Cross-Platform Resource Scheduling for Spark and MapReduce on YARN

Dazhao Cheng, Xiaobo Zhou, *Senior Member, IEEE*, Palden Lama, Jun Wu, and Changjun Jiang

Abstract—While MapReduce is inherently designed for batch and high throughput processing workloads, there is an increasing demand for non-batch processes on big data, e.g., interactive jobs, real-time queries, and stream computations. Emerging Apache Spark fills in this gap, which can run on an established Hadoop cluster and take advantages of existing HDFS. As a result, the deployment model of Spark-on-YARN is widely applied by many industry leaders. However, we identify three key challenges to deploy Spark on YARN, inflexible reservation-based resource management, inter-task dependency blind scheduling, and the locality interference between Spark and MapReduce applications. The three challenges cause inefficient resource utilization and significant performance deterioration. We propose and develop a cross-platform resource scheduling middleware, *iKayak*, which aims to improve the resource utilization and application performance in multi-tenant Spark-on-YARN clusters. iKayak relies on three key mechanisms: reservation-aware executor placement to avoid long waiting for resource reservation, dependency-aware resource adjustment to exploit under-utilized resource occupied by reduce tasks, and cross-platform locality-aware task assignment to coordinate locality competition between Spark and MapReduce applications. We implement iKayak in YARN. Experimental results on a testbed show that iKayak can achieve 50 percent performance improvement for Spark applications and 19 percent performance improvement for MapReduce applications, compared to two popular Spark-on-YARN deployment models, i.e., YARN-client model and YARN-cluster model.

Index Terms—Spark-on-YARN, resource scheduling, cross-platform, application performance, reservation-aware executor placement, dependency-aware resource adjustment, locality-aware task assignment

1 Introduction

In the past few years, MapReduce has revolutionized big data parallel and distributed processing. MapReduce has proven to be an effective platform to implement complex batch applications as diverse as sifting through system logs, running extract transform load operations, and computing web indexes. However, its one-pass computation model makes MapReduce a poor fit for low-latency applications and iterative computations, such as machine learning and graph algorithms. Recently, emerging Apache Spark [1] addresses such limitations by generalizing the MapReduce computation. Spark enables applications to reliably store the data in memory. Each Spark application has multiple processes, called *executors*, running on the cluster to load related data in memory on its behalf even when it is not running any job. It allows applications to avoid costly disk accesses, which is the key to the high performance of Spark.

- *D. Cheng is with the Department of Computer Science, University of North Carolina at Charlotte, Charlotte, NC 28223. E-mail: dazhao.cheng@uncc.edu.*
- *X. Zhou is with the Department of Computer Science, University of Colorado, Colorado Springs, CO 80918. E-mail: xzhou@uccs.edu.*
- *P. Lama is with the Department of Computer Science, University of Texas at San Antonio, 1 UTSA Circle, San Antonio, TX 78249. E-mail: palden.lama@utsa.edu.*
- *J. Wu and C. Jiang are with the Department of Computer Science & Technology, Tongji University, 1239 Siping Road, Shanghai 200092, China. E-mail: {wujun, cjjiang}@tongji.edu.cn.*

Manuscript received 24 June 2016; revised 20 Jan. 2017; accepted 1 Feb. 2017. Date of publication 14 Feb. 2017; date of current version 17 July 2017. Recommended for acceptance by M. Caccamo. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2017.2669964

Thus, many of today's big data deployments go beyond MapReduce by integrating Spark for interactive and streaming computations. However, dividing a unified resource pool into smaller pools for different applications would lead to inefficient utilization of resources. A single cluster manager with dynamic resource allocation may lead to better resource utilization. In particular, enterprises prefer to deploy the emerging Spark applications on their existing Hadoop clusters in order to leverage the established cluster, access to existing HDFS dataset, and take advantage of Hadoop's security environment. For example, eBay and Yahoo! [2] employ MapReduce to generate reports and answer historical queries, while deploying Spark at the same time to calculate key metrics in real-time. Hadoop YARN [3] is an emerging cross-platform cluster manager. It allows multiple computing platforms to co-exist and share resource on a single cluster and benefit from its fine-grained resource management scheme. However, we find that there exists a semantic gap between the reservation-based resource scheduling policy [4] of YARN and the dynamic need of Spark applications, which causes inefficient resource utilization and poor application performance. Specifically, Spark-on-YARN raises several key challenges as follows.

First, the reservation-based resource scheduling policy of YARN makes tasks with high resource demand very hard to obtain the required resource in time. Unfortunately, Spark is such kind of applications. Spark is based on the multi-thread programming model. This characteristic could lead to a scenario that a single executor of Spark occupies a large amount of resource at one time. Thus, an executor with high resource demand may have to wait a long time for the

resource reservation, leading to low resource utilization and poor performance. Even worse, starvation could happen for some jobs with very large resource demand (e.g., Spark streaming) when the required resource is to be reserved but cannot be satisfied in a long period of time.

Second, existing schedulers in YARN do not recognize the impact of dependency between map and reduce tasks. As the cluster resource is reserved and shared with Spark applications, MapReduce jobs that have already launched reduce tasks may not be able to have all their map tasks completed in time. Because the reduce tasks cannot execute their functions until all map tasks are completed, the launched reduce tasks will keep occupying the resource and waiting for the completion of the map tasks. Thus, it incurs low utilization of the resource that is allocated to the reduce tasks.

Third, both MapReduce and Spark applications try to place tasks or executors alongside their related HDFS blocks for locality awareness, while they need to negotiate with YARN for resource scheduling. In particular, jobs could exhibit poor data locality on the nodes that co-host Spark executors and MapReduce tasks. For example, when a count task is computed by a Spark application, the same count dataset could be needed by a MapReduce job. However, the MapReduce job may not be able to obtain the resource on the node with local dataset if the node is mostly occupied by the Spark executor. Thus, the locality interference due to the multi-tenant competition hurts performance of both Spark and MapReduce applications.

In this work, we propose and develop a cross-platform resource scheduling middleware, iKayak, that aims to improve the resource utilization and application performance in multi-tenant Spark-on-YARN clusters. iKayak relies on three key designs that leverage time-varying resource demands of different applications, inter-task dependency between map and reduce tasks, and cross-platform locality awareness to tackle the aforementioned challenges, respectively. We first design and develop a reservation-aware executor placement mechanism to select efficient hosting nodes for Spark executors to achieve shorter reservation time. It aims to satisfy the high resource demands of executors in a timely manner. We then design and develop a dependency-aware resource adjustment mechanism to adaptively control the resource underutilized by reduce tasks. It allows map tasks to preempt resource from the reduce tasks. We further design and develop a cross-platform task assignment mechanism to coordinate the locality awareness between MapReduce task assignment and Spark executor placement. It aims to increase local data access opportunities on the nodes that co-host Spark and MapReduce applications.

We implement iKayak on a 16-node Hadoop YARN cluster and evaluate its benefits using the Purdue MapReduce Benchmark Suite (PUMA) and "BigDataBench" benchmark with datasets collected from real applications. We compare the performance of iKayak with two popular Spark-on-YARN deployment models: YARN-client model and YARN-cluster model. Experimental results by running different workloads show that iKayak reduces the job completion time of Spark workloads by 50 and 30 percent compared to the two popular models, respectively. At the same time, it also reduces job completion time of MapReduce workloads by 14 and 19 percent while it increases the CPU utilization by 22 and 15 percent compared with the two models, respectively.

In the rest of paper, Section 2 gives case studies and motivations of Spark-on-YARN. Section 3 describes the design and development of iKayak. Section 4 gives implementation details. Section 5 presents the experimental results. Section 6 reviews related work. Section 7 concludes the paper.

2 Motivations

2.1 Background

Reservation-Based Scheduling in YARN. Compared to the first generation Hadoop, YARN adopts an fine-grained resource management, which means applications can configure their required resources like CPU and memory for individual tasks when submit jobs. If there are not sufficient resources available in the cluster as required, the *ResourceManager* will start to reserve resources on the selected node for the task. When other tasks on this node are completed, more available resources can be assigned to the upcoming task. Until the reservation is satisfied, the released resources on this node will not allow to be allocated to other applications. During the reservation process, the new released resources are accumulated but not allowed to use, which provides a good resource isolation capability but causes ineffective resource utilization.

