2019 IEEE International Conference on Big Data (Big Data)

Elastic Executor Provisioning for Iterative Workloads on Apache Spark

Donglin Yang, Wei Rang, Dazhao Cheng
University of North Carolina, Charlotte
dyang33, wrang, dazhao.cheng@uncc.edu

Yu Wang
Temple University
wangyu@temple.edu

Jiannan Tian, Dingwen Tao
The University of Alabama
jtian10@crimson.ua.edu, tao@cs.ua.edu

***Abstract*—In-memory data analytic frameworks like Apache Spark are employed by an increasing number of diverse applications—such as machine learning, graph computation, and scientific computing, which benefit from the long-running process (e.g. executor) programming model to avoid system I/O overhead. However, existing resource allocation strategies mainly rely on the peak demand normally specified by users. Since the resource usages of long-running applications like iterative computation vary significantly over time, we find that peak-demand-based resource allocation policies lead to low cloud utilization in production environments. In this paper, we present an elastic utilization aware executor provisioning approach for iterative workloads on Apache Spark (i.e., *iSpark*). It can identify the causes of resource underutilization due to an inflexible resource policy, and elastically adjusts the allocated executors over time according to the real-time resource usage. In general, iterative applications require more computation resources at the beginning stage and their demands for resources diminish as more iterations are completed. iSpark aims to timely scale up or scale down the number of executors in order to fully utilize the allocated resources while taking the dominant factor into consideration. It further preempts the underutilized executors and preserves the cached intermediate data to ensure the data consistency. Testbed evaluations show that iSpark averagely improves the resource utilization of individual executors by 35.2% compared to vanilla Spark. At the same time, it increases the cluster utilization from 32.1% to 51.3% and effectively reduces the overall job completion time by 20.8% for a set of representative iterative applications.**

I. Introduction

While many data analytic systems [21] are widely applied to process diverse workloads, various cloud management systems [17] are designed to effectively allocate physical resources according to underlying resource requirements. However, substantial disparities between the resource usage requested by jobs and the actual system resource utilization still exist in production clouds. Industrial studies [16] from Twitter and Google show that typical disparities are around 53% for CPU while the actual CPU utilization is only between 20% and 35%. Although many resource control policies have been proposed to improve the cloud resource utilization with different techniques, e.g., resource provisioning [17], job scheduling [11][19] and load balance [6][15], most of them focus on multi-process programming models like MapReduce [5], in which task is the basic scheduling unit. They often optimize the resource allocation via resizing containers or VMs (Virtual Machines), where JVMs (Java Virtual Machines) are launched to execute the computation of tasks. Then, different tasks are executed by different processes so that each JVM can initiate a new process with flexible resource allocation. However, compared to multi-process programming models, Apache Spark, one of the most popular in-memory data analytics platform, utilizes multiple threads instead of multiple processes to achieve parallelism on a single node. It adopts an executor-based multi-thread programming model. Different from container or VM, the executor is a long-running JVM process, in which multiple tasks can be executed concurrently via multiple threads. It avoids the memory overhead of several JVMs but leads to individual executors occupying a large amount of resources over a long time, which makes traditional dynamic resource control techniques for multi-process programming models inefficient in Spark.

Apache Spark's unique programming model provides intermediate data consistency in memory between computation tasks, which eliminates significant amount of disk I/Os and reduces data processing times. In particular, Spark runtime outperforms Hadoop runtime by more than ten times for machine learning or iterative applications [21]. This is the reason why iterative computations are increasingly common on data-parallel clusters. However, the diverse resource consumption patterns of various iterative tasks potentially result in mismatch in cluster resource usage, which should be handled properly. In particular, the data processing of many iterative applications is typically iterative with diminishing return [22][3]. In particular, it requires more computation resource at the beginning and then the demands for resource diminish as more iterations are completed. However, most existing policies primarily focus on resource fairness or load balance, which are agnostic to the real-time resource demand and job runtime.

Indeed, Spark applies peak demand based policies to allocate resources i.e., number of executors. The static allocation policy reserves resources upfront, while the dynamic allocation policy simply requests resources as much as application's demands. In both scenarios, these long-running executors keep alive until the entire Directed Acyclic Graph (DAG) is finished. In other words, they will hold all the requested resources once allocated to the job, though the job's demand may diminish in the later stages. A few recent studies [4] try to dynamically change the parallelism within each stage to improve the resource utilization. Unfortunately, they do not

978-1-7281-0858-2/19/$31.00 ©2019 IEEE

fundamentally address the underutilization problems when the long-running application requires time-varying resources. This disparity causes the formation of idle resources, producing inefficient cluster resource usage and an inability to provision computing services.

Many existing studies [11][3][10] have shown that dynamically tuning resources at runtime can effectively utilize computation resources and reduce resource contention. However, it is challenging to exploit time-varying resource demands of various iterative workloads on Apache Spark. First, resource allocation would be counterproductive while lacking accurate accounting information of the resource usage in the default system. Most existing schedulers [9] assign resources based on the peak demand estimation, which can not capture the real-time resource usage and further lead to resource underutilization. In contrast, an ideal scheduler should provision cluster resources in a timely manner if any resource under-utilization is detected during runtime. However, it is still difficult to decide the number of allocated executors and which executor to be evicted from the running list facing time-varying multi-resource demand across different stages. Second, it is necessary to ensure the data consistency when applying elastic resource scheduling strategies, e.g. adding and removing executors. In particular, removing running executors directly may cause the re-computation of lost data, which is a significant overhead to in-memory data analytics computation. To tackle the aforementioned challenges, we present an automated utilization aware executor provisioning approach for long-running iterative workloads on Apache Spark (i.e., iSpark). It monitors the real-time resource usage for individual executors and flexibly scales up or down the allocated resources to best fit the application's demand, i.e., minimizing the resource slack in the cluster. iSpark further preserves the cached intermediate result on the underutilized executors before removing them to ensure consistency. This policy can effectively avoid the additional overhead caused by implementing the scaling down decisions. Specifically, this paper makes the following contribution[1]:

- We empirically study and demonstrate the time-varying resource demand of iterative workloads running on Apache Spark. We find that the demands for resources of iterative applications are high at the beginning stage while diminishing in the later stages as more iterations are completed.
- We propose iSpark, an elastic executor management framework compatible with Apache Spark. It monitors the real time resource usage and adjusts the allocated resource strategically at runtime to capture varying workload demands and improve cluster utilization.
- We design and implement a new component with adaptive policy to preempt the underutilized executors gently. It preserves the cached intermediate data from underutilized executors to ensure the data consistency.
- We evaluate iSpark based on the representative workloads from HiBench [8]. Our results demonstrate that iSpark improves cluster's CPU utilization by 29% compared to default Spark resource allocation policies and effectively reduces the overall job completion time 20.8%.

