Understanding the Influence of Configuration Settings: An Execution Model-driven Framework for Apache Spark Platform

Nhan Nguyen, Mohammad Maifi Hasan Khan, Yusuf Albayram, Kewen Wang
Department of Computer Science and Engineering, University of Connecticut
Email: {nhan.q.nguyen, maifi.khan, yusuf.albayram, kewen.wang}@uconn.edu

***Abstract*—Apache Spark provides numerous configuration settings that can be tuned to improve the performance of specific applications running on the platform. However, due to its multi-stage execution model and high interactive complexity across nodes, it is nontrivial to understand how/why a specific setting influences the execution flow and performance. To address this challenge, we develop an execution model-driven framework that extracts key performance metrics relevant to different levels of execution (e.g., application level, stage level, task level, system level) and applies statistical analysis techniques to identify the key execution features that change significantly in response to changes in configuration settings. This allows users to answer questions such as "How does configuration setting X affect the execution behavior of Spark?" or "Why does changing configuration setting X degrade the performance of Spark application Y?". We tested our framework using 6 open source applications (e.g., Word Count, Tera Sort, KMeans, Matrix Factorization, PageRank, and Triangle Count) and demonstrated the effectiveness of our framework in identifying the underlying reasons behind changes in performance.**

***Keywords*-Apache Spark; Performance; Configuration Setting; Workload Characterization**

I. INTRODUCTION

Apache Spark [1] is a recently popularized large-scale data analytic platform that is currently being used by a large number of companies such as Amazon, eBay, and IBM[1]. Given its superior performance compared to Hadoop (i.e., an implementation of MapReduce model) for certain applications (e.g., clustering algorithms), researchers have been recently focusing on understanding its performance models. Among many, prior work investigated the problem of predicting performance [2], [3], understanding the effect of bottleneck components [4], and modeling interference [5]. However, understanding the influence of configuration settings on the performance of Spark is still an open research problem and is nontrivial for several reasons. For instance, as Spark provides a large number of configuration settings (which is common for large-scale cloud platforms [6]), it is extremely challenging to understand how each setting may influence the performance of a particular Spark application, leading to possible misconfiguration and/or suboptimal performance problem. Moreover, as Spark supports a wide variety of applications such as machine learning, graph computation, interactive queries, and stream applications, and each setting may influence different applications differently, it is important to develop a framework that allows users to investigate the performance influence model of each setting for different applications with minimal effort.

To address this void, we present an execution model-driven framework that allows automated identification of performance influence models of each setting for a specific application. The main idea is to leverage the multi-stage execution flow of Spark to identify relevant performance metrics for each stage and apply statistical analysis iteratively to determine the key execution features that get affected due to changes in settings. Our framework allows users to answer questions such as "How does configuration setting X affect the execution behavior of Spark?" or "Why does changing configuration setting X degrade the performance of Spark application Y?".

To evaluate our framework, as the same setting may affect the performance of different applications differently, we used six different workloads. Specifically, we used Word Count and Tera Sort (batch processing applications), KMeans and Matrix Factorization (machine learning algorithms), and PageRank and Triangle Count (graph computation algorithms). In each case, our framework correctly points to the underlying factors influencing the performance for individual settings.

II. THE DESIGN OF THE EXECUTION MODEL-DRIVEN FRAMEWORK

A. Execution Model of Spark

At a high level, Spark contains two main components: the driver program and the executors. The driver program (i.e., main program) receives an application (a.k.a. job) from a client and coordinates with a set of executors on worker nodes to run the submitted job. The executors can return the result to the driver program or directly write to a predefined output location. In this platform, a Spark job can be viewed as a directed acyclic graph of stages where each stage contains a group of tasks. Each task works directly with the resilient distributed dataset (RDD) partitions, and computes and outputs the intermediate result that can be used by the tasks in the following stage. There are two types of tasks in Spark: ShuffleMap Task and Result Task. A ShuffleMap Task executes a task and divides the task output into multiple partitions. A Result Task sends the result back to the driver

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

2159-6190/17 $31.00 © 2017 IEEE
DOI 10.1109/CLOUD.2017.119

program or outputs the result by itself. Stages in Spark are usually separated by shuffle operations where a task in a stage requires data from the previous stage. A shuffle operation is expensive as it can involve data partition, data serialization/de-serialization, and data transfer over the network.

