electronics

MDPI

Article

Performance Improvement of DAG-Aware Task Scheduling Algorithms with Efficient Cache Management in Spark

Yao Zhao [1], Jian Dong [1,*], Hongwei Liu [1], Jin Wu [2] and Yanxin Liu [1]

[1] School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China; zhaoyao@ftcl.hit.edu.cn (Y.Z.); liuhw@hit.edu.cn (H.L.); liuyanxin@ftcl.hit.edu.cn (Y.L.)
[2] School of Engineering, University of Georgia, Athens, GA 30602, USA; jinwu@uga.edu
* Correspondence: dan@hit.edu.cn

Abstract: Directed acyclic graph (DAG)-aware task scheduling algorithms have been studied extensively in recent years, and these algorithms have achieved significant performance improvements in data-parallel analytic platforms. However, current DAG-aware task scheduling algorithms, among which HEFT and GRAPHENE are notable, pay little attention to the cache management policy, which plays a vital role in in-memory data-parallel systems such as Spark. Cache management policies that are designed for Spark exhibit poor performance in DAG-aware task-scheduling algorithms, which leads to cache misses and performance degradation. In this study, we propose a new cache management policy known as Long-Running Stage Set First (LSF), which makes full use of the task dependencies to optimize the cache management performance in DAG-aware scheduling algorithms. LSF calculates the caching and prefetching priorities of resilient distributed datasets according to their unprocessed workloads and significance in parallel scheduling, which are key factors in DAG-aware scheduling algorithms. Moreover, we present a cache-aware task scheduling algorithm based on LSF to reduce the resource fragmentation in computing. Experiments demonstrate that, compared to DAG-aware scheduling algorithms with LRU and MRD, the same algorithms with LSF improve the JCT by up to 42% and 30%, respectively. The proposed cache-aware scheduling algorithm also exhibits about 12% reduction in the average job completion time compared to GRAPHENE with LSF.

Keywords: DAG aware; cache management; cache aware; Spark

check for updates

Citation: Zhao, Y.; Dong, J.; Liu, H.; Wu, J.; Liu, Y. Performance Improvement of DAG-Aware Task Scheduling Algorithms with Efficient Cache Management in Spark. *Electronics* **2021**, *10*, 1874. https://doi.org/10.3390/electronics10161874

Academic Editor: David Defour

Received: 15 July 2021
Accepted: 2 August 2021
Published: 4 August 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Spark is an in-memory data analytics framework that is used extensively in iterative data processing with low latency [1–4]. It uses resilient distributed datasets (RDDs) to cache and compute parallel data, which results in significant performance improvements compared to traditional disk-based frameworks [5]. The workflows in Spark can be presented as a directed acyclic graph (DAG) and the edges in the DAG represent the data dependencies in the workloads. The default FIFO task scheduler and LRU cache policy are oblivious to data dependencies and exhibit poor performance in Spark.

In recent years, DAG-aware scheduling for iterative data processing has attracted increasing interest [6,7]. Numerous efficient algorithms have been proposed to solve various scheduling tasks in specific systems [8–11]. HEFT [12] calculates the schedule priority based on the resource demands of the nodes and estimates the job completion time. GRAPHENE [13] considers both the complexity and heterogeneity of the DAGs and exhibits improved performance in multi-resource parallel computing. Branch scheduling [14] focuses on the influence of the data locality and provides superior adaptability with the cache policy compared to GRAPHENE. However, in an in-memory parallel computing system such as Spark, the cache policy also has a significant impact on the task scheduling. Current DAG-aware scheduling algorithms neglect the impact of the cache policy and fail to use resources efficiently in multi-resource computing.

The cache policy in Spark has also been studied in depth [15]. Several dependency-aware cache policies have been proposed for Spark, such as LRC [16], LCRC [17], and MRD [18]. All of these policies have led to progress in improving the cache hit ratio compared to the default LRU cache policy. LRC traverses the DAG and assigns each RDD a different caching priority according to its reference count, and RDDs with low reference counts tend to be evicted when the memory is full. LCRC and MRD further exploit the DAGs of jobs and consider the reference gap, which results in the cached RDD being more time sensitive and achieves a better hit ratio than LRC. However, MRD is designed for the default FIFO scheduler in Spark and cannot offer its advantage in the DAG-aware scheduling method.