[1]The source code is available at https://github.com/Young768/iSpark.

The rest of this paper is organized as follows. Section II gives background and motivations on resource allocation. Section III describes the detailed system design and embedded resource scheduling algorithm. Section IV presents experimental results. Section V reviews related work. Section VI concludes the paper.

II. Motivation

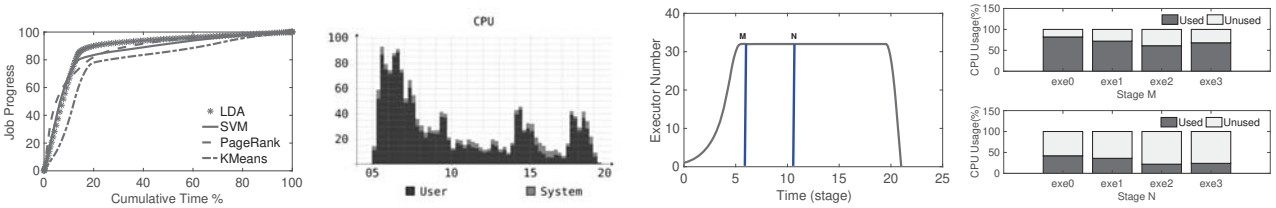
A. Background

Spark currently applies two peak demand based policies (i.e., static and dynamic allocation models) to allocate resources for task execution. In the mode of static allocation, users control the amount of application resource by configuring the number of executors. The application reserves CPU and memory resources over the full job duration. The dynamic allocation can only increase the number of executors simply to meet the peak demand of workloads until it reaches the upper number set by users. Note that the executor can exit only when the corresponding process becomes idle in both scenarios. Obviously, these peak demand based strategies in Apache Spark may lead to severe resource wastage when the application's resource usage varies significantly in different stages. Extremely, the idle executors are not allowed to be shut down because of the cached intermediate data on them. Otherwise, it will incur additional overhead by re-computation [2]. Thus, a more fine-grained and holistic design is expected for those iterative applications with diminishing resource demand.

B. Empirical Study

To quantify the time-varying resource demand of iterative workloads, we profile the resource usage of executors by using the benchmark from HiBench [8]. We conducted a case study based on a 9-nodes cluster, where 8 nodes serve as slave nodes and one serves as master node. We configured that each executor with 2 GB memory and one CPU core. We repeated each experiment for 10 times and collected the average data.

1) Diminishing Demand of Iterative Processing: Firstly, we study the resource demand nature of iterative applications under the default static resource allocation policy. In the experiment, we set the number of executors to be 8. We ran a number of iterative applications provided in HiBench [8], including LDA, SVM, PageRank and KMeans. Figure 1(a) shows the accumulative job progress achieved with the statically allocated resources. The result also shows that it takes 20% time for the SVM job to achieve progress by 90%, and 80% time to further finish training. The other three applications also exhibit similar features: the first several iterations generally boost the job progress very quickly. These features can be explained as the law of diminishing return, which has been applied in many other data analytic systems in addition to machine learning [22]. It implies that the



(a) Accumulative Job Progress. (b) Accumulative of CPU Usage. (c) Number of Executor. (d) CPU Utilization.

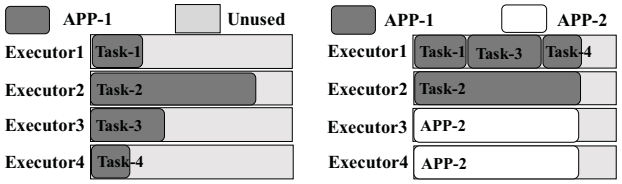
Fig. 1. Empirical study results of iterative job progress analysis, accumulative CPU usage and time-varying CPU utilization on executors.

iterations in later stages may result in marginal performance if still given the same amount of computational resources. To further demonstrate our observations, we collect the real-time CPU utilization trace of the cumulative CPU usage across all machines, which is illustrated in Figure 1(b). The results confirm that the CPU usage for iterative application is quite high at the beginning, which is nearly close to 80%. As the iterative processes going on, the CPU usage drops to below 30%. The above observations demonstrate that the default resource allocation is inefficient as they are agnostic to the actual resource demand within each job. Thus, we find an opportunity here that deallocating the resources from these applications in the later stages may not affect their performance but result in higher resource efficiency.

2) Underutilized Executors: We further enabled the dynamic allocation policy and set the maximum number of executors to be 32. We depicted the number of executors against time in Figure 1(c) and recorded the corresponding average CPU utilization at time M and N in Figure 1(d). It shows that the number of executors increased exponentially at the beginning stage until it reached the upper bound. The dynamic allocation policy allows Spark to request executors in rounds (i.e. stages), which is triggered when there are pending tasks backlogged for more than the duration set by user. The result in Figure 1(c) also demonstrates the effectiveness of such scaling up resource control policy. Figure 1(d) shows the average CPU usage at time M and N on four selected executors for PageRank application. The result illustrates that the average CPU usage achieves nearly 70% at stage M. However, the average CPU usage decreases to 30% at stage N, which is lower than half of peak value at stage M. The number of executors remained the same even though there are under-utilized nodes, e.g., exe2 and exe3. These results demonstrate that the current dynamic allocation policy cannot evict the under-utilized nodes effectively and lead to severe resource fragmentation and wastage. A key observation is that the number of allocated executors should be reduced when there are under-utilized nodes. The utilization of individual executors should be improved and then the deallocated resources can be released or utilized by other applications co-hosting in the cluster.