Based on the execution flow, we can represent the runtime of a Spark application as follows.

$$Runtime\_Application = Application\_Initialization + \sum_{i=1}^{S} Runtime\_Stage_i + Application\_Termination \quad (1)$$

Here S represents the number of stages of an application.

As each stage contains (possibly multiple) waves of parallel tasks, if the average run time of a task in a stage is $Task\_Runtime\_Average$, then we have the following.

$$\begin{aligned} &Runtime\_Stage_i \\ &= Finish\_Time\_Last\_Task - Submission\_Time\_First\_Task \\ &\approx \sum_{j=1}^{W} Longest\_Task\_Runtime\_in\_Wave_j \\ &\approx W * Task\_Runtime\_Average \end{aligned} \quad (2)$$

Here W represents the number of waves of tasks in a stage.

Assuming that we know the number of tasks (i.e., $Number\_of\_Tasks$) and the maximum number of parallel tasks (i.e., $Number\_of\_Parallel\_Tasks$), then the stage runtime in (2) can be estimated as follows.

$$Runtime\_Stage_i = \frac{Number\_of\_Task}{Number\_of\_Parallel\_Tasks} * Task\_Runtime\_Average \quad (3)$$

In Spark, each RDD is assigned to a task and the number of tasks is calculated as shown in (4) where *Input_Size* is the size of the input data and *Block_Size* is the size of the distributed file system block which Spark uses (i.e., HDFS in this work).

$$Number\_of\_Tasks = \frac{Input\_Size}{Block\_Size} \quad (4)$$

The value of $Block\_Size$ of HDFS is configurable but in this work we only focus on configuration settings of Spark and used the default value for $Block\_Size$ (i.e., 128MB). Based on the above model, we can see that a setting can affect the performance of an application at one or more levels (e.g., task, stage), making it harder to understand the underlying reasons behind changes in performance, which we try to automatically identify in this paper using simple statistical techniques.

B. Methodology for Identifying the Role of Configuration Settings in Spark

1) Selecting Configuration Settings Related to Performance: To demonstrate our approach, based on Apache Spark documentation[2], we first identify a subset of the configuration settings that are related to performance tuning and classify them into different types based on whether it is a basic application setting (i.e., Type Application) or a setting for tuning an individual phase of the Spark application (e.g., Shuffle, Execution) (column *Type* in Table I). The first three configuration settings listed in Table I are used to tune the performance of the driver and executor processes. The remaining configuration settings are used to tune the performance of Spark jobs at task-level. Configuration settings can be further grouped based on which resource it is likely to affect significantly (i.e., column *Resource*).

[2]http://spark.apache.org/docs/latest/configuration.html

Configuration Setting	Default Value	Type	Resource
spark.driver.cores	1	Application	CPU
spark.driver.memory	1g	Application	Memory
spark.executor.memory	1g	Application	Memory
spark.executor.cores	-	Execution	CPU
spark.task.cpus	1	Scheduling	CPU
spark.default.parallelism	-	Execution	CPU
spark.memory.fraction	0.6	Memory Management	Memory
spark.reducer.maxSizeInFlight	48m	Shuffle	Memory
spark.shuffle.compress	true	Shuffle	Memory
spark.shuffle.spill.compress	true	Shuffle	Memory
spark.speculation	false	Scheduling	-

TABLE I: Configuration settings related to the performance of Spark. In Column Default Value, symbol "-" denotes that the default value is set by Spark. In column Resource, symbol "-" indicates multiple resources.

Level	Metrics
Application	execution time, number of running executors, number of stages, execution time of each stage, number of tasks, total runtime of each type of tasks, shuffle read/write bytes over the network and on local disk, de-serialization/serialization time
Stage	execution time, number of tasks, total runtime of each type of tasks
Task	task type and task execution time
System	CPU utilization, RAM usage, Disk throughput, Network throughput

TABLE II: Execution metrics are organized in the following order (high to low): application, stage, task, and system.

2) Benchmarking and Collecting Data: Although we can classify configuration settings in Spark into different groups as described above, however, given the multi-stage execution model and different kinds of processes that are involved (e.g., executor, worker), it is nontrivial to understand "How" a particular setting affects performance. To understand how changing a setting actually affects application performance, in addition to collecting information related to system resource usage, we extract execution metrics from Spark logs and organize them at three levels based on execution model, namely, application level, stage level, and task level as shown in Table II.