In this paper, we present an efficient cache policy for DAG-aware scheduling and discuss the impact of the cache policy on task scheduling in Spark, with the aim of combining DAG-aware scheduling with the cache policy. The DAG-aware scheduling should be optimized according to the cache policy to improve resource utilization and to obtain a better job completion time (JCT) in the workflow.

Thus, we propose a novel cache policy and an efficient cache-aware scheduling algorithm based on our cache policy in Spark. In cache-aware scheduling, the cache results dynamically affect the scheduling policy, thereby resulting in an improved JCT. Our contributions are as follows.

Considering the limitations of current cache policies, we designed a priority-based cache policy for DAG-aware scheduling to improve the RDD hit ratio. Our cache policy is deeply integrated with the scheduling algorithm and is proven to be efficient in iterative data computing.

We investigated the impact of the cache policy on scheduling and proposed a cache-aware scheduling method in Spark, which increases the parallelism of tasks and exhibits better utilization of cluster resources. The schedule is based on the DAGs of the workloads and the dependencies between stages are guaranteed. Our solution further improves the resource utilization and reduces the JCT of the workloads.

We implemented our solution as a pluggable scheduler in Spark 2.4. We conducted extensive evaluations on a six-node cluster with five different data analysis workloads in SparkBench [19] to verify the efficiency of our method. The LSF cache management and cache-aware scheduling algorithm achieved high performance and exhibit significant advantages in memory-constrained situations on all benchmarks. According to our experimental results, compared to DAG-aware scheduling algorithms with LRU and MRD, the same algorithms with LSF improve the JCT by up to 42% and 30%, respectively. Our cache-aware scheduling algorithm also exhibits an average performance advantage of 12% over GRAPHENE with LSF.

The remainder of this paper is organized as follows. In Section 2, the background of the study is presented, and the inefficiencies of existing cache management policies and DAG-aware task scheduling algorithms are described. The design of our solution and its implementation details are presented in Section 3. The evaluation results are discussed in Section 4. Finally, conclusions are provided in Section 5.

2. Background and Motivation

In this section, we discuss the background of the data access in Spark jobs and present the motivation for introducing a novel cache-aware scheduling algorithm. Our discussions are limited to the context of Spark; however, the details are also applicable to other in-memory computing frameworks.

2.1. RDD and Data Dependency

Spark is a distributed, in-memory computing framework for big data, which provides an RDD as its primary abstraction in computing. RDDs are distributed datasets that are stored in memory. Spark can only transform an RDD into a new one with a transformation operation. The data workflows on parallel computing frameworks are determined by

DAGs that consist of RDDs. These DAGs contain rich information regarding data dependencies, which is crucial for data caching and has not been fully explored in default cache management policies [20].

For example, as a key abstraction in Spark, an RDD is a collection of objects that are partitioned across the nodes in a Spark cluster [5] and all of the partitions can be computed in parallel. In Spark, the operations are divided into transformation and action, all of which are based on the RDDs. As illustrated in Figure 1, the scheduler of Spark is composed of RDD objects, a DAG scheduler, task scheduling, and task operations. During construction of the RDD objects, the scheduler will analyze the RDDs of the incoming tasks and submit these to the DAG scheduler while the action operation is triggered. Subsequently, the DAG scheduler forms a DAG, thereby implying a task execution sequence that is divided into several stages. During the execution, dragging or failing tasks are recomputed. Therefore, cache replacement strategies have a significant influence on the recomputing costs.

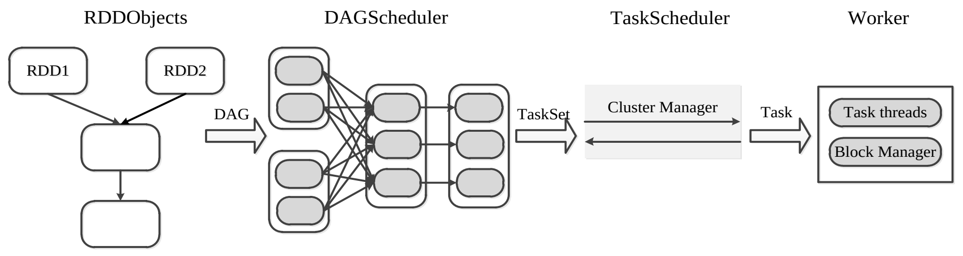


Figure 1. Scheduling in Spark.