C. Motivation of Elastic Provisioning

The above observations motivate us that the number of allocated executors should be dynamically tuned during runtime to avoid the resource wastage. This work focuses on the scaling down policy to handle the under-utilized problem in the later stage of iterative job execution. We intuitively explain our proposed scheme via a toy example in Figure 2. Note that the current dynamic allocation policy only starts to remove executors if there is no running tasks and cached data on the executor. The parallelism of the computation will maintain the initial setting during the runtime. However, for cases where the computation in the later stage requires less resources, it will not remove the corresponding executors (e.g. Executor 3 and 4 in Figure 2(a)) though their utilizations are quite low. This is because of the fact that running tasks and cached data are still on these underutilized executors. To overcome this problem, a more aggressive strategy is expected to combine the tasks running on the under-utilized Executor 3 and 4 into a single one executor (i.e., Executor 1), resulting in two high-utilized nodes as shown in Figure 2(b). At the same time, the cached intermediate data on Executor 3 and 4 should be preserved to ensure the data consistency. Then these released resource (i.e., Executor 3 and 4) can be utilized by other applications or shut down for energy efficiency.



(a) Peak Demand Base Allocation (b) Utilization Aware Allocation

Fig. 2. A toy example shows different allocation policies.

III. System Design

A. Architecture Overview

Figure 3 shows the architecture of iSpark. It includes three centralized components, i.e., *iMetricCollector*, *iController* and *iCacheManager* and two distributed mechanisms, i.e., *Monitor* and *iBlockManager*, which are implemented on each participating executor. The key functionality of *iSpark* is built on top of Apache Spark, which can timely scale up or scale down the number of executors in order to fully utilize the allocated resources while considering the multiple resource constraints. From the perspective of workflow, Spark usually launches a driver program together with a SparkContext object after job

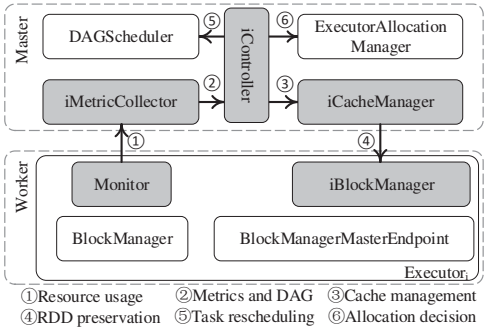


Fig. 3. Architecture of iSpark.

submssion. Within SparkContext, the proposed new component *iController*, $iMetricCollector$ and *iCacheManager* are instantiated along with existing components *DAGScheduler* and *ExecutorAllocationManager*. Then, Spark driver launches multiple executors across the cluster based on the resource demand, which will result in *Monitor* being deployed on each executor accordingly. We briefly describe the major components of the new mechanisms in iSpark.

- *iMetricCollector* collects the real-time information (e.g. CPU and memory metrics) and the operation logic information, i.e. RDD dependency, from DAG scheduler. *Monitor* will report the resources usage information of corresponding executors to *iMetricCollector* periodically via the system heartbeat.
- *iController* makes the provisioning decisions based on the metrics provided by *iMetricsCollector*, which will further request *ExecutorAllocationManager (EAM)* to perform the provisioning decision in terms of the number of executors.
- Centralized *iCacheManager* coordinates with *iController* to ensure the data consistency. *iCacheManager* is responsible for managing RDDs, which applies DAG-aware policy to preserve data partitions and updates the related RDD information to DAG scheduler.
- Distributed *iBlockManager* replicates the data blocks on these executors to be removed based on the provisioning decision and the DAG-aware policy provided by *iCacheManager*.

B. Utilization Aware Executor Provisioning

The resource provisioning policy embedded in *iController* aims to minimize the number of allocated executor, and satisfy the application's demand at the same time. The dynamic provisioning procedures of iSpark are based on the utilization information collected from the previous iterations of the running application, which is provided by $iMetricsCollector$. We first formulate the proposed resource provisioning problem and then address it based on the bin packing problem. We finally transform the multi-dimensional bin packing problem into the classic packing problem by using a dominant factor scheduling approach.

1) Problem Statement: For a given workload set, makespan is the duration between workload submission and completion. Equivalently, the makespan of applications can be minimized if all available resources can be fully utilized by applications along time. To achieve the above goal, iSpark aims to improve the resource utilization at both application and cluster level in this problem. In vanilla Spark, a user submits an application with parallelism of m requesting n executors (normally n $\leq$ m). Then the cluster manager allocates $\sum_{i}^{n} R_i$ resources for a specific iterative application (R_i is a multi-dimensional vector, with dimensions representing CPU and memory). We denote r_i as the peak resource request of a task x_i allocated on corresponding executor. Currently, vanilla Spark allocates resources for applications based on their peak demands. Initially, the allocated resource should satisfy that: $G = \{ x_i | \sum r_i \leq \sum_{i}^{n} R_i \}$. In iSpark, we use u_i to represent the estimated utilization of a task x_i running on the hosting machine and U_j to represent the utilization of executor j. We estimate u_i by using the historical utilization informations from tasks running on workers and provision sufficient resources at once based on the estimation.

Once an application is submitted at t, let x_{ij} denote that task i will be allocated on executor j and y_j be an indicator, where y_j represents that executor j is used. There are three constraints in the problem formulation as follows. First, at least one executor should be allocated for the computation during the job execution:

$$1 \leq \sum_{j}^{n} y_j. \tag{1}$$

Second, iSpark keeps the same parallelism of the application for the whole duration, which means the number of tasks or partitions will not be changed during the whole runtime:

$$\sum_{j} \sum_{i} x_{ij} y_i = m. \ \forall i, j. \tag{2}$$

Third, the resource usage of each executor j should not exceed its capacity. Here, we denote a dynamic threshold (i.e., $\min\{\theta,\ U_{max}\}$), which enables the system to automatically adjust its behavior depending on the workload patterns exhibited by the applications. θ represents each executor's fixed capacity, which is set as 100% by default. U_{max} is the highest usage of executor from previous iterations. This threshold represents a trade-off between efficiency and performance. On the one hand, it mitigates the impact from skewness to make the scheduling more efficient; On the other hand, it ensures that the system does not run out of resources. Each executor's estimated resource usage is $u_i(t - t_i)$ after the tasks have been executed since t_i. Thus,

$$\sum_{i} x_{ij} u_i (t - t_i) \ \leq \ y_i \min\{\theta, U_{max}\}. \tag{3}$$

The objective is formulated as below, which aims to maximize the usage of these allocated executors, i.e, minimizing the

number of executors to be assigned during the job execution.

