2017

AutoPath: Harnessing Parallel Execution Paths for Efficient Resource Allocation in Multi-stage Big Data Frameworks

Han Gao
ZHENGYU YANG, *Northeastern University*
Janki Bhimani, *Northeastern University*
Teng Wang
Jiayin Wang, et al.

AutoPath: Harnessing Parallel Execution Paths for Efficient Resource Allocation in Multi-stage Big Data Frameworks

Han Gao*, Zhengyu Yang*, Janki Bhimani*, Teng Wang†, Jiayin Wang†, Bo Sheng†, and Ningfang Mi*

*Dept. of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
†Dept. of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125

***Abstract*—Due to the flexibility of data operations and scalability of in-memory cache, Spark has revealed the potential to become the standard distributed framework to replace Hadoop for data-intensive processing in both industry and academia. However, we observe that the built-in scheduling algorithms in Spark (i.e., FIFO and FAIR) are not optimized for the applications with multiple parallel and independent branches in stages. Specifically, the child stage needs to wait and collect data from all its parent branches, but this wait has no guaranteed upper bound since it is tightly coupled with each branch's workload characteristic, stage order, and their corresponding allocated computing resource. To address this challenge, we investigate a superior solution which ensures all branches acquire suitable resources according to their workload demand in order to let the finish time of each branch be as close as possible. Based on this, we propose a novel scheduling policy, named AutoPath, which can effectively reduce the overall makespan of such kind of applications by detecting and leveraging the parallel path, and adaptively assigning computing resources based on the estimated workload demands during runtime. We implemented the new scheduling scheme in Spark v1.5.0 and evaluated it with selected representative workloads. The experiments demonstrate that our new scheduler effectively reduces the makespan and improves resource utilizations for these applications, compared to the current FIFO and FAIR schedulers.**

***Index Terms*—Spark, Scheduling, Resource Management, Task Assignment, Workload Evaluation & Estimation**

I. Introduction

Big data processing frameworks have evolved rapidly to fulfill the computing demands of various business and customers. Cluster-based platforms have been widely accepted and deployed to handle large-scale datasets in the new era of big data. Represented by Spark [1], the current big data frameworks are coupled well with the traditional SQL-like database operations, and support a large set of extended general data operations. A general concept embedded in the new paradigm is to support multi-stage data processing, i.e., an ordered sequence of operations are applied on input data, and each stage's output data is an input for the next stage. Each stage consists of multiple identical *tasks* that can be executed concurrently on different servers in a cluster. For instance, MapReduce [2] is a typical two-stage process. Nowadays, new platforms such as Spark support quite complex data flows represented by DAGs (directed acyclic graph) with multiple processing stages. These types of non-sequential data flows are commonly seen in practical applications and services. For example, a database 'join' operation could merge the results from two stages into one dataset that serves another stage.

This work was partially supported by National Science Foundation grant CNS-1552525, National Science Foundation Career Award CNS-1452751, and AFOSR grant FA9550-14-1-0160.

While DAG-based data flows provide users flexibilities and enriched functions to develop their application jobs, the resource allocation and task scheduling become challenging. Given the limited resources in the cluster and a complex DAG data flow, it is difficult to decide how many resources should be allocated to each stage. We have found that the default simple schedulers in Spark, i.e., FIFO and FAIR [3], do not perform well in terms of resource utilization and job execution time. One key issue that is not considered in the built-in schedulers is the dependency between consecutive stages, especially if a stage depends on the output from multiple other stages. Besides, there are other factors that may affect the system performance such as the resource demand of the tasks in each stage, and the run-time execution time of the tasks and stages.

In this paper, we present a new scheduler, named AutoPath, that targets on complex multi-stage data flows. Our main idea is to identify *parallel paths* in the DAG that can be concurrently executed, and try to align their finish times by allocating them resources proportional to their workloads. The main contributions of this paper are as follows: (1) we thoroughly investigate and study the performance issues of the existing schedulers and the causes of these issues via real experiments; (2) we introduce the concept of parallel execution paths in a DAG and present an algorithm that identifies the parallel paths; (3) our solution also includes a workload estimation module that derives the execution time of tasks, stages, and paths; and (4) finally, we present an adaptive task scheduler that allocates resources based on the workload of parallel paths with the objective of aligning their finish times. We implement *AutoPath* on Spark version 1.5.0 and evaluate its performance with a set of representative applications. Our experiments show that the new scheduler significantly improves the performance of jobs with complex data flows by reducing the execution time and increasing the resource utilization.

The rest of this paper is organized as follows. Sec. II presents the background of Spark and the motivation of our new scheduling algorithm. Sec. III formulates the multi-stage-parallel-path scheduling problem, and presents the details of our design. Experimental evaluation results and analysis are presented in Sec. IV. We present the related work and conclusions in Sec. V and Sec. VI.

II. Background and Motivation

In this paper, our design and evaluation are based on Spark that is a representative multi-stage platform. Therefore, in this section, we first present the background of Spark system, and define the relevant terms that will be used later. Then, we motivate our problem by showing and analyzing the experimental results of the current Spark default schedulers.

A. Background of Spark

To better understand how Spark works, we first introduce the following concepts in Spark framework:

- **Application and Job:** Spark applications are user developed codes that can be written in various programming languages. One application can contain multiple jobs, which are submitted to Spark by performing actions.
- **RDD:** As the core data structure in Spark, an RDD (Resilient Distributed Dataset) is a distributed immutable collection of objects which can be divided into multiple partitions residing in the memory or the disk of different machines. RDDs can be transformed into other RDDs if changes are needed. In Fig. 1, small solid boxes are RDDs.
- **DAG:** After analyzing the application, Spark's DAG scheduler will generate a direct acyclic graph (*DAG*) to represent a chain of RDD dependencies, which is also called "RDD lineage".
- **Stage and Task:** A *stage* in Spark is comprised of tasks based on partitions of the input data. Stages can be further pipelined and one partition of a stage running on each machine is called a *"task"*. Tasks can have different input sizes and workload heavinesses even in the same stage.
- **Two Types of Operations:** Operations in Spark are categorized into *transformation* and *action*. Transformation creates a new RDD from existing one or multiple RDDs (e.g., *flatMap* and *map* in Fig. 1). Transformations are executed on demand and computed lazily. Action triggers execution (e.g., *saveAsTextFile* in Fig. 1) by using the DAG to load the data, compute all intermediate transformations, and return final results.
- **Dependency:** *Narrow dependency* does not require the data to be shuffled across the partitions (e.g., *flatMap* and *map* in Fig. 1), while *wide dependency* requires the data to be shuffled across the cluster (e.g., *reduceByKey* in Fig. 1), so it basically determines the boundaries that divide stages.