In-Memory Computing with Spark. Spark is an up-and-coming big-data analytics solution developed by using highly efficient in-memory computing. It allows applications to explicitly cache a dataset in memory so that applications can access data from memory instead of disk, which can dramatically improve the performance. Compared to MapReduce, Spark utilizes multiple threads instead of multiple processes to achieve parallelism on a single node, avoiding the memory overhead of several JVMs but leading individual executors occupying a large amount of resources at one time. The resource demand (i.e., the number of executors and the resource request of each executor) of a specific Spark application should be configured by users when the job is submitted to the cluster. Spark can run in two popular modes: Spark-on-Mesos and Spark-on-YARN. Spark-on-Mesos [5] adopts the "all-or-nothing" resource management policy (e.g., allocating enough resource at one time or denying the request), which could cause starvations especially when the cluster is busy. However, YARN will reserve the resource instead of denying application to avoid such starvation.

Spark-on-YARN. In Spark-on-YARN deployment mode, Spark applications are similar to MapReduce "jobs". As shown in Fig. 1, MapReduce runs each task in its own process. When a task completes, the process goes away. In Spark, many tasks can run concurrently in a single process (i.e., executor), and this process sticks around for the lifetime of the Spark application, even when no jobs are running. Thus, MapReduce does not suffer from the reservation-based scheduling policy since the reservation periods of most map/reduce tasks are very short due to their small resource demands. However, Spark applications with huge resource demands of individual *containers* can suffer from too long resource reservation period under the current resource scheduling policy of YARN.

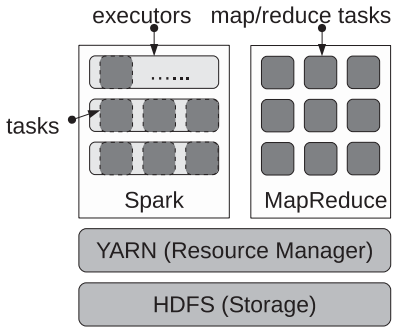


Fig. 1. [Spark+MapReduce]-on-YARN.

2.2 Case Study and Challenges

Although Spark-on-YARN provides good integration with YARN's cluster-wide resource management policies, it incurs significant challenges due to the different characteristics of the two programming modes. To illustrate the inefficiency caused by the hybrid deployment, we conduct a case study on a (Spark+MapReduce)-on-YARN cluster composed of six machines. We configured each slave node with 8 cores and 12 GB Memory, and the block size is set to 256 MB in the experiment. In the experiment, a representative MapReduce application from the PUMA benchmark [6], i.e., WordCount, is executed with 30 GB input data. Another representative Spark application, i.e., Logistic Regression [7], is executed with 10 GB input data from Wikipedia. The resource requirements of map and reduce tasks are configured to 2 cores and 2 GB memory. The resource requirement of individual executors is configured to 6 cores and 8 GB memory. We measured the different task completion times for various applications and observed the resource utilizations on the hybrid platform as follows.

Ineffective Resource Utilization. Fig. 2 shows the details of the resource scheduling diagram on a slave node during a short time window. When a Spark executor request is submitted at 20th second, *ResourceManager* finds there is no enough available resource on the node. Then *ResourceManager* starts to reserve resources for the submitted Spark application and reject to assign other MapReduce tasks to the selected node. Fig. 2 shows there exists a large amount of idle resource after Map 1 and Map 2 are completed. This is because the resource (i.e., 4 cores) released by Map 1 and Map 2 is less than the demand of the executor (i.e., 6 cores). The executor eventually starts to be executed on the node until Reduce 2 completed, when the resource accumulation on the node satisfies the demand of the Spark executor.

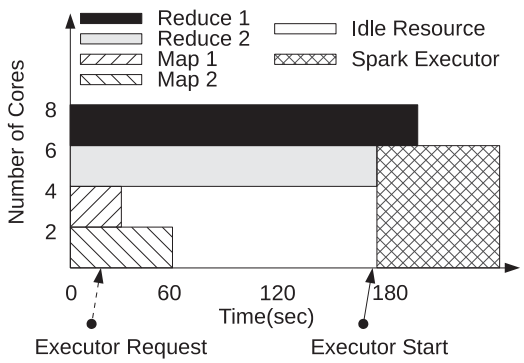


Fig. 2. Ineffective resource utilization.

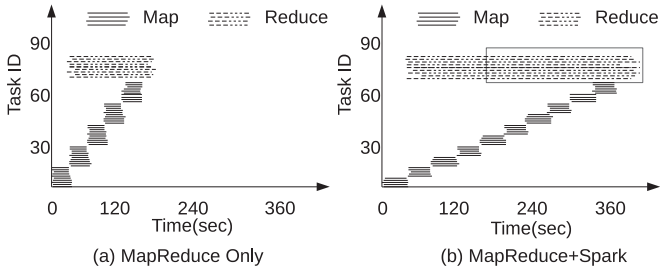


Fig. 3. Underutilized reduce tasks.

During this resource scheduling process, we find the reservation-based resource allocation policy is apparently ineffective and causes low resource utilization, leading poor performance of Spark applications as well. Such ineffective resource scheduling could be even worse if there are straggler tasks appeared on the selected nodes which typically occupy the resource for a long time period.

Underutilized Reduce Tasks. In the second case, we first run the MapReduce application solely and then run the MapReduce and Spark applications together on the cluster. Fig. 3 compares the tasks execution differences of MapReduce application under the two different scenarios. It shows that the mix deployment significantly increases the number of map waves for the MapReduce application (i.e., from 5 to 10 waves) and delays the reduce tasks (i.e., from 120 to 300 s). This is due to the fact that half of the cluster resource is allocated for the Spark application compared to the sole MapReduce deployment. In this case, we find the time of resource occupied by the reduce tasks is almost three times of the time spent in the sole MapReduce deployment while they complete the same task. Apparently, the resource reserved for reduce tasks in the hybrid environment is over-reserved than its need, leading significant under-utilization of reduce tasks. Such phenomenon could be even worse when the reduce tasks are configured to complete in multiple waves.

Locality-Missing Assignment. We further observed the locality awareness of the task assignment in both MapReduce and Spark executions. We find that the map tasks with data locality achieve 95 percent of the total completed map tasks when the MapReduce application runs solely in the dedicated cluster. However, the local map task rate is reduced to 68 percent when the MapReduce application is co-hosted with the Spark application. This is due to the fact that Spark executors typically occupy a large amount of resource on individual nodes, which significantly obstruct the locality-aware map task accesses for MapReduce application. For example, there are two nodes (i.e., M1 and M2) in the cluster and each node has 4 cores. Spark executor requires 3 cores and each map task requires 1 core. The data storage on M1 and M2 node is different with each other and then map tasks can be divided into two types, i.e., M1-local tasks and M2-local tasks. When the Spark executor occupies a big container (i.e., 3 cores) on the node M1, there is only 1 core for MapReduce task assignment. For map tasks, there are more free resources (i.e., 4 cores) on M2 node than that (i.e., 1 core) on M1 node. As MapReduce adopts a "pull-based" task assignment policy, some M1-local tasks are scheduled to M2 node. In this case, M1-local tasks that are assigned on the M2 node miss the locality awareness due to the competition from the Spark application.

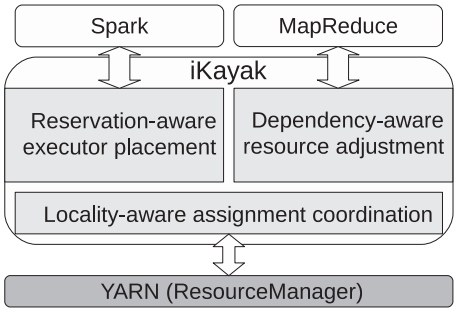


Fig. 4. Overview of iKayak.

HDFS usually has three data replicas in the cluster, which may improve the data locality. For example, there may exist another node having the same data file with M1 node, which provides the substitutes with locality for M1-local tasks. However, as the cluster size and the application quantity expand, such multi-replicas are still too limited and hard to avoid such locality-missing task assignment. In particular, both Spark and MapReduce apply Delay Scheduling [8] to schedule tasks based on the locality preference, i.e., they try to schedule tasks onto nodes with local data. So such locality awareness competition between Spark and MapReduce will make they both suffer from lack of locality awareness even there are multiple replicas in the cluster.

2.3 Opportunities

The new feature (i.e., reservation-based resource scheduling) of YARN fits well to deal with the small size MapReduce tasks due to its fine-grained resource management capability. However, running Spark on YARN incurs significant performance deterioration (e.g., inefficient resource utilization, dependency-blind resource allocation and locality-missing task assignment) due to its special characteristic, i.e., huge resource demand of individual executors. While Spark-on-YARN poses the above challenges, it also opens up new opportunities. Intuitively, we find three strategies to take the aforementioned three challenges respectively.