To investigate the effect of configuration change, we run benchmarks with different workloads and change one setting at a time (excluding the first three settings listed in Table I). During execution, we collect the relevant execution metrics (Table II). For numerical settings, we carefully select different values to stress the system. For a Boolean or categorical type, we try all possible values. We run each benchmark 5 times for a specific value and take the average of the observed metrics. For each value of a setting, we create a set of vectors where each vector contains all the metrics shown in Table II. In our case there are two types of metrics: scalar and vector. For example, application execution time, number of tasks, and number of stages are scalar items. Execution time of

stages and task duration are represented as vectors of numbers. Furthermore, we calculate statistical metrics such as max, min, mean for resource usage vectors such as CPU utilization, and disk I/O.

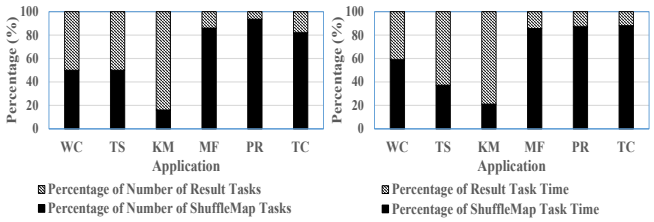
3) Analyzing Collected Data: If the change in performance is found to be not significant for different values of a particular setting, we do not perform any further analysis for that setting. In our case, we tested 20 settings and narrowed down to 11 settings as shown in Table I that appear to affect performance significantly when changed in isolation. Next, we study the reason why that setting affects the performance of Spark. For scalar metrics, we use Pearson correlation coefficient (r) to measure the correlation between the stage and task level metrics (e.g., number of task) and the performance (e.g., runtime) to determine Spark execution metrics that are related to performance changes. For vector metrics, we compare the probability distributions to determine whether these metrics change significantly when we change a setting. If the vector metrics such as task duration across multiple experiments appeared not to be normally distributed (Kolmogorov-Smirnov), we use the non-parametric Mann-Whitney U-tests to compare task duration obtained for two different values of a configuration setting.

III. Evaluation

For evaluation, all experiments were performed using Spark version 2.0.2 and HDFS version 2.7.1 on a cluster of 6 nodes. Each node has 12 CPU cores, 32 GB of RAM memory, and 1.8 TB hard drive. The network bandwidth is 1Gbps. In aggregate, the cluster has 60 cores, 192 GB of RAM memory, and 10.8 TB hard drive. One node is configured as the master node and the others as slave node for both HDFS and Spark. We use HDFS with a block size of 128 MB and a replication factor of 3. As Spark is extremely slow when the first three settings listed in Table I (i.e., *spark.driver.cores*, *spark.driver.memory*, *spark.executor.memory*) use default values, we set *spark.driver.cores* = 8, *spark.driver.memory* = 28 GB, and *spark.executor.memory* = 28 GB (which left 4 GB for the Operating system). These values were not changed across experiments.

A. Job Characteristics

In our study, we used Word Count (WC) and Tera Sort (TS) which are well-known batch processing applications for MapReduce-like frameworks, KMeans (KM) and Matrix Factorization (MF) which implement machine learning algorithms, and PageRank (PR) and Triangle Count (TC) which implement graph computation algorithms. Word Count counts the total number of occurrences of each word in a document. Tera Sort leverages map/reduce framework to sort data based on a total order. KMeans implements a clustering algorithm that classifies a given set of data points into a predefined number of groups. Matrix Factorization, a collaborative filtering technique, uses alternating least square modules provided by the Spark MLib package. PageRank implements a graph processing algorithm that measures the importance of each node in a graph. Triangle Count application counts the number of triangles in a graph. Each of these applications uses different Spark core operators and functions provided by the machine learning and graph library to implement the algorithms.



(a) Number of tasks. (b) Aggregated task execution time.

Fig. 1: Task characteristics for different applications.

Workload	Dataset	Size	SR	SW	Stages	Tasks
Word Count	Wikipedia	34 GB	6.4 GB	7.9 GB	2	534
Tera Sort	Synthetic	20 GB	17 GB	20 GB	4	800
KMeans	Synthetic	46 GB	25 MB	32 MB	19	6726
Matrix Factorization	Amazon Movie Review	4 GB	216 MB	1.9 GB	56	117
PageRank	Live Journal	20 GB	3 GB	5 GB	13	2050
Triangle Count	Amazon Movie Review	200 MB	413 MB	993 MB	11	22

TABLE III: Benchmarking dataset for different applications and their execution characteristics under the default setting. SR and SW represent the amount of shuffle read and shuffle write respectively.

