Noname manuscript No.
(will be inserted by the editor)

Design and Implementation of an Analytical Framework for Interference Aware Job Scheduling on Apache Spark Platform

Kewen Wang
Mohammad Maifi Hasan Khan
Nhan Nguyen
Swapna Gokhale

the date of receipt and acceptance should be inserted later

Abstract Apache Spark is one of the recently popularized open-source platforms that is increasingly being used for large-scale data analytic applications. However, while performance prediction in such systems is important for efficient job scheduling and optimizing resource allocation, interference among multiple Apache Spark jobs running concurrently in a virtualized environment makes it extremely difficult, which is addressed in this paper. Towards that, first, we develop data-driven analytical models to estimate the effect of interference among multiple Apache Spark jobs on job execution time in virtualized cloud environments. Next, we present the design of an interference aware job scheduling algorithm leveraging the developed analytical framework. We evaluated the accuracy of our models using four real-life applications (e.g., Page rank, K-means, Logistic regression, and Word count) on a 6 node cluster while running up to four jobs concurrently. Our experimental results show that the scheduling algorithm reduces the average execution time of individual jobs and the total execution time significantly, and ranges between 47% to 26% for individual jobs and 2% to 13% for total execution time respectively.[1]

Keywords Apache Spark · Job Scheduling · Performance Interference Modeling · Execution Time Prediction

Department of Computer Science and Engineering
University of Connecticut
E-mail: kewen.wang@uconn.edu, maifi.khan@uconn.edu,nhan.q.nguyen@uconn.edu, ssg@uconn.edu

1 Introduction

With significant advancement in wired and wireless sensor technologies and wide adoption of Internet connectivity, researchers are exploring increasingly diverse, complex, and extensible dynamic data driven application systems (DDDAS) such as smart cities [19], smart grid monitoring [19], infrastructure and asset management [19], and environmental applications such as target tracking [31,15], volcanic ash propagation and hazard analysis [25,7,27], just to name a few. As the scale of the applications and the volume of data that needs to be stored and processed continue to grow, service providers are increasingly adopting cloud-based solutions (e.g., Apache Hadoop [2], Apache Spark [33]) to provide reliable and scalable service while maximizing the resource utilization and minimizing the operating cost.

Among different cloud computing platforms, Apache Spark [35] is one of the recently popularized open-source platforms that is currently used by over 500 organizations, including companies such as Amazon, eBay and Baidu[2]. Apache Spark leverages the concept of resilient distributed datasets (RDDs) [34] and in-memory computation to enable fast processing of large volume of data, making it suitable for large-scale data analytic applications. However, while performance prediction in such systems is important to optimize resource allocation [17] [11] [13], it is nontrivial for Apache Spark jobs for several reasons as follows. First, the execution time of a particular job on Apache Spark platform can vary significantly depending on the input data type and size, design and implementation of the algorithm, and computing capability (e.g., number of nodes, CPU speed,

[1] This paper is a significantly extended version of the authors' prior work [32,33] and includes the design and evaluation of interference aware job scheduling algorithms, which is not presented in prior efforts.

[2] http://spark.apache.org/faq.html

memory size), making it difficult to predict job performance. Second and finally, with advancement in hardware technology, virtualization technique is increasingly being used to share resources among applications [28]. However, while virtualization isolates multiple applications running on separate virtual machines, the interference among these applications still affects the execution performance. Due to the aforementioned factors, modeling performance of multiple Apache Spark jobs running in a virtualized environment concurrently is extremely challenging. While our own prior effort looked into the problem of performance prediction for a single job running on Apache Spark platform [32], that approach does not address the challenge of performance modeling for multiple jobs running in parallel on the same cluster.

To address this void, in this paper, we focus on modeling interference among multiple Apache Spark jobs, and predict the execution time of a job when interfered with other jobs. In contrast to hard to interpret machine learning approaches that are often used to predict system performance leveraging past system execution data, we apply analytical approach that can provide a better understanding regarding the observed behavior (e.g., execution slowdown), exposing the underlying interactions among multiple jobs [23] [22] [38] [18]. Specifically, we use a simulation job (an Apache Spark job implemented by us) to predict the slowdown ratio while running multiple jobs concurrently, and use the slowdown ratio to predict the execution time. As Apache spark jobs follow a multi-stage execution model (more details are discussed in Section 3) and different stages have different characteristics (e.g., I/O intensive vs. CPU intensive), our framework develops interference models for each stage, and predicts execution time for each stage separately. Finally, as concurrent Apache Spark jobs can heavily interfere, we design and implement a scheduler that automatically schedules and executes submitted Spark jobs leveraging the performance prediction framework, minimizing interference and reducing job execution time significantly.

We evaluated our framework with four real-world applications, namely, Page Rank, K-means clustering algorithm, Logistic regression, and Word Count application. We varied the number of concurrent jobs up to 4 and predicted execution time for individual stages. While the prediction accuracy for individual stages varied, it ranged between 86% to 99% when the number of concurrent jobs was four and all started simultaneously, and ranged between 71% to 99% when the number of concurrent jobs was four and started at different times. Furthermore, the scheduling algorithm reduced the average execution time of individual jobs and the total execution time (i.e., completion time of the last job minus the start time of the first job) significantly, and ranged between 47% to 26% for individual jobs and 2% to 13% for total execution time respectively.

The rest of the paper is organized as follows. Section 2 describes prior research that is related to our work. Section 3 presents the models that are used to predict job performance and the interference aware job scheduling algorithm. Section 4 presents the experimental results. Limitations of our current work and future directions are discussed in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

With the proliferation of cloud computing platforms, significant volume of prior work looked into the problem of performance modeling in cloud settings and distributed systems in general [26,6,37,16,20,30,9,24,21,14]. Among these, PREDIcT [26] looks into the problem of predicting runtime for network intensive iterative algorithms and focuses on Hadoop MapReduce platform. Starfish [16] leverages analytical approaches to predict job performance based on job simulation data. CloudScope [10] is one of the more recent efforts that employs a discrete-time Markov Chain model to predict the performance interference of co-resident applications by modeling an application as a sequence of job slices and estimating the probability of a job moving from one state to another considering different factors such as current workloads and slowdown. Matrix [12] utilizes machine learning methods to predict application performance on virtual machines by applying clustering methods to classify applications and predicts the performance of new applications by comparing against the previously trained models.

Interference modeling among multiple applications running on MapReduce framework is tried before as well for the purpose of efficient job scheduling [8] that requires training using different combinations of applications, which can quickly become prohibitive. MIMP [36] presents a progress aware scheduler for Hadoop framework that applies regression model to train and predict task completion time based on past execution. HybridMR [29] presents another MapReduce scheduler for hybrid data center consisting of physical and virtual machines. This scheduler uses performance interference models to guide resource allocation, and applies linear and non-linear exponential regression model to capture CPU, I/O, and memory interference.