B. Spark Default Schedulers

In a Spark job, multiple independent stages (e.g., stage 0 and stage 1 to 14 in App 2 as shown in Fig. 1(b)) can be executed simultaneously. Spark *standalone mode* provides the following two schedulers to manage the execution of multiple stages.

- **FIFO:** (*first-in-first-out*) assigns the resources based on the stage submission order. When all the tasks in a stage are served, FIFO starts to consider the resource assignment for the next stage.
- **FAIR:** allows the system to assign different (fixed) weights to jobs and group them into pools based on their weights [3]. In a job, the stages that can be concurrently executed equally share the resources assigned to this job.

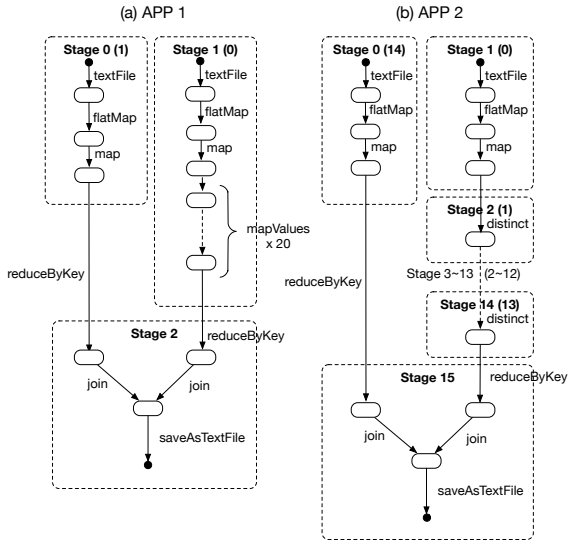

Fig. 1: Examples of DAGs: Pipeline of tasks, transformations and actions of two applications. FIFO-S and FIFO-L's stage IDs are shown in the outside- and inside-parenthesis, respectively.

However, we find that neither of them can work well with complex data flows. First, the performance of FIFO heavily depends on the order of stage submission which is often decided by user's application codes. A slight difference in the user codes with the same function can yield a significant difference in the performance[1]. In addition, while serializing the execution of all the stages, FIFO does not use the knowledge of the DAG structure to determine the execution order of the stages. This can certainly cause delays at the stages that need the input from other stages. Second, while FAIR allows parallel execution of multiple stages, it does not carefully consider the DAG structure, either. Moreover, the resources are equally distributed among the stages without considering their individual workload. As a result, FAIR may mitigate the performance issue in FIFO under some cases, but under other cases, as we show in Section II-D, FAIR could perform even worse than FIFO.

C. Impacts of Stage Orders on FIFO and FAIR

In this subsection, we execute two simple applications derived from the original *WordCount* benchmark, and analyze the performance of FIFO and FAIR schedulers. The DAGs of these two applications (i.e., App1 and App2) are shown in Fig. 1. They both have two stage paths in parallel before submitting to the final stage, i.e., stage 2 and stage 15, respectively. We consider a path in the DAG longer if it involves more stages or heavier workload (e.g., 20 *mapValue* in App 1 and 13 *distinct* transformations in App 2). In our examples, for both applications, the right path is longer than the left one.

[1]For example, the user can write *rdd1.join(rdd2)* or *rdd2.join(rdd1)*, both of which conduct the same data processing but will result in different orders of the stages.

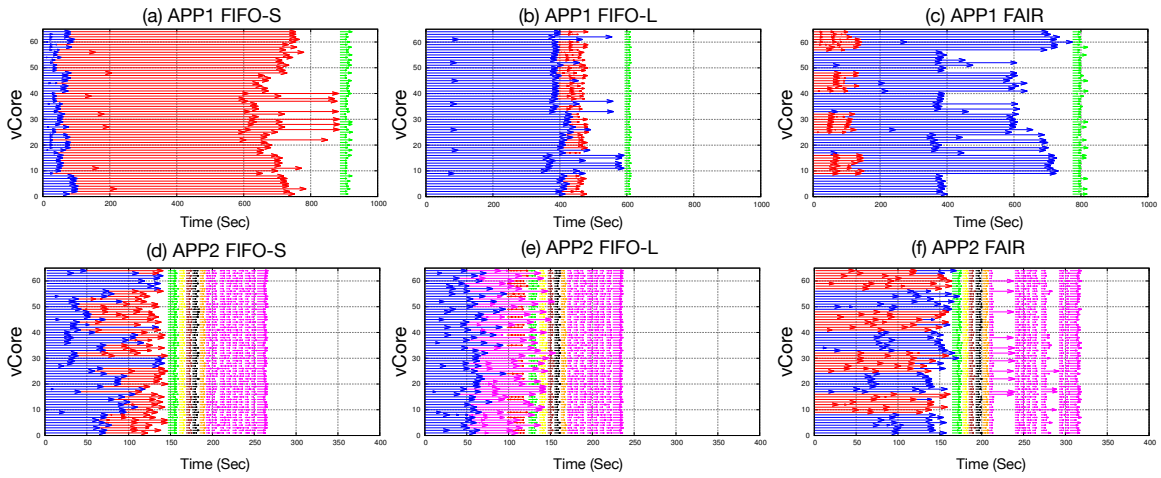

Fig. 2: Task execution flows of App 1 and 2 under different scheduling solutions.

Since the execution order of FIFO scheduler is uncertain, we consider the following two strategies for FIFO based on the path length:

- **FIFO-S:** execute the stages along the shortest path first.
- **FIFO-L:** execute the stages along the longest path first.

