IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 14 July 2022, accepted 6 September 2022, date of publication 12 September 2022, date of current version 19 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3205723

RESEARCH ARTICLE

An Optimized Straggler Mitigation Framework for Large-Scale Distributed Computing Systems

SAMAR A. SAID[1], SHAHIRA M. HABASHY[1], SAMEH A. SALEM[1,2], AND ELSAYED M. SAAD[1]

[1]Department of Computer and Systems Engineering, Faculty of Engineering, Helwan University, Cairo 11792, Egypt
[2]Egyptian Computer Emergency Readiness Team (EG-CERT), National Telecom Regulatory Authority (NTRA), Cairo 12577, Egypt

Corresponding author: Samar A. Said (samar_said@h-eng.helwan.edu.eg)

ABSTRACT Nowadays, Big Data becomes a research focus in industrial, banking, social network, and other fields. In addition, the explosive increase of data and information require efficient processing solutions. Therefore, Spark is considered as a promising candidate of Large-Scale Distributed Computing Systems for big data processing. One primary challenge is the straggler problem that occurred due to the presence of heterogeneity where a machine takes an extra-long time to finish execution of a task, which decreases the system throughput. To mitigate straggler tasks, Spark adopts speculative execution mechanism, in which the scheduler launches additional backup to avoid slow task processing and achieve acceleration. In this paper, a new Optimized Straggler Mitigation Framework is proposed. The proposed framework uses a dynamic criterion to determine the closest straggler tasks. This criterion is based on multiple coefficients to achieve a reliable straggler decision. Also, it integrates the historical data analysis and online adaptation for intelligent straggler judgment. This guarantees the effectiveness of speculative tasks by improving cluster performance. Experimental results on various benchmarks and applications show that the proposed framework achieves 23.5% to 30.7% execution time reductions, and 25.4 to 46.3% increase of the cluster throughputs compared with spark engine.

INDEX TERMS Spark, straggler, speculative execution, cluster throughput.

I. INTRODUCTION

In the last decade, the huge amount of digital data becomes a key issue to be stored, managed and analyzed. As a consequence, Large-Scale Distributed Computing Systems is a promising solution in many fields [1], [2], [3], [4], [5], [6], [7]. Many companies believe that these computing systems are the most effective and fault-tolerant method to store and handle enormous volumes of data [8]. Hadoop [9] and Spark [10] are two popular distributed computing systems that are widely used in the industry and academia. Hadoop is an open-source software framework for handling massive volumes of data, providing comprehensive processing and analytical capabilities. Hadoop core composed of a distributed file system storage and a MapReduce processing [11]. This data processing computing system consists of three stages: Map phase, Shuffle phase and Reduce phase. In this system, huge files are decomposed into several little pieces of similar size and distributed on the cluster for storage. Spark is an alternative distributed computing technology that is open-source and free to use. It is implemented on top of the Hadoop and its goal is to build a general-purpose programing model faster and more fault-tolerant than MapReduce. Resilient Distributed Dataset (RDD) [12] is a technology introduced by Spark that provides application program interfaces (APIs) that enable transformations and parallelization of data which can be adapted by users on basis of their applications. As a result, the performance of the batch, interactive, streaming and iterative computations can be increased by persisting RDD in memory. Furthermore, Spark offers a variety of sophisticated modules which are built on top of the Spark core including Spark Streaming [13], Spark SQL [14], GraphX [15] and MLlib [16]. Spark Streaming module allows Spark to build streaming applications, while Spark SQL module used for structured data processing. Also, GraphX is a graph API that allows you to do

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

graph-parallel computations. But, MLlib Spark's is a scalable machine learning library. However, Spark is extremely recommended for data analytics, the appearance of Straggler tasks and performance deterioration in parallel systems may occur [17], [18]. In this context, a task is considered straggler when a machine has a significant delay to finish the execution of that task compared with other tasks at the similar stage [19], [20]. This delay causes a degradation of the system throughput. Many studies introduced a speculative execution to overcome this problem, this leading to enhance the execution efficiency and the cluster performance [21], [22], [23], [24]. In this context, late tasks can run on another nodes using speculative execution. A speculative execution can be categorized into two prevalent techniques [21] named as the Cloning technique and Straggler Detection technique. The first technique [25] suggests full cloning of small jobs. In this technique, when the calculation of jobs' costs are predictable to be minimal as well as the system resources are available, the clones run concurrently with original tasks. Accordingly, the clones are launched in incidentally and rapacious manner. So, it is suitable only and most appropriate for clusters with light loads. But, in the latter technique [26], [27] the system observes the completion of each task and only commences backup copies whenever a straggler is found. Consequently, the straggler detection is more comprehensive and applicable for low and high cluster loads. In this paper, a new Optimized Straggler Mitigation Framework is proposed. The proposed framework introduces a dynamic criterion to evaluate the most suitable tasks for speculation. The proposed criterion is based on multiple coefficients to obtain the optimal straggler decision. In addition, it guarantees the effectiveness for speculative tasks by improving cluster performance. The proposed framework is composed of two modules. The first module named a Straggler Decision Engine Module that collects tasks execution logs to perform initial calculation and identifies an appropriate straggler detection threshold based on four weighted coefficients. The latter module is named as Straggler Alleviation Module that guarantees the effectiveness for speculative tasks for more efficient straggler mitigation. We can summarize the major contributions of that paper as follows:

(1) We propose a new Optimized Straggler Mitigation Framework that presents dynamic criterion to predict the tasks that suffer from straggler in an efficient manner.

(2) The proposed framework criterion is based on multiple coefficients which lead to finding the optimum straggler decision.

3) We evaluate the job execution and the improvement in the cluster throughput using several benchmarks.

4) We improve the cluster performance and guarantee the effectiveness of detecting and manipulating speculative tasks.

This paper is organized as follows: Section II describes the background and motivation. Section III demonstrates the system model and problem formulation. Section IV describes the implementation details of the proposed framework. Section V examines the complexity analysis. Section VI shows an illustrative example. Section VII explores the performance evaluation and detailed results. In section VIII, the paper is concluded with the main findings.

II. BACKGROUND AND MOTIVATION

This section presents a concise overview of Spark computing system. After that, the straggler problem as well as its solution in spark will be explored.

A. SPARK

Apache Spark [10] is a promising candidate in large scale distributed computing systems. Spark is intended to improve application execution as well as fulfill scalability and fault tolerance by using resilient distributed dataset (RDD) [12]. RDD is a read-only collection of objects partitioned among a number of machines that can be reconstructed when losing one of the partitions. Each Spark application launches a single master process known as the driver, which is in charge of task scheduling. It employs a hierarchical scheduling procedure that includes jobs, stages, and tasks, where the term "stages" refers to smaller groups of tasks that are separated from interdependent jobs. As shown in Fig. 1, A Spark cluster is made up of only master node as well as many slave nodes known as workers. Every Worker is handled on an execution node, which may incorporate one or many executors. Each executor has the ability to use many cores and execute tasks at the same time. In case of a Spark application is submitted, the master calls the resource manager for getting computing resources based on the application's needs. Once the resource is ready, tasks are assigned to all executors in parallel through Spark scheduler. Then, the master node will track the status of executors and gathers the results from worker nodes throughout this process. In this paper, Spark is used as the target framework to determine and predict the tasks that will suffer from straggler in an efficient manner.

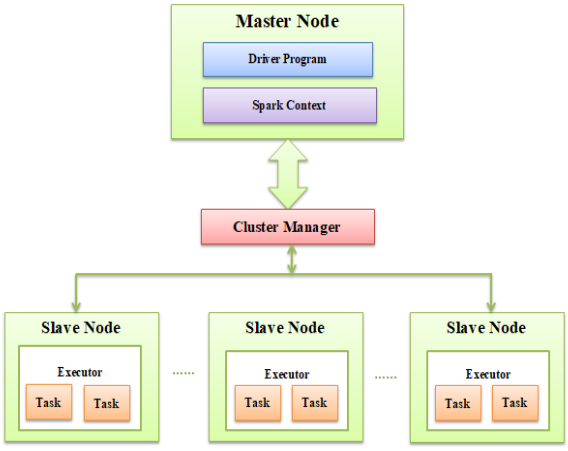

FIGURE 1. Spark architecture [28].