While these prior efforts provide invaluable insight to the problem of performance modeling, however, most of them use black-box approaches and can not be extended easily without retraining. Moreover, due to the

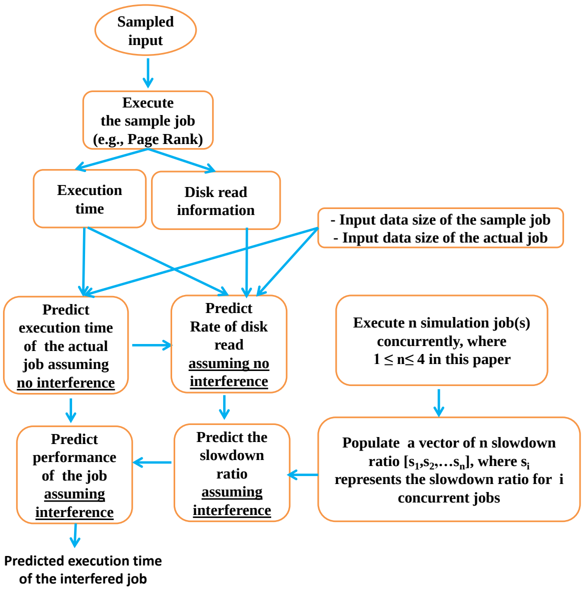
Sampled input
Execute the sample job (e.g., Page Rank)
Execution time
Disk read information
- Input data size of the sample job
- Input data size of the actual job
Predict execution time of the actual job assuming no interference
Predict Rate of disk read assuming no interference
Execute n simulation job(s) concurrently, where 1 ≤ n≤ 4 in this paper
Predict performance of the job assuming interference
Predict the slowdown ratio assuming interference
Populate a vector of n slowdown ratio [s1,s2,...sn], where si represents the slowdown ratio for i concurrent jobs
Predicted execution time of the interfered job

Fig. 1: Performance Prediction for Interfered Jobs

multi-stage execution model and in-memory computation feature of Apache Spark platform, it is non-trivial to apply these approaches without further modifications for predicting the effect of interference on job execution time. As such, we focus on developing data-driven analytical models for modeling interference among multiple Apache Spark jobs which is complementary to prior efforts.

3 Overview

Each Apache Spark job typically consists of multiple execution stages where each stage implements certain operations and is executed sequentially. Moreover, to facilitate parallel processing, input data set is partitioned into multiple sets and are distributed over multiple worker nodes. Within each worker node, multiple batches of tasks are launched to process the corresponding partition of the input data. The number of tasks within each node is determined based on the size of the input data and configuration settings of the program. For example, if the input data size of the PageRank job is 2.5 GB, the total number of input blocks will be 40 for a default block size of 64 MB. As the number of tasks is equal to the number of input blocks and the number of tasks in each stage is same within one Spark job, there will be 40 tasks in each stage. However, different CPU core may complete different number of tasks due to the differences in computing ability and uncertainty during the program execution.

Given the above multistage execution model, the main idea behind our work is as follows (Figure 1). First, for a given Apache Spark job, we predict the execution time for each stage leveraging the performance model developed based on the performance of the actual job with smaller input data set. Note that this model is presented in our prior work [32], and assumes that there are no interference in the system from other jobs. Next, we estimate the slowdown ratio for a given number of jobs running concurrently by executing our simulation job, which is implemented by us (more details in Section 4) and is different than any of the four jobs that we used for evaluation. However, as the slowdown ratio due to interference among simulated jobs can be different compared to the actual jobs, for a given job, we adjust the expected slowdown ratio by taking into account the actual job parameters such as input data size and disk I/O characteristics. Once we estimate the expected slowdown ratio, we estimate the execution time considering the interference. For completeness, we first briefly present the model that is used to predict execution time assuming no interference from our earlier work [32], and then present the model for predicting the slowdown ratio due to interference that allows us to predict the execution time in the presence of interference among multiple jobs.

3.1 Model for Estimating Execution Time

As an Apache Spark job is executed in multiple stages where each stage contains multiple tasks, we use the following notation to represent an Apache Spark job.

$$Job = \{Stage_i \mid 0 \leq i \leq M\} \tag{1}$$

$$Stage_i = \{Task_{i,j} \mid 0 \leq j \leq N\} \tag{2}$$

Here M is the number of stages in a job and N is the number of tasks in a stage.

Next, as different stages within a job are executed sequentially, we represent the execution time of a job as the sum of the execution time of each stage plus the job startup time and the job cleanup time as follows.

$$JobTime = Startup + \sum_{s=1}^{M} StageTime_s + Cleanup \tag{3}$$

Next, within each stage, as one CPU core executes one task at a time, in a cluster with H worker nodes, the number of tasks P that can run in parallel is calculated as follows.

$$P = \sum_{h=1}^{H} CoreNum_h \tag{4}$$

Here $CoreNum_h$ is the number of CPU cores of working node h and H is the number of working nodes in the cluster. Hence, within an execution stage, tasks in each stage are executed in batches where each batch consists of P tasks running in parallel. However, due to the differences in computing capabilities among different worker nodes in a cluster and inherent uncertainty in program execution, the execution time of different tasks may vary significantly. Therefore, the time spent in a particular stage can be calculated as the maximum of the sum of all the sequential tasks' time within a stage plus the stage startup time and the stage cleanup time as follows.

$$\begin{aligned} StageTime = Startup + \max_{c=1}^{P} \sum_{i=1}^{K_c} TaskTime_{c,i} \\ + Cleanup \end{aligned} \tag{5}$$

Here P is the number of total CPU cores, and K_c is the number of sequential tasks executed on CPU core c.

Finally, as different tasks in a stage follow the same execution pattern, the execution time of a task can be computed as follows.

$$\begin{aligned} TaskTime = DeserializationTime + RunTime \\ + SerializationTime \end{aligned} \tag{6}$$

Here $DeserializationTime$ is the time taken to deserialize the input data, $SerializationTime$ is the time taken to serialize the result, and $RunTime$ is the actual time spent performing operations on data such as data mapping, filtering, calculation, and analyses. Based on the above model, to predict job performance, the presented framework first executes the program on a cluster using limited amount of sample input data and collects performance metrics such as run time during the simulated run.

Next, to predict the execution time of the actual run based on the extracted performance metric from simulated run, we first calculate the number of tasks N where $N = InputSize/BlockSize$, $InputSize$ is the size of the input data, and $Blocksize$ is the size of one data block in HDFS. The tasks within a stage are scheduled to run batch by batch, and the number of tasks P in each batch is computed as shown in equation (4). In one batch of tasks, while the tasks may start simultaneously, they may not finish at the same time due to various factors such as data skew problem, and differences in computing capability of different worker nodes. Hence, using sampled data, we calculate the average execution time for a task for a given stage for a worker node h as follows.

$$\begin{aligned} TaskRunTime_{h,i} &= DeserializeTime_{h,i} \\ &\quad + RunTime_{h,i} \\ &\quad + SerializationTime_{h,i} \end{aligned} \tag{7}$$

$$AvgTaskTime_h = \frac{1}{n_h} \sum_{i=1}^{n_h} TaskRunTime_{h,i} \tag{8}$$

Here n_h is the number of tasks running in host h in a particular stage of the sample job. Moreover, during our experiment, we observed that the average execution time of the first batch is significantly different compared to the subsequent batches within the same stage, which we capture as follows.

$$Ratio_h = \frac{\frac{1}{n_h - P_h} \sum_{i=P_h+1}^{n_h} TaskTime_{h,i}}{\frac{1}{P_h} \sum_{j=1}^{P_h} TaskTime_{h,j}} \tag{9}$$

Here n_h is the number of tasks running in host h, and P_h is the number of tasks in the first batch. As tasks execute on different hosts in parallel, to predict the execution time for a particular stage during actual execution, stage $Startup$ time and $Cleanup$ time are viewed as constants which are extracted from simulation logs, and stage execution time is estimated as follows.

$$\begin{aligned} EstStageTime = Startup + \max_{c=1}^{P} \sum_{i=1}^{K_c} AvgTaskTime_{c,i} \\ + Cleanup \end{aligned} \tag{10}$$

$$EstTaskTime_{c,i} = \begin{cases} AvgTaskTime_c, & i = 1 \\ AvgLaterTaskTime_c, & i > 1 \end{cases} \tag{11}$$

Here P is the number of total CPU cores calculated in equation (4) and K_c is the number of sequential tasks running on CPU core c. $AvgTaskTime_c$ is the average time of the tasks of the first batch running on CPU core c for the corresponding host, which is calculated in equation (8). $AvgLaterTaskTime_c$ is the average time of the tasks of the following batches, which is calculated as $Ratio_h \times AvgTaskTime_h$.