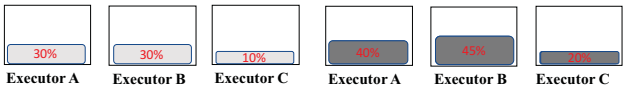
$$\min \sum_{i=1}^{n} y_i, \quad s.t.\ (1)\ (2)\ (3). \tag{4}$$

As the objective function and the constraint (3) are both nonlinear, it is difficult and computationally expensive to solve the problem directly. Thus, we introduce a heuristic algorithm as follows to tackle the aforementioned challenges.

Algorithm 1 Adaptive provisioning

```
1: update u_i via heartbeat from monitor;
2: sort U_j in decreasing order;
3: if there are pending jobs and U_j ≤ γ then
4:     for task i in taskset do
5:         arg max U_j based on DFS;
6:         if U_j ≤ min{θ, U_max} then
7:             pack task i into executor j;
8:             update and sort U_j in decreasing order;
9:             remove x_i from taskset;
10:        end if
11:    end for
12: end if
```

2) Adaptive Provisioning Policy: The optimization problem defined in Equation (4) can be transferred to a bin packing problem [20] with variable bin sizes and balls, where bins represent the executors and balls are the task set to be allocated. Bin sizes are available capacities of the executors and the size of balls corresponds to the resource usage of tasks. First, we draw an analogy with the solution in a one-dimensional space (both balls and bins). An effective heuristic for such problem is to repeatedly match the largest ball to fit in the current bin. When no more balls fit, a new bin will be opened. Intuitively, this approach reduces the unused space in each bin (i.e., reduces fragmentation in our problem) and therefore, reduces the total number of bins used, which can be equivalent to minimizing executor number *N* and fully utilizing the existing bins. As the bin packing problem is NP-Hard, we then apply a Best Fit Decreasing Packing (BFDP) algorithm, which requires no more than ($\frac{11}{9}\times OPT$+1) [20] bins, where OPT is the number of bins provided by the optimal solution. To efficiently trigger the algorithm, we introduce a lower bound γ, which is chosen empirically based on the sensitivity evaluation in Section IV-D. As shown in Algorithm 1, iSpark adaptively adjusts resource provisioning when there are pending jobs in the queue and underutilized executors simultaneously. For each task to be scheduled, the algorithm tries to pack it into the executor which has the least available capacity among all eligible executors, to maximize U_j based on BFDP (as shown in line 5-8). If it can satisfy the capacity's constraint (Eq. 3), the task will be packed into that executor directly. If not, the algorithm will keep searching for executor with enough available resources. After this, the executor will be classified into two categories: *receivers* or *givers*. *Givers* are defined as those executors who will give up their execution mission for the remaining stages and release the underutilized resource according to the new placement decisions. *Receivers* are defined as running executors who will be oversubscribed by tasks from *givers* for the remaining iterations to fully use resource slack. The task assignment results (as shown in line 7) and related U_j will be updated repeatedly until all the tasks have been scheduled.



(a) CPU Usage (b) Memory Usage

Fig. 4. An example of dominant factor from the view of CPU and memory.

Unfortunately, BFDP algorithm typically focuses on solving one-dimensional packing problem. To solve our problem for multiple resources, we transform the multi-dimensional bin packing problem into the classic packing problem by using Dominant Factor Scheduling (DFS) approach. For instance, if we simply apply BFDP algorithm to pursue an optimal number of executors to fully utilize CPU cores, it may be too greedy and introduce high contention for other resources, such as memory. Intuitively, as shown in Figure 4, we use an example to explain how DFS works. There are three running executors, whose CPU usage are 30%, 30% and 10%, and memory usage are 40%, 45% and 20%, respectively. If we schedule the resources by applying BFDP based on the CPU usage, the optimal solution would be to pack the tasks from both executors B and C into A, resulting in that the estimated utilization of executor A will be around 70% in the following stages. However, from the perspective of memory, it is highly possible that the memory usage of executor A will be overloaded (105%), causing the tasks failure in the following stages, which is unacceptable. In this case, CPU is actually a soft constraint (task is slower than expected with higher resource usage), but memory should be a hard constraint (task will fail due to Out of Memory). Thus, it is better to only pack the tasks from underutilized C into B to fully utilize the resources slack, avoiding the potential task failure. Correspondingly, B and C will be the *receivers* and *givers*, respectively. In iSpark, the dominant factor is defined as the bottleneck resource when we pack the task sets into executors, which ensures that our reallocation will not cause task failures. As shown in Line 5 of Algorithm 1 , we use the maximal utilization as the DFS for BFDP. Here, we define maximal utilization as maximizing $\langle U_i^c(t),\ U_i^m(t)\rangle$, where $U_i^c(t)$ and $U_i^m(t)$ are the used memory and CPU at time t.

3) Resource Reprovisioning: Another concern is reclaiming resource slack aggressively (i.e., under-provisioning) may cause severe resource contention (with possible task failures). In particular, CPU provisioning can only tolerate over 100% utilization in a short bursty duration and should be avoided. Thus, the resource reprovisioning is necessary when the average resource utilization keeps higher than h times

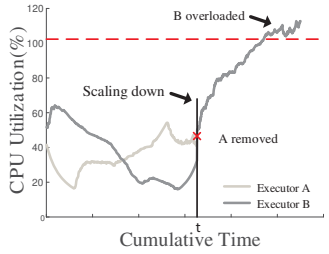


Fig. 5. An overprovisioning example to motivate the reprovisioning operation.

