A Dynamic Resource Allocation Framework for Apache Spark Applications

Kewen Wang, Mohammad Maifi Hasan Khan, and Nhan Nguyen
Department of Computer Science and Engineering
University of Connecticut
kewen.wang@uconn.edu, maifi.khan@uconn.edu, nhan.q.nguyen@uconn.edu

***Abstract*—In this paper we design and implement a middleware service for dynamically allocating computing resources for Apache Spark applications on cloud platforms, and consider two different approaches to allocate resources. In the first approach, based on limited execution data of an application, we estimate the amount of resource adjustment (i.e., Δ) for each application separately a priori which is static during the execution of that particular application (i.e., Approach - I). In the second approach, we adjust the value of Δ dynamically during runtime based on execution pattern in real-time (i.e., Approach - II). Our evaluation using six different Apache Spark applications on both physical and virtual clusters demonstrates that our approaches can improve application performance while reducing resource requirements significantly in most cases compared to static resource allocation strategies.**

***Index Terms*—Resource Allocation, Apache Spark, Resource Scaling, Performance Prediction**

I. Introduction

An increasing number of organizations are utilizing cloud platforms to provide services and host third party computing applications [1], [2]. In this rapidly changing technological landscape, cloud management, especially cloud resource allocation, is becoming increasingly important to minimize operating cost [3]–[11]. However, as workload and computing resource requirements often vary across different categories of applications (e.g., graph processing, image processing), static resource allocation strategies can lead to resource wastage and/or suboptimal performance. Prior efforts attempted to address this by developing dynamic resource allocation strategies [12]–[15]. However, given that multiple applications are often executed concurrently on cloud platforms, commonly used prediction-based allocation strategies that require prior training data often suffer from poor scalability.

To address this challenge, as resource demand for an application often varies with time, we present an application-level dynamic resource allocation scheme that is designed to adjust the computing resource according to the changes in workloads and resource demands during runtime. Specifically, as reassigning resources during runtime affects the performance of the current timeslot and thereby affects the application resource requirements for the subsequent time slots, we design a closed-loop algorithm that combines runtime prediction with resource allocation. In our framework we leverage the current and prior resource usage to predict future demand, and use either an a priori calculated (Approach-I) or dynamically adjusted (Approach-II) threshold value (i.e., Δ) to adjust the amount of allocated resources for the future time intervals. Our experiments using six different Apache Spark data processing applications on both physical and virtual clusters demonstrates that our approach can improve application performance while reducing resource demand significantly in most cases compared to static allocation strategies. Furthermore, in case of multiple concurrent applications, even when performance is affected negatively (by at most 13%) in a small number of cases, it led to significant reduction in resource requirements (i.e., reduction in resource requirements ranged between 49% to 84% in cases where the performance were affected negatively).

The rest of this paper is organized as follows. Relevant prior work is discussed in Section II. Section III presents the detailed design and implementation of our system. Experimental result is presented in Section IV. Finally, Section V concludes the paper.

II. Related Work

A significant volume of prior work exists that has looked at dynamic resource allocation using various techniques such as performance modeling, optimization of cloud configurations, and job scheduling [6]–[8], [12]–[19]. Among these, PARIS [7] applies data-driven modeling approach to choose the optimal virtual machine based on target performance and cost constraints. It combines offline and online stages to construct a machine learning model (i.e., random forest model), and estimates task performance and the resulting cost to select the most cost effective VM type. In the offline stage, it runs a large set of benchmarks and collects profiling data for each VM. In the online stage, given the input of a representative task by a user, this system builds a performance model, invokes performance predictors, and estimates the cost of each VM to enable a high-level policy to choose a VM. However, it needs sufficient benchmark workload data to train the model in order to achieve high accuracy prediction for VM selection.

In a complimentary approach, a group of work exists that treats resource allocation problem as a cloud configuration optimization problem [15], [20]–[23]. For instance, CherryPick [15] attempts to optimize configuration settings such as the number of VMs, number of CPU cores, and RAM size in the context of various cloud applications (e.g., Spark regression

978-1-7281-7303-0/20/$31.00 ©2020 IEEE
DOI 10.1109/COMPSAC48688.2020.0-141

application, MapReduce applications). Towards that, it applies a Bayesian Optimization engine to build performance models, designs a search controller to orchestrate the configuration selection process including providing candidate configurations to the Bayesian Optimization engine and searching over multiple iterations, and uses a cloud controller to facilitate the control of running workloads. However, CherryPick needs to be rerun to build a new model each time the workload changes.