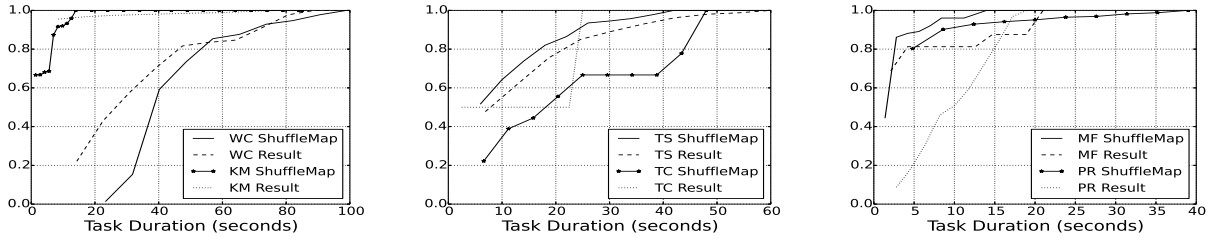
The data sets used by these applications are listed in Table III. Dataset Wikipedia is downloaded from Wikimedia website[3]. Synthetic datasets (Tera Sort and KMeans) are created using the data generator classes provided by the Spark. Amazon Movie Review and Live Journal data sets are downloaded from Stanford Network Analysis Project website[4].

As shown in Figure 1 (a) and Figure 1 (b), the ratio of the number and execution time of ShuffleMap Tasks and Result Tasks for different applications are quite different. Also, different applications spend different amounts of time in performing the ShuffleMap Tasks and Result Tasks respectively.

The cumulative distributions of task durations for these applications are presented in Figure 2. As can be seen, Word Count has the largest task duration with the value of 100 seconds and Matrix Factorization has the smallest task duration with the value of 2 seconds. As task is the fundamental unit of execution in Spark, the duration of the longest task has a significant impact on the runtime of a Spark application (Equation (2)). From Figure 2, we can see that the duration of ShuffleMap Tasks and Result Tasks varies significantly across different applications.

In terms of network load (column shuffle read *(SR)* in Table III), once again different applications exhibit different characteristics. For example, while KMeans does not transfer too much data over the network (25 MB) although the input data size is 46 GB, the size of the data transferred by Tera Sort over the network is quite large (17 GB) though its input data size is smaller (20 GB) compared to KMeans. Other relevant

[3]https://dumps.wikimedia.org/enwiki/
[4]https://snap.stanford.edu/

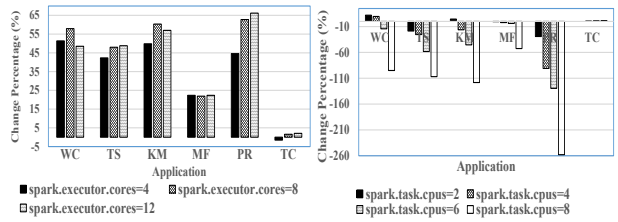


(a) Word Count and KMeans. (b) Tera Sort and Triangle Count. (c) Matrix Factorization and PageRank.

Fig. 2: Cumulative distributions of task durations for different applications.

Configuration Setting	Base value	Tuned values	Affected Metrics	WC	TS	KM	MF	PR	TC
spark.executor.cores	1	4, 8, 12	task duration	↑↑↑	↑↑↑	↑↑↑	↑↑↑	↑↑↑	- - -
spark.task.cpus	1	2, 4, 6, 8	task duration	↑↑↓↓	↓↓↓↓	- ↓↓↓	- - - ↓	↓↓↓↓	- - -
spark.default.parallelism	1	4, 8, 100	task duration	↑↑↑↑	↑↑↑↑	↑↑↑↑	↑↑↑↑	↑↑↑↑	↑↑↑↑
spark.memory.fraction	0.6	0.3, 0.5, 0.7, 0.9	input data size, shuffle time	- ↑↑↓	↓↓↓↓	↑ - ↑↑	- - - -	↓↓↓↓	- ↑ - -
spark.reducer.maxSizeInFlight	48MB	24MB, 96MB, 144MB	shuffle time	- ↓↓	↓↓↓	↓↑↑	- - -	↓↓↓	- ↑↑
spark.speculation	false	true	number of tasks	↓	↓	↓	-	-	-
spark.shuffle.compress	true	false	de-serialization time, shuffle time	↓	↓	↑	↓	↓	↓
spark.shuffle.spill.compress	true	false	de-serialization time, shuffle time	↑	↓	↑	-	↓	-

TABLE IV: Configuration settings and their effect on different applications. Symbol "↑", "↓", "-" indicates whether the performance of Spark is improved, degraded, or remain almost unchanged respectively when a setting is changed to a new value (each arrow corresponds to the respective value in column "Tuned values") compared to the baseline performance using the base value (column "Base value").