Fig. 2 depicts the execution flows of tasks running on 64 vCores (virtual CPU cores) under FIFO-S, FIFO-L and FAIR of two applications. In our experiments, we use 9 nodes in the Spark cluster (1 for master node and 8 for worker nodes), and each worker node has 8 vCores and 8GB memory. In Fig. 2, different colors represent different stages, and each task requests one vCore for execution.

We compare the performance of these three scheduling schemes using the makespan, which is defined as the execution time of the entire job. Apparently, FIFO-L in Fig. 2 (b) and (e) is superior to the other two in both applications. From the figure, we can find that the main reason is that FIFO-L yields a good (efficient) overlapping phase between consecutive stages in both jobs. It leads to a better alignment of the two parallel paths than the other two schemes. For both jobs, the finish times of the two parallel paths are the closest in Fig. 2 (b) and (e). It motivates our design intuition of aligning the execution of parallel paths and making them reach their common child node in the DAG as close as possible.

D. Another Synthetic Case Study

In fact, neither FIFO nor FAIR can achieve the optimal resource allocation because they assign resources with a fixed ratio, i.e., FAIR shares resources equally among stages and FIFO dedicates all resources to one stage at a moment. To better understand how to achieve an optimal schedule, we further illustrate the performance comparison in several synthetic cases in Fig. 3.

All jobs consist of two parallel paths, and each task in all stages demands 1 vCore resource for execution. Consider a small cluster with 4 vCores, and assume that with prior knowledge, task execution times can be normalized into numbers of time units (T). Besides FAIR and FIFO, we also show the optimal scheduler where we assign resource proportional to each path with the ratio of their workload (3:1) in this case[2]. For example, in case 1, we find that when FAIR equally shares 4 vCores between two paths, path 2 finishes earlier at $3T$, however, the second stage of path 1 cannot occupy the two idle vCores at $3T$ because its precedent stage is still running. As a result, the total execution length is $6T$ and we have two idle vCores at both the time of $3T$ and $6T$. The same execution length is obtained when we schedule path 1 first, i.e., FIFO-1. The optimal scheduler assigns the number of vCores to two paths approximately proportional to the number of tasks in each path, and the execution length of this job is successfully reduced to $5T$ and 4 vCores are fully utilized across the duration of job execution. This implicates that the stage or path with more tasks should acquire a larger portion of resources to avoid or reduce the lag between the finish times of two paths.

Similarly, cases 2 and 3 indicate that the stage with large tasks (i.e., long execution time) and the path with more stages should get more resources. Case 4 illustrates a more realistic case where the tasks in the same stage yield different execution times. This variance causes idle periods under FAIR and FIFO. In fact, this is common because a stage often includes different transformations. In Fig. 3, the optimal solution proportionally assigns resources to two paths to align the execution of these paths and avoid the occurrence of idle resources.

Above all, the design intuition in this paper is to align the execution of parallel paths, i.e., being finished at the same time. To achieve this goal, our solution needs to adaptively allocate resources to each path based on the estimation of its execution length.

III. Algorithm Design

In this section, we first formulate the task scheduling problem in a multistage framework, and then present our

[2]Considering discrete ratio setting, there are only 4 choices in our example. FAIR represents 2:2.

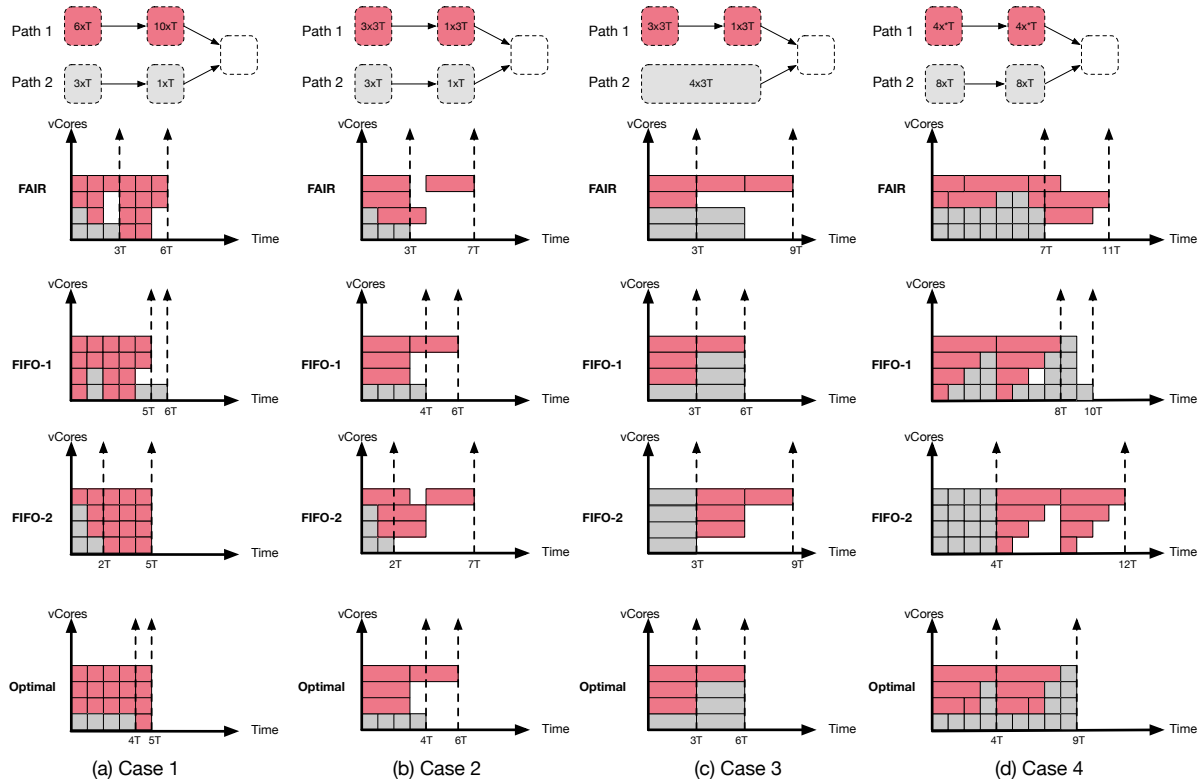

Fig. 3: DAG diagrams and execution flows for different cases under FAIR (1:1), FIFO-1 (path 1 first), FIFO-2 (path 2 first) and the optimal (3:1) scheme. Tasks in path 1 and path 2 are marked with red and gray color, respectively. The dash arrows in execution flows indicate the ending time of each path.

solution *AutoPath* that appropriately manages the execution of parallel paths.