Along with modeling based approaches, there are prior efforts that attempt to allocate resources based on application resource demands [12], [14], [18], [19], [24]. For instance, Pocket [12], which is close in spirit to our work, investigates the challenge of dynamically allocating resources along multiple dimensions (e.g., CPU, network, storage) for serverless applications. This system is designed for ephemeral data sharing. To ensure that serverless applications are not constrained by I/O bottleneck, Pocket uses a control plane to determine job I/O requirement based on "hints" from registered jobs regarding resource requirements, and selects storage tier (such as DRAM, disk) and the number of storage servers. It controls virtual machines hosting containers of serverless applications for automatic and dynamic resource allocation, thereby providing high I/O performance with lower cost. Prior efforts also looked at data stream processing (DSP) systems where system needs to be scaled up or down based on incoming data rate and resource usage. For instance, [18] investigated a controller based approach to reduce the cost of scaling operations by making the scaling decisions based on changes in data rate and system states. It utilizes an Extended Kalman Filter (EKF) for smoothing the measured system states to reduce the number of operations for scaling system, and performs predict-update iterations for auto scaling in response to changes in system states.

While a significant volume of prior work exists that looked at dynamic resource allocation for cluster workloads, in contrast to prior efforts, our work treats each cloud application separately, and uses an application-level dynamic resource allocation approach to improve application performance while minimizing resource requirement.

III. Overview

In this work we focus on dynamically allocating resources for cloud applications (e.g., map reduce framework, Apache Spark) that often execute in multiple stages. For instance, Apache Spark application (which is the focus of our work) typically consists of multiple execution stages that are executed sequentially where each stage implements a distinct operation of an application program. As such, during the execution of an application, the resource usage patterns often vary across execution stages of the application. One way to allocate resources for such applications is to construct fine grained stage-specific prediction models, and use that to predict and allocate resources. However, this approach requires constructing models for each application separately, which is not scalable.

To address this, in this paper we design a dynamic resource allocation middleware service that leverages resource utilization information at the application level to allocate resources during runtime. In contrast to application-level analytical performance models, leveraging run time traces allows us to infer resource contention at the system level and address that without understanding the inner workings of a target application. In our work, to capture changes in resource requirements during the execution of an application, we calculate resource usage profiles dynamically without knowing the stage boundaries within an application. Specifically, in our framework, for a distributed data processing application, we construct H resource usage profiles for an application running on a cluster of H working nodes where each resource usage profile consists of a series of timestamped resource usage vectors as shown in equation 1. There is one resource usage vector for every time interval. The number of the resource usage vectors (i.e., N_h) is determined by the execution duration of an application and the sampling interval, which is calculated as in equation 2.

$$Resource = \{Resource_h \mid 1 \leq h \leq H\} \quad (1)$$

$$Resource_h = \{ResourceUsage_{h,i} \mid 1 \leq i \leq N_h\}$$

$$N_h = \frac{AppDuartion_h}{TimeInterval_h} \quad (2)$$

Here, H is the number of working nodes in the cluster. N_h is the number of $ResourceUsage$ vectors, and $TimeInterval_h$ is the time gap between two resource usage vectors on host h (i.e., $1s$ is used as the default value). $AppDuartion_h$ is the application execution time on host h.

For a given time interval, there is one resource usage vector that represents the resource usage (e.g., CPU, Memory, and I/O) for an application as shown in equation 3. To construct these resource utilization vectors, we leverage $/proc$ filesystem which can be analyzed with minimal overhead. Specifically, for CPU utilization, we use the percentage of CPU utilization at the user level (application) and the system level (kernel) as shown in equation 4. While there are different resources that may impact performance and can be included in our framework (e.g., memory utilization, I/O), in our current implementation we focus on tuning CPU resource as it turned out to be the most influential in impacting application performance in our case.

$$ResourceUsage_{h,i} = (CPU_{h,i}, MEM_{h,i}, IO_{h,i}) \quad (3)$$

$$CPU_{h,i} = (usr_{h,i}, sys_{h,i}) \quad (4)$$

Here $1 \leq i \leq N_h$, and N_h is the number of time points calculated as in equation 2.

For CPU, there are two important parameters, namely, $cpu.cfs\_period\_us$ and $cpu.cfs\_quota\_us$, that can be adjusted to tune the CPU resource allocation. These can be also set to allow an application to use all the available computing resources in the cluster. While these can be set statically a priori for applications that exhibits mostly constant resource usage patterns, however, this is often not suitable as resource

requirements for most applications vary during runtime. For such applications, static resource allocation will result in resource wastage if higher resource cap than needed is set, or lead to performance degradation if lower cap than needed is set. To avoid this problem, we design a dynamic resource allocation middleware service, and use cgroup (a linux kernel feature) [25] to control the maximum CPU resource an application can acquire during runtime. Details are presented below.