B. STRAGGLER PROBLEM

In Spark, a job is broken down into one or more stages. Afterwards, stages are divided into separate tasks. A task is considered as a unit of execution that runs on the Spark worker in the cluster. When a task in execution becomes slower than other tasks in the same job, this task is called a ''straggler task'' which prolong the entire job and the cluster throughput will be affected. There are numerous causes that make a task take a long time in execution and turned into a straggler [29], [30]. These causes like hardware heterogeneity, overawed machines, network congestion, bad code and contention of resources between tasks running on the straggler machine. This problem solution is the speculative execution [31]. Although it seems that speculative execution mechanism is a simple matter. It allows you to restart the straggler tasks on another machine, in actuality it is a complicated issue because speculative tasks consume resources which may affect other running tasks. As a consequence, if a straggler task is not detected correctly or a backup task is finished earlier than the original task, this will consumes resources with no use. Also it is leading to increase the job execution time and degrading the cluster throughput [32]. Speculative execution algorithms faces some challenges like their methodology for correctly detecting straggler tasks, and stragglers should be identified as early as possible to save resources. Also, the choice of nodes for run backup tasks is very important factor to avoid unnecessary resource consumptions. In this context, if these factors are not met, the system will perform poorly. It should be noted that the Spark computing system allows the speculation by default. The default speculation implements simple technique to deal with stragglers. Initially, the speculation parameter ''spark.speculation'' should be set by true. This helps in identifying slower running tasks in a stage according to a precalculated threshold that is based on the average number of successful tasks multiplied by ''spark.speculation.multiplier''. After that, a copy of speculative tasks is ready to run on idle nodes. It is worth noting that the parameter ''spark.speculation.quantile'' identifies slow tasks when a certain amount of tasks are completed. Also, it is continuously applied based on an interval through ''spark.speculation.interval'' parameter. Despite the fact that spark's default speculation has improved performance in a heterogeneous environment, but it has many defects. The speculation decision becomes less accurate since it is based on a fixed time as well as it does not consider the processing capacity of various nodes. Furthermore, it may be unnecessary to launch clones of speculation tasks across the cluster.

III. SYSTEM MODEL AND PROBLEM FORMULATION

This section introduces the architecture of the proposed Straggler Mitigation Framework. The purpose of the introduced framework is to minimize job execution time by eliminating the impact of the straggler. This boosts the cluster throughput by applying the speculative execution mechanism. To achieve this goal, the proposed framework uses a dynamic criterion to evaluate the most suitable tasks for speculation. This criterion is based on multiple coefficients to achieve a reliable straggler decision. Four coefficients are used by the proposed framework, namely job quality of service limitation, stage proceeding behavior, processing bandwidth, and cluster utilization level.

The Proposed Mitigation Framework Architecture: The proposed framework is designed to work in conjunction with Spark parallel data processing platform. The main modules of the proposed framework include a *Straggler Decision Engine Module* and *Straggler Alleviation Module*. The *straggler decision engine Module* is constructed from two components; the initial historical calculator component that collects tasks execution logs to perform initial calculations, and the Dynamic Weighted Straggler Decision component that determines the best threshold for identifying stragglers. After that, the decision for straggler is made based on four weighted coefficients. The latter module, *Straggler Alleviation Module*, foresees the machine performance to provide further and effective straggler mitigations. Additionally, it guarantees the effectiveness to speculative tasks. Figure 2 shows the architecture of the proposed straggler mitigation framework.

IV. THE IMPLEMENTATION DETAILS OF THE PROPOSED FRAMEWORK

A. STRAGGLER DECISION ENGINE MODULE

1) THE INITIAL HISTORICAL CALCULATION

The straggler decision engine module include the initial historical component that used to gather the information of task execution logs and preliminary calculations. In Spark, the lifetime of an executed task is comprised into three time periods (TP_1, TP_2, and TP_3). These periods are deserialization of task period, running task period, and serialization of task results period respectively. The deserialization of task period (TP_1) is the elapsed time spent to deserialize the task object and data. Also, the running task period (TP_2) is the elapsed time that spent running this task. This includes the time of fetching the shuffle data. While TP_3 is the serialization of task result period which is the elapsed time spent in serializing the task result. It should be noted that the information of tasks' execution is collected for each node n. So, the mean execution time of each period for a node n is recorded as $\overline{TP1}_n$, $\overline{TP2}_n$, and $\overline{TP3}_n$. Therefore, the total time of each task for all periods can be computed as follows in (1)

$$Task_{time} = \sum_{j=1}^{3} TP_n(j) \tag{1}$$

where j is the period id and n is the node id.

Also, the total time for all completed and successful tasks on node n is defined as $\overline{MT}_n$. In this context, the average total time of all successful tasks is ($\overline{MT}$).

2) DYNAMIC WEIGHTED STRAGGLER DECISION CRITERION

According to the proposed framework, the judgment of a task to be a straggler or not is dynamically identified. To increase the speculation efficiency, the straggler decision criterion is

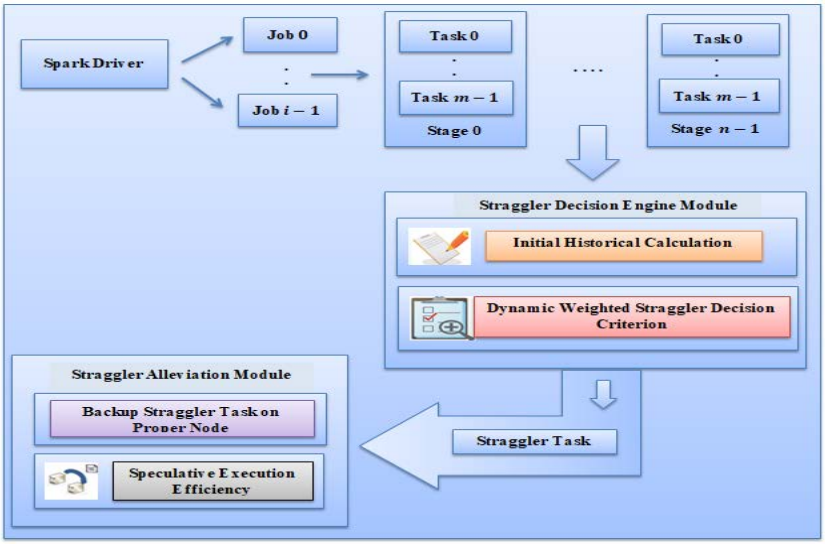

FIGURE 2. The architecture of the proposed straggler mitigation framework.

taken according to four weighted coefficients. These coefficients are the job quality of service limitation (C_Q), the stage proceeding behavior (C_P), the processing bandwidth (C_{BW}), and the cluster utilization level (C_U). Afterwards, on the basis of the system administrator preferences, the priorities of the above parameters are determined through the weights W_0, W_1, W_2 and W_3. The Straggler decision (D_S) can be computed in Eq. (2).

$$D_S = W_0 C_Q + W_1 C_P + W_2 C_{BW} + W_3 C_U \tag{2}$$

The straggler decision depends on the residual time to complete a task (Rt) as in Eq. (3).

$$Rt = TimeFactor_n\,(p) \times \sum_{j=p}^{3} \overline{TP}_n\,(j) \tag{3}$$

where the $TimeFactor_n\,(p)$ is computed as the ratio of the running time at this period of that task to the mean value of the running time within that period. Finally, if the total running time of a task is greater than $D_S \times \overline{MT}$. then, the task is added to a speculation queue. The pseudo-code for the Dynamic Weighted Straggler Decision Module is illustrated in Algorithm 1.

The coefficients calculations can be illustrated as follows:

The job quality of service limitation coefficient (C_Q): This coefficient is one of the crucial considerations for a straggler decision. This coefficient depends on the application nature, where some jobs' deadline could not be caught. These jobs lead to poor application performance. So, it is preferable to execute such tasks rather than preserve resources on the cluster. The following Algorithm computes the value of the coefficient considering the limitation time for a job quality of service as defined in Eq. (4).

$$C_Q = \begin{cases} \dfrac{Q_T}{MT} & if\ (Q_T \geq \max\,(Task_{time})) \\ \dfrac{min(Task_{time})}{MT} & otherwise \end{cases} \tag{4}$$

where Q_T is the required time for a task and considered as the quality of service coefficient. When Q_T value is more than the maximum $Task_{time}$, this indicates a long limitation time for quality of service. Therefore, there is no need to make a large number of clones because there are no risks of intensive performance implications. In this case, the coefficient is computed as the division between the quality of service time limitation (Q_T) and the mean time (MT). When Q_T value is less than the maximum $Task_{time}$, the coefficient is set to the minimal $Task_{time}$ divided by the mean time (MT) as in Eq. (4). The pseudo-code for calculating the job quality of service limitation coefficient (C_Q) is illustrated in Algorithm 2.