- Spark executors should be placed on the suitable nodes to satisfy their resource reservations in time, and avoid too long time waiting and low resource utilization.
- The resources allocated to reduce tasks should be rebalanced based on their time-varying demands to avoid the resource wasting when any Spark applications is submitted into the cluster.
- The task assignment of both Spark and MapReduce applications should be optimized to reduce the locality-aware competition while considering the locality awareness of both Spark and MapReduce applications.

All these motivate us to develop a holistic resource scheduling approach to improve the cluster utilization and the application performance in Spark-on-YARN. In next section, we focus on exploiting these opportunities to optimize the resource scheduling for Spark-on-YARN.

3 iKayak Design

iKayak is a cross-platform resource scheduling middleware that aims to optimize the resource management in Spark-on-YARN clusters. It takes advantage of the different resource demand characteristics of Spark and MapReduce applications, and dynamically optimizes resource scheduling while considering the data locality for both applications. Fig. 4 shows the architecture of iKayak. It has three main components: reservation-aware executor placement, dependency-aware resource adjustment, and locality-aware task assignment coordination. We briefly describe their major features.

- *Reservation-aware executor placement* adaptively places Spark executors on efficient hosting nodes that can satisfy high resource demand of individual executors in a timely manner.
- *Dependency-aware resource adjustment* dynamically exploits the resource allocated to reduce tasks to mitigate low resource utilization due to resource over provisioning, especially when reduce tasks are idle and waiting for the intermediate data from map tasks.
- *Locality-aware task assignment coordination* improves the data locality awareness of both Spark and MapReduce applications so that they can share the limited local data access opportunities on the multi-tenant cluster nodes.

3.1 Reservation-Aware Executor Placement

When a Spark job is submitted to the cluster, the placement of executors is crucial to the application performance and cluster utilization. We propose and develop a reservation-aware executor placement mechanism to mitigate the inefficient resource utilization issue due to high resource demand of Spark. The key insight is that Spark executors should be placed on nodes that either have enough available resource or host small map tasks. Note that reduce tasks usually occupy resource for much longer time than map tasks. If executors have to wait for the released resource from reduce tasks, they should be placed on the nodes that host reduce tasks approaching completion.

Spark provides two default executor placement schedulers, i.e., *SpreadOut* and *Non-SpreadOut* [9], to place executors on multiple nodes in the cluster. Both of them have no capability to place executors in a resource reservation aware manner. When running Spark on YARN, each Spark executor runs in a YARN container. The Spark *ApplicationMaster* is responsible to negotiate resource requests with YARN and find a set of efficient hosts to place and run the executors. However, YARN does not have the capability to recognize efficient worker nodes for Spark applications because it was originally designed for MapReduce computations. MapReduce schedules a container and fires up a JVM for each task, while Spark hosts multiple tasks within the same container for multi-threading. Correspondingly, the resource demand of an individual container for Spark is much more than that of an individual task for MapReduce. So the default "pull" based task scheduling approach for MapReduce jobs does not fit well for the executor placement of Spark applications.

3.1.1 Resource Reservation Analysis

Resource reserved for Spark executors may come from three sources on a selected node n, i.e., available free resource

(R^n_{free}), running map tasks (R^n_{map}), and running reduce tasks (R^n_{reduce}). We define an efficient worker node for hosting a Spark application as the node that can timely satisfy the resource demand of individual executors of Spark. We analyze the characteristics of the three sources as follows.

- The free resource on the selected node can be used for executors immediately after reservation, which is the most efficient source for executor resource reservation.
- The resource from running map tasks can be used for executors in a very short time, i.e., several seconds or a couple of minutes. This is due to the fact that map tasks are light weight and generally small. They will release the occupied resource after completion.
- The resource from running reduce tasks can be used for executors in an uncertain time, which is dependent on the reduce task execution progress.

We develop an effective executor placement algorithm based on the resource reservation awareness of each possible destination node in the cluster. The algorithm aims to make the executor run as soon as possible once it is placed on the selected worker node. As shown in Algorithm 1, it first selects the efficient worker node with enough free resource to satisfy the demand of individual executors. If there is not enough free resource on the node, iKayak selects the efficient worker node that hosts enough running map tasks to place executors. If there is still not enough resource, iKayak finally selects a node that hosts reduce tasks approaching completion.

Algorithm 1. Reservation-Aware Executor Placement

```
repeat
  if Any Spark executor requests resource: R_executor then
    Evaluate R^n_free, R^n_map, R^n_reduce, n ∈ N
    if R^n_free ≥ R_executor then
      best = arg max_n[R^n_free]
      Select the node with maximum free resource
    end if
    if R^n_free < R_executor ≤ [R^n_free + R^n_map] then
      best = arg max_n[R^n_free + R^n_map]
      Select the node with maximum map tasks
    end if
    if [R^n_free + R^n_map] < R_executor then
      Select the node with minimum waiting time
    end if
    Place executor on the selected node
  end if
until Satisfy the demand of Spark
```

The first two scenarios are straight forward so that we focus on analysis of the third scenario. When most MapReduce workloads in the cluster are reduce-heavy workloads, Spark executors may have to wait the resource released by reduce tasks to satisfy their demands. In this scenario, iKayak aims to find the destination nodes with the reduce tasks approaching completion so that the Spark application does not have to wait for long time. Spark executor is not simply assigned to the first resource slot freed by reduce tasks since it is no guarantee that other reduce tasks on this machine can complete in a short time. In order to to find which reduce tasks are approaching completion, it is necessary to predict the execution progress and remaining time of reduce tasks.

3.1.2 Estimating Reduce Remaining Time

Although there are many existing prediction approaches for MapReduce execution, we develop a self-adaptive fuzzy model to predict a reduce task's remaining execution time based on its input size and resource allocation. The fuzzy model is often used to capture the complex relationship between resource allocations and a task's fine-grained execution progress. However, a task's progress can be affected by many factors. First, reduce task progress is not uniform at different execution phases, e.g., shuffle, sort and reduce phases. Second, even within the same phase, data skew among tasks leads to different task execution speed at different intervals. Finally, co-running tasks may unpredictably interfere with a task's execution, making the mapping of resource to task progress variable. Many existing prediction approaches do not consider the multi-tenant interferences from other co-existing applications. Therefore, we design an online self-adaptive fuzzy model based on real-time measurements.

Fuzzy Model. The reduce task remaining execution time (i.e., $y(t)$) in the control interval t is represented as the input-output NARX type (Nonlinear Auto Regressive model with eXogenous inputs),

$$y(t) = F(u(t), d, \xi(t)). \tag{1}$$

F is the relationship between the input variables and the output variable. The input variables are the current resource allocation $u(t)$, the job input size d, and the regression vector $\xi(t)$. Here, $u(t)$ represents the resource allocation for the reduce task. The regression vector $\xi(t)$ contains a number of lagged outputs and inputs of the previous control periods. It is represented as

$$\begin{aligned}\xi(t) = [&(y(t-1), y(t-2), \ldots, y(t-n_y)),\\ &(u(t), u(t-1), \ldots, u(t-n_u))]^T,\end{aligned} \tag{2}$$

where n_y and n_u are the number of lagged values for outputs and inputs, respectively. Let ρ denote the number of elements in the regression vector $\xi(t)$, that is,

$$\rho = n_y + n_u. \tag{3}$$

F is the rule-based fuzzy model that consists of Takagi-Sugeno rules [10]. A rule F_j is represented as

$$\begin{aligned}F_j : \text{IF } &\xi_1(t) \text{ is } \Omega_{j,1}, \xi_2(t) \text{ is } \Omega_{j,2}, \ldots, \text{ and } \xi_\rho(t) \text{ is } \Omega_{j,\rho}\\ &u(t) \text{ is } \Omega_{j,\rho+1} \text{ and } d_j \text{ is } \Omega_{j,\rho+2}\\ \text{THEN } &y_j(t) = \zeta_j \xi(t) + \eta_j u(t) + \omega_j d_j + \theta_j.\end{aligned} \tag{4}$$

Here, Ω_j is the antecedent fuzzy set of the jth rule, which is composed of a series of subsets: $\Omega_{j,1}, \Omega_{j,2}, \ldots, \Omega_{j,\rho+2}$. ζ_j, η_j and ω_j are parameters, and θ_j is the offset. Their values are obtained by offline training. Each fuzzy rule characterizes the nonlinear relationship between allocated resources and performance for a specific reduce task type.

Online Self-Learning. Due to the dynamics of MapReduce job behaviors (e.g., data skews, different phases and multi-tenant interferences), we design an online self-learning module to adapt the fuzzy model. It aims to minimize the prediction error of the fuzzy model $e(t)$, which is the error between actual measured job progress and predicted value.