A. Problem Formulation

Given a DAG of a job $G(V, E)$ (where V and E denote sets of nodes and edges, respectively) such as Fig. 4 , we call the final result stage the *root* (stage 14). For any node in the DAG, its parents are the nodes that feed its input data, and its output data becomes its children nodes' input. For example, stage 11's *ParentsSet* is $\{3, 6\}$, and its *ChildrenSet* is $\{12\}$. A node with more than one parents is called a *crunode*.

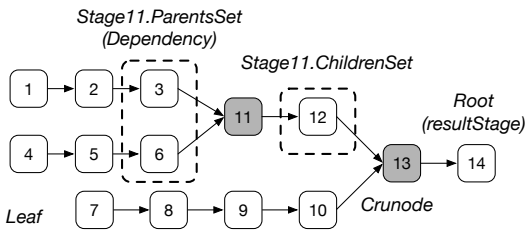

Fig. 4: DAG of an application with multiple paths.

For any node $v \in V$, we use e_v to indicate the execution time of the processing operation, and r_v to represent the average amount of resources allocated to the node. In addition, let S_v and F_v represent the start time and finish time of the processing operation associated with node v, $F_v = S_v + e_v$. The best scheduling algorithm can be derived by solving the following optimization problem:

$$\begin{aligned} \text{Minimize:} \quad & F_{Root} \\ \text{Subject to:} \quad & \forall u, v, \text{ where } u \text{ is } v\text{'s parent}, S_v > F_u; \\ & \forall t > 0, \sum_{S_v < t < F_v} r_v \leq RC, \end{aligned}$$

where RC is the resource capacity in the processing cluster.

There are some critical challenges in developing such scheduling algorithms:

Challenge 1: NP-hard Problem. By using Graham's notation [4], the task scheduling problem in Spark with multiple stages can be formated as a *generally parallel machine problem* (e.g. $Q|intree|C_{max}$ [5] or more specifically $P2|intree|C_{max}$ if given two equal capacity machines [6]). However, many of these scheduling problems are proven as NP-hard and existing solutions [4], [7] can not meet the decision making temporal requirement for the Spark framework.

Challenge 2: Complex Dependencies in Spark. DAGs of multi-stage jobs often involve different types of nodes (such as root and leaf nodes, and crunodes) and complex dependencies among these nodes. Scheduling in such DAGs is not as straightforward as MapReduce jobs that only contain one map phase and one reduce phase. Given complex dependencies and data flows, it becomes challenging to identify all independent paths in the DAG that are disjoint and can be executed in

parallel and determine active independent paths that should be synchronized at run time.

Challenge 3: No Prior Knowledge. Many related factors, such as number of operations in the path, number of tasks (i.e., parallelism) of its operations, and resource demands, can affect the execution of an independent path. In addition, operations in multi-stage jobs often have widely varying demands for different resources.

In this paper, we present a heuristic and efficient solution that focuses on the resource allocation ratio among paths in the DAG. The notations that will be used in the discussion are listed in Table I.

TABLE I: Summary of Notations.

Notation	Description
e_v	Execution time of node v in the DAG.
r_v	Average amount of resources allocated for node v.
S_v/F_v	Start/end time of node v.
$p_i/s_j/t_k$	Path i/stage j/task k.
P	Set of path p_i.
S_{p_i}	Set of stages belonging to p_i.
T_{s_j}	Set of tasks belonging to of stage s_j.
T_{t_k}	Execution time of task t_k.
n_{t_k}	Number of RDDs in task t_k.
pw_i	Evaluate workload of p_i.
pc_i	Capacity of resources (i.e. vCores) assigned to p_i.
rn_i	Number of running tasks for path p_i.
$\overline{t_{rdd_i}}$	Average transformation time between RDDs in path p_i.
SL	Set of *ready-to-execute* (i.e., current "leaf" nodes).
T_M	Entire makespan of an application.
T_{P_i}	Time of parallel paths at phase i to be finished. which is bottlenecked by the slowest path.
T_R	Time of the root node to be finished.
$\mathbb{P}_i$	path set for phase i.
c/c_i	Total available resource (vCores)/Resource assigned to path i.

B. Our Approach

Our major design intuition is to align the execution of parallel paths by allocating appropriate amounts of resources to the stages on each path. Because of the challenges mentioned earlier, we simplify the problem by the following two changes on the execution path:

Relaxed definition of parallel paths. Instead of considering independent parallel paths that merge to crunodes, we consider every path from a leaf node to the root, i.e., the number of initial parallel paths is the number of leaf nodes.

Temporal paths. We divide the execution time to multiple epochs/phases, and the definition of parallel paths is specific to each phase, i.e., along the execution, the set of parallel paths may change.

For example, in Fig. 5, there are n leaf nodes in the DAG, thus n parallel paths before the execution. We divide the execution into m phases, and after phase 6, the first four paths merge into two paths. Therefore, in phase 7, the set of parallel paths becomes smaller.