The stage proceeding behavior coefficient (C_P): Ideally, a speculation should preferably be detected early in the job lifecycle. This saves cluster resources and that reflects on the job completion time. As a consequence, it is vital to examine the stage proceeding behavior to get effective straggler detection. The proceeding behavior in the stage can be computed as the ratio between the number of completed tasks in the stage (n) and the total number of tasks in that stage (m) as follows in (5).

$$Proceeding = \frac{n}{m} \tag{5}$$

Then, the average proceeding $Proceeding_{avg}$ for all stages can be computed, which indicates the present proceeding in the entire job lifecycle. After that, the computation of the stage proceeding behavior coefficient (C_P) at time t is given in (6)

$$C_P = Proceeding_{avg} - P_{th} \tag{6}$$

Algorithm 1 Dynamic Weighted Straggler Decision Module

Input:

- $\overline{TP1}_n$: Mapping between node and the mean deserialization of task period
- $\overline{TP2}_n$: Mapping between node and the mean running task period
- $\overline{TP3}_n$: Mapping between node and the mean serialization of task result
- Parallel jobs in Spark $J = \{J_1, J_2, \ldots, J_i\}$
- The list of nodes in spark cluster $n = \{n_1, n_2, \ldots, n_x\}$
- W_0, W_1, W_2 and W_3: The coefficients weights can be specified by system administrator
- $\overline{MT}$: The average total time of all successful tasks
- Rt: The residual time for a task to complete
- C_Q : The job quality of service limitation coefficient // defined in Algorithm 2
- C_P: The stage proceeding behavior coefficient // defined in Algorithm 3
- C_{BW}: The proceeding bandwidth coefficient // defined in Algorithm 4
- C_U : The cluster utilization level coefficient // defined in Algorithm 5

Output: find speculation task

1. Get_C_Q ( ) //as in Algorithm 2
2. Get_C_P ( ) //as in Algorithm 3
3. Get_C_{BW} ( ) //as in Algorithm 4
4. Get_C_U ( ) //as in Algorithm 5
5. $D_S = W_0 C_Q + W_1 C_P + W_2 C_{BW} + W_3 C_U$
6. For ($\tau = 0$ *to* $k - 1$) do
7. If τ is not completed then
8. Get $\overline{MT}$ [τ]
9. If τ in the first period
10. $Rt = \frac{Run\ time(\tau)}{\overline{TP1}_\eta} \times (\overline{TP2}_n + \overline{TP3}_n)$
11. Else if τ is in the second period
12. $Rt = \frac{Run\ time(\tau)}{\overline{TP2}_n} \times (\overline{TP3}_n)$
13. End if
14. End if
15. If ($Run\ time + Rt > D_S \times \overline{MT}$)
16. If ($Spark - speculation\ (\tau) == false$)
17. $AddInSpeculationQueue(\tau)$
18. End if
19. End if
20. End if
21. End for
22. /* end of Dynamic Weighted Straggler Decision Algorithm*/

where P_{th} is the proceeding threshold that indicates the specified maximum point during the lifespan, which eligible for straggler decisions. The value of P_{th} is variable $\in$ [0:1] as the administrator preference. As a result, when a task slows down at its final stages, the created replica has less chances of finishing before the straggler. As a consequence, to avoid ineffective speculation, it is reasonable to raise the threshold value in response to late progress. Also, it is acceptable to reduce the threshold value early in the task lifecycle to motivate replica generation. This is because the replica should have a greater chance of surpassing the original task. In such situations, it is preferable to run these tasks instead of conserving resources on the cluster. The pseudo-code for calculation the stage proceeding behavior coefficient (C_P) is illustrated in Algorithm 3.

Algorithm 2 The Job Quality of Service Limitation Coefficient (C_Q)

Input:

Q_T : Quality of service limitation time

Output: The job quality of service limitation coefficient (C_Q)

Get_C_Q ( )

1. Get $\overline{MT}$
2. Get max ($Task_{time}$)
3. Get min ($Task_{time}$)
4. If ($Q_T \geq$ max ($Task_{time}$)
5. $C_Q = \frac{Q_T}{MT}$
6. Else
7. $C_Q = \frac{\min(Task_{time})}{MT}$
8. End if
9. /* end of job quality of service limitation coefficient*/

Algorithm 3 The Stage Proceeding Behavior Coefficient (C_P)

Input:

- Parallel jobs in spark app $J = \{J_1, J_2, \ldots, J_i\}$
- A job decomposed into single or multiple stages $S = \{S_1, S_2, \ldots .S_j\}$
- n : the number of completed tasks in the stage.
- m : the total number of tasks in that stage

Output: The stage proceeding behavior coefficient (C_P)

Get_C_P ( )

1. For ($S = 0$ *to* $j - 1$) do
2. $Proceeding[s] = \frac{n}{m}$
3. $Proceeding_{sum} += Proceeding[s]$
4. End for
5. $Proceeding_{avg} = \frac{Proceeding_{sum}}{j}$
6. $C_P = Proceeding_{avg} - P_{th}$
7. /* end of job quality of service limitation coefficient*/

$*Proceeding_{avg}$: The average proceeding for all stages.
$*P_{th}$: The proceeding threshold

The pseudo-code for calculation the stage proceeding behavior coefficient (C_P) is illustrated in Algorithm 3.

The processing bandwidth coefficient (C_{BW}): This coefficient measures a job's process speed in order to identify slow tasks more quickly. The amount of processed data for a given time period is used to calculate the process speed. The processing bandwidth ($Processing_{BW}$) for a stage can be computed as the ratio between processed data size in the stage

Algorithm 4 The Proceeding Bandwidth Coefficient (C_{BW})

Input:

- Parallel jobs in spark app $J = \{J_1, J_2, \ldots, J_i\}$
- A job decomposed into single or multiple stages $S = \{S_1, S_2, \ldots .S_j\}$

Output: The proceeding bandwidth coefficient (C_{BW})

Get_C_{BW}()

1. For ($S = 0$ *to* $j - 1$) do
2. $Processing_{BW}[s] = \frac{Processeddatasize}{Processingtime}$
3. $Rate_{BW}[s] = \frac{Processing_{BW}}{data}$
4. $Rate_{sum} += Rate_{BW}[s]$
5. End for
6. $Rate_{avg} = \frac{Rate_{sum}}{j}$
7. $C_{BW} = Rate_{avg} - Rate_{th}$
8. /* end of job the proceeding bandwidth coefficient */

$*$ $Processing_{BW}$: The processing bandwidth
$*$ $Rate_{BW}$: The processing bandwidth rate
$*$ $Rate_{th}$: The processing bandwidth rate threshold

to the processing time in that stage as in Eq. (7).

$$Processing_{BW} = \frac{Processed\ data\ size}{Processing\ time} \tag{7}$$

After that, the processing bandwidth rate ($Rate_{BW}$) can be computed by dividing $Processing_{BW}$ to the size of data to be processed ($data$) as in Eq. (8).

$$Rate_{BW}(t) = \frac{Processing_{BW}}{data} \tag{8}$$

Then, the average bandwidth rate $Rate_{avg}$ for all stages can be computed and indicates the present bandwidth rate for the entire job lifecycle. The computation of the processing bandwidth coefficient is defined in Eq. (9).

$$C_{BW}(t) = Rate_{avg} - Rate_{th} \tag{9}$$

where $Rate_{th}$ is the processing bandwidth rate threshold that indicates the specified maximum point during the lifespan eligible for straggler decision. The value of $Rate_{th}$ is variable $\in$ [0:1] as the system administrator preference. The pseudo-code for computing the processing bandwidth coefficient (C_{BW}) is illustrated in Algorithm 4.

The cluster utilization level coefficient (C_U): The overhead incurred by speculations is considered as a significant factor while dealing with stragglers. It should be noted that the creation of replicas in high resource utilization cluster may increase straggler occurrence. While in low cluster utilization levels, an additional speculation might be needed for enhancing the job execution time. Therefore, the current cluster utilization level should be considered for getting efficient dynamic straggler calculations. Many parameters are considered by cluster utilization level coefficient such as CPU utilization, Memory utilization, Disk I/O utilization and Network bandwidth utilization as in Eqs. (10), (11), (12) and (13) [17].