If $e(t) \neq 0$, we apply a recursive least squares (RLS) method to adapt the parameters of the current fuzzy rule. The technique updates the model parameters as new measurements are sampled from the runtime system. It applies exponentially decaying weights on the sampled data so that higher weights are assigned to more recent observations.

We express the fuzzy model output in Eq. (1) as follow:

$$y(t) = \phi(t)X + e(t), \tag{5}$$

where $e(t)$ is the error between the actual output and predicted output. $\phi(t) = [\phi_1^T, \phi_2^T, .., \phi_\rho^T]$ is a vector composed of the model parameters. $X = [\sigma_1 X(t), \sigma_2 X(t), .., \sigma_\rho X(t)]$ where σ_j is the normalized degree of fulfillment or firing strength of jth rule and $X(t) = [\xi(t)^T, u(t)]$. The parameter vector $\phi(t)$ is estimated so that the error function in Eq. (6) is minimized. We apply both the current error $e(t)$ and the previous error $e(t-1)$ to estimate the parameter vector,

$$Error = e(t)^2 + \tau e(t-1)^2. \tag{6}$$

Here τ is called the discount factor as it gives higher weights on more recent samples in the optimization. It determines in what manner the current prediction error and old errors affect the update of parameter estimation.

3.1.3 Executor Placement with Reduce Tasks

Based on the estimated remaining time of reduce tasks on different nodes, we use Algorithm 2 to select the hosting node for executor placement. As shown in Algorithm 2, we assume that reduce tasks m, ($m \in M$), are running on nodes n, ($n \in N$). We first evaluate the remaining execution times (i.e., y_m^n) of reduce tasks on each node based on Eq. (1). We then sort the remaining execution times of reduce tasks within each node, and calculate the minimum accumulated resource (i.e., $\sum_{m=1}^{m=x} u_m^n$) from reduce tasks on the nth node to satisfy the executor's demand. We obtain the corresponding estimated waiting time ($Wait^n$) to accumulate the required resources from the nth node. Finally, the best candidate to place the executor is the node with the minimum waiting time.

Algorithm 2. Executor Placement with Reduce Tasks

```
if [R^n_free + R^n_map] < R_executor then
  repeat
    Evaluate y^n_m, n ∈ N, m ∈ M by Eq. (1)
    Sort remaining times: y^n_m, n ∈ N, m ∈ M
    /*y^n_1, ≤ ··· , ≤ y^n_x, ≤ ··· , ≤ y^n_M*/
    if Σ_{m=1}^{m=x} u^n_m ≥ [R_executor − (R^n_free + R^n_map)] then
      Wait^n = arg max_{m∈x} [y^n_m]
    end if
  until Obtain Wait^n, n ∈ N
end if
best = arg min_{n∈N} [Wait^n]
Place executor on the selected node
```

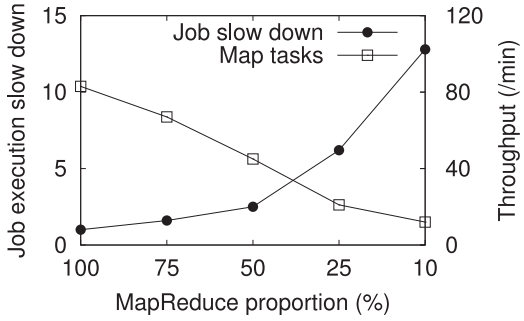


Fig. 5. Impact of workload proportion.

3.2 Dependency-Aware Resource Adjustment

We identify the dependency in Spark-on-YARN at two levels, i.e., application level and task level. At the application level, there exists apparent dependency, i.e., resource competition between Spark and MapReduce applications in the multi-tenant cluster. We measured various MapReduce job completion times and throughputs while giving different input data proportions between the Spark and MapReduce applications. Three MapReduce applications from the PUMA benchmark [6], i.e., WordCount, Terasort and Grep, with various input data sizes, were run on the cluster. Another three representative Spark applications from the BigDataBench [7] benchmark, i.e., K-means, PageRank and Index, with the input data sets from Wikipedia, were run on the cluster too. We quantified the slowdown of MapReduce job execution by comparing various job completion times to the job completion time achieved in the dedicated cluster with sufficient resources (i.e., 100 percent). Fig. 5 shows that MapReduce execution slows down as its workload proportion decreases in the cluster. It also shows that the throughput of completed map tasks is significantly lower as MapReduce workload decreases in the cluster. As a result, at the task level, the delayed map task executions further affect the resource demands of the reduce tasks due to the dependency between the map and reduce tasks. We develop a dependency-aware resource adjustment mechanism to dynamically control the resource allocated to reduce tasks.

We want to exploit the resource allocated to reduce tasks after a MapReduce job starts execution, when the cluster resource allocated to the MapReduce workload changes because of co-hosting Spark applications in the cluster. We define a delay factor $\lambda = \frac{R_{Spark}}{R_{Spark}+R_{MR}}$ to represent the delay of map task execution caused by the dynamic resource allocation in the multi-tenant cluster environment. Here, R_{Spark} and R_{MR} represent the resource allocations for Spark and MapReduce applications, respectively. The delay factor is event driven by application submissions.

We use JVM *suspend* function to suspend the execution of reduce tasks when their allocated resource is not fully utilized due to the delay of their map task execution. The length of suspension is dependent on the delay factor λ. Let T represent the control interval. The suspension length is calculated as $T_{suspend} = \lambda^v \times T$, where v is weight parameter. A larger λ means there is more resource occupied by Spark applications than that occupied by MapReduce applications so that the suspension time of reduce tasks should be longer. During the suspension period, reduce task executions are suspended and their CPU resource is released. Memory resource cannot

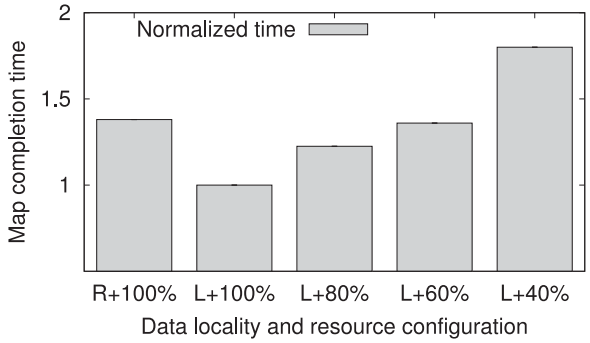


Fig. 6. Data locality and resource configuration.

be released immediately because the related data of the process is still kept in memory to avoid data loss when the reduce task is resumed. Note that the related data of the suspended reduce JVM can be swapped out by the operating system with overhead [11]. Thus, we use $T_{suspend} = [\lambda^v - \theta] \times T$ to determine the suspension length, where θ is a threshold value representing the suspension overhead.

When reduce tasks of a job are suspended, iKayak utilizes the released resource to assign map tasks of the same job to the node for improving resource utilization. First, more intermediate data produced by the map tasks can be used to accelerate reduce tasks, since reduce tasks need the intermediate data for execution. Second, many map tasks are CPU intensive so that the resource released by suspended reduce tasks can be utilized by map tasks more efficiently.

3.3 Locality-Aware Assignment Coordination

Both Spark and MapReduce applications prefer to assign executors and tasks together with their data blocks to achieve local data access. Spark applications are greedy of the data locality from two aspects. First, executors stick around for the lifetime of the application, even when no jobs are running. Second, unlike MapReduce tasks, individual Spark executors typically occupy a large amount of resource on the selected worker node. Correspondingly, there are often locality interference when co-hosting Spark executors and MapReduce tasks.

The performance of Spark degrades significantly if there is not enough memory to store the data of a job [1]. Thus, the resource configuration of Spark executors is more inflexible compared to that of MapReduce tasks. As map tasks are more sensitive of the data locality than reduce tasks, iKayak focuses on the locality coordination of map tasks for MapReduce workloads. When the local data access opportunity is limited for map tasks, `ResourceManager` has to make a choice between allowing remote data access and allocating less resource than its demand but with local data access. For example, there are 2 cpu cores and 2 GB memory available on the M1 node. The resource demand of individual MapReduce tasks is configured as 2 cpu cores and 2 GB memory. There are two M1 local map tasks have to be assigned at the moment. There are two potential solutions for task assignment. Solution a is to assign only one local map task on the node M1 while allowing another task to be assigned on the node M2 without data locality. Solution b is to shrink the resource demands of both map tasks to (1 cpu core, 1 GB memory) and then they both can be assigned on M1 node with local data access. Correspondingly, the containers for the two map tasks have to be proportionally shrank when they are initialized. However, shrinking the resource allocation for tasks definitely deteriorates performance.