(a) Setting *spark.executor.cores* (base value =1). (b) Setting *spark.task.cpus* (base value=1).

Fig. 3: Changes in Spark performance.

job characteristics information for different applications are listed in Table III.

The values of different settings that are used in our experiments along with the metrics that are significantly affected when we change the settings are listed in Table IV. The details are discussed below.

B. How does spark.executor.cores influence performance?

Setting *spark.executor.cores* defines the number of cores that can be used by each executor process. The number of executors per node is decided a priori based on this setting, total available memory in the system, and the setting *spark.executor.memory*. In our case, the number of executors per node was fixed at 1. Hence, intuitively, if we increase the value of *spark.executor.cores*, it should allow the executor to run multiple tasks in parallel, and reduces the execution time. To test this, we set *spark.executor.cores* equal to 4, 8, and 12 and compared the performance against *spark.executor.cores* = 1 (Figure 3 (a)). We can see that the performance changes significantly when the value is changed from 1 to 4, and remains similar beyond that point. When we vary the value of this setting, the correlation analysis shows that the number of tasks is significantly correlated to the runtime (Pearson's r = 0.373, p = 0.043), implying that the applications that have a large number of tasks to run gained significantly more improvement by increasing the value of *spark.executor.cores*.

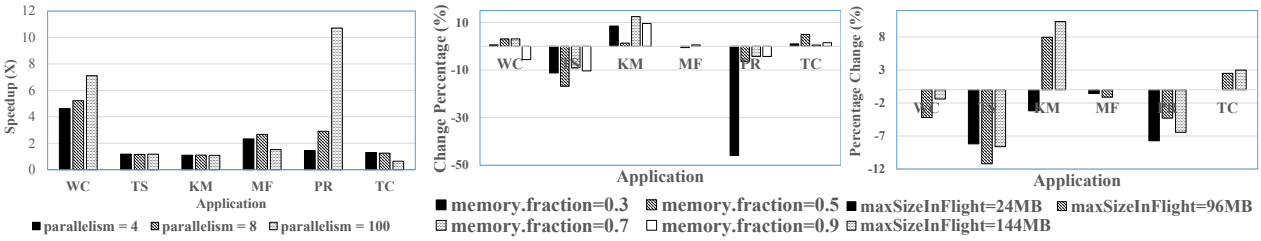
C. How does spark.task.cpus influence performance?

Setting *spark.task.cpus* defines the number of CPU cores that can be used to execute tasks, which in turn determines the number of parallel tasks in the system (Equation (5)). In Spark, the value of $Number\_of\_Parallel\_Tasks$ affects the number of waves of tasks (i.e., the larger number of waves might lead to more time for a particular stage). Therefore, if we increase the value of *spark.task.cpus*, the number of waves of tasks is likely to increase, which can significantly reduce the performance of Spark if an application has many tasks. Not surprisingly, the number of tasks and the run time is found to be positively correlated in our study (r = 0.443, p = 0.027). We also found the number of shuffle tasks to be significantly correlated to the performance of Spark when we change the value of this setting (r = 0.437, p = 0.029).

$$Number\_of\_Parallel\_Tasks = \frac{Number\_of\_Cores}{spark.task.cpus} \quad (5)$$

Figure 3 (b) illustrates how the performance is affected due to changing this setting. Interestingly, for Word Count, changing *spark.task.cpus* to 2 and 4 reduces the runtime of tasks including the longest task, which helps to improve the performance. In contrast, the performance of PageRank, which has the largest number of shuffle tasks (1,920 tasks), is reduced significantly when we increase the value of *spark.task.cpus*. As such, based on our experiments, it appears that increasing the value of *spark.task.cpus* may be beneficial if an application has long task duration (e.g., Word Count in Figure 2 (a)).