- *CPU utilization* (n) is defined as the ratio of CPU busy time to the time interval for a node n.

$$cpu_{util} = \frac{cpu\_busy}{time\_interval} \tag{10}$$

- *Memory utilization* (n) is the ratio of the maximum memory accessed during the time interval by tasks to the size of physical memory for a node n.

$$mem_{util} = \frac{\max\_mem\_accessed}{physical\_mem\_size} \tag{11}$$

- *Disk I/O utilization* (n) is the ratio of the disk's read/write volumes consumed by tasks during the time interval to the effective maximum bandwidth of disc I/O for a node n.

$$disk_{util} = \frac{read\_vol + write\_vol}{max\_disk\_bandwidth} \tag{12}$$

- *Network bandwidth utilization* (n) is the ratio of receiving/sending tasks' traffic over a given period of time to the effective maximum bandwidth for a node n.

$$net_{util} = \frac{rec\_vol + send\_vol}{max\_bandwidth} \tag{13}$$

The calculation of cluster utilization level coefficient (C_U) at time t is given as in Eq 14.

$$C_U(t) = avg\left(\begin{pmatrix}\left(\frac{Sum_{cpu}}{n} - cpu_{th}\right), \left(\frac{Sum_{mem}}{n} - mem_{th}\right),\\ \left(\frac{Sum_{disk}}{n} - disk_{th}\right), \left(\frac{Sum_{net}}{n} - net_{th}\right)\end{pmatrix}\right) \tag{14}$$

where cpu_{th} is the CPU utilization threshold, mem_{th} is the memory utilization threshold, $disk_{th}$ is the disk utilization threshold, net_{th} is the network bandwidth utilization threshold and n is the total number of cluster nodes. The values of cpu_{th}, mem_{th}, $disk_{th}$, and net_{th} are variable $\in$ [0:1] as the administrator preference. When the average utilization parameter exceeds the utilization thresholds specified by the user, this leads having positive C_U values. The pseudo-code for computing the cluster utilization level coefficient (C_U) is illustrated in Algorithm 5.

B. STRAGGLER ALLEVIATION MODULE

1) BACKUP STRAGGLER TASK ON PROPER NODE

One of the major challenges of speculative execution is the backup of tasks at appropriate nodes. Since every node's capability may vary, it is essential to have an appropriate metric to measure the performance of heterogeneity nodes. Therefore, the capability of a node can be obtained through the amount of tasks completed and total tasks processed as in (15) [33]:

$$capability_{node} = \frac{Number\ of\ Completed\ Tasks}{Number\ of\ Processed\ Tasks} \tag{15}$$

TABLE 1. The time and space complexity analysis for the proposed Algorithms.

Module Name	Algorithm No.		Time Complexity	Space Complexity
Straggler Decision Engine Module "Algorithm 1"	Algorithm 2		$3O(k) + O(1)$	$4O(1)$
	Algorithm 3		$O(j) + 2O(1)$	$O(j) + 4O(1)$
	Algorithm 4		$O(j) + 2O(1)$	$2O(j) + 8O(1)$
	Algorithm 5		$O(x) + 2O(1)$	$10O(x) + 10O(1)$
	Algorithm 1	Overall	$3O(k) + 2O(j) + 3O(x) + 7O(1) + O(k^2) + 4O(1)$	$O(k) \times [3O(j) + 10O(x) + 26O(1)]$
		Approx.	$O(k^2) \quad \forall\, k \gg j, k \gg x$	$O(k \times (j + x))$
Straggler Alleviation Module "Algorithm 6"	Algorithm 6	Overall	$O(k^2) + 3O(1)$	$6O(1)$
		Approx.	$O(k^2)$	$O(1)$

Algorithm 5 The Cluster Utilization Level Coefficient (C_U)

Input:

The list of nodes in spark cluster $n = \{n_1, n_2, \ldots., n_x\}$

Output: The cluster utilization level coefficient (C_U)

1. Get_$C_U()$
2. For $(n = 1\ to\ x)$ do
3. $Sum_{cpu} += \frac{cpu\_busy[n]}{time\_interval[n]}$
4. $Sum_{mem} += \frac{\max\_mem\_accessed[n]}{physical\_mem\_size[n]}$
5. $Sum_{disk} += \frac{read\_vol[n]+write\_vol[n]}{max\_disk\_bandwidth[n]}$
6. $Sum_{net} += \frac{rec\_vol[n]+send\_vol[n]}{\max\_bandwidth[n]}$
7. End for
8. $avg_{Util} = avg\left(\left(\frac{Sum_{cpu}}{x} - cpu_{th}\right), \left(\frac{Sum_{mem}}{x} - mem_{th}\right), \left(\frac{Sum_{disk}}{x} - disk_{th}\right), \left(\frac{Sum_{net}}{x} - net_{th}\right)\right)$
9. $C_U = avg_{Util}$
10. /∗ end of the cluster utilization level coefficient ∗/

It should be noted that the completed tasks are the tasks that successfully finished execution, while failed tasks and the tasks lost by speculative execution are included in the processed tasks.

Algorithm 6 Speculative Execution Efficiency Algorithm

Input:

- Parallel jobs in spark app $J = \{J, J_2, \ldots., J_i\}$
- A job decomposed into single or multiple stages $S = \{S_1, S_2, \ldots .S_j\}$

Output:

1. For each task in the straggler task queue
2. Get_$Rt()$ //as in Eq. 3
3. Get_$Bt()$ //as in Eq. 16
4. If $(Rt < Bt)$
5. Ignore this task
6. End if
7. Get profit of backup and profit of not backup
8. Select the task which will do the maximum profit and assign it on the highest node capability //as in Eq. 15
9. If (the cluster is overloaded && $Rt \geq 2 * Bt$) then
10. Delete the original task
11. End if
12. End for
13. /* end of Speculative Execution Efficiency Algorithm */

2) SPECULATIVE EXECUTION EFFICIENCY

It is preferable to develop an efficient method for determining whether backup tasks should be started or not. Therefore for each task in the straggler tasks queue, we compute the benefit of having backup and the benefit of not having backup and picking up a task which give maximum benefit. In [34], the cost parameter is considered as the busy time for computing resources. But, when the speculative execution saves time, it is considered as a benefit parameter. The benefit of backing up a task will be measured with taking into account assigning another slot for back up task. Since both the original and the backup must continue to run till the task is completed. While conserving one slot is equal to the difference between the residual time and the backup time. The residual time to complete a task can be computed as referred in (3). The mean execution time of three periods of all tasks is adjusted by

TABLE 2. Illustrative example for the straggler decision criteria in low and high state.

Task	Task time (s)	coefficient			D_S		Decision	
					Low	High	Low	High
T1	2	C_Q	Low	1.14	0.035	0.838	**S**	**N**
			High	2				
		C_P	Low	-0.4				
			High	0.45				
		C_{BW}	Low	-0.25				
			High	0.42				
		C_U	Low	-0.35				
			High	0.48				
T2	5	C_Q	Low	1.14	0.035	0.838	**S**	**N**
			High	2				
		C_P	Low	-0.4				
			High	0.45				
		C_{BW}	Low	-0.25				
			High	0.42				
		C_U	Low	-0.35				
			High	0.48				
T3	4	C_Q	Low	1.14	0.035	0.838	**S**	**N**
			High	2				
		C_P	Low	-0.4				
			High	0.45				
		C_{BW}	Low	-0.25				
			High	0.42				
		C_U	Low	-0.35				
			High	0.48				
T4	6	C_Q	Low	1.14	0.035	0.838	**S**	**N**
			High	2				
		C_P	Low	-0.4				
			High	0.45				
		C_{BW}	Low	-0.25				
			High	0.42				
		C_U	Low	-0.35				
			High	0.48				
T5	19	C_Q	Low	1.14	0.035	0.838	**S**	**S**
			High	2				
		C_P	Low	-0.4				
			High	0.45				
		C_{BW}	Low	-0.25				
			High	0.42				
		C_U	Low	-0.35				
			High	0.48				

Data factor to compute the backup time. *Data factor* denotes the proportion of the task's input size to the mean input size of all tasks. The backup time is computed as follows in Eq. (16)

$$Bt = Data\ factor \times \sum_{p=1}^{3} \overline{TP}_n\,(p) \tag{16}$$

Otherwise, the cost of not backing will be one slot of residual time which is consumed with no benefit. Eqs. (17) and (18) show the $benefit_{backup}$ and $benefit_{not\_backup}$ as follows:

$$benefit_{backup} = \alpha \times (Rt - Bt) - \beta \times 2 \times Bt \tag{17}$$

$$benefit_{not\_backup} = \propto \times 0 - \beta \times Rt \tag{18}$$

where α and β are benefit and cost weights respectively. Similraly, Rt and Bt are the residual and backup times respectively. When the $benefit_{backup}$ is greater than the $benefit_{not\_backup}$, the task is considered as a slow task as in Eq. (19).

$$benefit_{backup} > benefit_{not\_backup} \leftrightarrow \frac{Rt}{Bt} > \frac{\propto +2\beta}{\alpha+\beta} \tag{19}$$

when replacing $\frac{\beta}{\alpha}$ with γ, we can obtain:

$$\frac{Rt}{Bt} > \frac{1+2\gamma}{1+\gamma} \tag{20}$$

where

$$\gamma = load_{factor} = \frac{number_{pending\,tasks}}{number_{free\,slots}} \tag{21}$$

When there are a lot of pending tasks, γ gives higher values close to two. Consequently, fewer tasks will be backed up. But, when the cluster contains multiple free slots converges to zero, resulting in no cost for speculative execution. As a result, the use of the load factor γ conforms and meets the requirements. As a consequent, the original task will be deleted in case of the cluster is overloaded and there are no free slots. Also, when the residual time is large enough for safe restart, the residual time is at least double the backup time as in Eq. (22).

$$Rt \geq 2 * Bt \tag{22}$$

Algorithm 6 illustrates the pseudo-code for Speculative Execution Efficiency Algorithm.