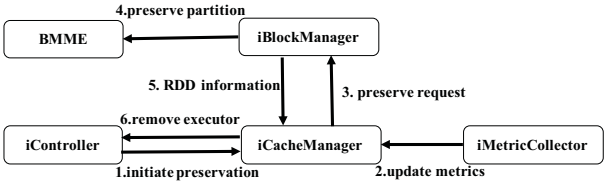


Fig. 6. Overview of RDD preservation workflow in iSpark.

of the expected upper threshold for one monitoring epoch. For example, as shown in Figure 5, the CPU usages of both executor A and B are underutilized before time t. However, the scaling down policy leads the executor overloaded after the executor A is removed directly at time t. To avoid these cases, we set h to 1.1 for CPU usage specifically, which is experimentally determined to balance application performance and resource utilization. The detailed sensitivity analysis is discussed in Section IV-D. In each provisioning interval, we update the threshold $\min\ \{\theta,\ U_{max}\}$ based on the latest information. To prevent disastrous task failures caused by resource demand variation, the reprovisioning action is required if any executor's actual utilization is larger than $h{\times}\min\ \{\theta,\ U_{max}\}$. Then iSpark requests one more executors by extending the existing scaling up policy. This heuristic allows iSpark to quickly correct resource provisioning decision to avoid overloaded workers.

C. Preempt Executors Gently

1) Data Consistency in iSpark: Due to the consistency semantics of RDD, if any intermediate data from previous iterations has not finished or been lost (due to removing running executors), Spark will not automatically use these cached RDDs for the following computations. Therefore, iSpark introduces a new automatic mechanism *iCacheManager* to entail that some RDD partitions need to be preserved when the provisioning is triggered. It leverages the DAG information generated by the task scheduler for selecting preservation candidates. After the preservation, *iCacheManager* will inform the DAG scheduler to update RDD information. With this functionality, iSpark can safely implement the resource provisioning decisions while avoiding additional and unnecessary re-computation overhead.

The key idea of keeping the data consistency is to preserve the intermediate results in memory before removing a running

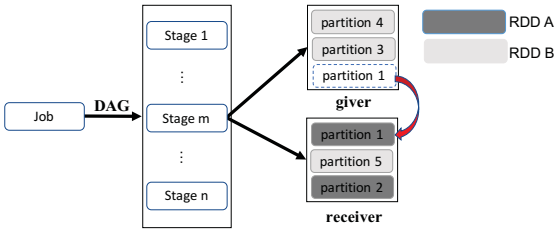


Fig. 7. An illustration of VIP policy: when stage m is finished, the future stages have dependency on RDD A while no dependency on RDD B. So partition 1 is prioritized to be preserved from *giver* to *receiver*.

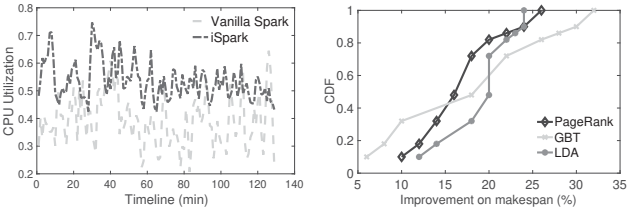
executor, i.e, to preempt a running processes safely. The detailed procedure is illustrated in Figure 6. Once a $remove$ request is initiated by *iController*, *iCacheManager* creates a new RPC (Remote Procedure Call) thread, which holds a list of *givers* that are being removed. Then *iCacheManager* collects DAG information and instantiates a new distributed component *iBlockManager* on that *givers* correspondingly. *iBlockManager* will send a request to BlockManagerMasterEndpoint (BMME) to preserve the prioritized RDD partitions. So these partitions can be preserved to *receivers* according to the scheduling decision. *iBlockManager* then responds whether there are more RDD blocks to be preserved. If all RDD partitions have been preserved, *iCacheManager* will inform *iController* that the executor can be removed safely. EAM will further release that executor and update the RDD information to DAG scheduler and task scheduler. If *iBlockManager* finds that there are more data partitions, *iBlockManager* will keep preserving process.

2) RDD Preservation: Once the resource provisioning is determined, task scheduler is performed to assign individual tasks to each executor. To make our proposed architecture compatible with default task scheduling policies, we let *iBlockManager* to preserve the RDD partitions from the executors and avoid data inconsistency. Considering the limited memory space, iSpark leverages both the metrics from *Monitor* and the DAG information generated by task scheduler for selecting the best candidates. Here we introduce a DAG-aware preservation policy to utilize memory efficiently.

Spark typically divides a job into several stages based on RDD dependencies, and submits the stages one by one. Each stage has a group of tasks that concurrently execute the computation. These tasks are generated based on different localities of RDD partitions. In iSpark, *iCacheManager* obtains RDD dependencies via the metrics from *iMetricsCollector*. *iBlockManager* then associates this information to support partition-level preservation. Thus, in each stage of a job, we have a collection of tasks with their dependent RDDs and refer these partitions as $\underline{VIP}$ ($\underline{V}$ery $\underline{I}$mportant $\underline{P}$artitions). When implementing a *remove* request, *iBlockManager* firstly scans the current RDDs cached in memory on the executors to be removed. As shown in Figure 7, it will prioritize $\underline{VIP}$ partition 1 to be preserved from *giver* to *receiver* according to the demand of future stages. If there is no more available $\underline{VIP}$, iSpark preserves the RDDs with highest partition ID,

TABLE I
WORKLOADS USED IN EVALUATION.

Workload	PageRank	GBT	LDA
Input data size(GB)	22.8	32.4	25.6
Fraction	40%	30%	30%



(a) Cluster CPU utilization comparison between iSpark and Vanilla Spark (b) The CDFs of improvement on makespan of iSpark vs. Vanilla Spark

Fig. 8. Performance effectiveness on multiple applications.

which is based on the observation that Spark assigns tasks according to the partition ID in an ascending order. The goal of this policy is to utilize RDD dependency information to achieve better memory utilization and avoid additional I/O overhead. The second scenario happens in *receivers* when the cache storage is full or has not sufficient space. To remedy this, we have to utilize limited storage memory space to hold the incoming partitions. iSpark evicts partitions whose dependent stages have finished before spilling others. As shown in this example, partition 5 of RDD B will be evicted, i.e., spilled to hard disk by default. Then partition 1, which belongs to RDD A, will be moved to *receiver* to ensure the data consistency.