In our problem setting, the scheduler can adjust the resource allocation at the beginning of each phase. Now, the problem becomes how to allocate resources for each phase to minimize execution time of each phase, so that the overall makespan is minimized:

$$\text{Minimize: } T_M = \sum_{i=1}^{m} T_{P_i} \tag{1}$$

$$\text{Subject to: } \forall i \in [1, m], \quad n_i = |\mathbb{P}_i| \tag{2}$$

$$T_{P_i} = max(\frac{pw_1}{c_1}, \frac{pw_2}{c_2}, ... \frac{pw_{n_i}}{c_{n_i}}) \tag{3}$$

$$\sum_{i=1}^{n_i} c_i = c, \quad c_i \geq 0, \quad pw_i \geq 0 \tag{4}$$

The objective function (Eq. 1) aims to minimize the entire makespan T_M, which consists of makespan of the parallel paths at all phases $\sum_{i=1}^{m} T_{P_i}$. For each phase, its makespan is determined by the bottleneck of the slowest path (Eq. 3). Furthermore, if we ensure that during each phase if each path can finish at the same time, then the overall $\sum_{i=1}^{m} T_{P_i}$ will be the minimal. Specifically, Eq. 2 and Eq. 3 calculate the makespan of each phase, where $n_i = |\mathbb{P}_i|$ is the total number of paths during phase i, and pw and c are the workload and allocated resource capacity of each path during each phase, respectively. It can be easily proven that when $\frac{pw_1}{c_1} = \frac{pw_2}{c_2} = ... = \frac{pw_n}{c_n}$, the optimal solution is reached. This indicates that at the beginning of each phase, we should adjust the resource capacity ratio according to the proportion of their demand *inside* each phase (i.e., $c_1 : c_2 : ... : c_n = pw_1 : pw_2 : ... : pw_n$). However, in real implementation, we need to answer the following two questions:

Question 1: How to estimate path workload? In order to calculate the minimal makespan, we need to analyze the workload of each path *within* each phase (i.e., T_{P_i} in Eq. 3). For example, as shown in Fig. 5, during phase 1, for path 1, we need to consider both the workload demand and the resource to assign to only the left half part of stage 1_1 which appears in that phase. Moreover, we also need to monitor the "path set" ($\mathbb{P}_i$) during each phase, since the number of paths (n_i in Eq. 2) may vary during runtime.

Question 2: How to select phase size? Obviously, the more fine-grained the phase size is, the closer to the optimal solution, i.e., when $m \to 0$, we have $T_M \to T_{M_{min}}$. However, considering the trade-off between performance and overhead, we cannot set the phase size to infinity small. As a result, in our implementation, *AutoPath* triggers a new phase when a new stage is launched, and a path set is also updated for each phase to reflect any path changes (e.g., path merge).

Notice that under this setting, "phase sizes" will vary during runtime, which makes it hard to know the workloads *within* each phase. Thus, we further use the sum of the remaining workloads of each path to estimate the workload of the current phase.

C. Online Parallel Path Detection

In order to meet the goal of optimal scheduling (i.e., align the completion of parallel paths and reduce the overall makespan), one of the main tasks in *AutoPath* is to identify the entire information of all existing parallel paths. *AutoPath*

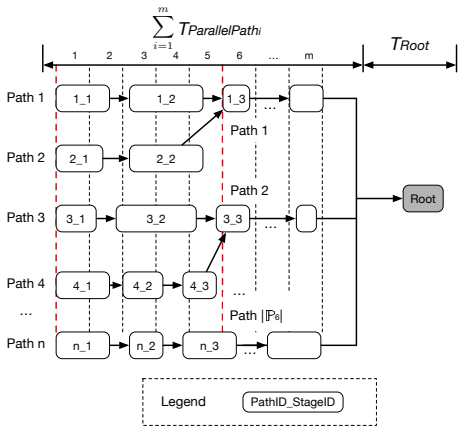

Fig. 5: Reasoning for the intuition of our proposed approach.

will *periodically* traverse the DAG of application to generate the path table, an example is shown in Fig. 4 and Table II. This path detection is a dynamic process and will be triggered to update once a stage starts to execute during runtime. For instance, at the beginning (T_0), there are three paths 1→14, 4→14, and 7→14, but after the task in stage 11 begins to execute at time T_x, the path set reduce to 2 paths since 1→14 and 4→14 are merged into one path 11→14 as shown in fourth column. Based on this path set, *AutoPath* further determines the priorities for each path in the set according to their estimated workload (e.g., paths with higher weights) and assigns resources to tasks of each path proportionally to their priorities. Our implementation of this path detection procedure is described in Alg. 1, where the recursive function $TraversalStageTree$ is called to search all parallel paths from the root to leaves.

Algorithm 1: Online Parallel Stage Path Detection

Input: DAG, t
Output: pathSet T_t at time t
$T_t \leftarrow \emptyset$, $path \leftarrow \emptyset$;
traversalStageTree(root, path);
return T_t
Procedure `TraversalStageTree`$(S, path)$
  if *S has finished or $S == null$* **then**
    $T_t \leftarrow path$;
    return;
  $path \leftarrow S$;
  foreach $parentStage \in S.parentSet$ **do**
    $TraversalStageTree(parentStage, path)$;

TABLE II: Online detected parallel paths.

PathID	Path at T_0	...	Path at T_x	...
1	1→2→3→11→12→13 →14	...	11→12→13→14	...
2	4→5→6→11→12→13 →14	...	9→10→13→14	...
3	7→8→9→10→13→14	...		...

D. Path Workloads Estimation

To construct the adaptive scheduling algorithm, *AutoPath* needs to capture the workload of each path during runtime. In order to comprehensively reflect each path's characteristics (such as numbers of stages/tasks and total estimated times of operations between RDDs inside each path, etc.), *AutoPath* defines the workload (i.e., pw) of a path by estimating the *remaining total execution time of each path*, which is based on the runtime statistics of up-to-now average transformation time between RDDs in each path (i.e., $\overline{t_{rdd}}$). Specifically, for the ith path, we have:

$$pw_i(t) = \sum_{s_j \in S_{p_i}} \sum_{t_k \in T_{s_j}} (n_{t_k} - 1) \cdot \overline{t_{rdd_i}(t)}, \tag{5}$$

where S_{p_i} is the set of remaining stages in the path, and T_{s_j} is the set of tasks of stage j in S_{p_i} (i.e., s_j). In the dynamic process of profiling and estimation, the workload of each path will be updated once each task finishes. Such task completion information on each worker node is sent to the master node through system heartbeat messages. During the application execution, fewer and fewer stages remain in the path set and eventually the set becomes empty, and pw_i will also become 0. Moreover, as mentioned in Sec. II-D, different operations can have different execution times based on their types and input data sizes, therefore in real implementation the estimation granularity of *AutoPath* can be tuned from the coarse-grained path level (i.e., $t_{rdd_i}(t)$ is calculated based on each path) to the fined-grained per-stage level (i.e., further calibrate $t_{rdd_i}(t)$ based on each stage).