While we can apply the prediction model presented in this section to estimate the execution time for a single job assuming no interference [32], we still need a way to predict the slowdown ratio when interfered with other jobs, which we address as follows.

3.2 Modeling Interference

As different stages of a job is expected to have different characteristics in terms of resource utilization (e.g., CPU, I/O, memory), different stages of multiple jobs running concurrently on a system is expected to result in different interference patterns, affecting the execution time differently. Based on this observation, we model the slowdown ratio due to interference among multiple jobs for each stage separately. Towards that, in our model, each stage is represented as a vector consisting of execution time, CPU usage, disk I/O rate, and network I/O rate as follows.

$$Res_i = (RunTime_i, CPU_i, DiskIO_i, NetIO_i) \tag{12}$$

Here $1 \leq i \leq M$, and M is the number of stages in a job. Memory was not one of the bottleneck resources in our case. As such, we only considered Disk I/O and did not consider memory utilization in our model, which can be incorporated if needed for certain platforms/scenarios. Next, the slowdown resulting from interference with other applications for a particular stage is represented as follows.

$$SlowdownRatio(Stage_{i,k}) = f(Res_{i,k}, Res_{Otherjobs}) \tag{13}$$

Here $1 \leq k \leq J$, $1 \leq i \leq M$, J is the number of jobs running in parallel, and M is the number of stages in the Apache Spark job. $Res_{Otherjobs}$ represents the resources consumed by other jobs that are running concurrently with $Stage_{i,k}$. Simply put, this slowdown ratio is the ratio between execution time with interference over execution time without interference for a particular stage. Hence, once we estimate the value of the slowdown ratio and the expected execution time when there is no interference, we can estimate the execution time if there are interference with other jobs.

As the slowdown happens primarily due to contention for bottleneck resources in the system, to better understand the underlying reasons behind the slowdown, we ran a series of experiments and collected job event logs and resource consumption data, and then extracted the resource usage profile for each stage. Job event log is generated by Apache Spark platform, and resource consumption data is collected using system monitoring tool dstat [1]. Apache Spark log records the time line of different stages of a running job, which was used to determine the submission and completion time of different stages of a job. The resource usage for different stages of a job is represented as below:

$$CPU_i = (CPUusr_i, CPUsys_i, CPUidle_i, CPUwait_i) \tag{14}$$

$$DiskIO_i = (RateofDiskRead_i, RateofDiskWrite_i) \tag{15}$$

$$NetIO_i = (RateofNetReceive_i, RateofNetSend_i) \tag{16}$$

Here $1 \leq i \leq M$, and M is the number of stages in a Apache Spark job. As an Apache Spark job uses in-memory data processing to reduce execution time, in the first stage of a job, it reads the input data to memory, and then analyzes the in-memory data in the subsequent stages. Due to this characteristic, in the first stage, frequent I/O is expected, leading to longer I/O wait. Based on this observation, as bulk of the disk I/O happens in the first stage, in our model, we calculate the slowdown ratio for the first stage only, and assume that the slowdown ratio in cases where the first stage interferes with the following stages from another job is 1.0 (i.e., the slowdown due to interference is expected to be minimal). Note that, while this assumption is not accurate for certain jobs and stages, the error introduced due to this assumption in prediction accuracy is not significant in our case.

As most of the time spent in the first stage is due to reading data from disk to memory, we represent the relationship between the amount of data read in the first stage (e.g., size of input data), the rate of disk read, and the execution time of the first stage as follows:

$$RunTime_{Stage1} = c \times \frac{InputDataSize}{RateofDiskRead_{Stage1}} \tag{17}$$

Now, if we assume that we execute the same job twice, once with the reduced input data set (i.e., sample job) and once with the complete input data set (i.e., complete job), from equation 17, we can have the following. The word *Int.* refers to *Interference* in the following equations.

$$\begin{gathered}\frac{InputDataSize_{Samplejob}}{InputDataSize_{Completejob}} = \\ \frac{RateofDiskRead_{Samplejob,Stage1}}{RateofDiskRead_{Completejob,Stage1}} \times \\ \frac{RunTime_{SampleJob,Stage1}}{RunTime_{CompletejobwithoutInt.,Stage1}}\end{gathered} \tag{18}$$

Based on equation 18, we can have the following equation for predicting the rate of disk read for a complete job.

$$\begin{gathered}PredictedRateofDiskRead_{CompleteJobwithoutInt.,Stage1} = \\ RateofDiskRead_{SampleJob,Stage1} \times \\ \frac{RunTime_{SampleJob,Stage1}}{RunTime_{CompleteJobwithoutInt.,Stage1}} \\ \times \frac{InputSize_{CompleteJob}}{InputSize_{SampleJob}}\end{gathered} \tag{19}$$

In the above equation, we can estimate the value of $RunTime_{CompleteJobwithoutInt.,Stage1}$ using the model described in Section 3.1 [32]. Once we predict the rate of disk read for a complete job with no interference, next, we need to model the relation between the rate of disk read and the slowdown ratio when there is interference. For that, first, we run a simulation program (written by us as described in Section 4) to collect the runtime information with and without interference and calculate the parameter β_n as follows.

$$\beta_n = \frac{1}{RateofDiskRead_{SimulationRunwithoutInt.,Stage1}} \times \\ (\frac{RunTime_{SimulationRunwithInt.fornjobs,Stage1}}{RunTime_{SimulationRunwithoutInt.,Stage1}} - \\ \left\lfloor \frac{RunTime_{SimulationRunwithInt.fornjobs,Stage1}}{RunTime_{SimulationrunwithoutInt.,Stage1}} \right\rfloor) \tag{20}$$

In this paper, we assume that there can be at most 4 concurrent jobs in a system, and varied n between 2 to 4 to calculate β_2, β_3, and β_4. Running the simulation job and calculating β_n take only few minutes and need to be done only once for a given environment. Next, we use β_n to estimate the slowdown ratio when there are n concurrent jobs in the system as follows.

$$SlowdownRatio(Stage_{(1,k)}) = \\ \frac{RunTime_{CompleteJobwithInt.,Stage1}}{RunTime_{CompleteJobwithoutInt.,Stage1}} = \beta_n \\ \times PredictedRateofDiskRead_{CompleteJobwithoutInt.,Stage1} \\ + \left\lfloor \frac{RunTime_{SimulationrunwithInt.,Stage1}}{RunTime_{SimulationRunwithoutInt.,Stage1}} \right\rfloor \tag{21}$$

3.3 The Cascading Effect

Given the above formulation, if we assume that all the jobs are of same type and start at the same time, modeling interference is straightforward as they all have the same execution behavior in each stage. However, for interference among different types of jobs possibly starting at different times, this is a bit more complicated due to the possible cascading effect. For example, the slowdown of stage 1 of job A may push this stage to interfere with stage 2 of job B. Hence, a dynamic interference estimation algorithm is designed to solve this problem. The main idea behind the algorithm is as follows. First, the algorithm uses the execution time line of each job as input, and calculates the slowdown ratio for each stage of different jobs within the same time slot, and generates the execution time line of each job under interference condition. Based on that, the algorithm identifies the job that will finish its first stage the earliest, removes that job from the list, and recalculates the effect of interference for the remaining jobs for the remainder of the execution time. The algorithm applies this repeatedly until the list becomes empty. This dynamic interference estimation algorithm is described in Algorithm 1 (Appendix).