A. Resource Allocation

The main idea behind our middleware service is to allocate computing resource for the next interval based on the resource usage of the current and last interval. Specifically, if the resource usage for the current interval is lower than the last interval, we predict that the resource requirement has decreased, and therefore reduce the maximum resource limit for the next time interval. However, if the resource usage is as high as the current maximum value, it is possible that the maximum limit is set to too low. In that case, in order to allow resource scaling-up for an application, we adjust the maximum limit by adding a value Δ to the current limit as shown in equation 5.

$$ResourceQuota = currentUsage + \Delta, \quad 0.1 \leqslant \Delta \leqslant max\_res \tag{5}$$

In this equation, max_res is the maximum resource that can be allocated. For CPU resource, it is the value of total available CPU cores on one working node.

In our work, we consider two different approaches to estimate the value of Δ. Specifically, in the first case, we calculate the value of Δ for each application separately which is static during the execution of that particular application (i.e., Approach - I). Note that, while Δ is static during the execution of an application, it is different for each application and estimated based on limited execution data of each application using our developed algorithm. In the second case, the value of Δ changes dynamically during execution based on resource usage pattern in real-time (i.e., Approach - II). We present each of these approaches below.

1) Approach - I: Use of Application-specific Δ: As resource consumption varies across time, ideally, there should be a (possibly different) resource limit (resource quota) for each time interval as shown in equation 6. The time interval is set to one second in our case, which can be set to other values to tune the reactivity of the algorithm depending on resource usage patterns.

$$ResQuota = \{quota_i \mid 1 \leq i \leq N_h\} \tag{6}$$

$$quota_i = usg_{i-1} + \Delta \tag{7}$$

Here, N_h is the number of time intervals for the target application on host h. In equation 7, resource quota $quota_i$ is determined based on previous resource usage and Δ.

While having a (possibly) different Δ for each time interval may lead to near-optimal resource allocation, it is difficult to estimate Δ for each time interval for each application. As such, we attempt to estimate a single value for Δ for each application that may minimize the prediction error. Towards that, we designed a search algorithm as shown in Algorithm 1 that iterates through a series of values within a given range to find a value for Δ that minimizes a given *cost function*. Specifically, to estimate Δ for a specific application (e.g., PageRank), we first execute the application (e.g., PageRank) to construct the resource usage profile as a sequence of values for that application as shown in equation 8. The number of data points (i.e., N) in the resource usage profile is determined based on the total execution time of the sample application and the time interval as shown in equation 9.

$$SampleResUsage = \{usg_i \mid 1 \leq i \leq N\} \tag{8}$$

$$N = \frac{SampleApplicationDuration}{TimeInterval} \tag{9}$$

The next step is to find the $ResQuota$ that minimizes $Distance$ defined in equation 10, which is calculated as the summation of the differences between the allocated resource quota (i.e., $quota_i$) and resource usage values.

$$Distance = \sum_{i=1}^{N} (quota_i - usg_i)^2 \tag{10}$$

Here $quota_i$ is a value within the range of $[0.1, CoreNum]$ where $CoreNum$ is the number of total CPU cores.

As shown in Algorithm 1, the idea is to try each δ value in a given range (i.e., $[0.1, CoreNum]$) with step size inc, and identify the δ that minimizes the distance calculated in equation 10. Further, as setting the resource quota too small can hurt performance, to avoid this, we add a constraint to keep the quota value from being too low. Specifically, $diff\_sum$ is calculated as in equation 11, and $diff\_sum > 0$ is used as a constraint to ensure that the sum of quota values is positive.

$$diff\_sum = \sum_{i=1}^{N} (quota_i - usg_i) \tag{11}$$

The time complexity of this algorithm is $O(N \times M)$, where N is the number of time points in the sample application as calculated in equation 9, and M is the number of possible δ within the range which is calculated as $M = CoreNum/inc$. Here inc is the step size. Compared to actual application execution, the execution time of the sample application is smaller. As N and M are relatively small, the time complexity of this algorithm is low (i.e., N=33 and M=60 in our implementation).

2) Approach - II: Use of Dynamic Δ: In our second approach, instead of calculating a single Δ value for each application, we design an algorithm to dynamically adjust Δ at each time interval based on the current value of Δ, allocated resources, and the actual resource usage as shown in equation 12 and 13.