E. Adaptive Resource Allocation Scheme

Finally, we present an online adaptive algorithm that dynamically allocates resources to each stage and path to improve the overall performance (i.e., the makespan of jobs). The key idea of our algorithm is to use the periodically updated estimation of path workloads as weights to make resource assignment decisions for each path. In particular, *AutoPath* has two sessions, i.e., *training allocation* and *online allocation*. The first session is executed for a preset window of time (i.e., training length T_w). Without any prior knowledge of path workloads, all vCores in the cluster are available to the application, and *AutoPath* adopts the proportional share of the capacity to assign vCores to the first stage of each path based on the ascending order of their path ID. *AutoPath* also considers a boundary scenario where the first stage of each path may not fully utilize the recourse quota based on its workload estimation proportion (i.e., this stage has fewer parallel tasks to run than the quota). In this case, *AutoPath* collects those extra vCores and further allocates them to other *ready-to-execute* stages (e.g., the unfulfilled stages in "leaf nodes" with current quota).

In the second session, *AutoPath* re-calibrates the path workload estimation based on the remaining stages of each path with the "per-task" frequency (i.e., re-calibration is triggered upon the completion of a task). For assigning tasks, *AutoPath* adopts *two-steps* assign order (i.e. select stage and task in circulation style) within the pending task pool. Specifically, when the built-in "resourceOffer" function is periodically called by Spark system to inquire for available resources, *AutoPath*

TABLE III: Cluster Hardware Configuration.

Machine		Server 1	Server 2	Server 3
Host	Memory(GB)	48	48	48
	CPU(Core)	24	24	24
VM	Role	Master	Worker	Worker
	Number of VMs	1	4	4
	Memory Per VM(GB)	4	8	8
	CPU Per VM(Core)	4	8	8

TABLE IV: Application Configuration.

App Name	Path Number	Stage Per Path	Represented User Cases
App 1	Mulitple	Single	PrimeNumCal, PathSim [8], MTPS [9]
App 2	Multiple	Multiple	Apriori [10], Parallel Matrix Factorization [11]

checks the available vCores on each worker node and strives to allocate these vCores to *ready-to-execute* stages in the stage list $SL(t)$ without violating each path capacity. Unlike the first session, the online allocation session is focused on assigning the reclaimed vCores released by previous finished tasks in each stage in a fine-grained task granularity. Specifically, *AutoPath* allocates vCores to those *ready-to-execute* stages whose current path quota increases or capacity is unfulfilled. At the end, *AutoPath* updates the status of resource and running tasks in the path.

IV. Performance Evaluation

We implemented *AutoPath* as a new scheduler in Spark v1.5.0. Table III gives the cluster configuration and hardware information for experiments. We also use the applications listed in Table IV as workloads to validate the effectiveness of *AutoPath*. We compare *AutoPath* with two default schedulers, i.e., FIFO and FAIR.

A. Makespan Speedup

In the first set of experiments, we tune the input data size (e.g., 1GB, 10GB, and 20GB) for two applications. The main metric we use for evaluating performance improvement is the *makespan speedup* with respect to FIFO-S. The results of the makespan speedup are shown in Fig. 6, where we can see that *AutoPath* outperforms with the relative improvement of 27.7%, 14.2% and 21.5% for App 1 and 10.1%, 18.0% and 24.5% for App 2 under three different input data sizes. This is because *AutoPath* proportionally shares resources among parallel paths during runtime and thus prevents the long path being dis-aligned by short tasks, as shown in the execution flow results of Fig. 7(d) and (h). On the other hand, as expected, FIFO-S and FAIR achieve the worst performance. We observe that tasks in the short path under FIFO-S cannot finish at the same time, so they break the alignments of tasks in the long path and cause more idle intervals in the long path, as shown in Fig. 7(a) and (e). Similarly, FAIR ignores the difference in the path workloads and equally allocates resources to the short and long paths. As a result, the alignments among tasks of both short and long paths are broken, as shown in Fig. 7(c) and (g).

B. Evaluation on Resource Allocation

To further investigate how *AutoPath* allocates computing resource among paths, we also plot the total number of running

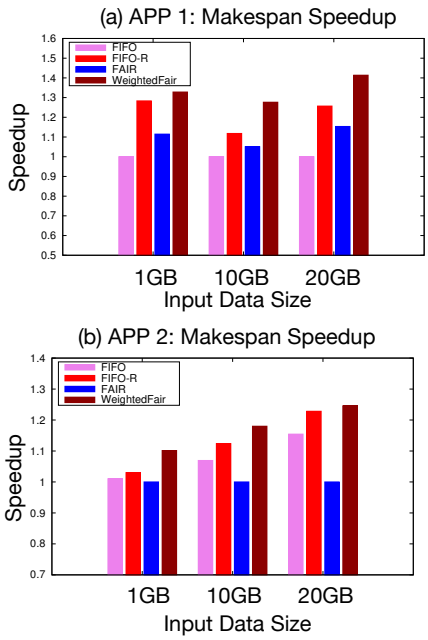

Fig. 6: Makespan speedup for different scheduling algorithms with various input data sizes.

tasks (i.e. rn), the estimated residue workload (i.e. pw) and the allocated resource ratio under *AutoPath* during runtime in Fig. 8. Firstly, from Fig. 8(a), we can observe that the number of running tasks (rc) drops from 64 tasks (i.e., fully using vCore) to 0 only within about 80 sec (i.e., from about 220 sec. to 300 sec) in *AutoPath*, which is much faster than FIFO-S and FAIR. This demonstrates that *AutoPath* helps to align long path and short one very well which further reduces the idle period in the cluster. However, it is hard for App 2 to obtain similar benefit under *AutoPath* since its multiple-stage short path still suffers from the cross-stage asynchronous issue, as shown in Fig. 8(d). Secondly, Fig. 8(b) and (e) show that although there is a delay for path workload estimation at beginning (e.g., path 1 in App 2 is underestimated before 100 sec, even though the stage in path 1 has more transformations), as time pass by, *AutoPath* can self-correct and finally accurately estimate the actual path workload. Thirdly, Fig. 8(c) and (f) further illustrate that *AutoPath* can dynamically assign an appropriate amount of CPU resource to each path; which is proportional to that path's online estimated residue workload.