3.4 Interference Aware Job Scheduling

Finally, as concurrent Apache Spark jobs often heavily interfere, especially at the first stage, to minimize interference and job execution time, we design and implement a scheduler that automatically schedules and executes submitted Spark jobs leveraging the performance prediction framework presented earlier. Specifically, when a new job arrives in the system, if there is no existing job in the system, the scheduler locates available servers that can execute the job and starts the job immediately. However, if there are existing jobs running in the system with possibly more jobs waiting in the queue, the scheduler calculates the waiting time (if any) of the new job and readjusts the waiting time of the jobs that are already in the queue (if needed) to determine the best scheduling plan and updates the scheduling file accordingly (Note that jobs are not executed on first-come-first serve basis in our system).

The process of calculating the waiting time for a job in multiple steps is illustrated in Figure 2. Here we assume that the first job J_1 is submitted at time point T_1, and is started immediately as there is no other job in the system. The second job J_2 is submitted at time T_2. At that time, the scheduling algorithm calculates the amount of time already executed by J_1 (i.e., $T_2 - T_1$) to decide whether J_2 can be started immediately or needs to wait to avoid interference. If J_2 needs to wait, the algorithm calculates the tentative wait time for J_2 based on the interference model, which is ΔT, and updates the scheduling file by writing the waiting period ΔT for J_2.

Now, let us assume that, at time point T_3, the third job J_3 arrives in the system. At that point, the algorithm first calculates the execution time (which is different than the wall clock time) of the two jobs already running in the system. As J_1 executed alone before S_1 and executed concurrently with J_2 between S_1 and T_3, we calculate the execution time of J_1 since the last job submission time (i.e., T_2) as $ExeTime_{J1} = (S_1 - T_2) + (T_3 - S_1) \times Ratio_{J_1}$, and the execution time for J_2 as $ExeTime_2 = (T_3 - S_1) \times Ratio_{J_2}$, where $Ratio_{J_1}$

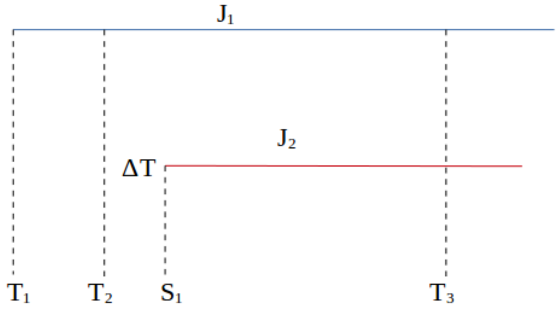

Fig. 2: A Scheduling Example

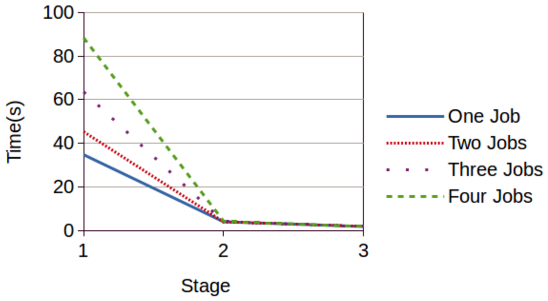

Fig. 3: Execution Time for Different Number of Simulation Jobs

and $Ratio_{J_2}$ are the slow down ratio of J_1 and J_2 respectively when interfered with one job. Note that the slowdown ratios are different for different jobs due to differences in job characteristics. Also, as the job profiles in the scheduling file are updated every time a new job is submitted, we only need to estimate the execution time for each running job since the last job submission time.

After calculating the execution times for these two jobs (which tell us what stage each job is at currently), we update the job profiles for J_1 and J_2, and then decide whether J_3 can start immediately or not, based on the possibility of interference with the currently running jobs.

While calculating the execution time, we have to consider the possibility that each job may interfere with different jobs at different points in time during the execution. To handle this possibility, the algorithm saves the start and end time point of the first stage for each job (as the significant interference happens in the first stage of a job), and sorts the list based on start time points, and determines which set of jobs interfere at a particular point in time, and calculates the execution time incrementally.

The algorithm to determine the waiting time for a job is shown in Algorithm 2 (Appendix) which has two main parts. The first part of the algorithm consists of the function depicted in Algorithm 3 (Appendix) that calculates the execution time of the previously submitted jobs and updates the job schedule file (Algorithm 4 (Appendix)). The second part consists of Algorithm 5(Appendix) that searches the combination of waiting times and job schedules that will minimize the total execution time.

4 Evaluation

To evaluate the accuracy of our modeling framework and the performance of the job scheduling algorithm, we used a cluster of 6 machines. Each machine had 8 CPU cores (Intel(R) Xeon(R) CPU E5620, 2.40GHz) and 22 GB of RAM memory where each machine hosted 4 virtual machines. We used Xen hypervisor [5] to create up to four virtual machines on each physical machine. Each virtual machine was configured with 4GB of RAM memory and 1 CPU core. For the deployed Apache Spark platform, one machine served as the master node, and the remaining five machines served as the working nodes. We created multiple clusters leveraging virtual machines to execute multiple Apache Spark jobs in parallel.

In our evaluation, for prediction, first, we need to estimate the parameter β_n in equation (20). Towards that, we implemented our own Apache Spark job and executed that on our cluster to obtain the execution time and resource consumption information. This simulation job consists of three stages executing distinct(), groupByKey(), and count() operation respectively. Distinct() implements a mapping function and parses the input data, groupByKey() processes the output of distinct() operation, and count() is a CPU intensive operation performing data summarization. This simulation job is executed with 2.5 GB of sample data where the first stage implementing the Distinct() operation involves significant I/O compared to the following two stages. To measure β_n, we executed n (n=1,2,3,4) instances of this simulation job in parallel. As shown in Figure 3, the effect of interference is significant for the first stage but minimal for the subsequent stages.

Once we estimated the value of β_n, subsequently, we used our formulation to predict the execution time for each stage of each job separately considering different execution scenarios and added up the prediction error for each stage to calculate the total prediction accuracy R as below.

$$R = |1 - \frac{\sum_{i=1}^{M} |PredictedTime_i - MeasuredTime_i|}{\sum_{j=1}^{M} MeasuredTime_j}|$$

Table 1: Prediction Accuracy for Interference Among Same Jobs

JobName	Job Number	First Stage	Whole Job
PR	2	0.97	0.80
	3	0.96	0.85
	4	0.92	0.82
KM	2	0.75	0.70
	3	0.71	0.68
	4	0.98	0.92
LR	2	0.74	0.78
	3	0.79	0.81
	4	0.97	0.97
WC	2	0.87	0.86
	3	0.96	0.94
	4	0.95	0.94

Table 2: Execution Time Prediction for Four Interfered PageRank Jobs

StageNo	1	2	3	4	5	6
Actual Time (s)	652.7	37.3	32.3	23.9	23.2	58.8
Predicted Time (s)	594.3	43.0	10.6	5.4	5.0	28.3

Table 3: Execution Time Prediction for Four Interfered K-Means Jobs

StageNo	1	2	3	4	5	6	7	8	9	10	11	12
Actual Time (s)	688.4	7.1	19.9	8.1	16.5	7.4	15.9	7.2	15.8	7.0	15.6	7.0
Predicted Time (s)	708.3	7.4	25.2	18.1	22.9	9.7	23.6	8.4	23.2	9.2	24.8	6.2

Table 4: Execution Time Prediction for Four Interfered Logistic Regression Jobs

StageNo	1	2	3	4	5	6	7	8	9	10
Actual Time (s)	649.6	7.3	7.5	7.3	7.4	7.2	7.5	7.2	7.5	7.4
Predicted Time (s)	669.2	7.1	7.2	7.6	6.6	7.0	6.5	7.9	6.9	6.3

Table 5: Execution Time Prediction for Four Interfered WordCount Jobs

StageNo	1	2
Actual Time (s)	598.8	66.8
Predicted Time (s)	631.8	77.8

(22)

Here M is the number of stages in a job, $PredictedTime_i$ is the predicted execution time for $stage_i$, and $MeasuredTime_i$ is the actual execution time of $stage_i$. Different evaluation scenarios are presented below.