2.2. Inefficient Cache Management in Spark

LRU, which is a classic history-based cache management method, is used as the cache replacement algorithm in Spark. LRU tracks the data in memory and evicts the blocks that have not been accessed for the longest periods. However, LRU is oblivious to the lineage of Spark jobs, which results in poor efficiency in the eviction of RDDs. Several DAG-based cache management policies have been proposed. LRC and MRD are both representative DAG-based cache policies that have been proven to perform effectively on common benchmarks. LRC traverses the lineage and tracks the dependency count of each RDD. This count is updated continuously as a priority for evicting RDD blocks from memory while the Spark jobs run. An RDD with a higher reference count is more likely to be used in future computations and should be cached in memory. To save the maximum memory space, the RDD with the lowest dependency count should be evicted from memory. Compared to the default LRU policy, LRC improves the cache hit ratio and provides an improved comprehensive application workflow. MRD is time sensitive and performs better than LRC in the FIFO scheduler. However, it does not exhibit significant advantages over other cache policies in DAG-aware scheduling.

We use a DAG to illustrate the possible problems in the current cache and scheduling policies. In Figure 2, each box represents a stage with RDD blocks. The resource demands and durations are indicated above each stage. The capacity is assumed to be 10 for the executor. The schedule order and execution time for the FIFO and DAG-aware scheduling are listed in Table 1. In a DAG-aware scheduler such as GRAPHENE, a job is scheduled in the order of stage1, stage3, (stage4, stage0), (stage0, stage2, stage5) to achieve better resource utilization and JCTs compared to FIFO. However, MRD still conducts its prefetch policy according to the FIFO schedule order, which is the ascending order of stage numbers. The RDD blocks that are prefetched in the cache are not accessed immediately, which leads to frequent cache replacement and performance degradation. Cache management, which is characteristic of DAG-aware scheduling, offers advantages in such situations.

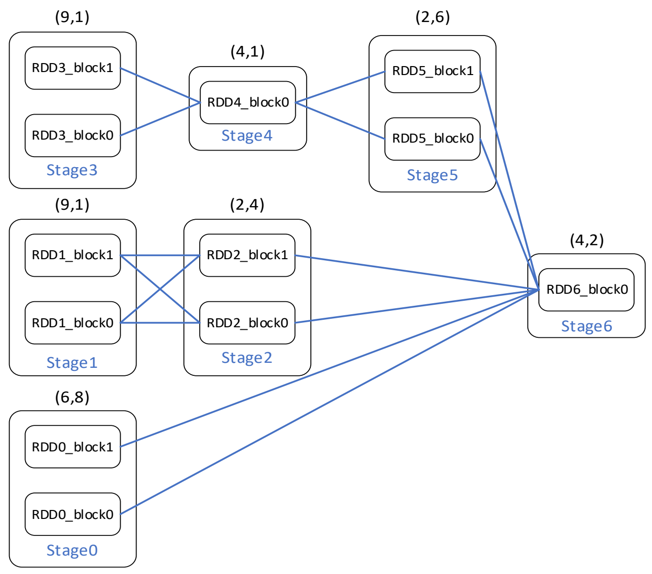


Figure 2. DAG with resource demands and durations.

Table 1. Scheduling order and execution time.

Algorithm	Scheduling Order (Represented by Stage ID)	Execution Time
FIFO	0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6	23
GRAPHENE	1 -> 3 -> (4, 0) -> (0, 2, 5)	12

2.3. Cache-Oblivious Task Scheduling

DAG-aware scheduling has been studied extensively for both homogeneous and heterogeneous systems. HEFT and GRAPHENE are representative DAG-aware scheduling algorithms that have been proven to perform effectively in many parallel computing frameworks. HEFT traverses the DAG and assigns a rank value to each node according to the resource demands and durations of its child nodes. A node with a higher rank value implies a higher schedule priority. HEFT considers the resource demands of the nodes and estimated JCT to obtain an efficient scheduling order. Several optimization algorithms have been proposed for HEFT, such as HEFT with lookahead [21], to improve the resource utilization and JCT of applications further. GRAPHENE uses the concept of troublesome tasks, which are long-running tasks that are usually difficult to pack and play a decisive role in the task completion time. GRAPHENE identifies troublesome tasks and divides the other tasks into different task sets based on their dependencies with troublesome task sets. Subsequently, GRAPHENE schedules tasks in sets based on their resource demands, without violating the task dependencies. GRAPHENE further investigates the features of DAG-aware scheduling and improves the JCT. Inspired by GRAPHENE algorithm, the scheduling priority of long-running tasks has become the focus of many studies related to DAG-aware scheduling. For example, Stage Delay Scheduling [10] traverses jobs' DAG and calculates the bounds of execution time in each path. Then, it schedules stages in the long-running execution path with high priority by delaying other stages in limitation time to minimize the resource fragments and maximize the benefits of resource interleaving. These DAG-aware scheduling algorithms offer advantages in parallel computing frameworks compared to FIFO and capacity-based schedule policies. However, these algorithms neglect the impact of the cache management policy in execution and can be further optimized in in-memory data analytics frameworks such as Spark. An efficient cache management policy can improve the cache hit ratio to reduce the JCT by making full use of the memory. As mentioned previously, the scheduling order that is determined by DAG-aware scheduling algorithms is mainly dependent on the resource utilization and possible completion time of long-running tasks, which are strongly influenced by cache management policies. Tasks in Spark can be executed more efficiently if the scheduling algorithm dynamically updates the resource utilization and estimates the JCT according to the cache management policy.