D. Implementation

iMetricCollector: We use two lightweight system-level tools *psutil* and *MemoryMXBean* in Linux to implement the monitor. The monitor at each worker communicates with the $iMetricCollector$ at the master node via RPC. The data points for training the models are gathered from the class *shuffleReadMetrics* in *TaskMetrics* file.

iController We modify *maxNumExecutorsNeeded()* function to implement the provisioning strategy. For the preemption, we add a function in *ExecutorAllocationClient.scala* file to mark executors to be shut down. We invoke the API *doKillExecutors* to preempt specific executors.

iCacheManger: We add a new file *iCacheManager.scala* to implement the component *iCacheManager*. The API *prePartition()* will request *iBlockManager* to preserve RDD partitions based on the source and destination list provided by *iController*. While preserving the RDD partitions, we implement *checkMem()* to check if there is enough memory space on destination executors.

iBlockManager: We add a new class *preemptID* to represent the executors to be preempted, which can not accept any partition from others. We add a new API *preserveRDDBlock()* in *BlockManagerMasterEndpoint.scala* files to implement the preservation process.

IV. EVALUATION

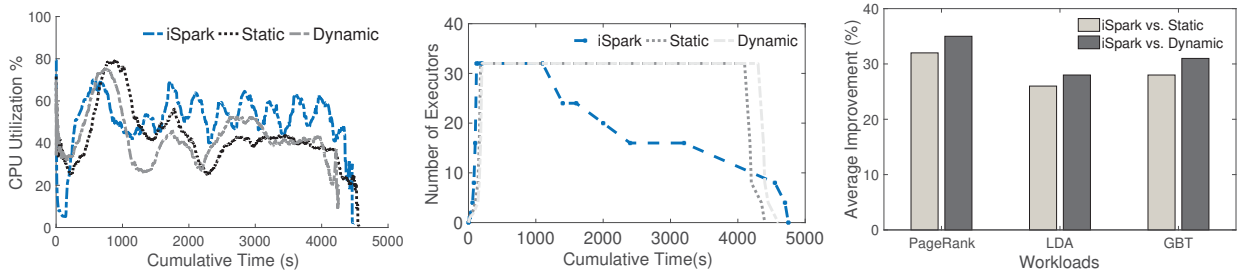
A. Experiment Setup

The evaluation testbed consists of 9 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR3 RAM, running Ubuntu 16.04 LTS operating system with kernel version 4.0, Scala 2.10.0, and Hadoop YARN 2.8.0 for cluster management. One node serves as the *master*, and all the other 8 nodes serve as *slaves*. These nodes are connected with Gigabit Ethernet. Each executor is configured with a bunch of $\{1$ core, 2G RAM$\}$ resources. To test our prototype, we use three workloads from the HiBench big data benchmarking suite [8], which are commonly used in existing work. For each workload we use different input data sizes. In each experiment run, we sequentially submitted a total number of 120 applications, including PageRank, GBT and LDA, with 22.8, 32.4 and 25.6 G input data set, as shown in Table I, whose arrival follows a Poisson process with mean inter-arrival time of 5 seconds [13]. We compare iSpark with two existing resource scheduling strategies which are provided by Vanilla Spark. (1) Static allocation statically reserves CPU and memory for each executor according to the peak demand, and launches a fixed number of executors according to user configuration. (2) Dynamic allocation is a built-in policy, which can only scale up the number of executors based on the workload as much as possible. For these policies, we configure the same number of CPU cores and memory size for each executor.

B. Analysis on Multiple Applications

Figure 8(a) shows the resource utilization achieved by Vanilla Spark and iSpark in terms of accumulative cluster usage, which is an averaged value of all the running workers. For fairness, the default dynamic allocation policy is enabled in Vanilla Spark. The results show that iSpark achieves apparent CPU usage increment, with the average CPU utilization increasing from 32.1% to 51.3%. For Vanilla Spark, the overall utilization is relatively low from 40th to 80th minutes, which is mainly caused by starvation. Vanilla Spark requests resource based on the peak demand, but it steps into trough stages after the short duration of peak usage and leads low utilization. In this case, the running application already acquires all desired resources from the cluster manager, which prevents or delays other applications to share the cluster resources. Compared with Vanilla Spark, iSpark can release underutilized resource in time, which allows more applications running concurrently in the cluster to minimize the fragmentation.

We further compared the makespan between Vanilla Spark and iSpark, and the improvement of makespan is shown in Figure 8(b). Here, makespan is defined as the time elapsed between application submission and completion. Figure 8(b) shows that the median reduction of PageRank, GBT and LDA on makespan is around 20.8%. The improvement on the makespan results from that iSpark can effectively reduce the waiting times of those applications pending in the submission



(a) Utilization traces of LDA (b) Number of executors when running LDA (c) CPU utilization comparison

Fig. 9. Real-time traces and effectiveness analysis on a single application.

queue. Peak demand based allocation policy in Vanilla Spark hurts the makespan of applications due to lower cluster utilization, and it keeps other applications in the waiting queue for longer time. Compared with Vanilla Spark, iSpark can make the underutilized resources available for pending applications, resulting in shorter makespan.

C. Analysis on a Single Application

To further evaluate the effectiveness of the proposed provisioning policy, we specifically investigate the real-time traces of CPU utilization and number of executor during running LDA workload by profiling the execution log files. We measure and plot the averaged CPU utilization of all the running executors as shown in Figure 9(a). Here, we define the CPU utilization for a single application as $\frac{actual\ CPU\ Time}{reserved\ CPU\ Time} \times 100\%$. We compare iSpark with Vanilla Spark under the default static and dynamic allocation policies. The results in Figure 9(a) show that the average utilizations of the default static and dynamic allocation are 43.2% and 41.5% respectively. Figure 9(b) further verifies our previous investigation in case study that these two policies in Vanilla Spark are both peak demand based. They over-allocate CPU cores and can not reclaim the under-utilized resources even though the utilization decreases obviously after 900th second in the job duration. The average CPU usage is lower than half of the peak usage and the related duration is nearly 3 times of the peak duration, resulting the severe resource wastage. However, compared with Vanilla Spark, there are less resource usage fluctuations in iSpark, whose average utilization is 58.4%. This is because iSpark can reduce the reserved CPU resource when the utilization decreases. We notice that the job execution time (neglecting the waiting time in queue) is negligibly delayed by 3% for PageRank application, which are mainly from the potential resource contention for higher utilization and overhead from the data preservation. Figure 9(b) also shows that iSpark can grant sufficient resources earlier than the other two schemes, which benefits from the polynomial regression model at the beginning stage. Figure 9(c) further shows the average utilization improvement of PageRank, GBT and LDA. Here we define average improvement as $\frac{average\ load - baseline}{baseline} \times 100\%$. Compared with static allocation, the overall resource utilization improvement by iSpark are 32.3%, 26.1% and 28.6% for PageRank, GBT and LDA, respectively. Indeed, the dynamic allocation is lower efficient than the static allocation as it starves for more resources at the beginning stages. Compared to the dynamic allocation, the overall resource utilization by iSpark achieves higher improvements, i.e., 35.2%, 28.3% and 31.8% for PageRank, GBT and LDA, respectively.