C. Sensitivity Analysis on Parallel Degree

An important question which is always raised by developers and datacenter managers is whether setting more partitions in Spark can help improve performance or not. To answer it, we conduct the sensitivity analysis on parallel degree (i.e., the number of partitions or tasks in a stage) under both the default schedulers and our new scheduler and present the makespan speedup of these schedulers in Fig. 9. Note that we did not present the case of the partition number less than 64 in Fig. 9 because in that case, the parallel computing resource is not fully utilized and Spark is expected to get worse performance.

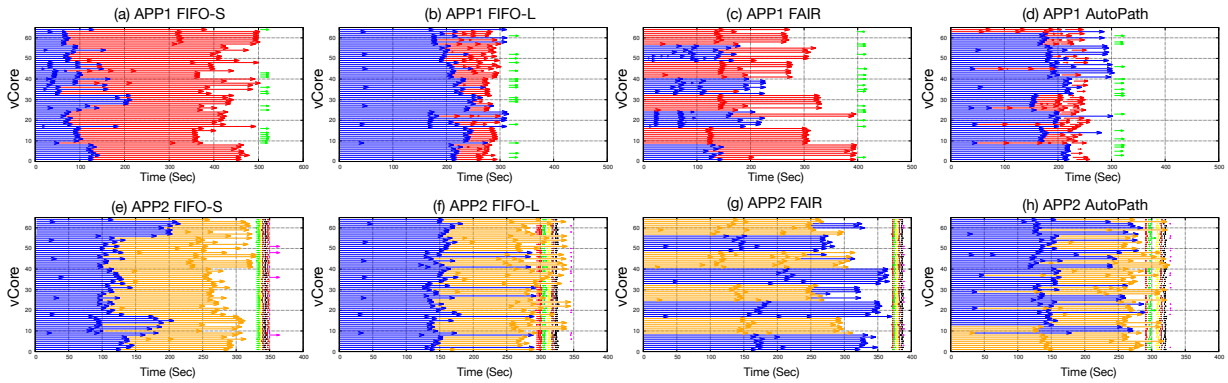

Fig. 7: Task execution flows under different schedulers with 10GB dataset and the parallel degree (i.e., number of partitions) is 81.

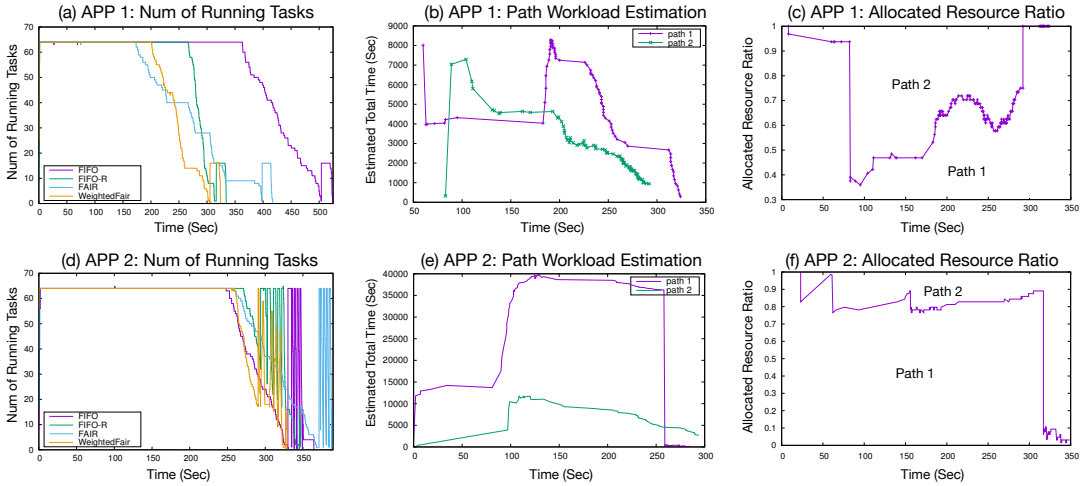

Fig. 8: Runtime measurements in the experiment with 10GB dataset and the parallel degree (i.e., number of partitions) is 81.

We first observe that when the partition number is slightly larger than the total vCore capacity (e.g., $spark.core.max = 64$ in our case), *AutoPath* achieves the best improvement. When the number of partitions increases (e.g., 162 and 234), the relative improvement of *AutoPath* decreases. But, we can still see that *AutoPath* performs the best among these schedulers. We interpret these results by observing that assigning more partitions reduces the input data size of each task and thus reduces the load for each task to process. As a result, the variation of task execution times decreases, which mitigates the Spark delay scheduling issue introduced by dependency between stages and thus limits the improvement space for *AutoPath*.

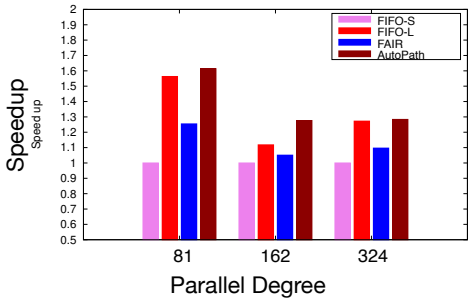

Fig. 9: Makespan speedup for different scheduling algorithms under various partition numbers.