V. COMPLEXITY ANALYSIS

In this section, we examine the time and space complexity analysis for the proposed algorithms from Algorithm 1 to 6. As demonstrated, the main modules of the proposed framework include a Straggler Decision Engine Module and Straggler Alleviation Module. For a Straggler Decision Engine Module which is represented by Algorithm 1, where it uses algorithms 2 to 5 as in lines 1 to 4 in addition to the decision criterion in lines 5 to 21 as in Algorithm 1. As a consequence, Table 1 shows the Straggler Decision Engine Module requires $O\left(k^2\right)$ and $O\left(k \times (j+x)\right)$ for the time and space complexity respectively. Similarly, Straggler Alleviation Module requires time complexity of $\mathrm{O}\left(k^2\right)$ and space complexity of $O\left(1\right)$ where k the number of tasks, j the number of stages and x the number of nodes.

VI. AN ILLUSTRATIVE EXAMPLE

For further clarification, an illustrative example is introduced to demonstrate the key idea of the proposed framework. This example shows the proposed criterion for changing the straggler decision threshold based on different coefficients for different scenarios. One scenario clarifies the straggler decision threshold when the job at a low state, where $Proceeding_{avg}$, $Rate_{avg}$ or avg_{Util} are less than 0.25. Another scenario shows the straggler decision threshold for high states, where the values for $Proceeding_{avg}$, $Rate_{avg}$ or avg_{Util} are more than 0.75. For clarification purpose, we consider a particular job having five tasks $T1, T2, T3, T4$ and $T5$, and the values of the weights W_0, W_1, W_2 and W_3 are $\in$ [0:1] based on the administrator preferences in such a way that $\sum_i W_i = 1$. In this case study, we use equal weights set to 0.25. Also, the threshold parameters as P_{th}, $Rate_{th}$, cpu_{th}, mem_{th}, $disk_{th}$ and net_{th} are set to 0.5. These settings characterize a common configuration for the two scenarios, and can be customized for different purposes. Then, the coefficients of the stage proceeding behavior (C_P), the processing bandwidth (C_{BW}), and the cluster utilization level (C_U)can be computed as in Eqs. (6), (9) and (14). Therefore Straggler decision (D_S) can be obtained as in Eq. 2. In this context, we can denote each task to be either a straggler task (S) or a none straggler task (N). Table 2 shows the straggler decision (D_S) for low and high states with $Q_T = 8$ and 14 respectively. For low states, the proposed criterion encourages replica creation in early stages of the life cycle by generating smaller coefficients while ensuring that the quality of service. While in high states, the proposed criterion creates fewer replicas to avoid overload of the system. Also, it guarantees the realization of quality of service requirement.

TABLE 3. Cluster configurations.

Cluster configurations			
Node	Main Memory	CPU cores	Storage
Master	16 G	12	100 G
Slave-1	12 G	8	50 G
Slave-2	8 G	4	50 G
Slave-3	4 G	4	50 G

TABLE 4. Software configurations.

Software configurations	
Operating System	Ubuntu 18.04
Spark	3.0.0
Hadoop	2.7.3
JDK	1.8
Scala version	2.12.10

TABLE 5. Benchmark applications, and Workload types and sizes.

Application	Benchmark category	Workload type	Workload size
TeraSort	Micro Benchmark	CPU and I/O intensive	3GB,6GB,12GB
WordCount	Micro Benchmark	CPU intensive	10GB,20GB,30GB
K-means	Machine learning	CPU and I/O intensive	1K,10K,100K samples

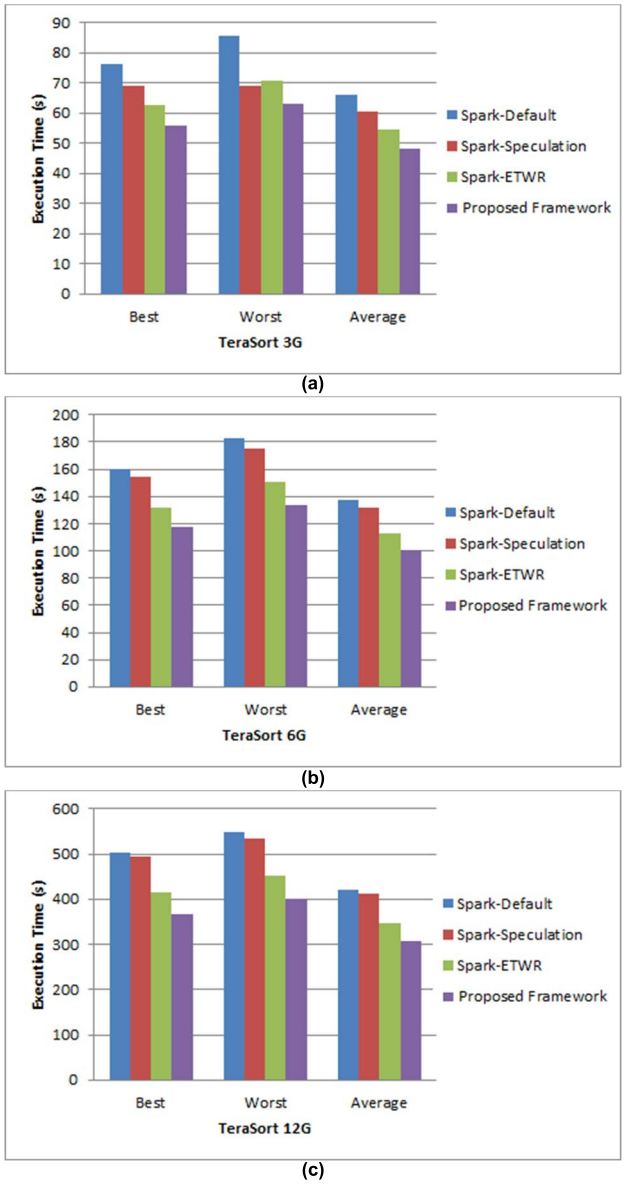

FIGURE 3. The execution time of TeraSort at workload: (a) 3 GB, (b) 6 GB, and (c) 12 GB.

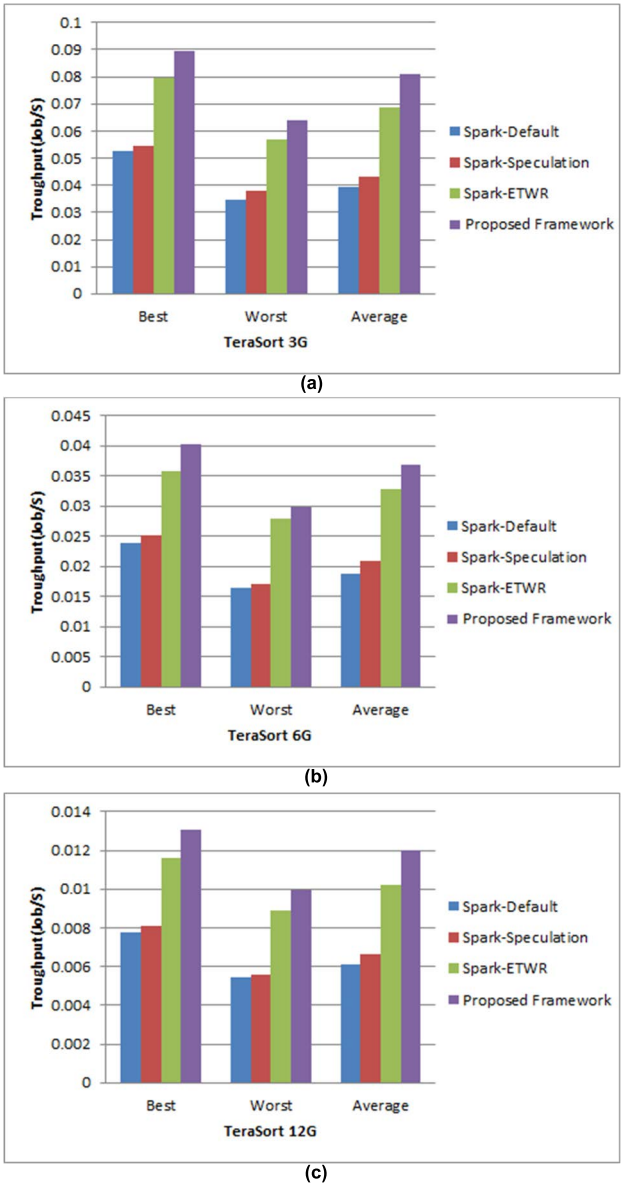

FIGURE 4. The throughput of TeraSort at workload: (a) 3 GB, (b) 6 GB, and (c) 12 GB.