TABLE II
IMPACT OF DOMINANT FACTOR.

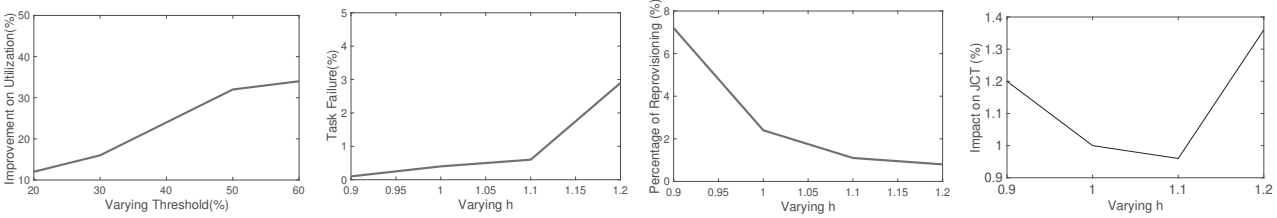
DF vs. CF	average	median	highest
PageRank	4.2%	4.8%	9.7%
GBT	3.3%	5.2%	8.4%
LDA	4.7%	6.3%	11.2%

D. Sensitivity Analysis

To evaluate sensitivity of parameter γ and h, we use the PageRank application as workload, and vary γ from 20% to 60% and h from 0.9 to 1.2. Figure 10(a) captures the effectiveness impacts by varying γ. Here the baseline is Vanilla Spark with default dynamic allocation policy. The figure shows that the utilization improvement marginally increases with the higher h. The improvement on utilization is 13%, 17%, 24%, 32% and 34% when we set the γ as 20%, 30%, 40%, 50% and 60%. Thus, we recommend that the γ should be 50% in case of variability of workloads and frequent resource reallocation. Figure 10(b) and 10(c) show the sensitivity of the parameter h. When h is set to be 0.9, 1.0 and 1.1, the corresponding task failure is less than 1.0%. However, when we increase h to 1.2, task failure grows to 2.9% because of resource under-provision. We also evaluate the impact of h in terms of Job Completion Time (JCT). Here the baseline is h=1. We measure the value $\frac{actual\ JCT}{baseline}$ and plot it in Figure 10(d). When h equals 1.2, the job execution is delayed to 1.3 times of baseline, which is mainly caused by task failure. When we drop h to 0.9, the job execution is also delayed, which is mainly caused by a higher percentage of executors involving in resource reprovisoning 10(c), which takes a large portion of time to request new executors [14].

E. Effectiveness of Dominant Factor

To evaluate the effectiveness of dominant factor, we compare the job completion time between iSpark with dominant



(a) Average utilization. (b) Percentage of task failures (c) Percentage of reprovisioning (d) Impact on JCT

Fig. 10. Sensitivity analysis on varying threshold γ and parameter h.

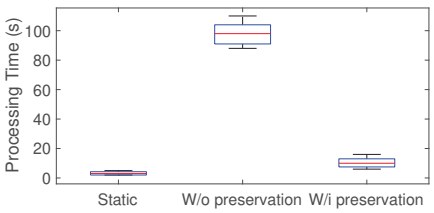
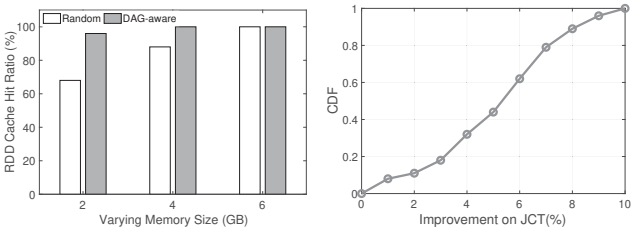


Fig. 11. Evaluation on overhead and effectiveness of preservation.



(a) Impact on Cache Hit Ratio (b) Impact on JCT

Fig. 12. Effectiveness of DAG-Aware Policy.

factor (DF) and iSpark with CPU factor only (CF). Here, CF means it only takes each executor's CPU capacity into consideration when we apply the BFDP algorithm to minimize the number of executors. Table II shows that dominant factor scheduling can achieve the average and median improvement on Job Completion Time (JCT) at 4.2% and 4.8% for PageRank, 3.3% and 5.2% for GBT, 4.7% and 6.3% for LDA, respectively. Compared with the dominant factor, the CPU factor approach reclaims the underutilized resources much more aggressively, which neglects the possible bottleneck by memory resource. It's aggressiveness invokes more resource reprovisioning to make up the possible under-provision, which takes a noticeable amount of time to request and ramp up new executors and degrades the job performance correspondingly.

F. Overhead and Effectiveness of Preemption

To evaluate the effectiveness of executor preemption, we run two wordcount applications in an interleaved way and only evaluate the action phase, in which the result stage needs to perform operations on cached intermediate data. Each application starts with 8 executors initially. We elastically change the number of executors between 4 and 32. Here, we set the processing period for each application to 1s, which means that every second only one application can be executed. Then the CPU utilization of executors occupied by another application is 0. Under the dynamic allocation modes, the executor number will be reduced to 4 when CPU usage is 0. Figure 11 shows that elastic provisioning with disabled *iCacheManager* is much worse than the other two modes in terms of processing time. The reason is that some intermediate data results have been lost, which needs averagely 88 seconds to recompute these intermediate data. However, iSpark with preservation component only accounts 3 to 5 seconds compared to the static policy, which demonstrates that the proposed component can ensure the data consistency with minor preservation overhead.