Input: ResourceProfile res
Output: Recommended Δ_{opt}
Function *ParameterSearch*
  $res = \{usg_i \mid 1 \leq i \leq N\}$;
  Initialize $\delta = 0.1, inc = 0.1, max\_dis = inf$;
  while $\delta \leqslant CoreNum$ **do**
    $quota = \delta$;
    $dis = 0$;
    $diff\_sum = 0$;
    $notLow = True$;
    for usg *in* res **do**
      $diff = quota - usg$;
      $diff\_sum = diff\_sum + diff$;
      if $diff\_sum < 0$ **then**
        $notLow = False$;
        $break$;
      end
      $dis = dis + diff^2$;
      $quota = min(usg + \delta, CoreNum)$;
    end
    if $dis < max\_dis$ *and* $notLow = True$ **then**
      $max\_dis = dis$;
      $\Delta_{opt} = \delta$;
    end
    $\delta = \delta + inc$;
  end
end

Algorithm 1: Search Algorithm for Parameter Δ

$$\Delta \quad = \quad \{\delta_i \mid 1 \leq i \leq N_h\} \tag{12}$$

$$\delta_{i+1} = \begin{cases} \delta_i - (quota_i - usg_i) \times dec, & quota_i > usg_i \\ \delta_i + (max_{res} - usg_i) \times inc, & quota_i \leqslant usg_i \end{cases} \tag{13}$$

In equation 13, δ_{i+1} is adjusted depending on which of the two conditions is satisfied. Specifically, in the first condition, if the currently allocated resource quota (i.e., $quota_i$) is larger than the current resource usage (i.e., usg_i), we decrease δ_i to avoid over allocation. The difference is determined based on the currently allocated resource quota (i.e., $quota_i$) and resource usage (i.e., usg_i). To control the rate of decrease, we use a ratio dec which is set to 0.5. In the second condition, if the currently allocated resource quota (i.e., $quota_i$) is not larger than the current resource usage, we increase δ_i. The difference is determined based on the maximum allowed resource max_res and the current resource usage usg_i. Under this condition, we use a ratio inc to control the rate of increase which is set to 0.2 in our current implementation. The value of dec and inc is determined empirically and can be adjusted to tune the sensitivity of the algorithm.

B. System Implementation

The middleware is implemented using Linux cgroups features. As shown in Figure 1, this middleware is deployed on each worker node. After an application is launched, the corresponding process is detected by the "Application Listener" module. Next, the "Resource Monitor" module is triggered to collect resource usage metrics, which is used to dynamically allocate resource by the "Resource Allocation" module at each time interval. Resource usage is extracted by analyzing $/proc$ files. Each application process running on different working nodes is monitored and controlled separately. The overhead of running this tool is minimal and requires less than 1% CPU usage on our cluster.

TABLE I: Cluster Setting

Configuration	Physical Cluster	Virtual Cluster
Number of Nodes	4	4
Number of CPU cores per Node	20	6
RAM Memory size per Node	16GB	4GB

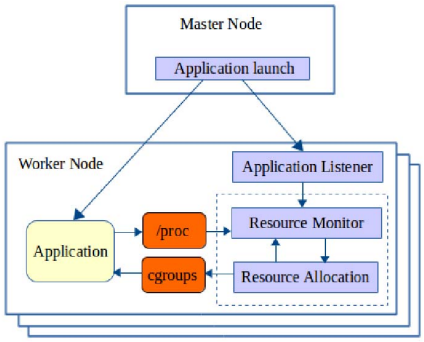


Fig. 1: Tool Architecture

IV. Evaluation

We evaluated the performance of our approach by executing multiple Apache Spark applications [26] on a cluster that included four physical machines. One machine was configured as the master node and the remaining three as the working nodes. The configuration of this cluster is listed in Table I. Each machine had 20 CPU cores and 16GB of RAM memory. The maximum CPU utilization limit was 2000% ($20 \times 100\%$) for each machine on this cluster. In addition to evaluating performance for a single application, to evaluate the performance for applications running in parallel on the cluster, on each physical machine we set up 3 virtual machines using Vagrant [27]. This allowed us to execute 3 applications in parallel.

In our experiments, Apache Spark (version 2.1.0) applications were executed on top of Hadoop Distributed File System (HDFS, version 2.7.3) in standalone mode [28]. Using standalone mode enabled us to apply our approach without external control and conflict. To evaluate our approach, we used different types of Apache Spark applications as listed in Table II. For input, Graph processing applications PageRank and ConnectedComponents used 50GB of graph data generated from LiveJournal network dataset downloaded from SNAP [29]. TriangleCount used a smaller subset of this data

TABLE II: Apache Spark Applications

Application	Abbrev.	Category	Input Size
PageRank	PR	Graph Processing	50GB
ConnectedComponents	CC	Graph Processing	50GB
TriangleCount	TC	Graph Processing	10GB
WordCount	WC	Text Analysis	80GB
K-MeansClustering	KM	Machine Learning	50GB
LogisticRegression	LR	Machine Learning	50GB

to avoid out-of-memory (i.e., OOM) problem. WordCount application used 80 GB of Wikipedia data downloaded from public source [30]. Two classical ML applications such as KMeans clustering and Logistic Regression used 50GB data of Color-Magnitude Diagram of Sloan Digital Sky Survey (SDSS) [31].