D. How does spark.default.parallelism influence performance?

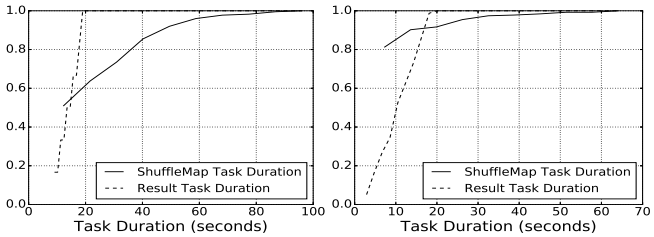


(a) Setting *spark.default.parallelism* (base value=1).

(b) Setting *spark.memory.fraction* (base value=0.6).

(c) Setting *spark.reducer.maxSizeInFlight* (base value=48).

Fig. 4: Changes in Spark performance.



(a) *spark.default.parallelism* = 8. (b) *spark.default.parallelism* = 100.

Fig. 5: Cumulative distribution of task durations for different values of setting *parallelism* for PageRank.

Setting *spark.default.parallelism* defines the number of partitions returned by the shuffle operations, which defines the number of tasks in the second and later stages. The number of parallel tasks indicates how many tasks can be executed in parallel by an executor. For this setting, analysis at the third level (task level) helps to find out why the setting affects the performance of Spark. Specifically, after applying Mann-Whitney U test, we found that the task duration when *spark.default.parallelism* = 8 is significantly different compared to when *spark.default.parallelism* = 100 (U = 75961, $p < 0.001$). Figure 5 shows the distributions of task durations when we run PageRank with *spark.default.parallelism* = 8 (Figure 5 (a)) and *spark.default.parallelism* = 100 (Figure 5 (b)). We can see the difference between the ShuffleMap Task duration in these two figures. Intuitively, the small value of *spark.default.parallelism* leads to an increase in the duration of ShuffleMap Tasks but does not affect the duration of Result Tasks. The change in duration of ShuffleMap Tasks explains the slowdown of Spark when *spark.default.parallelism* = 8.

Figure 4 (a) shows the effect of setting *spark.default.parallelism* on different applications. Based on our experiments, we conclude that this setting can help if an application has long ShuffleMaptask duration such as Word Count (Figure 2 (a)) and Page Rank (Figure 2 (c)).

E. How does spark.memory.fraction influence performance?

Setting *spark.memory.fraction* defines how much memory Spark uses for execution and storage. For this setting, we found that input data size (r = 0.639, $p < 0.001$) and shuffle time (r = 0.587, $p < 0.001$) are significantly positively correlated to performance. Figure 4 (b) shows the performance change of Spark when we varied the value of this setting. While the performance of Tera Sort and PageRank reduces significantly (up to 48%) when the value of *spark.memory.fraction* is small, the performance of the other applications is less affected by the same value of the setting. The value of 0.7 helps to improve the performance of KMeans up to 12% and also has less negative effect on the performance of PageRank. The experiment shows that the performance of applications which have small shuffle time (e.g., Matrix Factorization) or small input data size (e.g., Triangle Count) does not vary much when we change this setting.

F. How does spark.reducer.maxSizeInFlight influence performance?

Setting *spark.reducer.maxSizeInFlight* determines the amount of data that can be fetched from map outputs by each reducer. We found shuffle time (r = 0.548, $p = 0.001$) to be an important factor related to the application execution time. Figure 4 (c) shows that this setting reduces the performance of Tera Sort and PageRank if we set this setting to a value different than the default value (i.e., 48MB). We can see that the performance of the applications which have large amount of data to be read/written by shuffles (e.g., Tera Sort and PageRank as shown in Table III) heavily depends on this setting (depends on the shuffle time). If we set this setting to a small value, it is likely to benefit a system which has a small amount of memory but will require more number of transactions between machines. As our cluster has a large amount of memory, the small value for *spark.reducer.maxSizeInFlight* (24MB in this work) does not improve Spark performance significantly.

G. How does spark.speculation influence performance?

Setting *spark.speculation* is used to enable the feature that runs speculative execution of tasks. In particular, if *spark.speculation* = true, Spark automatically checks if there is any task that is running slowly and re-launch that task automatically. In general, this setting is used to mitigate the effect of straggler tasks on the performance of Spark. However, Figure 6 shows that the performance of Spark degrades if we set this setting to true, especially for KMeans. We compared cumulative distributions and characteristics of KMeans when Spark has different values for *spark.speculation*. Although U test shows that the cumulative distributions are not significantly different, the number of tasks which is significantly correlated with the run time (r = 0.723, $p = 0.005$) increased when *spark.speculation* = true. This increase in the number of tasks causes Spark to run longer.