4.1 Interference Among Multiple Jobs of the Same Type Starting Simultaneously

In a shared cluster, multiple Apache Spark jobs running on separate virtual machines hosted on the same physical machine can cause significant interference, impacting the performance of each job. However, as different jobs may have different execution patterns and resource requirements, estimating the effect of interference on performance is nontrivial. As such, to demonstrate the generalizability of our work, we validated our model using four different Apache Spark jobs, namely, PageRank, K-Means, Logistic Regression and WordCount. The jobs vary in terms of the number of stages, the number of tasks, and the library functions they use to implement the job. For example, WordCount counts the word frequency for a given text file. K-Means implements a clustering algorithm, and Logistic Regression implements the Logistic Regression algorithm, which are examples of machine learning jobs. Finally, PageRank is an example of graph analyzing and processing jobs. For testing, we used the LiveJournal network dataset from SNAP [4] for PageRank, which is processed through mapping each node id onto a longer string to form a 20 GB input data set. K-Means and Logistic Regression used 20 GB of numerical Color-Magnitude Diagram data of galaxy from Sloan Digital Sky Survey (SDSS) [3]. WordCount application used 20 GB of Wikipedia dump data.

In this part of the evaluation, we present the accuracy of prediction while modeling the effect of interference among multiple jobs of the same type (e.g., interference between n instances of job x).

For prediction, we first executed the sample job (e.g., Page rank) with 2.5 GB of input data which was extracted from the actual input data to collect the job execution profile. We ensured that no other job was running during this execution. The collected execution trace was then used to predict the execution time assuming no interference. Finally, we used our framework to adjust the prediction assuming interference. The prediction accuracy is summarized in Table 1. In the table, PR, KM, LR, and WC refers to PageRank, K-Means, Logistic Regression, and WordCount application respectively. Column Job number (e.g., 2, 3, 4) indicates the number of jobs that were executed in parallel. For instance, a value of 2 indicates that two instances of the same job were executed in parallel. As can be seen, prediction accuracy is highest for Logistic regression application (97%) and lowest for K-means (68%).The predicted execution time and the actual execution time when we executed four instances of the same job in parallel are shown in Table 2, Table 3, Table 4, and Table 5 for PageRank, K-Means, Logistic Regression, and WordCount respectively.

The predicted execution time and the actual execution time when we executed four different jobs in parallel are shown in Table 6, Table 7, Table 8, and Table 9 for PageRank, K-Means, Logistic Regression, and WordCount respectively.

4.2 Interference Among Multiple Jobs of Different Types Starting Simultaneously

In this section, we present the accuracy of prediction while modeling the interference among n different jobs concurrently, where n was varied from 2 to 4. For example, when $n = 2$, we execute two different jobs concurrently. The prediction accuracy while running two different jobs in parallel is summarized in Table 10. As shown in the table, there are a total of 6 combinations to consider. As can be seen, prediction accuracy ranges between 97% and 69% for the whole job, and between 99% and 70% for the first stage, which incurs the bulk of the execution time.

For n=3, we execute three different jobs concurrently. The prediction accuracy while running three different jobs in parallel is summarized in Table 11. As shown in the table, there are a total of 4 combinations to consider. As can be seen, prediction accuracy ranges between 90% and 79% for the whole job, and between 99% and 83% for the first stage.

Finally, for n=4, we execute four different jobs concurrently. The prediction accuracy while running four different jobs in parallel is summarized in Table 12. As shown in the table, there was only one combination to consider. As can be seen, prediction accuracy ranges between 99% and 86% for the whole job, and between 99% and 92% for the first stage.

4.3 Interference Among Multiple Jobs Starting at Different Times

Finally, to test the prediction accuracy of our model where different jobs may arrive and start at different times, we use the four Apache Spark jobs and input data set as before, and start them randomly at different times. To ensure that each job will interfere with at least one other job while executing, we set the starting time for each job as $startingTime \in [minstagetime/10, minstagetime/2]$, where $minstagetime$ represents the smallest execution time for the first stage among all the jobs. In our case, $minstagetime = 190sec$, causing the starting time for different jobs to be between 19 sec and 95 sec.

Given the above range, for evaluation, we randomly pick one job and start at time 0, and then set the starting time for the remaining three jobs between 19 sec and 95 sec randomly. We considered four scenarios where the starting job is different in each scenario. The prediction accuracy for the whole job while running four different jobs in parallel starting at different times is summarized in Table 13. As shown in the table, in our evaluation, prediction accuracy ranges between 99% and 71% for the whole job, and between 99% and 72% for the first stage. The predicted execution time and the actual execution time for PageRank, K-Means, Logistic Regression, and WordCount under Scenario - I are shown in Table 14, Table 15, Table 16, and Table 17 respectively.

Table 6: Execution Time Prediction for PageRank Job Interfered with Other Three Jobs

StageNo	**1**	**2**	**3**	**4**	**5**	**6**
Actual Time (s)	646.4	36.9	30.4	22.3	21.8	26.9
Predicted Time (s)	594.3	43.0	10.6	5.4	5.0	28.3

Table 7: Execution Time Prediction for K-Means Job Interfered with Other Three Jobs

StageNo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Actual Time (s)	657.4	7.1	20.6	9.6	18.1	7.3	17.8	7.0	17.4	7.1	17.7	7.1
Predicted Time (s)	657.4	7.4	25.2	18.1	22.9	9.7	23.6	8.4	23.2	9.2	24.8	6.2

Table 8: Execution Time Prediction for Logistic Regression Job Interfered with Other Three Jobs

StageNo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Actual Time (s)	651.9	7.3	7.4	7.1	7.9	7.1	7.3	7.2	7.7	7.1
Predicted Time (s)	646.0	7.1	7.2	7.6	6.6	7.0	6.5	7.9	6.9	6.3

Table 9: Execution Time Prediction for WordCount Job Interfered with Other Three Jobs

StageNo	**1**	**2**
Actual Time (s)	660.2	40.2
Predicted Time (s)	622.2	77.8

Table 10: Prediction Accuracy for Two Different Jobs

JobName	**Interfered Job**	**First Stage**	**Whole Job**
PR	KM	0.91	0.79
	LR	0.93	0.81
	WC	0.99	0.85
KM	PR	0.89	0.80
	LR	0.80	0.73
	WC	0.75	0.69
LR	PR	0.97	0.97
	KM	0.73	0.77
	WC	0.70	0.75
WC	PR	0.96	0.87
	KM	0.93	0.84
	LR	0.96	0.88

4.4 Performance of Interference Aware Job Scheduling Algorithm

To evaluate the performance of the job scheduling algorithm, we used the four Apache Spark jobs as before (e.g., PageRank (PR), K-Means (KM), Logistic Regression (LR) and WordCount (WC)). The order of job submission was varied in different experiments and the submission time was randomly chosen between 0 to 100 sec. The performance of our algorithm is compared against the default condition where each job is started immediately after submission.

In the first experiment, the four jobs were submitted in the order of PR, KM, LR, WC, and the input data size was set to 20GB for each of them. PR was submitted at $0s$, KM was submitted at $16s$, LR was submitted at $47s$, and WC was submitted at $94s$. As shown in Figure 4, the average execution time of individual jobs and total execution time (i.e., completion time of the last job minus the start time of the first job) were reduced by 47% and 10% respectively.