We explore the performance impact of data locality and resource configuration for individual map tasks. Fig. 6 compares the map task completion time achieved with remote data access and that achieved with local data access and different resource configurations. In this case study, map tasks are configured with 2 cores and 2 GB memory resource demand (presented as $<2core, 2G>$). "R+100 percent" means map tasks rely on remote data access and have 100 percent resource, i.e., $<2core, 2G>$. "L+100 percent" means map tasks rely on local data access and have 100 percent resource, i.e., $<2core, 2G>$. It shows that map tasks that have local data access with 100 percent resource configuration have the shortest completion time. It also demonstrates that map tasks (i.e., L+80 percent) with less resource configuration than its demand but with local data access have better performance than map tasks (i.e., R+100 percent) that have full amount of needed resource but have to do remote data access.

To coordinate the locality awareness, we develop a per-task configuration approach [12] for iKayak that allocates different amount of resource to map tasks instead of maintaining the identical resource configuration of containers for all map tasks. As shown in Algorithm 3, iKayak periodically checks the resource status of each node by heartbeat connections. If there is any free resource available in a node, `ResourceManager` assigns map tasks with data locality to the node. Suppose there are $P_{requested}$ M-local map task assignments requested by `ApplicationManager`. The node M has R_{free} free resource and each task requests R_{map}^{demand} resource. At the beginning, iKayak accepts all map tasks. If there is not enough resource to satisfy all of the map tasks at the moment, iKayak gradually reduces the number of accepted tasks and shrinks the resource configuration of individual tasks. Eventually iKayak accepts $P_{accepted}$ M-local map tasks on the node M with the shrunk resource configuration R_{map}^{shrink}. To avoid performance deterioration, we set a shrinking ratio μ to prevent iKayak from shrinking the resource configuration of individual tasks too much. The shrinking ratio μ of map tasks is defined as $\mu = \frac{N_{local}}{N_{local}+N_{remote}}$, where N_{local} and N_{remote} represent the number of local and remote map task access respectively.

Algorithm 3. Map Task Shrinking Algorithm

```
1: repeat
2:   if Any free resource is available on node M^n then
3:     P_accepted = P_requested /*accepts all requested tasks*/
4:     if R^n_free < p × R^demand_map then
5:       repeat
6:         /*Shrink resource allocation of map tasks*/
7:         R^shrink_map = R^n_free / P_accepted
8:         Reduce the number of accepted map tasks.
9:       until R^shrink_map ≥ μ × R^demand_map
10:    end if
11:    Accept P_accepted M-local map tasks.
12:    Assign tasks with new configuration: R^shrink_map.
13:  end if
14: until All running jobs completed
```

TABLE 1
MapReduce Parameter Configuration

Parameter	*Wordcount*	*Grep*	*Terasort*
io.sort.factor	10	10	10
io.sort.mb	100 mb	200 mb	150 mb
io.sort.record.percent	0.35	0.25	0.15
io.sort.spill.percent	0.8	0.6	0.75
io.file.buffer.size	4 k	16 k	32 k
mapred.child.java.opts	200	300	250

4 Implementation

4.1 YARN Modification

We implemented iKayak by modifying classes `ResourceManager`, `ApplicationManager` and `LaunchTaskAction` based on Hadoop version 2.6.0. We added a new interface `exePlace`to implement the reservation-aware executor placement algorithm. When an application submits its register request to *ApplicationMaster* of Spark, it will call the method *exePlace* to allocate the resource for *Executors*. The method *exePlace* is responsible for allocating the available resource to applications, which is event-driven by any application submissions or cluster resource changes. Additionally, we added another new interface `taskRes`, which is used to specify the resource configuration of individual tasks while assigning them to slave nodes. Each executor placement and task resource allocation is tagged with its corresponding `AttemptTaskID`. During job execution, we created a method `taskAnalyzer` to collect the status of completed tasks by using `TaskCounter` and `TaskReport`. The resource utilizations and the execution times of tasks are reported by `TaskTrackers` via heartbeat connection periodically.

4.2 Experiment Setup

We evaluate iKayak on a cluster composed of 3 T110 (8-core CPUs and 16 GB RAM), 2 T420 (24-core CPUs and 32 GB RAM), 1 T320 (12-core CPUs and 24 GB RAM), 2 T620 (24-core CPUs and 32 GB RAM), and 8 Dell desktops (8-core CPUs and 16 GB RAM). Each machine has 1 TB hard disk. The master node is hosted on one Dell desktop in the cluster. The servers are connected with Gigabit Ethernet. The block size of HDFS is set to 256 MB and the number of replicas of HDFS is set to 3. iKayak is based on the version 2.6.0 of Hadoop implementation and 1.4.0 of Spark implementation. We implement iKayak on the *Master* node of the cluster. The suspension control interval (T) is set to 5 minutes, which is a trade-off between the map task and the reduce task completion time. The suspension parameter v and the overhead (θ) are empirically determined to be 1 and 0.15 in the experiment.

4.3 Real-World Workloads

To understand the effectiveness of iKayak in a production environment, we used a synthetic workload, "MicroSoft-Derived (MSD)", which models the production workload of 174,000 jobs in Microsoft datacenter in a single month in 2011 [13] as MapReduce applications. MSD mimics the distribution characteristics of the production jobs by running *Wordcount*, *Terasort* and *Grep* applications from the PUMA benchmark [6] with various input data sizes. It is a scaled-down version of the workload studied in [13] since our cluster is significantly smaller. We scale down the workload in two ways: we reduce the overall number of jobs to 87, and eliminate the largest 10 percent of jobs and the smallest 20 percent of jobs. We used a set of representative Spark applications from the "BigDataBench [7]" benchmark, i.e., K-means, PageRank and Index, with the input data sets from Wikipedia. We mixed MapReduce workload and Spark workload at a ratio of 1:1 in the experiments, which is based on the workload analysis from industry [2], [14].

TABLE 2
Resource Configurations of MapReduce

Task	*Wordcount*	*Grep*	*Terasort*
Map	1core, 1 GB	1core, 2 GB	1core, 2 GB
Reduce	1core, 1 GB	1core, 2 GB	1core, 2 GB

TABLE 3
Resource Configurations of Spark

K-means	*PageRank*	*Index*
4core, 4 GB	2core, 8 GB	4core, 6 GB

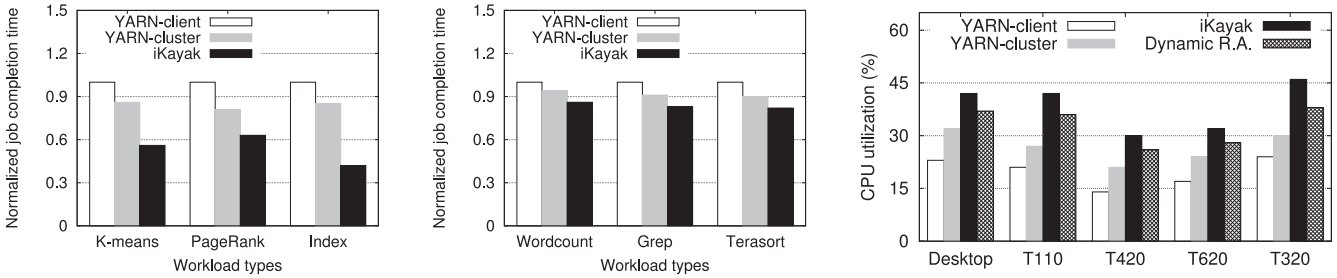
As shown in Table 1, we set the Hadoop configurations according to the rules recommended by Cloudera [15]. We used the same configurations for evaluating various approaches in the experiment. In YARN, there is no "slot" which is the building block in the old versions, and the system no longer distinguishes map and reduce tasks when allocating resources. Instead, each task specifies a resource request in the form of $<1core, 2GB>$ (i.e., requesting 1 cpu core and 2 GB memory), and it will be assigned to a node with sufficient capacity. We configured the resource demands of MapReduce workloads (as shown in Table 2) and Spark workloads (as shown in Table 3) based the previous studies [1], [16] and our experimental experiences.

5 Experimental Evaluation

5.1 Effectiveness of iKayak

We compare the performance of iKayak with two representative deployment models for running Spark on YARN clusters [17]: YARN-cluster model and YARN-client model.