To demonstrate the effectiveness of our middleware, we compared the application execution time and total allocated CPU resources for the presented approaches against static allocation strategies. Execution time is calculated as the total run time of each application. Resource is calculated as the amount of allocated resources at each time point, and this value determined the upper bound of resource usage for that time point. For example, CPU quota 1 means 100% CPU usage is the limit, and CPU quota 20 means the application can reach at most 2000% CPU usage. The total CPU resource allocated is calculated as the summation of allocated resource quota at each time point during the application duration as shown in equation 14.

$$TotalResAllocated = \sum_{h=1}^{H} \sum_{i=1}^{N_h} ResQuota_{i,h} \quad (14)$$

Here H is the number of total working nodes in the cluster, and N_h is the number of time intervals calculated in Equation 2.

A. Results for a Single Application

To evaluate the performance of our approach in the context of a single application running in isolation, we executed six different Apache Spark applications one at a time under five different conditions as follows.

- CPU resource statically allocated at 2000% (i.e., Res-2000)
- CPU resource statically allocated at 500% (i.e., Res-500)
- CPU resource statically allocated at 300% (i.e., Res-300)
- CPU resource allocated using "Approach - I"
- CPU resource allocated using "Approach - II"

CPU resource was set to 2000% to compare against the maximum available CPU resource. 500% and 300% CPU usage limit were chosen empirically as the average CPU usage fell within that range for these applications.

In case of Approach - I, we first ran the corresponding sample application for each application where the input data size of each application was set to 2GB to minimize runtime of the sample application. We used this sample application trace to find the corresponding Δ. The execution time for each sample application and corresponding Δ are listed in Table III.

TABLE III: Overhead and Δ for Single Application ("Approach-I")

Application	Sample Time	*Delta*
PageRank	82s	1.4
ConnectedComponents	33s	0.7
TriangleCount	41s	0.7
WordCount	29s	1.5
K-MeansClustering	175s	0.6
LogisticRegression	75s	0.4

The performance improvements are presented by comparing the execution times and allocated resources for these applications under different resource allocation strategies. The execution time for each application under each condition is displayed in Figure 2. We compared the total allocated CPU quota for different conditions in Figure 3.

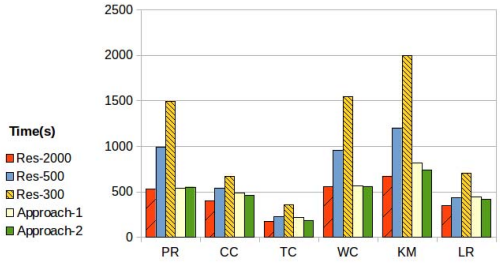


Fig. 2: Performance Improvement for Single Application

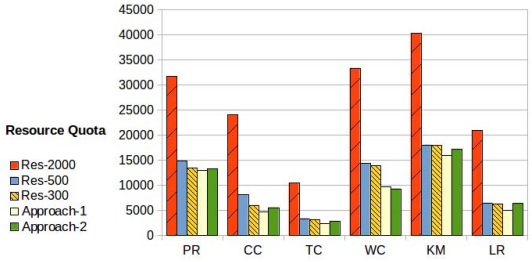


Fig. 3: Resource Allocation for Single Application

The performance for different approaches are listed in Table IV. As can be seen, Approach-I reduced resource requirement in all cases whereas Approach-II reduced resource requirement in all cases except for Res-300 condition for LogisticRegression which increased by 1%. Approach-I improved performance in all but seven conditions (i.e., (PR, Res-2000); (CC, Res-2000); (TC, Res-2000); (WC, Res-2000); (KM, Res-2000); (LR, Res-2000); (LR, Res-500)). Approach-II improved performance in all but six conditions (i.e., (PR, Res-2000); (CC, Res-2000); (TC, Res-2000); (WC, Res-2000); (KM, Res-2000); (LR, Res-2000)). Notably, while performance is reduced by at most 19% using Approach-II compared to the Res-2000 condition, it led to significant reduction in resource requirements (i.e., reduction in resource requirements ranged between 57% to 77% in cases where the performance were affected negatively).