In the second experiment, the four jobs were submitted in the same order (PR, KM, LR, WC), but the input data size were set to 20GB for PR, 15GB for KM, 10GB

Table 11: Prediction Accuracy for Three Different Jobs

JobName	**Interfered Jobs**	**First Stage**	**Whole Job**
PR	KM, LR	0.96	0.87
	KM, WC	0.99	0.90
	LR, WC	0.99	0.90
KM	PR, LR	0.84	0.79
	PR, WC	0.92	0.87
	LR, WC	0.83	0.80
LR	PR, KM	0.84	0.85
	PR, WC	0.87	0.88
	KM, WC	0.83	0.84
WC	PR, LR	0.93	0.87
	PR, KM	0.93	0.87
	KM, LR	0.94	0.89

Table 12: Prediction Accuracy for Four Different Jobs

JobName	**Interfered Jobs**	**First Stage**	**Whole Job**
PR	KM, LR, WC	0.92	0.86
KM	PR, LR, WC	0.99	0.95
LR	PR, KM, WC	0.99	0.99
WC	PR, KM, LR	0.95	0.90

Table 13: Prediction Accuracy for Interference Among Different Jobs Starting at Different Times

Run	**JobName**	**Starting Time(s)**	**First Stage**	**Whole Job**
Scenario - I	PR	0	0.91	0.81
	KM	38	0.99	0.94
	LR	26	0.94	0.94
	WC	78	0.83	0.82
Scenario - II	PR	91	0.90	0.82
	KM	0	0.79	0.77
	LR	48	0.87	0.88
	WC	53	0.99	0.93
Scenario - III	PR	20	0.99	0.90
	KM	87	0.98	0.91
	LR	0	0.84	0.85
	WC	48	0.98	0.91
Scenario - IV	PR	77	0.93	0.85
	KM	25	0.72	0.71
	LR	86	0.99	0.99
	WC	0	0.99	0.93

Table 14: Execution Time Prediction for PageRank Job in Scenario - I

StageNo	**1**	**2**	**3**	**4**	**5**	**6**
Actual Time (s)	575.3	36.4	33.7	24.1	22.5	55.8
Predicted Time (s)	522.9	43.0	10.6	5.4	5.0	28.3

Table 15: Execution Time Prediction for K-Means Job in Scenario - I

StageNo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Actual Time (s)	611.8	6.9	21.0	8.2	18.0	7.5	17.6	7.2	17.5	7.1	17.3	6.9
Predicted Time (s)	610.7	7.4	25.2	18.1	22.9	9.7	23.6	8.4	23.2	9.2	24.8	6.2

Table 16: Execution Time Prediction for Logistic Regression Job in Scenario - I

StageNo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Actual Time (s)	576.8	7.4	7.3	7.4	7.2	7.1	7.8	7.3	7.2	7.1
Predicted Time (s)	617.1	7.1	7.2	7.6	6.6	7.0	6.5	7.9	6.9	6.3

Table 17: Execution Time Prediction for WordCount Job in Scenario - I

StageNo	**1**	**2**
Actual Time (s)	615.2	61.8
Predicted Time (s)	506.5	77.8

for LR and 5GB for WC. PR was submitted at $0s$, KM was submitted at $59s$, LR was submitted at $88s$, and WC was submitted at $97s$. As shown in Figure 5, the average execution time of individual jobs and total execution time were reduced by 34% and 13% respectively.

In the third experiment, the four jobs were submitted in the order of KM, WC, PR, LR, and the input data size were set to 10GB for KM, 20GB for WC, 15GB for PR and 5GB for LR respectively. KM was submitted at $0s$, WC was submitted at $30s$, PR was submitted at $51s$, and LR was submitted at $71s$. As shown in Figure 6, the average execution time of individual jobs and total execution time were reduced by 26% and 2% respectively.

In the fourth experiment, the four jobs were submitted in the order of LR, WC, KM, PR, and the input data size were set to 15GB for LR, 10GB for WC, 20GB for KM and 15GB for PR respectively. LR was submitted at $0s$, WC was submitted at $8s$, KM was submitted at $53s$, and PR was submitted at $64s$. As shown in Figure 7, the average execution time of individual jobs and total execution time were reduced by 39% and 8% respectively.

In the fifth and last experiment, the four jobs were submitted in the order of WC, PR, LR, KM, and the input data size were set to 20GB for WC, 15GB for PR, 10GB for LR and 10GB for KM respectively. WC was submitted at $0s$, PR was submitted at $7s$, LR was submitted at $27s$, and KM was submitted at $71s$. As shown in Figure 8, the average execution time of individual jobs and total execution time were reduced by 40% and 8% respectively.

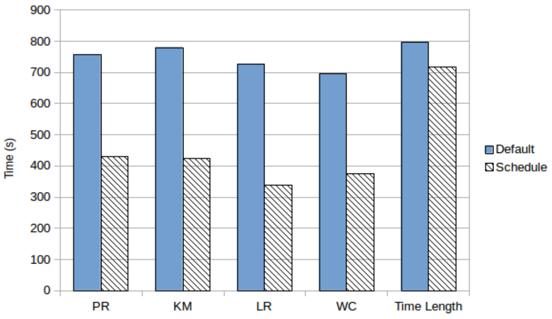

Fig. 4: Result of Scheduling Experiment 1. Column Time length represents the total execution time (completion time of the last job minus the start time of the first job).

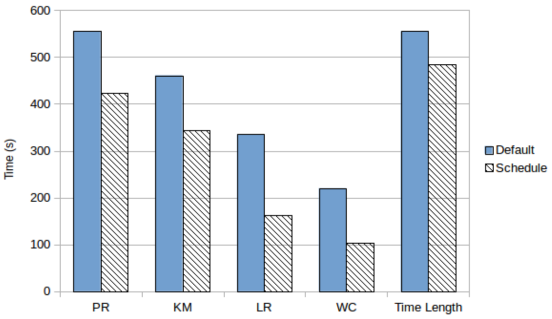

Fig. 5: Result of Scheduling Experiment 2. Column Time length represents the total execution time (completion time of the last job minus the start time of the first job).

5 Discussion

While our framework can predict performance degradation due to interference among multiple Apache Spark jobs with high accuracy and reduces the average execution time for individual jobs significantly, we do ac-

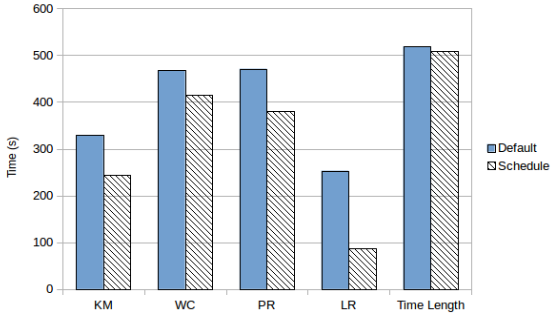

Fig. 6: Result of Scheduling Experiment 3. Column Time length represents the total execution time (completion time of the last job minus the start time of the first job).

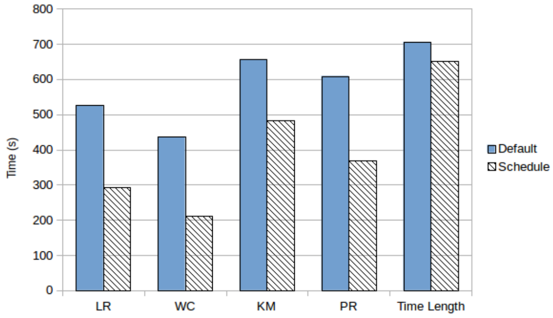

Fig. 7: Result of Scheduling Experiment 4. Column Time length represents the total execution time (completion time of the last job minus the start time of the first job).