Reducing Spark Job Completion Time. Fig. 7a shows iKayak significantly improves the job completion times of K-means, PageRank and Index workloads about 50 and 30 percent compared with YARN-client model and YARN-cluster model, respectively. Spark jobs perform better than MapReduce jobs due to two factors. First, as Spark is in-memory computing application, Spark jobs are more sensitive of the data locality than MapReduce jobs. The performance of Spark degrades significantly if there is not enough memory to store the data of a job. So our work focuses on shrinking the resource configurations of map tasks for MapReduce workloads while maintaining sufficient resource for Spark applications. Second, it benefits from the capability of iKayak that can adaptively search efficient worker nodes to place executors based on the reservation time awareness. The results also reveal that iKayak is more effective for CPU intensive workload (i.e., K-means and Index) than I/O intensive workload (i.e., PageRank). This is due to the fact that the memory configurations of the machines in the cluster



(a) Spark completion time. (b) MapReduce completion time. (c) Various CPU utilizations.

Fig. 7. Effectiveness of iKayak for spark, MapReduce and the cluster resource utilization.

are relatively limited. The reservation time of I/O intensive workload is longer than that of CPU intensive workload.

Reducing MapReduce Job Completion Time. Fig. 7b shows iKayak improves the overall job completion time of MSD workload by 14 and 19 percent compared with YARN-client mode and YARN-cluster model, respectively. This is due to the fact that iKayak is more flexible about the resource management and task scheduling of MapReduce applications in a task dependence and locality aware manner. On the other hand, Fig. 7b shows that YARN-client model achieves better performance than YARN-cluster model does. This is because that YARN-client model hosts only one Spark application at one time. It allocates more cluster resources to MapReduce applications than YARN-cluster model does.

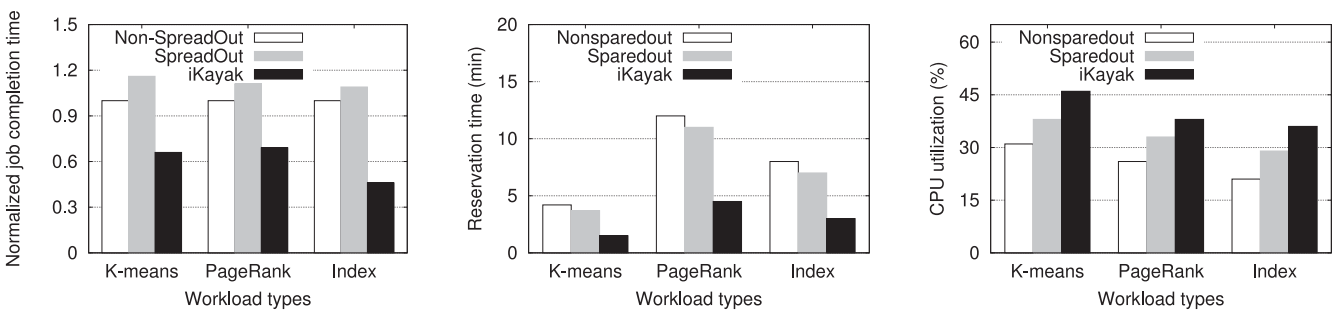
Increasing Cluster Resource Utilization. Fig. 7c shows the CPU utilizations of the various type machines that resulted from different cluster resource scheduling approaches. It demonstrates iKayak improves the overall CPU utilization of all type machines in the cluster by 22 and 15 percent compared with YARN-client model and YARN-cluster model, respectively. Note that iKayak achieves a lower utilization improvement on T420 and T620 machines compared to others in the cluster. This is due to its relatively lower memory resource availability and powerful CPU. Thus, it is hard to fully utilize the CPU especially for memory intensive computing. We further compare the CPU utilization under the dynamic resource allocation (Dynamic R.A.) policy of Spark and the proposed iKayak approach. The coarse-grained dynamic resource allocation policy improves the CPU utilization by 16 and 9 percent compared with YARN-client model and YARN-cluster model, respectively. The results demonstrate that the fine-grained resource allocation of iKayak provides higher resource utilization than the dynamic allocation policy does.

5.2 Benefit of Executor Placement

Fig. 8 compares the job completion times and CPU utilizations achieved by different executor placement strategies (i.e., *SpreadOut*, *Nonspreadout* and iKayak) while running the three Spark workloads on the cluster. The result demonstrates the proposed reservation-aware executor placement approach significantly reduces the job completion times and reservation times of the Spark jobs while increasing the CPU utilization of the hosting machines.

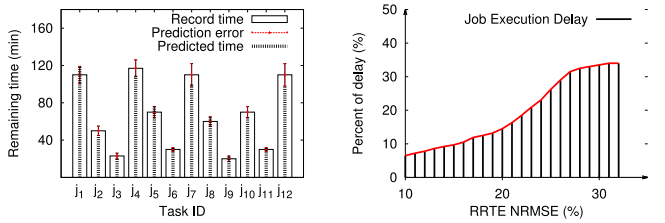
Reducing Job Completion Times. Fig. 8a shows iKayak significantly reduces the job completion times of K-means, PageRank and Index workloads up to 55 and 37 percent compared with *NonspreadOut* policy and *SpreadOut* policy, respectively. It can automatically select the destination machines to host Spark executors, which aims to avoid the unnecessary waiting time for resource reservation. In particular, the reservation-aware executor placement mechanism effectively reduces the opportunity of placing executors on the worker nodes that host long running reduce tasks. The result also shows that *NonspreadOut* approach achieves better performance than *SpreadOut* policy in terms of job completion time. This is due to the fact that *NonspreadOut* encourages executors to be placed on a subset of machines in the cluster so that it can reduce the intermediate data traffic to improve performance.

Reducing Reservation Times. Fig. 8b shows the reservation-aware executor placement mechanism of iKayak significantly reduces the reservation times of K-means, PageRank and Index workloads up to 67 and 53 percent compared with *NonspreadOut* policy and *SpreadOut* policy, respectively. It also illustrates that the reservation time of the executors with fewer resource demands (i.e., K-means) is apparently less than that of the executors with more resource demands (i.e., PageRank and Index). This is due to



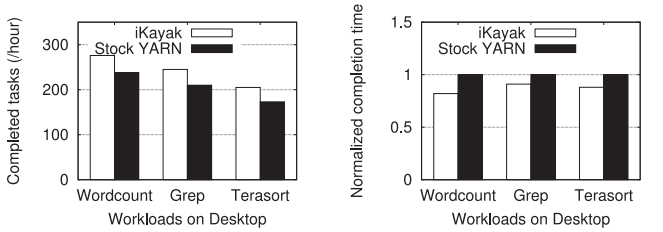
(a) Reduce job completion time. (b) Reduce reservation time. (c) Increase CPU utilization.

Fig. 8. Effectiveness of the reservation-aware executor placement for Spark applications.



(a) Prediction error. (b) Impact on job execution.

Fig. 9. Accuracy of task remaining time estimation.



(a) Throughout of Map tasks. (b) Reduce completion time.

Fig. 10. Effectiveness of reduce adjustment.

the fact that small resource demand can be satisfied in time by the current reservation based resource management scheme in YARN. The result also reveals that *SpreadOut* approach achieves better performance than *NonspreadOut* policy in terms of reservation time. The reason is that the centralized deployment of *NonspreadOut* causes more executors to be co-hosted with reduce tasks than *SpreadOut* does, leading long time to wait for resource reservation.

Increasing CPU Utilization. Fig. 8c shows the executor placement mechanism slightly increases the CPU utilizations (i.e., 10 percent) for the three different workloads since most of Spark workloads are memory intensive application and have limited impact on CPU resource. On the other hand, *SpreadOut* incus higher CPU utilization than *NonspreadOut* does. This is due to the fact that *SpreadOut* policy tries to spread out all executors on the whole cluster. It leads executors to be located on most nodes in the cluster, which results in higher CPU utilization than *NonspreadOut*.

5.3 Accuracy of Remaining Time Estimation

iKayak applies fuzzy models to estimate the reduce task remaining times while selecting efficient hosting machines for Spark executors. To evaluate the accuracy of the fuzzy models, we compare the error between the predicted reduce task remaining times and the actual remaining times. The accuracy is measured by the normalized root mean square error (NRMSE), a standard metric for deviation. Three MapReduce applications from the PUMA benchmark [6], i.e., *Terasort*, *Grep* and *WordCount*, each with 300 GB input data, were run on the four different machines (i.e., T420, T320, T620 and T110) in the cluster. Accordingly, we recorded the actual remaining times and predicted remaining times of 12 reduce tasks as shown in Fig. 9a. It shows that the prediction was quite accurate, with on average 7.8 percent NRMSE. Fig. 9b shows that as the remaining time prediction error is increased from 10 to 35 percent, the Spark job execution times increased from 7.5 to 34 percent compared to the execution times achieved with iKayak's real prediction accuracy. This is due to the fact that inaccurate remaining time estimation may incur additional Spark executor waiting time for resource reservation. This observation confirms that the reservation-aware executor placement, and the estimation accuracy of reduce task remaining time are critical to reducing Spark job execution times, and the executor placement waiting times.