G. Effectiveness of DAG-aware Policy

To evaluate the effectiveness of DAG-aware block preservation policy, we run SVM application with 32G input dataset. We compare the performance achieved by iSpark with DAG-aware policy and iSpark with random selection policy separately. We keep the fraction of storage memory as default value, i.e., 0.6, but vary the configured memory size of individual executors from 2G to 6G. As shown in Figure 12(a), DAG-aware policy performs better than the random selection policy in terms of cache hit ratio. The cache hit ratio of random selection policy is 68% and 84% when the memory size is 2G and 4G respectively. In contrast, the proposed DAG-aware policy can improve the cache hit ratio to 92% and 98%, respectively. This is due to the fact that the DAG-aware policy can reserve the limited memory space for the data with higher priority with the help from RDD dependency information. There is no apparent difference while given sufficient memory space. The reason is that there is enough memory to cache all the intermediate data. We further evaluate the effectiveness of DAG-aware policy in terms of JCT when the memory of executor is configured to 2G. Figure 12(b) shows the reduction on JCT achieved by iSpark with DAG-aware policy is 5.2% compared to random selection policy. This result shows that DAG-aware policy can efficiently utilize the limited memory space when preserving the RDD partitions.

V. Related Work

Many popular cluster schedulers primarily focus on resource fairness [7], cluster utilization [12] or resource reservations [17]. Recently, capacity scheduling [1] is proposed to ful-

fill an efficient quota-based resource sharing among multiple jobs. The objective is the enforcement of scheduling invariants for heterogeneous applications, with policing/security utilized to prevent excessive resource occupation. Although these works achieve higher cluster utilization in practice, they cannot cope well with Spark executors which execute the entire DAG of applications. Furthermore, most of these works only focus on initial allocation of resources, which cannot dynamically provision resources at runtime for those applications with time-varying demands.

There are some recent works to optimize the scheduling of long-running processes, i.e. executors. Prophet [18] performs executor scheduling to fully match the demand of applications with the available resources. Morpheus [10] automatically assigns resources for jobs based on the historical data to meet application's demand. MEDEA [6] targets at making good placement decisions of long-running processes. However, both Prophet and Morpheus cannot dynamically adjust the computing resource amounts at runtime. They do not cope well with Spark since they may require additional launching and computational cost if any reallocation happens. MEDEA works well for task placement but does not consider the possible time-varying resource usage of applications during runtime. Our work differs from these researches in that we dynamically provision the executors of Spark without additional launching and computational cost if any reallocation happens.

In particular, Elasecutor [11] and PAF [3] focus the dynamic resource demands of workloads. Elasecutor elastically resizes executors to avoid over-allocation, and places executors strategically to minimize resource fragmentation. However, it may not guarantee the original submission order of jobs and further affect the fairness among jobs. Considering the resource elasticity, PAF starts with the fair allocation policy and then judiciously adjusts it by iteratively transferring resources from one job to another, so as to improve resource utility. However, it cannot cope with the diminishing resource demand within each application. SLAQ [22] is another quality-driven scheduling system designed for large-scale iterative training jobs. The scheduler automatically predicts the resource consumption and adjusts allocation to achieve better quality. It aims to improve the throughput and speed up the convergence of a single job. In contrast, iSpark aims to improve the performance of multiple iterative applications and the overall cluster utilization.

VI. Conclusion

In this paper, we focus on iterative workloads running on Apache Spark with diminishing resource demands during the job duration. We identify the causes of resource underutilization due to inflexible resource policy. To this end, we propose a flexible utilization aware executor provisioning approach to elastically adjust the allocated executors according to the real-time resource usage. iSpark preempts the underutilized executors and preserves the cached intermediate data from them to ensure the data consistency. Testbed evaluations demonstrate that iSpark improves the resource utilization of individual executors 35.2% compared to vanilla Spark. At the same time, it increases the cluster utilization from 32.1% to 51.3% and effectively reduces the overall job completion time by 20.8% for a set of representative iterative applications.

Acknowledgement

This work was supported by NSF grant CCF-1908843.

References

[1] Hadoop: Capacity scheduler. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[2] Spark dynamic allocation policy. https://spark.apache.org/docs/latest/job-scheduling.html.

[3] C. Chen, W. Wang, and B. Li. Performance-aware fair scheduling: Exploiting demand elasticity of data analytics jobs. *Proc. of IEEE INFOCOM*, 2018.

[4] Y. Cho, C. A. C. Guzman, and B. Egger. Maximizing system utilization via parallelism management for co-located parallel applications. In *Proc. of PACT*. ACM, 2018.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[6] P. Garefalakis, K. Karanasos, P. R. Pietzuch, A. Suresh, and S. Rao. Medea: scheduling of long running applications in shared production clusters. In *Proc. of ACM EuroSys*, 2018.

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of USENIX NSDI*, 2011.

[8] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proc. of IEEE ICDEW*, 2010.

[9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proc. of ACM SOSP*, 2009.

[10] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *Proc. of USENIX OSDI*, 2016.

[11] H. X. Libin Liu. Elasecutor: Elastic executor scheduling in data analytics systems. In *Proc. of ACM SoCC*, 2018.

[12] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proc. of SOSP*. ACM, 2017.

[13] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *Proc. of USENIX NSDI*, 2015.

[14] A. Pi, W. Chen, X. Zhou, and M. Ji. Profiling distributed systems in lightweight virtualized environments with logs and resource metrics. In *Proc. of IEEE IPDPS*, 2018.

[15] P. Pinyoanuntapong, M. Lee, and P. Wang. Delay-optimal traffic engineering through multi-agent reinforcement learning. In *2019 IEEE INFOCOM Workshop: NI 2019*, 2019.

[16] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of ACM SoCC*, 2012.

[17] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of ACM SoCC*, 2013.

[18] G. Xu, C.-Z. Xu, and S. Jiang. Prophet: Scheduling executors with time-varying resource demands on data-parallel computation frameworks. In *Proc. of IEEE ICAC*, 2016.

[19] D. Yang, W. Rang, and D. Cheng. Joint optimization of mapreduce scheduling and network policy in hierarchical clouds. In *Proc. of ICPP*. ACM, 2018.

[20] M. Yue. A simple proof of the inequality ffd (l) 11/9 opt (l)+ 1, l for the ffd bin-packing algorithm. *Acta mathematicae applicatae sinica*, 1991.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *Proc. of USENIX HotCloud*, 2010.

[22] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proc. of ACM SoCC*, 2017.