., Montes, C.T., Torres, J., 2017. Dynamic configuration of partitioning in spark applications. IEEE Transactions on Parallel and Distributed Systems (TPDS) 28 (7), 1891–1904.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I., 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI).

Hunt, P., Konar, M., Junqueira, F.P., Reed, B., 2010. Zookeeper: Wait-free coordination for internet-scale systems. In: Proceedings of the USENIX Annual Technical Conference.

Islam, M.T., Karunasekera, S., Buyya, R., 2017. dspark: Deadline-based resource allocation for big data applications in apache spark. In: Proceedings of the 13th IEEE International Conference on e-Science (e-Science).

Jyothi, S.A., Curino, C., Menache, I., Narayanamurthy, S.M., Tumanov, A., Yaniv, J., Mavlyutov, R., Goiri, I.n., Krishnan, S., Kulkarni, J., Rao, S., 2016. Morpheus: Towards automated slos for enterprise clusters. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI).

Kc, K., Anyanwu, K., 2010. Scheduling hadoop jobs to meet deadlines. In: Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science.

Lakshman, A., Malik, P., 2010. Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. 44 (2).

Nelder, J.A., Mead, R., 1965. A simplex method for function minimization. Comput. J. 7 (4), 308–313.

Ross, G.T., Soland, R.M., 1975. A branch and bound algorithm for the generalized assignment problem. Math. Program. 8 (1), 91–103.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. The hadoop distributed file system. In: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST).

Sidhanta, S., Golab, W., Mukhopadhyay, S., 2016. Optex: A deadline-aware cost optimization model for spark. In: Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).

Soualhia, M., Khomh, F., Tahar, S., 2017. Task scheduling in big data platforms: a systematic literature review. J. Syst. Softw. 134, 170–189.

Tian, C., Zhou, H., He, Y., Zha, L., 2009. A dynamic mapreduce scheduler for heterogeneous workloads. In: Proceedings of the 8th International Conference on Grid and Cooperative Computing.

Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al., 2013. Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th ACM Annual Symposium on Cloud Computing.

Wang, G., Xu, J., He, B., 2016. A novel method for tuning configuration parameters of spark based on machine learning. In: Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC).

Wang, K., Khan, M.M.H., 2015. Performance prediction for apache spark platform. In: Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications (HPCC).

Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., et al., 2014. Bigdatabench: A big data benchmark suite from internet services. In: Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA).

Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I., 2009. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55. EECS Department, University of California, Berkeley.
Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I., 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI).
Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I., 2016. Apache spark: a unified engine for big data processing. Commun. ACM 59 (11), 56–65.

Muhammed Tawfiqul Islam is a Ph.D. candidate with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He received the BS and MS degree in Computer Science from the University of Dhaka in 2010 and 2012, respectively. He joined as Lecturer in the Department of Computer Science & Engineering, the University of Dhaka in 2014. Before joining academia, he also worked as a Software Engineer for Internet Society and REVE Systems. His research interests include resource management, cloud computing, and big data.

Satish Narayana Srirama is a Research Professor and the head of the Mobile & Cloud Lab at the Institute of Computer Science, University of Tartu, Estonia and a Visiting Professor at the University of Hyderabad, India. He received his Ph.D. in computer science from RWTH Aachen University, Germany, in 2008. His current research focuses on cloud computing, mobile web services, mobile cloud, Internet of Things, fog computing, migrating scientific computing and enterprise applications to the cloud and large scale data analytics on the cloud. He is an IEEE Senior Member, an Editor of Wiley Software: Practice and Experience journal and a program committee member of several international conferences and workshops.

Shanika Karunasekera received the B.Sc. degree in electronics and telecommunications engineering from the University of Moratuwa, Sri Lanka, in 1990 and the Ph.D. degree in electrical engineering from the University of Cambridge, Cambridge, U.K., in 1995. From 1995 to 2002, she was a Software Engineer and a Distinguished Member of Technical Staff with Lucent Technologies, Bell Labs Innovations, USA. In January 2003, she joined the University of Melbourne, Victoria, Australia, and currently, she is a Professor in the School of Computing and Information Systems. Her current research interests include distributed system engineering, distributed data-mining, and social media analytics.

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory in the School of Computing and Information Systems at the University of Melbourne, Australia. He has authored over 625 publications and seven textbooks including “Mastering Cloud Computing” published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. Microsoft Academic Search Index ranked Dr. Buyya as #1 author in the world (2005–2016) for both field rating and citations evaluations in the area of Distributed and Parallel Computing. “A Scientometric Analysis of Cloud Computing Literature” by German scientists ranked Dr. Buyya as the World’s Top-Cited (#1) Author and the World’s Most-Productive (#1) Author in Cloud Computing. Dr. Buyya is recognized as a “2016 Web of Science Highly Cited Researcher” by Thomson Reuters, a Fellow of IEEE, and Scopus Researcher of the Year 2017 with Excellence in Innovative Research Award by Elsevier for his outstanding contributions to Cloud computing.