TABLE IV: Performance Improvement and Resource Quota Reduction for Single Application

App.	Condition	Approach-I		Approach-II	
		Time	Resource	Time	Resource
PR	Res-2000	-3%	59%	-5%	58%
	Res-500	45%	12%	44%	11%
	Res-300	64%	3%	63%	1%
CC	Res-2000	-22%	80%	-14%	77%
	Res-500	10%	41%	15%	32%
	Res-300	27%	21%	32%	8%
TC	Res-2000	-24%	77%	-6%	73%
	Res-500	4%	29%	19%	15%
	Res-300	40%	25%	49%	11%
WC	Res-2000	-2%	71%	-1%	72%
	Res-500	41%	32%	42%	35%
	Res-300	63%	30%	64%	33%
KM	Res-2000	-22%	60%	-10%	57%
	Res-500	32%	11%	38%	5%
	Res-300	59%	11%	63%	4%
LR	Res-2000	-27%	76%	-19%	69%
	Res-500	-2%	23%	4%	1%
	Res-300	37%	21%	41%	-1%

TABLE V: Groups of Multiple Concurrent Applications

Group	Application	Start Time	Input Size
Group I	PageRank	87s	20GB
	WordCount	98s	20GB
	K-MeansClustering	2s	20GB
Group II	ConnectedComponents	40s	20GB
	TriangleCount	11s	1GB
	LogisticRegression	3s	20GB

B. Results for Multiple Concurrent Applications

To evaluate the performance of our approach for multiple concurrent applications that may lead to interference across applications, we created three virtual clusters on our physical cluster by creating three virtual machines on each physical machine using Vagrant (configuration shown in Table I). For evaluation, TriangleCount used 1 GB input data, and the other five applications used 20 GB of input data each (listed in Table V). These six applications were executed in two separate groups, Group I included PageRank, WordCount, and KMeansClustering, and Group II included Connected-Components, TriangleCount and LogisticRegression. These applications were randomly assigned in each group. All three applications of a particular group were executed in parallel on the virtual cluster. In each group, these three applications were executed with different starting time T, which is randomly selected from the range $[0, 100)$ (listed in Table V). This was done to ensure that the stages that interfere across applications are unpredictable.

To compare against our approach, we executed each group of applications under five different conditions:

- CPU resource statically allocated at 600% (i.e., Res-600)
- CPU resource statically allocated at 300% (i.e., Res-300)
- CPU resource statically allocated at 200% (i.e., Res-200)
- CPU resource allocated using "Approach - I"
- CPU resource allocated using "Approach - II"

CPU resource was set to 600% to compare against the maximum available CPU resource on the virtual cluster. 300% and 200% CPU usage limit were chosen empirically as the average CPU usage fell within that range for these applications on the virtual cluster.

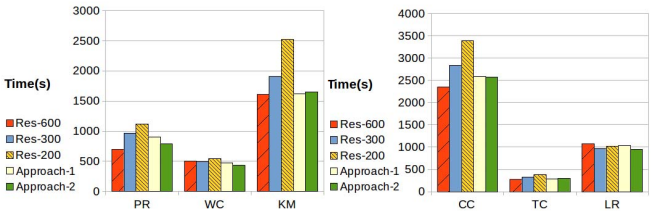
TABLE VI: Overhead and Δ for Multiple Applications ("Approach-I")

Application	Sample Time	*Delta*
PageRank	102s	0.7
WordCount	33s	0.5
K-MeansClustering	72s	0.5
ConnectedComponents	38s	0.6
TriangleCount	34s	0.6
LogisticRegression	45s	0.6

In case of Approach - I, we first ran the corresponding sample application for each application where the input data size of each application was set to 2GB except for TriangleCount which used 1GB of input data. We used this sample application trace to find the corresponding Δ. The execution time for each sample application and corresponding Δ are listed in Table VI.

The performance for each application under each condition is shown in Figure 4. We calculated total CPU resource quota using equation 14, and compared the total allocated resource quota under different conditions in Figure 5.

The performance for different approaches are listed in Table VII. Approach-I reduced resource requirement in all cases except for Res-200 condition for K-MeansClustering which increased by 1% whereas Approach-II reduced resource requirement in all cases. Approach-I improved performance in all but five conditions (i.e., (PR, Res-600); (CC, Res-600); (TC, Res-600); (LR, Res-300); (LR, Res-200)). Approach-II improved performance in all but four conditions (i.e., (PR, Res-600); (KM, Res-600); (CC, Res-600); (TC, Res-600)). Notably, while performance is reduced by at most 13% using Approach-II compared to the Res-600 condition, it led to significant reduction in resource requirements (i.e., reduction in resource requirements ranged between 49% to 84% in cases where the performance were affected negatively).