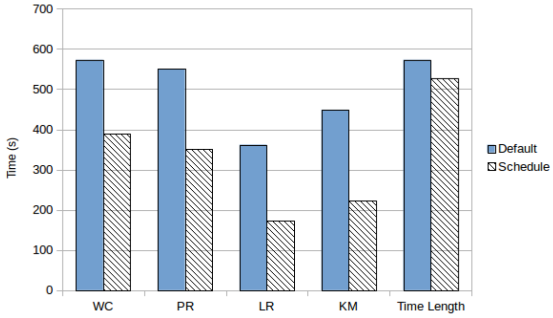

Fig. 8: Result of Scheduling Experiment 5. Column Time length represents the total execution time (completion time of the last job minus the start time of the first job).

knowledge several limitations of our current work as follows.

First, our current work models the interference for the first stage only. However, as the presented framework predicts performance for each individual stage, the model can be easily extended for cases where the interference happens in later stages.

Second, as this work focuses on modeling interference among multiple Apache Spark jobs, we assume that all the VMs running on the same machine are running Apache Spark jobs. As such, if multiple VMs consolidated on the same physical machine are running different kinds of jobs, our model may not work as the model of interference will be different. While we can extend our approach for such scenarios, it will require adaptation of model parameters.

Finally, the model was evaluated on a 6 node cluster with 4 concurrent jobs, which is smaller compared to the size of real-life clusters. None the less, our modeling framework demonstrates the feasibility of modeling interference for Apache Spark platform on a virtualized cluster and should work well once the parameters are estimated for different cluster size and number of concurrent jobs in a system.

6 Conclusion

In this paper, to predict the execution time of Apache Spark jobs interfered with other jobs, we develop an interference model. This model combines the execution information and resource consumption profile for each stage of Apache Spark jobs to calculate the slowdown ratio resulting from the interference, and then predicts the execution time when interfered with other jobs. Furthermore, an interference aware job scheduling algorithm leveraging the analytical framework is designed for Apache Spark platform. The developed models and the algorithm are evaluated using four real-life applications (e.g., Page rank , K-means, Logistic regression, Word count) on a 6 node cluster while running up to four jobs concurrently. Experimental results demonstrate that our framework can achieve high prediction accuracy and reduces average execution time for individual jobs significantly, thereby improving the system utilization.

7 Acknowledgment

This material is based upon work supported by the Air Force Office of Scientific Research award number FA9550-15-1-0184 under the DDDAS program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency.

References

1. Dstat: Versatile resource statistics tool. URL `http://dag.wiee.rs/home-made/dstat/`
2. https://hadoop.apache.org/
3. Sloan Digital Sky Survey. URL `http://www.sdss.org/`
4. Stanford SNAP. URL `http://snap.stanford.edu/`
5. Xen Project. URL `http://www.xenproject.org/`

6. Barbierato, E., Gribaudo, M., Iacono, M.: Performance evaluation of NoSQL big-data applications using multiformalism models. Future Generation Computer Systems **37**, 345–353 (2014)
7. Brun, C., Artées, T., Margalef, T., Cortées, A.: Coupling Wind Dynamics into a DDDAS Forest Fire Propagation Prediction System. Procedia Computer Science **9**, 1110–1118 (2012)
8. Bu, X., Rao, J., Xu, C.z.: Interference and Locality-Aware Task Scheduling for MapReduce Applications in Virtual Clusters. In: Proceedings of the 22nd international symposium on High-performance parallel and distributed computing, pp. 227–238. ACM (2013)
9. Chaisiri, S., Lee, B.S., Niyato, D.: Optimization of Resource Provisioning Cost in Cloud Computing. Services Computing, IEEE Transactions on **5**(2), 164–177 (2012)
10. Chen, X., Rupprecht, L., Osman, R., Pietzuch, P., Franciosi, F., Knottenbelt, W.: CloudScope: Diagnosing and Managing Performance Interference in Multi-Tenant Clouds. In: Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on, pp. 164–173. IEEE (2015)
11. Cheng, D., Rao, J., Jiang, C., Zhou, X.: Resource and Deadline-Aware Job Scheduling in Dynamic Hadoop Clusters. In: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, pp. 956–965. IEEE (2015)
12. Chiang, R.C., Hwang, J., Huang, H.H., Wood, T.: Matrix: Achieving Predictable Virtual Machine Performance in the Clouds. In: 11th International Conference on Autonomic Computing (ICAC 14) (2014)
13. Delimitrou, C., Kozyrakis, C.: Quasar: Resource-Efficient and QoS-Aware Cluster Management. ACM SIGPLAN Notices **49**(4), 127–144 (2014)
14. Didona, D., Quaglia, F., Romano, P., Torre, E.: Enhancing Performance Prediction Robustness by Combining Analytical Modeling and Machine Learning. In: Proceedings of the International Conference on Performance Engineering (ICPE). ACM (2015)
15. Fujimoto, R., Guin, A., Hunter, M., Park, H., Kanitkar, G., Kannan, R., Milholen, M., Neal, S., Pecher, P.: A Dynamic Data Driven Application System for Vehicle Tracking. Procedia Computer Science **29**, 1203–1215 (2014)
16. Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S.: Starfish: A Self-tuning System for Big Data Analytics. In: CIDR, vol. 11, pp. 261–272 (2011)
17. Khan, M., Jin, Y., Li, M., Xiang, Y., Jiang, C.: Hadoop Performance Modeling for Job Estimation and Resource Provisioning. IEEE Transactions on Parallel and Distributed Systems **27**(2), 441–454 (2016)
18. Lai, C.A., Wang, Q., Kimball, J., Li, J., Park, J., Pu, C.: IO Performance Interference among Consolidated n-Tier Applications: Sharing Is Better Than Isolation for Disks. In: Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on, pp. 24–31. IEEE (2014)
19. Li, S., Da Xu, L., Zhao, S.: The internet of things: a survey. Information Systems Frontiers **17**(2), 243–259 (2015)
20. Mozafari, B., Curino, C., Jindal, A., Madden, S.: Performance and Resource Modeling in Highly-Concurrent OLTP Workloads. In: Proceedings of the 2013 acm sigmod international conference on management of data, pp. 301–312. ACM (2013)
21. Mozafari, B., Curino, C., Madden, S.: DBSeer: Resource and Performance Prediction for Building a Next Generation Database Cloud. In: CIDR (2013)
22. Noorshams, Q., Busch, A., Rentschler, A., Bruhn, D., Kounev, S., Tuma, P., Reussner, R.: Automated Modeling of I/O Performance and Interference Effects in Virtualized Storage Systems. In: Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on, pp. 88–93. IEEE (2014)
23. Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.G., ICSI, V.: Making Sense of Performance in Data Analytics Frameworks. In: NSDI, vol. 15, pp. 293–307 (2015)
24. Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D.A., Kern, R., Kumar, H., Zikos, M., Wu, H., et al.: Ananta: Cloud Scale Load Balancing. In: ACM SIGCOMM Computer Communication Review, vol. 43, pp. 207–218. ACM (2013)
25. Patra, A., Bursik, M., Dehn, J., Jones, M., Pavolonis, M., Pitman, E.B., Singh, T., Singla, P., Webley, P.: A DDDAS Framework for Volcanic Ash Propagation and Hazard Analysis. Procedia Computer Science **9**, 1090–1099 (2012)
26. Popescu, A.D., Balmin, A., Ercegovac, V., Ailamaki, A.: PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics. Proceedings of the VLDB Endowment **6**(14), 1678–1689 (2013)
27. Prudencio, E.E., Bauman, P.T., Williams, S., Faghihi, D., Ravi-Chandar, K., Oden, J.T.: A Dynamic Data Driven Application System for Real-time Monitoring of Stochastic Damage. Procedia Computer Science **18**, 2056–2065 (2013)
28. Sharma, B.P., Wood, T., Das, C.R.: HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers. In: Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on, pp. 102–111. IEEE (2013)
29. Sharma, B.P., Wood, T., Das, C.R.: HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers. In: Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on, pp. 102–111. IEEE (2013)
30. Tan, Y., Nguyen, H., Shen, Z., Gu, X., Venkatramani, C., Rajan, D.: Prepare: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. In: Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on, pp. 285–294. IEEE (2012)
31. Vodacek, A., Kerekes, J.P., Hoffman, M.J.: Adaptive Optical Sensing in an Object Tracking DDDAS. Procedia Computer Science **9**, 1159–1166 (2012)
32. Wang, K., Khan, M.M.H.: Performance Prediction for Apache Spark Platform. In: High Performance Computing and Communications (HPCC), 2015 IEEE 17th International Conference on, pp. 166–173. IEEE (2015)
33. Wang, K., Khan, M.M.H., Gokhale, S.: Modeling Interference for Apache Spark Jobs. In: Proceedings of IEEE International Conference on Cloud Computing (CLOUD). San Francisco, USA (2016)
34. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: 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, pp. 2–2. USENIX Association (2012)
35. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (2010)