3. System Design

We propose a new cache management policy known as Long-Running Stage Set First (LSF), which is efficient in DAG-aware task scheduling algorithms. Based on our cache management policy, we further propose a cache-aware task scheduling algorithm to reduce the resource fragmentation and the overall JCT of applications. Furthermore, we describe our implementation in Spark.

As mentioned previously, Spark benefits from its cache management policy in computing. Default cache management is scheduling-oblivious and exhibits poor performance in DAG scheduling. A DAG-based scheduler with suitable cache management will offer the advantage of improving the overall performance of Spark. Our solution is divided into two parts according to task dependencies.

The cache management policy for DAG-aware task scheduling makes cache eviction and prefetching decisions based on the characteristics of the DAG-aware task assignment policy.

The cache-aware schedule algorithm assigns tasks based on the dependency between the stages and cache management policy in the current job. It overlaps with the execution of long-running chains of stages to ensure more efficient resource utilization in the clusters.

3.1. Cache Management Policy for DAG-Aware Task Scheduling

The goal of the DAG-aware scheduling algorithm is to reduce the JCT. The resource demands and execution times of the stages are key factors in DAG-aware scheduling. DAG-aware scheduling algorithms generally aim to run the maximal number of pending tasks that fit within the available resources. Moreover, the scheduling overlaps the long-running chains of stages to guarantee the JCT. The cache management policy for DAG-aware scheduling should consider both factors in caching and prefetching data blocks.

Definition 1. *(Cache urgency). For an RDD_i, the cache urgency (CU) is defined as the remaining resource occupation of the longest DAG components that are derived from the stages including RDD_i. An RDD may have various CUs if it is computed in different stages.*

The CU of the RDD can be used to measure the priority of the data block eviction and prefetching. It is mainly dependent on the completion times and resource demands of the previous stages. The CU of RDD_i can be defined as follows:

$$CU(RDD_i) = CT(stage_m) \times R(stage_m) + Max\left(\sum_{stage_{i \in SuccPath(stage_m)}} (CT(stage_i) \times R(stage_i)) \right), \quad (1)$$

where $CT(stage_m)$ and $R(stage_m)$ denote the execution time and the resource demand of $stage_m$, which includes RDD_i, respectively. $SuccPath(stage_m)$ denotes the set of stages starting from $stage_m$ and ending with the final stage of the current job, which can be viewed as the possible execution path starting from $stage_m$ in the current job.

The DAG of the jobs is traversed, and several stage sets are established according to the resource demands of the stages. Each stage set includes stages that can be executed in parallel, without exceeding the resource demands. The RDDs in a stage set share the same cache and prefetching priority in the workflow. The CU of RDD_i in stage set m can be defined as follows:

$$CU(RDD_i) = Max(CU(RDD_j)) RDD_i \in stage\ set\ m. \quad (2)$$

Stage sets may have various combinations. DAG-aware scheduling overlaps long-running stages in the workflow. First, the stage sets, including the long-running stage, are listed. The set with the maximum resource utilization (RU) is selected as a candidate scheduling stage set in our algorithm. The RU of stage_set_i can be defined as follows:

$$RU(Stage\ set_i) = (\sum_{stage\ j \in Stage\ set_i} CT(stage_j) \times R(stage_j))/(R \times CT(stage_{lr})), \quad (3)$$

where $stage_{lr}$ denotes the long-running stage. The CUs of the RDDs are updated after the candidate scheduling stage sets have been determined. RDDs with a higher CU are more likely to be scheduled first according to the preference of the DAG-aware scheduling method. The cache management policy tracks the process of the stages and updates the CUs of the RDDs. The RDDs with the lowest cache urgency are evicted and that with the highest urgency is prefetched to achieve better computing performance. An RDD that is processed in different stages has various CUs, and the highest value among them is defined as its current CU.