VII. PERFORMANCE EVALUATION

In this section, the performance of the proposed framework is assessed on a spark cluster with a diverse set of nodes. Also, the proposed framework is compared with Spark-Default, Spark-Speculation and the work in [33] (marked as Spark-ETWR) in various benchmarks at different input sizes. The performance is evaluated in terms of the job execution time that refers to the elapsed time from the beginning to the end of the job in seconds. Also, the number of completed jobs per second for a cluster is denoted as the cluster throughput. In the following sub-sections, we provide an overview of the platform, and the benchmarks that are used to evaluate the performance as in Section VI-A, Section VI-B respectively. Finally, section VI-C illustrates the experimental results.

A. PLATFORM

The experimental big data cluster used in this work consists of four virtual machines that are made up of a master and three workers. The cluster classification and the software configurations are demonstrated in Table 3. The cluster applied with 28 cores and 40 GB memory space.

The software configurations are illustrated Table 4. Also, the block size of the Hadoop Distributed File System (HDFS) used is 128 MB.

B. BENCHMARKS

There are many spark applications incur a lot of resources. Based on many researches that conducted on spark

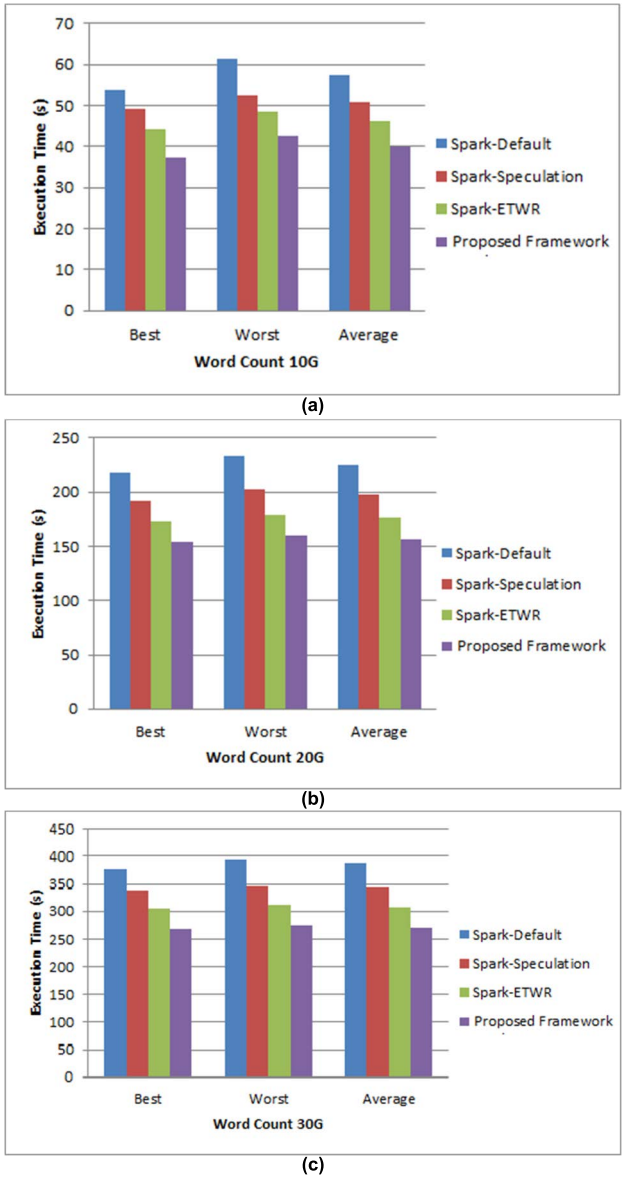

FIGURE 5. The execution time of WordCount with data size: (a) 10 GB, (b) 20 GB, and (c) 30 GB.

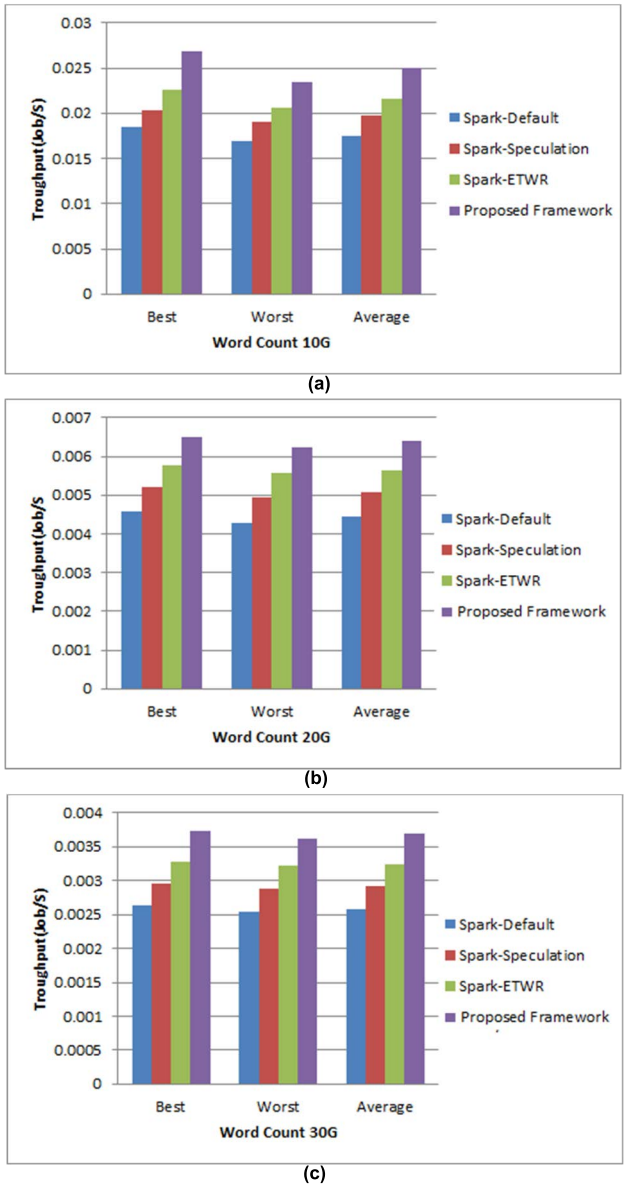

FIGURE 6. The throughput of WordCount at workload: (a) 10 GB, (b) 20 GB, and (c) 30 GB.

performance [35], the proposed framework is applied on three different benchmarks. In this paper, the benchmarks are applied with various workloads to deliver a comprehensive performance evaluation. Table 5 shows different benchmark applications with various workloads and sizes.

1) TeraSort

It is a common benchmark [36] that measures the time required to sort a given amount of randomly distributed data on a particular cluster. It consists of three functions as TeraGen, TeraSort and TeraValidate. The TeraGen function is written in Java that generates the input data. The TeraSort function uses MapReduce to sort the data, and the TeraValidate function verifies the sorted data output. The workload size we used in our experiments are 3, 6 and 12 GB text data.

2) WordCount

This application is a popular benchmark that counts the number of times each word appears in the input data file. It has two stages: stage 0 and stage 1. Stage 0 read data from the HDFS, and performs map and reduce operations. Stage 1 read the output data of stage 0 through shuffler, and performs reduce operations. In our experiment, we use the real dataset, *enwiki dump progress* [37] of size 10, 20 and 30 GB.