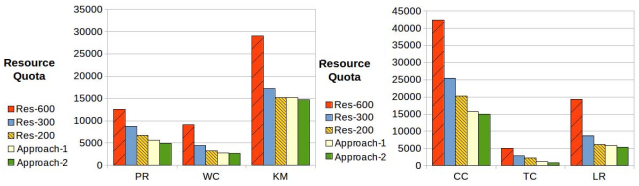


(a) GroupI (b) GroupII

Fig. 4: Performance Improvement for Multiple Concurrent Applications

C. Allocated Resource vs. Actual Usage

We collected the detailed trace of allocated CPU resources for each application, and compare that against the actual resource usage when executed under two different resource allocation approaches (i.e., Approach - I, Approach - II) for



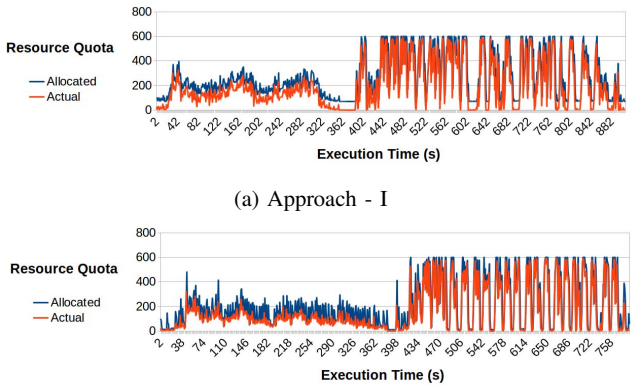
(a) GroupI (b) GroupII

Fig. 5: Resource Allocation for Multiple Concurrent Applications

TABLE VII: Performance Improvement and Resource Quota Reduction for Multiple Concurrent Applications

App.	Condition	Approach-I		Approach-II	
		Time	Resource	Time	Resource
PR	Res-600	-30%	55%	-13%	61%
	Res-300	6%	35%	19%	43%
	Res-200	19%	16%	29%	26%
WC	Res-600	6%	69%	13%	70%
	Res-300	4%	37%	12%	39%
	Res-200	13%	14%	20%	18%
KM	Res-600	0%	47%	-2%	49%
	Res-300	15%	11%	14%	14%
	Res-200	36%	-1%	35%	3%
CC	Res-600	-10%	63%	-9%	65%
	Res-300	9%	38%	9%	41%
	Res-200	24%	22%	24%	26%
TC	Res-600	-4%	77%	-6%	84%
	Res-300	10%	61%	8%	72%
	Res-200	24%	51%	23%	64%
LR	Res-600	4%	70%	12%	72%
	Res-300	-6%	33%	2%	39%
	Res-200	-1%	5%	7%	13%

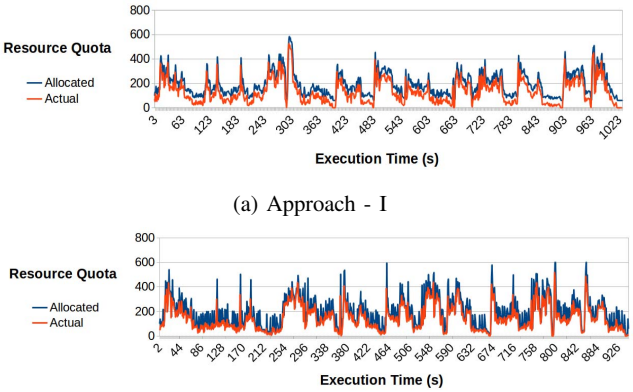
each time interval. Due to space constraint, we present the result for PageRank and LogisticRegression in Figure 6 and Figure 7 respectively that exhibit different resource usage patterns. As can be seen, the allocated CPU resource closely follows the actual CPU resource usage, and adapts to the pattern of CPU resource usage over time.