5.4 Benefit of Resource Adjustment

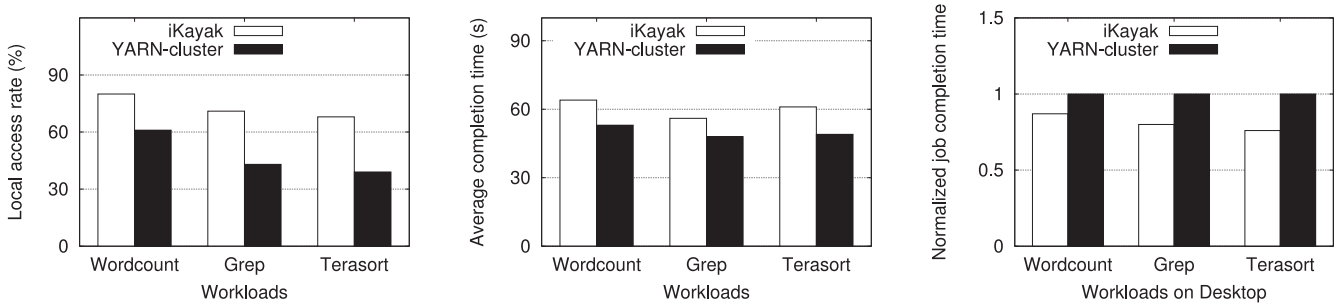
Fig. 10 depicts the throughput of MapReduce tasks achieved by iKayak and stock YARN in the experiment. We turn off other features of iKayak except the resource adjustment mechanism. The result demonstrates that the resource adjustment mechanism of iKayak effectively increases the throughput of map tasks and slightly reduces the completion times of reduce tasks. Fig. 10a shows that the throughput of map tasks on the Desktop machine is significantly increased about 17 percent compared to stock YARN since the dependence-aware resource adjustment mechanism gives more resource to map executions. The resource adjustment mechanism periodically suspends the reduce tasks when there are not enough intermediate data to process and releases the redundant resource to the related map tasks.

The reduce suspending provides more cluster resource for the corresponding map tasks. It further produces more intermediate data for their reduce tasks. Thus, suspending reduce tasks indeed accelerates these reduce task executions in this case. Fig. 10b demonstrates that the suspending policy has reduced the reduce completion time 8 percent than stock YARN does on the Desktop machine. This is due to the fact that the resource allocated to reduce tasks are over provisioning and would be idle if there are not enough intermediate data to feed.

5.5 Benefit of Locality Coordination

Fig. 11 demonstrates that the proposed locality-aware task assignment coordination module effectively improves the local data access rate of MapReduce tasks and accordingly reduces job completion times. Fig. 11a compares the local data access rates of map tasks achieved by the locality-aware coordination of iKayak and YARN-cluster model in the experiment. The result demonstrates that iKayak increases the local data access by 27 percent compared to YARN-cluster model for the workload Wordcount, Grep and Terasort. This is due to the fact that the coordination mechanism enforces map tasks to be assigned on the hosting nodes with local data access when MapReduce competes the locality awareness with Spark applications. Thus, the coordination mechanism allows more local map executions than the YARN-cluster deployment.

As described before, shrinking the resource allocations for map tasks to achieve local data access deteriorates individual map task performance. Fig. 11b shows the average map task completion time increases slightly by 9 percent when apply the locality-aware coordinator. However, the performance deterioration of individual tasks contributes more opportunities to run map tasks in parallel (i.e., increasing map "slots"). When shrinking some tasks to prefer data locality, there are more resources on other slave machines in the cluster will be freed. Those available resources can be used to run more other tasks in parallel than



(a) Local access rate. (b) Map task completion time. (c) MapReduce job completion time.

Fig. 11. Impact of cross-platform locality-aware assignment coordination (LAC).

without task shrinking. Increasing parallelism of map execution does not only compensate the performance losses of the shrunk map tasks, but also significantly improve job level performance. The whole job execution progress will be speeded up correspondingly. Fig. 11c demonstrates the job completion times of Wordcount, Grep and Terasort are effectively reduced when apply the locality-aware task assignment coordination mechanism.

5.6 Impact of Parameters and Configurations

We change the values of the suspension weight parameter v and the overhead θ to study their impact on the performance improvement in terms of job completion time. Fig. 12a shows that the job completion time initially decreases as the suspension parameter v increases. However, increasing the v further leads to performance degradation. This tells that a very large suspension parameter v may lead to job completion time deterioration. A large overhead θ (e.g., 2) leads significant performance deterioration due to the instability of suspension control. Thus, we empirically set the suspension parameter v to 1 in the experiment. Fig. 12b shows that tuning the suspension overhead θ has the similar phenomenon with tuning the parameter v. We empirically set the overhead θ to 0.15 without affecting system stability. It is a tradeoff between the reduce task suspension time cost and the job completion time.

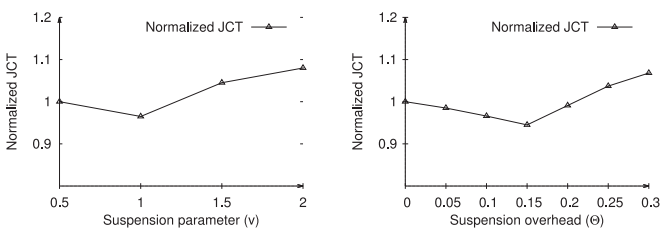
We further explore the effectiveness of the proposed iKayak under the various configurations (i.e., applying early start and delay scheduling) in the experiment. We first compared the performance improvements under the different early start percentages. Fig. 13a shows the job completion times changed as we altered the values of the early start percentage under the different approaches, i.e., iKayak and original YARN. The result demonstrates that iKayak outperforms the original YARN about 7 percent in terms of job completion time when tuning the early stage percentage. It shows that IO intensive workload (e.g., terasort) prefers small early start percentages since the intermediate data from map tasks can be directly shuffled to save the bandwidth. Non-IO intensive workload (e.g., wordcount) prefers a large early start percentage because it prevents reduce tasks to occupy the reduce slots without execution. We then compared the local data access rates while applying different scheduling policies (i.e., Delay (D) and Non-delay (N)) on MapReduce and Spark, respectively. The results in Fig. 13b shows that the delay scheduling increases the local access rate 12–15 percent if only one application (i.e., Spark or MapReduce) adopts it. However, when both Spark and MapReduce apply the delay scheduling, the local access rates are similar with the scenario that both of them do not apply the delay scheduling.

5.7 iKayak Overhead and Scalability

The overhead of iKayak mainly comes from the time required to perform the executor placement algorithm, the time required to suspend and resume reduce tasks, and the time required to reconfigure individual map tasks. Experimental results show that the overhead of the executor placement algorithm is between 120 to 150 ms, which is very small compared to Spark job execution time. We also measure the suspension overhead in each control interval. iKayak took on average of 2.7 seconds to suspend and resume a reduce task. The overhead is relatively negligible compared to the 5-minute control interval. Finally, the overhead of the task-level reconfiguration is quite stable (i.e., 45-55 ms) and independent of the testbed size. iKayak is scalable and applicable to larger scale clusters. The revisions on YARN components do not affect other computing systems.

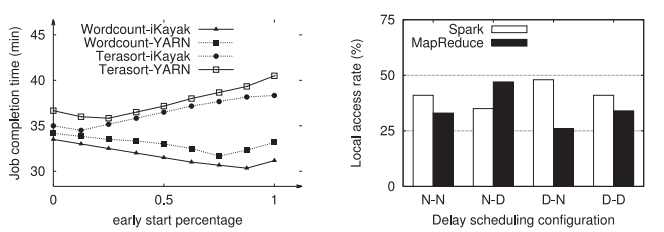
6 Related Work

Big-Data Resource Management. Modern big-data clusters apply a diverse mix of resource managers, e.g., YARN,



(a) Impact of v. (b) Impact of θ.

Fig. 12. Sensitivity of suspension parameter tuning.



(a) Impact of early start. (b) Impact of delay scheduling.

Fig. 13. Impact of early start and delay scheduling.

Corona, Omega and Mesos. These different computing frameworks have inherently different scheduling needs. YARN [3], the second generation of Hadoop, added a resource management layer in Hadoop. It allows different applications to be allocated with different number of task containers. Corona [18] is developed by Facebook and provides more flexibilities to manage the cluster resources based on the different resource demands of workloads. Omega [19], a new parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, to achieve both implementation extensibility and performance scalability. Mesos [5] abstracts CPU, memory, storage, and other compute resources away from machines, enabling fault-tolerant and elastic distributed systems to easily be built and run. Among these Big-data resource managers, YARN is the only one that supports security and leverages the existing HDFS dataset at the same time.