3) K-MEANS CLUSTERING ALGORITHM

It partitions a set of data point into K clusters. It is a commonly used on large data sets which automatically classifies the input data points into K clusters. So, it is appropriate candidate for parallelization. To generate the datasets,

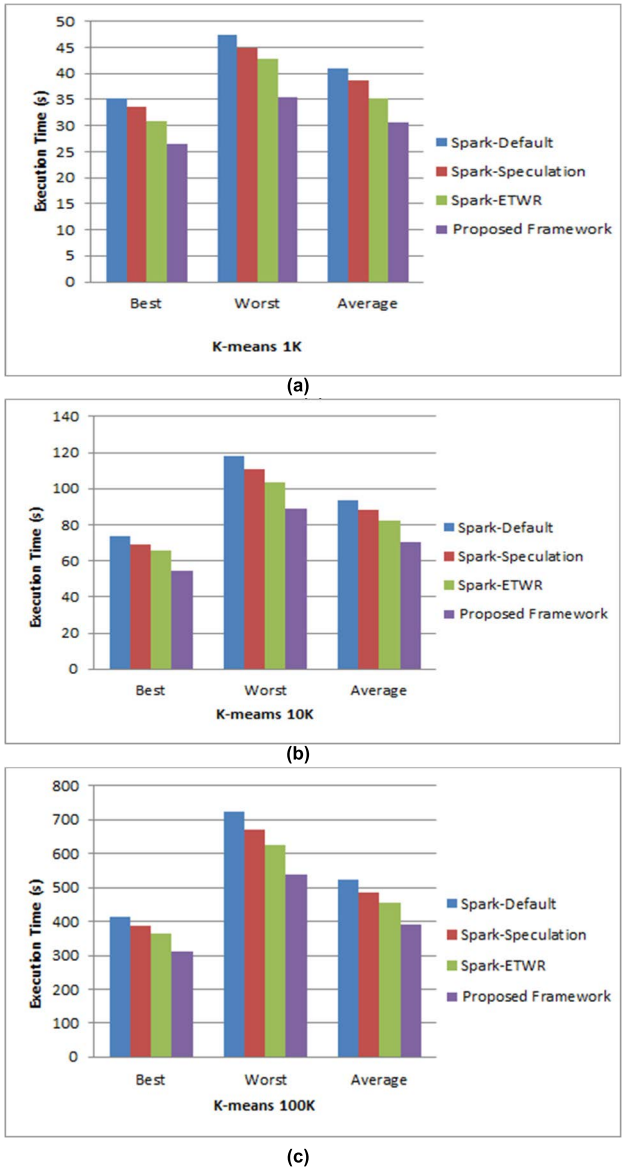

FIGURE 7. The execution time of K-means at samples of size: (a) 1K, (b) 10K, and (c) 100K.

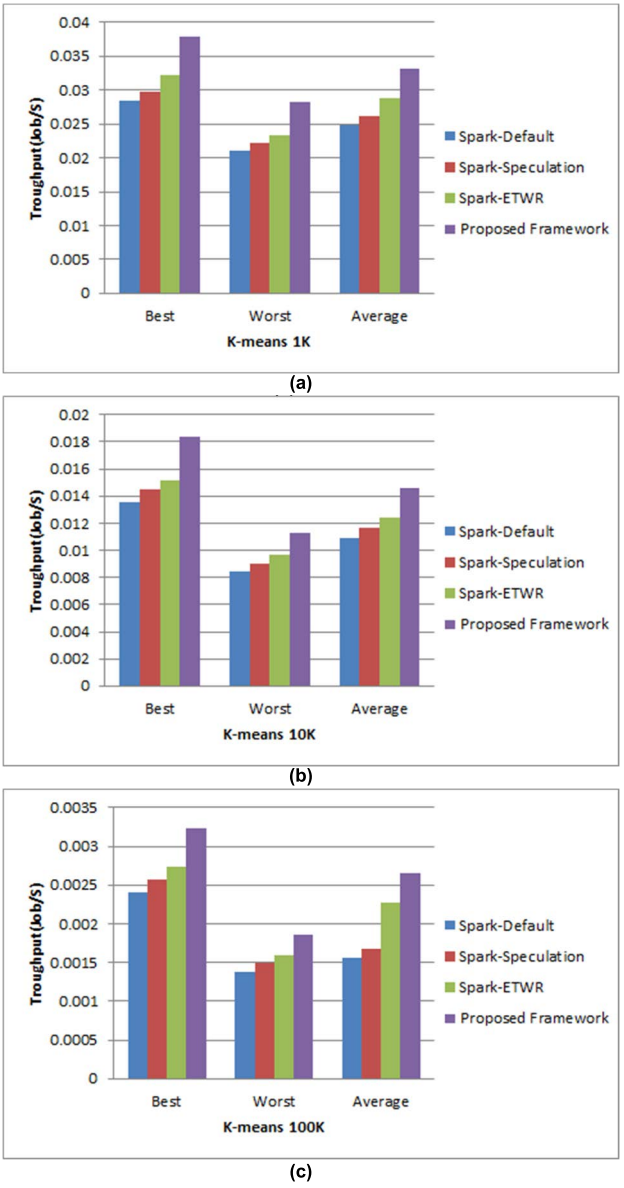

FIGURE 8. The throughput of K-means 100K at samples of size: (a) 1K, (b) 10K, and (c) 100K.

the scikit-learn python library is used to generate data points with clustering tendency, and sizes 1000, 10000, and 100000 points. In addition, the experiments are conducted at $K = 13$, and *maximum iterations* $= 50$.

C. EXPERIMENTAL RESULTS AND DISCUSSIONS

In this section, the results obtained after running the benchmarks are evaluated based on Spark framework. Each experiment is performed 20 times and the average value was calculated. For the purpose of comparisons, the proposed framework is compared with Spark-Default, Spark-Speculation, Spark-ETWR. The performance is evaluated in terms of job execution time as well as the cluster throughput. We have conducted a number of experiments to examine the total execution time. It is worth noting that the execution time is influenced by many parameters as the benchmark category, the input data sizes and the assigned node capability. Furthermore, the throughput is calculated likewise, the job numbers over the execution time. In TeraSort benchmark, Figure 3 shows the performance comparisons with Spark-Default, Spark-Speculation, and Spark-ETWR at workloads 3 GB, 6 GB and 12 GB.

On average, the proposed framework achieves 26.8% less execution time than Spark- Default. Also, it gives 24.3% execution time reduction against Spark-Speculation, and 14% reduction compared with Spark- ETWR. Furthermore, show that the cluster throughput increased by 46.3%, 42.8% and 15% with respect to the competitive methods respectively.

Similarly for WordCount benchmark as in Fig. 5, the proposed framework achieves a reduction of 28.7% in the

average execution time compared with Spark-Default. Additionally, it provides 21.5% execution time reduction compared with Spark-Speculation and 13.6% reduction with Spark- ETWR.

Also, Figure 6 is showing that the cluster throughput is improved by 30.7%, 24.2% and 15.7% compared with other competing methods.

For K-means clustering algorithm as in Fig. 7 and 8, the results are consistent with TeraSort and WordCount benchmarks, where, the proposed framework achieves 23.5%, 19.6% and 11% execution time reduction, and 25.4%, 22.2% and 15.4% throughput improvements against Spark-Default, Spark-Speculation and Spark-ETWR respectively.

As demonstrated, the proposed framework is clearly outperforming compared with Spark-Default, Spark-Speculation and Spark-ETWR, where the proposed framework uses a dynamic criterion to determine the closest straggler tasks. This criterion is based on multiple coefficients to achieve a reliable straggler decision. It shows that the suggested criterion gives an intelligent decision that can locate the straggler more precisely. As a consequent it improves the throughput and reduces the execution time.

VIII. CONCLUSION

This paper has introduced a new Optimized Straggler Mitigation Framework. The proposed framework's purpose is to minimize job execution time through speculative execution to reduce the impact of the straggler, and consequently boosting cluster throughput. To achieve this goal, the proposed algorithm uses a dynamic criterion to evaluate the most suitable tasks for speculation. This criterion is based on multiple coefficients to achieve a reliable straggler decision. As well as, it guarantees the effectiveness of detecting and treating speculative tasks to improve the cluster performance. As a consequence, the consumed resources can be reduced by restarting tasks to achieve the optimal goal. To examine the reliability of the proposed framework, several experiments have been carried out on various benchmarks having CPU bound and I/O intensive with different workloads. Results showed that the proposed framework achieved 25.4% to 46.3% higher cluster throughputs with lower applications' execution time of 23.5% to 30.7% reduction compared with Spark-3.0.0. In the future, further investigation will be carried out to consider effective machine learning techniques for enhancing the robustness and reliability of the proposed framework.

REFERENCES

[1] Z. Chen, W. Huang, L. Ma, H. Xu, and Y. Chen, "Application and development of big data in sustainable utilization of soil and land resources," *IEEE Access*, vol. 8, pp. 152751–152759, 2020.

[2] G. Mokhtari, A. Anvari-Moghaddam, and Q. Zhang, "A new layered architecture for future big data-driven smart homes," *IEEE Access*, vol. 7, pp. 19002–19012, 2019.

[3] I. Kotenko, I. Saenko, and A. Branitskiy, "Framework for mobile Internet of Things security monitoring based on big data processing and machine learning," *IEEE Access*, vol. 6, pp. 72714–72723, 2018.

[4] S. Sakr, Z. Maamar, A. Awad, B. Benatallah, and W. M. P. van der Aalst, "Business process analytics and big data systems: A roadmap to bridge the gap," *IEEE Access*, vol. 6, pp. 77308–77320, 2018.

[5] Y. Xu, Y. Sun, J. Wan, X. Liu, and Z. Song, "Industrial big data for fault diagnosis: Taxonomy, review, and applications," *IEEE Access*, vol. 5, pp. 17368–17380, 2017.