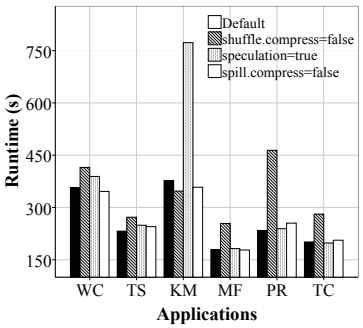


Fig. 6: Spark runtime for *spark.speculation*, *spark.shuffle.compress*, and *spark.shuffle.spill.compress*.

H. How does spark.shuffle.compress influence performance?

Settings *spark.shuffle.compress* and *spark.shuffle.spill.compress* determine whether Spark needs to compress map output and data spilled when performing shuffle operations respectively. For these settings, executor de-serialization time (i.e., the amount of time Spark executors spend to de-serialize data) (r = 0.733, $p = 0.001$) and shuffle time (r = 0.705, $p = 0.001$) are found to be significantly correlated with the run time. We note that this setting should be set after considering the trade-offs between CPU performance and network/disk performance. In particular, while compressing data helps to reduce the size of data sent over the network or stored on disks, it will cost CPU resource to compress data. Therefore, if the data to be read/written by shuffles is small like KMeans (Table III), compressing data is likely to increase the total runtime (Figure 6).

IV. Related Work

While configuration tuning for Apache Spark platform is not well studied yet, quite a few work exist that looked into the problem of performance modeling for Apache Spark platform [3]–[5], that includes the release of a benchmarking suite for performance measurement [7]. Among these few, Min Li *et. al* illustrated the effect of configuration settings on performance of Spark [7]. Another recent study demonstrated the influence of different factors such as system resources (i.e., Disk, Network) and task stragglers (i.e., slow tasks) on performance of Spark [4]. Norbert Siegmund et al. [8] used machine learning and sampling heuristics to construct performance influence models for configurable systems that help to describe how configuration settings and their interactions influence the performance of a system. In particular, the authors proposed sampling methods to select samples from a large configuration space and used linear regression to construct performance models. Performance modeling, resource allocation in various other cloud settings are studied as well [6], [9], [10].

While several of these prior efforts are close in spirit to our work, however, we focus on understanding the effect of configuration settings on Spark applications at the execution level. To the best of our knowledge, we are the first to investigate an execution model-driven framework that attempts to explain how and why different configuration settings affect application performance.

V. Conclusion

In this paper we present an execution model-driven framework for understanding the performance influence models of individual Apache Spark setting. By using a representative set of open source applications, we demonstrate that it is possible to reason about the impact of settings by characterizing the workloads and applying statistical analysis techniques. While this paper does not facilitate automated tuning of settings or investigate the interactions among multiple settings, we strongly believe that the presented framework can be leveraged to gain insights regarding the underlying reasons behind observed performance variations in response to changes in settings.

VI. Acknowledgment

This material is based upon work supported by the Air Force Office of Scientific Research award number FA 9550-15-1-0184 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] 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*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[2] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 363–378.

[3] K. Wang and M. M. H. Khan, "Performance Prediction for Apache Spark Platform," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*.

[4] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 293–307.

[5] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale, "Modeling Interference for Apache Spark Jobs," in *Proceedings of IEEE International Conference on Cloud Computing (CLOUD)*, ser. CLOUD'16, 2016.

[6] N. Nguyen, M. M. H. Khan, and K. Wang, "CSMiner: An Automated Tool for Analyzing Changes in Configuration Settings across Multiple Versions of Large Scale Cloud Software," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 472–480.

[7] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF '15. New York, NY, USA: ACM, 2015, pp. 53:1–53:8.

[8] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence Models for Highly Configurable Systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015.

[9] D. Kimura, E. Numata, and M. Kawatsu, "Performance Modeling to Divide Performance Interference of Virtualization and Virtual Machine Combination," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 256–263.

[10] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A Self-tuning System for Big Data Analytics." in *CIDR*, vol. 11, 2011, pp. 261–272.