V. Related Works

With the emerging of parallel computing frameworks (such as Hadoop [12] and Spark [13]), improving the performance of large data analysis in distributed clusters becomes the new attention in recent years. In a heterogeneous cluster with multiple users, Fair [3] scheduler not only gives each job an equal share of the cluster resources, but also considers multiple user scenarios by providing the pool for different user. Quincy [14] and Delay scheduling [3] optimize data locality in the case of Fair scheduling. The Coupling scheduler in [15] [16] [17] mitigates the starvation of reduce slots under Fair scheduling. [18] presents an efficient data redistribution to speedup big data analytics in large systems. [19] develops an interface for performance environment auto-configuration framework. ARIA [20] allocates the appropriate amounts of resources to jobs to meet the predefined deadline. [21]–[26] investigate storage related resource management problems in big data platforms. [27]–[30] further investigate "slow" node

in a cluster and start redundant tasks on other nodes to guarantee the reliability in Hadoop. [31] develops an analytical model to estimate the effect of interference among multiple Apache Spark jobs running concurrently on job execution time. Recently, [32] analyzes the DAG used by Spark to predict the stage execution times and the overall application execution time. However, it dose not consider the resource allocation optimization problem.

VI. Conclusion

We presented *AutoPath*, a novel scheduling scheme for Spark framework. The primary goal of *AutoPath* is to improve the makespan (i.e., the total execution length) for the application with multiple parallel stages. The source of *AutoPath*'s benefits lies in the constructed application DAG with multiple parallel paths and the runtime path workload estimation. *AutoPath* adaptively allocates resources to each parallel path according to their runtime workload estimation, so that all parallel paths can complete at around the same time to avoid idled resource waste. We implemented *AutoPath* in Spark v.1.5.0 as a pluggable module and evaluated this scheme by running representative Spark applications. The experimental results demonstrated that *AutoPath* delivers an effective improvement of the performance in terms of makespan speedup. In the future, we plan to improve the accuracy of workload estimation by using more comprehensive machine learning algorithms. Moreover, the frequency for path workload recalibration can also be increased to adapt to workload changes.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[2] J. Dean, S. Ghemawat, and G. Inc, "Mapreduce: simplified data processing on large clusters," in *OSDI'04*, 2004.

[3] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.

[4] P. Brucker and P. Brucker, *Scheduling algorithms*. Springer, 2007, vol. 3.

[5] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, 2002.

[6] H. Zhao and R. Sakellariou, "An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm," in *European Conference on Parallel Processing*. Springer, 2003, pp. 189–194.

[7] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of discrete mathematics*, vol. 5, pp. 287–326, 1979.

[8] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu, "Pathsim: Meta path-based top-k similarity search in heterogeneous information networks," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 992–1003, 2011.

[9] R.-Z. Liang, L. Shi, H. Wang, J. Meng, J. J.-Y. Wang, Q. Sun, and Y. Gu, "Optimizing top precision performance measure of content-based image retrieval by learning similarity function," in *Pattern Recognition (ICPR), 2016 23st International Conference on*. IEEE, 2016, pp. 2954–2958.

[10] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," *Principles of Data Mining and Knowledge Discovery*, pp. 13–23, 2000.

[11] A. Gupta, G. Karypis, and V. Kumar, "Highly scalable parallel algorithms for sparse matrix factorization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 5, pp. 502–520, 1997.

[12] Apache hadoop yarn. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[13] Apache spark. [Online]. Available: https://spark.apache.org

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 261–276.

[15] J. Tan, X. Meng, and L. Zhang, "Performance analysis of coupling scheduler for mapreduce/hadoop," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2586–2590.

[16] ——, "Delay tails in mapreduce scheduling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp. 5–16, 2012.

[17] ——, "Coupling task progress for mapreduce resource-aware scheduling," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 1618–1626.

[18] L. Cheng and T. Li, "Efficient data redistribution to speedup big data analytics in large systems," in *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*. IEEE, 2016, pp. 91–100.

[19] L. Men, B. Hadri, and H. You, "Interface for performance environment autoconfiguration framework," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 1356–1356.

[20] A. Verma, Ludmila Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *ICAC'11*, 2011, pp. 235–244.

[21] Z. Yang, J. Tai, J. Bhimani, J. Wang, B. Sheng, and N. Mi, "GREM: Dynamic SSD Resource Allocation In Virtualized Storage Systems With Heterogeneous VMs," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[22] J. Tai, D. Liu, Z. Yang, X. Zhu, J. Lo, and N. Mi, "Improving Flash Resource Utilization at Minimal Management Cost in Virtualized Flash-based Storage Systems," *Cloud Computing, IEEE Transactions on*, no. 99, p. 1, 2015.

[23] Z. Yang, M. Awasthi, M. Ghosh, and N. Mi, "A Fresh Perspective on Total Cost of Ownership Models for Flash Storage," in *2016 IEEE 8th International Conference on Cloud Computing Technology and Science*. IEEE, 2016.

[24] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[25] Z. Fan, D. H. Du, and D. Voigt, "H-arc: A non-volatile memory based cache policy for solid state drives," in *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*. IEEE, 2014, pp. 1–11.

[26] Z. Yang, J. Wang, D. Evans, and N. Mi, "AutoReplica: Automatic Data Replica Manager in Distributed Caching and Data Processing Systems," in *1st IEEE International workshop on Communication, Computing, and Networking in Cyber Physical Systems (CCNCPS)*. IEEE, 2016.

[27] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "Omo: Optimize mapreduce overlap with a good start (reduce) and a good finish (map)," in *Computing and Communications Conference (IPCCC), 2015 IEEE 34th International Performance*. IEEE, 2015, pp. 1–8.

[28] ——, "Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 761–768.

[29] J. Wang, T. Wang, Z. Yang, N. Mi, and S. Bo, "eSplash: Efficient Speculation in Large Scale Heterogeneous Computing Systems," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[30] J. Wang, T. Wang, Z. Yang, Y. Mao, N. Mi, and B. Sheng, "SEINA: A Stealthy and Effective Internal Attack in Hadoop Systems," in *International Conference on Computing, Networking and Communications (ICNC 2017)*. IEEE, 2017.

[31] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale, "Modeling interference for apache spark jobs," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 423–431.

[32] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna, "Stage aware performance modeling of dag based in memory analytic platforms," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 188–195.