36. Zhang, W., Rajasekaran, S., Wood, T., Zhu, M.: MIMP: Deadline and Interference Aware Scheduling of Hadoop Virtual Machines. In: Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, pp. 394–403. IEEE (2014)
37. Zhang, Z., Cherkasova, L., Loo, B.T.: Performance Modeling of MapReduce Jobs in Heterogeneous Cloud Environments. In: Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing, pp. 839–846. IEEE Computer Society (2013)
38. Zhu, Q., Tung, T.: A Performance Interference Model for Managing Consolidated Workloads in QoS-Aware Clouds. In: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on. IEEE (2012)

8 Appendix

```
Input: List $JobProfiles$ listing Execution Information without Interference
Output: List $JobTime$ listing Execution Time with Interference
Function PredictJobExecution
    Initialize List $Phases$, List $JobTime$;
    for all $job \in JobProfiles$ do
        $Phases$.add($job$.getStage(0)); //first stage
    end
    while $Phases.size > 0$ do
        Initialize MinTime $\leftarrow$ MaxValue;
        for all $phase \in Phases$ do
            r $\leftarrow$ $phase$.calculateSlowdownRatio($Phases$);
            phaseTime $\leftarrow$ $phase$.getStageTime() $\times$ r;
            $phase$.setPhaseTime(phaseTime);
            $phase$.setSlowdownRatio(r);
            if $phaseTime < MinTime$ then
                MinTime $\leftarrow$ phaseTime;
            end
        end
        for all $phase \in Phases$ do
            if $phase.getPhaseTime = MinTime$ then
                StageTimeInterfere $\leftarrow$ MinTime + $phase$.getPartialTime();
                $JobTimes$.add($phase$ ,StageTimeInterfere);
                $Phases$.remove($phase$);
                if $JobProfiles$.hasNextStage(phase) then
                    nextStage $\leftarrow$ $JobProfiles$.nextStage($phase$);
                    $Phases$.add(nextStage);
                end
            else
                $phase$.setStageTime($phase$.getStageTime() $- \frac{MinTime}{phase.getSlowdownRatio()}$);
                $phase$.setPartialTime($phase$.getPartialTime + MinTime);
            end
        end
    end
end
```

Algorithm 1: Interference Estimation Algorithm

```
Input: List $JobProfiles$ listing Execution Information
       without Interference
Output: List $JobsWaitingTime$ listing Job Waiting
        Time before scheduling
Function Scheduling
    $JobProfiles \leftarrow$
    CalculateExecutedTime($JobProfiles$) ;
    $JobsWaitingTime \leftarrow$
    FindWaitingTime($JobProfiles$) ;
end
```

Algorithm 2: Job Scheduling Algorithm

```
Input: List $JobProfiles$ listing Execution Information
       without Interference
Output: Remaining $JobProfiles$ after previous
        scheduling
Function CalculateExecutedTime
    Initialize $SchInfo$, List $JobTimeLine$, Map
    $SlowdownRatios$, $ExeTime$;
    $SchInfo \leftarrow$ read($ScheduleFile$);
    $LastExe \leftarrow SchInfo$.ExeTime();
    $LastWait \leftarrow SchInfo$.WaitTime();
    JobsUpdate($JobProfiles$,$LastExe$,$LastWait$);
    $interval \leftarrow$ CurrentTime $- SchInfo$.SubTime();
    for all $job \in JobProfiles$ do
        $JobTimeLine$.add($job$.getStage(0).getBegin());
        $JobTimeLine$.add($job$.getStage(0).getEnd());
        $R \leftarrow$
        $job$.getStage(0).calculateSlowdownRatios();
        $SlowdownRatios$.put($job$,$R$);
    end
    Initialize List $ParallelJobs$;
    for all $Timepoint \in JobTimeLine$ do
        if $Timepoint > interval$ then
            $Timepoint$.setValue($interval$);
        end
        $gap \leftarrow Timepoint - LastTimepoint$;
        $job \leftarrow Timepoint$.getJob();
        for all $job \in ParallelJobs$ do
            $p \leftarrow ParallelJobs$.size() $- 1$;
            $ExeTime$.put($job$,$ExeTime$.get($job$)
            $+gap \times SlowdownRatios$.get($job$).get($p$));
        end
        if $Timepoint.isBegin()$ then
            $ParallelJobs$.add($job$);
        else
            $lefttime \leftarrow job$.getDuration()
            $-job$.getStage(0).getDuration();
            if $lefttime > interval - Timepoint$ then
                $ExeTime$.put($job$,$ExeTime$.get($job$)
                $+interval - Timepoint.$);
            else
                $ExeTime$.put($job$,$ExeTime$.get($job$)
                $+lefttime$);
            end
            $ParallelJobs$.remove($job$);
        end
        $LastTimepoint \leftarrow Timepoint$;
    end
    JobsUpdate($JobProfiles$,$LastWait$,0);
    JobsUpdate($JobProfiles$,$ExeTime$,0);
end
```

Algorithm 3: Calculate Executed Time

```
Input: List $JobProfiles, ExeTime, WaitTime$
Output: Updated List $JobProfiles$
Function JobsUpdate
    for all $job \in JobProfiles$ do
        if $ExeTime.get(job) > 0$ then
            $job$.minus($ExeTime$.get($job$));
        end
        if $WaitTime.get(job) > 0$ then
            $job$.addStart($WaitTime$.get($job$));
        end
    end
end
```

Algorithm 4: JobProfile Update

```
Input: List $JobProfiles$ listing Execution Information
       without Interference
Output: List $JobsWaitingTime$ listing Job Waiting
        Time before scheduling
Function FindWaitingTime
    Initialize $WaitN$, $minTime$, Map $JobsWait$, List
    $JobsWaitingTime$;
    $JobTime \leftarrow PredictJobExecution(JobProfiles)$;
    for all $job \in JobProfiles$ do
        $waitTimeMax \leftarrow JobTime$.getMax()
        $-job$.getDuration();
        for $i \leftarrow 0$ to $WaitN$ do
            $Jobwait$.add($\frac{i}{WaitN} \times waitTimeMax$) ;
        end
        $JobsWait$.put($job$,$Jobwait$);
    end
    for $Jobswait \in \prod_{i=1}^{N} JobsWait.get(i)$ do
        JobsUpdate($JobProfiles$,0,$Jobswait$);
        $JobTime \leftarrow PredictJobExecution(JobProfiles)$;
        if $JobTime.getMax() < minTime$ then
            $minTime \leftarrow JobTime$.getMax() ;
            $JobsWaitingTime \leftarrow Jobswait$;
        end
    end
end
```

Algorithm 5: Find Waiting Time