(a) Approach - I

(b) Approach - II

Fig. 6: PageRank



(a) Approach - I

(b) Approach - II

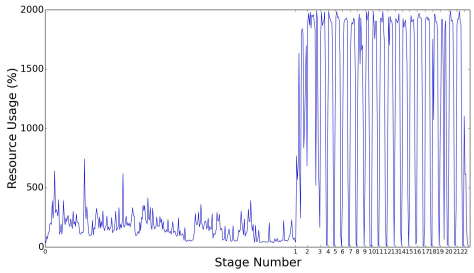
Fig. 7: LogisticRegression

D. Discussion

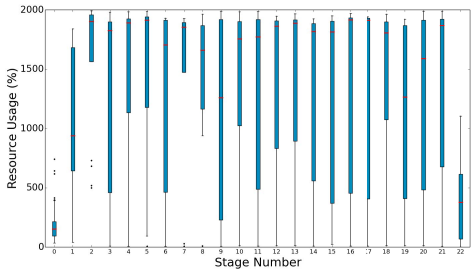
In our experiments, our presented approaches were able to improve application performance while reducing resource requirements. To better understand the underlying reasons behind such improvement, consider the example of PageRank application. As shown in Figure 8, the PageRank application contains 23 stages (i.e., phases defined by its inner computing pipeline). Among these, the first stage (labeled 0) is the longest as shown in Figure 8a. The median resource usage of Stage 0 is 152%, which is less than 500% allocated under Res-500 condition, and 300% allocated under Res-300 condition. This leads to resource wastage in Stage 0 under static allocation scheme. In contrast, our tool can allocate resources as needed, thereby avoiding over provisioning of resources. Furthermore, resource usage within each stage also exhibits significant variations over time, which is evident from the resource usage distribution per stage shown in Figure 8b. As such, allocating resources dynamically both within a stage and across multiple stages reduce the amount of allocated resources without sacrificing the application performance in our scheme.

V. Conclusion

In this paper we present a middleware service to dynamically allocate computing resources for distributed data processing applications running on a cluster. The presented strategy is oblivious to the application type and able to adjust resource quota based on changes in resource requirement throughout the lifetime of an application while incurring negligible runtime overhead. Experimental evaluation using multiple different types of Apache Spark applications demonstrates the effectiveness of our approach in improving application performance while reducing resource requirement. We believe that the presented approaches can be extended for other cloud platforms as well, and will help to improve resource utilization while reducing operating costs significantly in cloud settings.



(a) Stage Timeline



(b) Resource Usage Distribution

Fig. 8: Resource Usage for PageRank Application

REFERENCES

[1] 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 *NSDI*, vol. 11, no. 2011, 2011.

[2] B. Du, C. Wu, and Z. Huang, "Learning resource allocation and pricing for cloud profit maximization," in *The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, 2019.

[3] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[4] 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 *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[5] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 127–144.

[6] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167.

[7] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 452–465.

[8] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 199–208.

[9] M. Babaioff, Y. Mansour, N. Nisan, G. Noti, C. Curino, N. Ganapathy, I. Menache, O. Reingold, M. Tennenholtz, and E. Timnat, "Era: A framework for economic resource allocation for the cloud," in *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee, 2017.

[10] M. Shahrad, C. Klein, L. Zheng, M. Chiang, E. Elmroth, and D. Wentzlaff, "Incentivizing self-capping to increase cloud utilization," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017.

[11] C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-efficient provisioning in shared cloud systems," *Acm Sigplan Notices*, vol. 51, no. 4, pp. 473–488, 2016.

[12] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 427–444.

[13] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 3.

[14] A. Chung, J. W. Park, and G. R. Ganger, "Stratus: cost-aware container scheduling in the public cloud," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 121–134.

[15] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017.

[16] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker, "Monotasks: Architecting for performance clarity in data analytics frameworks," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 184–200.

[17] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya *et al.*, "Hydra: a federated resource manager for data-center scale analytics." in *NSDI*, 2019, pp. 177–192.

[18] M. Borkowski, C. Hochreiner, and S. Schulte, "Minimizing cost by reducing scaling operations in distributed stream processing," *Proceedings of the VLDB Endowment*, vol. 12, no. 7, pp. 724–737, 2019.

[19] J. Ortiz, B. Lee, M. Balazinska, J. Gehrke, and J. L. Hellerstein, "Slaorchestrator: reducing the cost of performance slas for cloud data analytics," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 547–560.

[20] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: heterogeneous cloud storage configuration for data analytics," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 759–773.

[21] C.-J. Hsu, V. Nair, T. Menzies, and V. Freeh, "Micky: A cheaper alternative for selecting cloud instances," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 409–416.

[22] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies, "Arrow: Low-level augmented bayesian optimization for finding the best cloud vm," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 660–670.

[23] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "Towards seamless configuration tuning of big data analytics," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1912–1919.

[24] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2019, pp. 158–164.

[25] Cgroups. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[27] Vagrant. [Online]. Available: https://www.vagrantup.com/

[28] Apache hadoop. [Online]. Available: https://hadoop.apache.org/

[29] Snap: Stanford network analysis project. [Online]. Available: http://snap.stanford.edu/

[30] Wikimedia downloads. [Online]. Available: https://dumps.wikimedia.org/

[31] Sloan digital sky survey. [Online]. Available: http://www.sdss.org/