[6] S. K. Roy, R. Devaraj, A. Sarkar, and D. Senapati, "SLAQA: Quality-level aware scheduling of task graphs on heterogeneous distributed systems," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 5, pp. 1–31, Sep. 2021.

[7] S. K. Roy, R. Devaraj, A. Sarkar, K. Maji, and S. Sinha, "Contention-aware optimal scheduling of real-time precedence-constrained task graphs on heterogeneous distributed systems," *J. Syst. Archit.*, vol. 105, May 2020, Art. no. 101706.

[8] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154300–154316, 2019.

[9] P. Mika and G. Tummarello, "Web semantics in the clouds," *IEEE Intell. Syst.*, vol. 23, no. 5, pp. 82–87, Sep./Oct. 2008.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2010, pp. 1–7.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2012, pp. 15–28.

[13] L. Jin, W. Fu, M. Ling, and L. Shi, "A fast cross-layer dynamic power estimation method by tracking cycle-accurate activity factors with spark streaming," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 4, pp. 353–364, Apr. 2022.

[14] X. Li, B. Yu, G. Feng, H. Wang, and W. Chen, "LotusSQL: SQL engine for high-performance big data systems," *Big Data Mining Anal.*, vol. 4, no. 4, pp. 252–265, Dec. 2021.

[15] J. Wang, X. Wang, C. Ma, and L. Kou, "A survey on the development status and application prospects of knowledge graph in smart grids," *IET Gener., Transmiss. Distrib.*, vol. 15, no. 3, pp. 383–407, Feb. 2021.

[16] E. Nagy, R. Lovas, I. Pintye, Á. Hajnal, and P. Kacsuk, "Cloud-agnostic architectures for machine learning based on apache spark," *Adv. Eng. Softw.*, vol. 159, Sep. 2021, Art. no. 103029.

[17] S. A. Said, M. S. El-Sayed, S. A. Salem, and S. M. Habashy, "A speculative execution framework for big data processing systems," in *Proc. Int. Conf. Inf. Technol. (ICIT)*, Jul. 2021, pp. 616–621.

[18] S. Deshmukh, K. T. Rao, and M. Shabaz, "Collaborative learning based straggler prevention in large-scale distributed computing framework," *Secur. Commun. Netw.*, vol. 2021, pp. 1–9, May 2021.

[19] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters," *IEEE Trans. Serv. Comput.*, vol. 12, no. 1, pp. 91–104, Jan. 2019.

[20] S. Lu, X. Wei, B. Rao, B. Tak, L. Wang, and L. Wang, "LADRA: Log-based abnormal task detection and root-cause analysis in big data processing with spark," *Future Gener. Comput. Syst.*, vol. 95, pp. 392–403, Jun. 2019.

[21] H. Xu and W. Cheong Lau, "Optimization for speculative execution in big data processing clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 530–545, Feb. 2017.

[22] S. S. Gill, X. Ouyang, and P. Garraghan, "Tails in the cloud: A survey and taxonomy of straggler management within large-scale cloud data centres," *J. Supercomput.*, vol. 76, no. 12, pp. 10050–10089, Dec. 2020.

[23] Q. Liu, W. Cai, Z. Fu, J. Shen, and N. Linge, "A smart strategy for speculative execution based on hardware resource in a heterogeneous distributed environment," *Int. J. Grid Distrib. Comput.*, vol. 9, no. 2, pp. 203–214, Feb. 2016.

[24] S. Deshmukh and K. T. Rao, "Straggler identification approach in large data processing frameworks using ensembled gradient boosting in smart-cities cloud services," *Int. J. Syst. Assurance Eng. Manag.*, vol. 13, no. S1, pp. 146–155, Mar. 2022.

[25] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 185–198.

[26] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: Trimming stragglers in approximation analytics," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 289–302.

[27] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using Mantri," in *Proc. 9th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2010, pp. 1–14.
[28] S. Tang, B. He, C. Yu, Y. Li, and K. Li, "A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 71–91, Jan. 2022.
[29] C. Li, H. Shen, and T. Huang, "Learning to diagnose stragglers in distributed computing," in *Proc. 9th Workshop Many-Task Comput. Clouds, Grids, Supercomput. (MTAGS)*, Nov. 2016, pp. 1–6.
[30] T.-D. Phan, G. Pallez, S. Ibrahim, and P. Raghavan, "A new framework for evaluating straggler detection mechanisms in MapReduce," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 4, no. 3, pp. 1–23, Sep. 2019.
[31] H. Xu and W. C. Lau, "Speculative execution for a single job in a MapReduce-like system," in *Proc. IEEE 7th Int. Conf. Cloud Comput.*, Jun. 2014, pp. 586–593.
[32] J. Mathew, "Cluster performance by dynamic load and resource-aware speculative execution," in *Inventive Systems and Control*. Singapore: Springer, 2021, pp. 877–893.
[33] Z. Fu and Z. Tang, "Optimizing speculative execution in spark heterogeneous environments," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 568–582, Jan. 2022.
[34] Q. Chen, C. Liu, and Z. Xiao, "Improving MapReduce performance using smart speculative execution strategy," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 954–967, Apr. 2014.
[35] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops (ICDEW)*, Mar. 2010, pp. 41–51.
[36] O. O'Malley, "Terabyte sort on Apache Hadoop," Yahoo, Sunnyvale, CA, USA, Tech. Rep., 2008, pp. 1–3. [Online]. Available: https://sortbenchmark.org/YahooHadoop.pdf
[37] *Dataset of Enwiki Dump Progress*. Accessed: May 2022. [Online]. Available: https://dumps.wikimedia.org/enwiki

SAMAR A. SAID received the B.Sc. and M.Sc. degrees in electronics, communications, and computer engineering from the Faculty of Engineering, Helwan University, in 2010 and 2018, respectively, where she is currently pursuing the Ph.D. degree. She is also a Teaching Assistant in the university and involved in many of its research projects. Her research interests include big data processing systems, machine learning, mobile cloud computing, algorithms, cloud computing, parallel programming, and data structure.

SHAHIRA M. HABASHY received the B.Sc., M.Sc., and Ph.D. degrees in communications and electronics engineering from Helwan University, Cairo, Egypt, in 1997, 2000, and 2006, respectively. She is currently a Professor with the Department of Computer and Systems Engineering, Helwan University. She is also a Manager of the Technology Innovation Commercialization Office, Helwan University. Her research interests include clustering algorithms, parallel computing, cloud computing, machine learning, and image processing.

SAMEH A. SALEM received the B.Sc. and M.Sc. degrees in communications and electronics engineering from Helwan University, Cairo, Egypt, in 1998 and 2003, respectively, and the Ph.D. degree in electrical engineering and electronics from the University of Liverpool, U.K., in 2008, respectively. In 2014, he got an Honorary Research Fellow Position at the Department of Electrical Engineering and Electronics, University of Liverpool. He was the Head of the Electronics, Communications and Computer Engineering Department, and the Computer and Systems Engineering Department, Helwan University. He is selected to be a Co-ordinator and an Academic Advisor at the Department of Communication and Information Technology, Uninettuno University, Italy, incorporation with the Faculty of Engineering, Helwan University. Furthermore, he is reviewing several proposals and research projects at the National Telecommunication Regulatory Authority (NTRA), Egypt. Since 2018, he has been a Malware Analysis Consultant at the Egyptian Computer Emergency Response Team (EG-CERT). He is currently a Professor of cyber-security and the Director of Research and Technical Solutions at the EG-CERT. His research interests include cyber-security, machine learning, data mining, the Internet of Things (IoT), and parallel and cloud computing.

ELSAYED M. SAAD received the B.Sc. degree in electrical engineering (communication section) from Cairo University, in 1967, the M.Sc. degree from the Electronic and Communication Engineering Department, Cairo University, in January 1974, and the Dip.-Ing. and Dr.-Ing. degrees in electrical engineering from Stuttgart University, in 1977 and 1981, respectively. He has a Military Service, from December 1969 to September 1972. He is currently a Professor of electronic circuit with the Faculty of Engineering, Helwan University. He is an Inventor of Saad's single amplifier SC structure. He has been an Engineering Consultant of the Supreme Council of Universities, since August 2002. He is an International Scientific Member of the ECCTD, in 1983. He is a member of the National Radio Science Committee. He is the author and/or coauthor of 190 articles. He is a member of the Egyptian Engineering Syndicate. He is a member of the European Circuit Society (ECS). He is a member of the Society of Electrical Engineering (SEE). He is a member of the Helwan University Council for Award of Scientific Research. He is a Judge of the National Scientific Award (Egypt National Level).