MapReduce Optimization. There are growing interests on MapReduce performance optimization with various techniques, e.g., resource provisioning [20], [24], job scheduling [22] and self-tuning configuration [12]. Jinda et al. [21] proposed a new data layout, coined Trojan Layout, that internally organizes data blocks into attribute groups in order to improve data access times. Dittrich et al. proposed Hadoop++ [26], a new index and join technique to improve runtime of MapReduce jobs. They are able to schedule incoming MapReduce jobs to data block replicas with the most suitable Trojan Layout. Recently, a few studies start to explore that how to optimize Hadoop configuration to improve MapReduce performance. Herodotou et al. proposed Starfish [23], an optimization framework that hierarchically optimizes from MapReduce jobs to work flows by searching for good parameter configurations. None of these approaches consider to optimize MapReduce performance by dynamic job scheduling in the Spark-on-YARN environment.

Task Scheduling. Many prior studies have shown that MapReduce performance can be significantly improved by various scheduling techniques [22], [25]. The default FIFO Scheduler in Hadoop implementation may not work well since a long job can exclusively take the computing resource on the cluster, and cause large delays for other jobs. This is the reason that many schedulers, e.g., Capacity Scheduler, Fair Scheduler, can share resources among multiple jobs. Recently, a few studies [20] start to optimize the performance of MapReduce jobs with respect to their performance goals. Wolf et al. described FLEX [22], a flexible and intelligent allocation scheme for MapReduce workloads. It is proposed as an add-on module that worked synergistically with Fair Scheduler to provide performance guarantees. Curino et al. designed a reservation-based scheduling [20], that provides support for prioritized MapReduce scheduling of the jobs with various deadlines. Our work differs from these efforts in that we consider running Spark and MapReduce applications together on YARN clusters.

7 Conclusions and Future Work

We observed that running Spark and MapReduce on YARN clusters incurs significant performance deterioration due to the semantic gap between the reservation-based resource allocation scheme of YARN and the dynamic application demands. Therefore, we design and develop a cross-platform resource scheduling middleware, iKayak, that aims to improve the cluster resource utilization as well as application performance for Spark-on-YARN deployment. It relies on three key mechanisms, i.e., reservation-aware executor placement, dependency-aware resource adjustment and locality-aware assignment coordination. iKayak leverages time-varying resource demands of different applications, inter-task dependency between map and reduce tasks, and cross-platform locality awareness to tackle the aforementioned challenges. Experimental results show that iKayak achieves up to 50 percent performance improvement for Spark applications and 19 percent performance improvement for MapReduce applications compared to the two popular Spark on YARN deploy models, i.e., YARN-client model and YARN-cluster model. In future work, we will explore more cross-platform resource scheduling approaches while deploying other processing paradigms (e.g., Shark, Pig, Storm and Hive) on Hadoop YARN.

Acknowledgments

This research was supported in part by US National Science Foundation research grant CNS-1422119. Xiaobo Zhou is the corresponding author.

References

[1] M. Zaharia, et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 2–2.
[2] Yahoo!, "Let spark fly: Advantages and use cases for spark on hadoop," 2014. [Online]. Available: https://spark-summit.org/2014/
[3] V. K. Vavilapalli, et al., "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. ACM 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 5.
[4] Pivotal Cloud Foundry, "YARN resource management," [Online]. Available: http://pivotalhd.docs.pivotal.io/docs/yarn-resource-management.html
[5] B. Hindman, et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.
[6] PUMA, "Purdue MapReduce benchmark suite," 2012. [Online]. Available: https://engineering.purdue.edu/ puma/datasets.htm
[7] L. Wang, et al., "BigDataBench: A big data benchmark suite from internet services," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2014, pp. 488–499.
[8] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[9] Spark 1.4.0 Documentation, "spark.deploy.spreadOut," 2015. [Online]. Available: https://spark.apache.org/docs/1.4.0/spark-standalone.html
[10] Y. Chen, B. Yang, A. Abraham, and L. Peng, "Automatic design of hierarchical Takagi Sugeno type fuzzy systems using evolutionary algorithms," *IEEE Trans. Fuzzy Syst.*, vol. 15, no. 3, pp. 385–397, Jun. 2007.
[11] Y. Wang, C. Xu, X. Que, X. Li, and W. Yu, "JVM-bypass shuffling for Hadoop acceleration," in *Proc. IEEE 7th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 569–578.
[12] D. Cheng, J. Rao, Y. Guo, and X. Zhou, "Improving MapReduce performance in heterogeneous environments with adaptive task tuning," in *Proc. ACM/IFIP/USENIX 15th Int. Middleware Conf.*, 2014, pp. 97–108.
[13] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, "Scale-up versus scale-out for hadoop: Time to rethink?" in *Proc. ACM Symp. Cloud Comput.*, 2013, Art. no. 20.
[14] eBay, "Using spark to ignite data analytics," 2014. [Online]. Available: http://www.ebaytechblog.com/2014/05/28/using-spark-to-ignite-data-analytics/U-qUSPldUbw

[15] Cloudera, "Configuration parameters," 2012. [Online]. Available: http://blog.cloudera.com/blog/author/aaron/
[16] M. Li, et al., "MRONLINE: Mapreduce online performance tuning," in *Proc. ACM Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 165–176.
[17] Cloudera, "Apache Spark resource management and YARN app models," 2014. [Online]. Available: http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/
[18] Facebook, "Hadoop corona: The next version of MapReduce," 2013. [Online]. Available: https://github.com/facebookarchive/hadoop-20/tree/master/src/contrib/corona
[19] M. Schwarzkopf, A. Konwinski, M. Abd-el-Malek , and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 351–364.
[20] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!" in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.
[21] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts: Right shoes for a running elephant," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, Art. no. 21.
[22] J. Wolf, et al., "FLEX: A slot allocation scheduling optimizer for MapReduce workloads," in *Proc. ACM/IFIP/USENIX 11th Int. Middleware Conf.*, 2010, pp. 1–20.
[23] H. Herodotou, et al., "Starfish: A self-tuning system for big data analytics," in *Proc. Conf. Innovative Data Syst. Res.*, 2011, pp. 261–272.
[24] D. Cheng, C. Jiang, and X. Zhou, "Resource and deadline-aware job scheduling in dynamic Hadoop clusters," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2015, pp. 956–965.
[25] D. Cheng, P. Lama, C. Jiang, and X. Zhou, "Towards energy efficiency in heterogeneous Hadoop clusters by adaptive task assignment," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 359–368.
[26] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *Proc. VLDB Endowment*, vol. 3, pp. 518–529, 2010.

Dazhao Cheng received the BS and MS degrees in electronic engineering from Hefei University of Technology, in 2006 and the University of Science and Technology of China, in 2009, respectively, and the PhD degree from the University of Colorado, Colorado Springs, in 2016. He is currently an assistant professor in the Department of Computer Science, University of North Carolina, Charlotte. His research interests include cloud and data intensive computing.

Xiaobo Zhou received the BS, MS, and PhD degrees in computer science from Nanjing University, in 1994, 1997, and 2000, respectively. Currently, he is a professor and the chair of the Department of Computer Science, University of Colorado, Colorado Springs. His research lies broadly in computer network systems, specifically, cloud computing and datacenters, BigData parallel and distributed processing, autonomic and sustainable computing, and scalable Internet services and architectures. He received the NSF CAREER Award in 2009. He is a senior member of the IEEE.

Palden Lama received the BTech degree in electronics and communication engineering from the Indian Institute of Technology, in 2003 and the PhD degree in computer science from the University of Colorado, Colorado Springs, in 2013. Currently, he is an assistant professor in the Department of Computer Science, University of Texas, San Antonio. His research interests include cloud computing and big data processing in the cloud.

Jun Wu received the BS degree in information engineering and the MS degree in communication and electronic system from Xidian University, in 1993 and 1996, respectively, and the PhD degree from Beijing University of Posts and Telecommunications, in 1999. He is a professor in the Computer Science and Technology Department, Tongji University, China. His research interests include wireless communication and signal processing.

Changjun Jiang received the PhD degree from the Institute of Automation, Chinese Academy of Sciences, Beijing, China, in 1995. Currently, he is a professor in the Department of Computer Science, Tongji University, Shanghai. He is also the director of Professional Committee of Petri Net of China Computer Federation. His current areas of research are concurrency and parallelism, BigData, Petri net, and intelligent transportation systems.