For example, in Figure 2, LSF traverses the parallel chains in the DAG and establishes stage sets (stage0, stage4), (stage0, stage2), (stage0, stage5), (stage2, stage5), and (stage0, stage2, stage5). According to the principle of considering the Long-Running Stage Set First, (stage0, stage2, stage5) is selected as a candidate stage set. The CU of each stage in the candidate stage set is set to be the same as that of the long-running stage0. LSF updates the priority of each RDD for caching and prefetching based on the new CUs. The CU of each RDD is listed in Table 2. The pseudo-code for the data eviction and prefetching policy is presented in Algorithm 1.

Table 2. CUs of RDDs.

RDD	Stage ID	CU
0	0	56
1	1	65
2	2	56
3	3	69
4	4	60
5	5	56
6	6	8

It is noted that, to achieve the basic cache priority, the DAG of the entire application needs to be traversed. However, in systems such as Spark, the applications usually consist of several jobs and only the DAG of the current job can be obtained from the Spark scheduler. Therefore, it is challenging to achieve the entire DAG of the applications. To solve this problem, we reconsider our cache policy in two situations.

Applications that run on in-memory data-parallel systems are mainly recurring and they usually repeat certain jobs with the same DAG to process different datasets. Therefore, it is feasible to learn the entire DAG from previous jobs to achieve higher performance of our cache policy in these applications.

For non-recurring applications with jobs that have different DAGs, LSF operates in the same manner as in recurring applications in each single job, but the cache priority should be recomputed when new jobs arrive. The hit ratio decreases by a certain percentage compared to that for recurring applications.

3.2. Cache-Aware Scheduling Algorithm

DAG-aware schedule algorithms, such as GRAPHENE, HEFT, and branch scheduling, have been studied extensively and several efficient methods have been proposed. Such algorithms exhibit superior performance in the processing of DAG-based applications compared to traditional scheduling methods. However, they are oblivious to the cache policy of Spark and should be optimized for improved performance. Thus, we propose a cache-aware scheduling algorithm that considers the effects of LSF cache management policy. Compared to the current DAG-aware scheduling algorithms, our solution can obtain better JCTs in applications that are processed on Spark.

Algorithm 1 LSF cache management policy.

```
Input: DAG of application, stage_set with long_running tasks
Candidate_Stage_set: Stage sets overlap long-running stages
Candidate_RDD: RDDs ready for prefetching
//Computing CU of each RDD
1:   for each $RDD_i$ of DAG do
2:   $CU(RDD_i) = CT(stage_m) \times R(stage_m) + Max\left(\sum_{stage_{i \in SuccPath(stage_m)}} (CT(stage_i) \times R(stage_i)\right)$
3:   CU_Table <- $CU(RDD_i)$
4:   end for
5:   for each $Stage\ set_i$ including long-running tasks do
6:   $RU(Stage\ set_i) = (\sum_{stage\ j \in Stage\ set_i} CT\left(stage_j\right) \times R\left(stage_j\right))/(R \times CT(stage_{lr}))$
7:   end for
8:   Candidate_Stage_set <- Stage set with highest RU
9:   for each $RDD_i$ in Candidate_Stage_set do
10:  $CU(RDD_i) = Max\left(CU\left(RDD_j\right)\right)\ RDD_j \in stage_{lr}$
11:  end for
12:  for each stage i of SuccPath do
13:  update(CU_Table)
14:  end for
//Eviction policy of cache management
15:  if data block size of $RDD_i$ > free memory then
16:      for each $RDD$ cached in memory do
17:          $RDD_j$ <- $RDD$ with lowest cache urgency
18:          if($CU\left(RDD_j\right)$ < $CU(RDD_i)$) then
19:              if($CU\left(RDD_j\right)$) is not 0) then
20:                  evict($RDD_j$)
21:                  write $RDD_j$ to disk
22:                  update(CU_Table)
23:              else
24:                  evict($RDD_j$)
25:              end if
26:          end if
27:      end for
28:  end if
//Prefetch RDD
29:  for each stage unprocessed do
30:      Candidate RDD <- RDD with highest CU in CU_Table
31:      if size of (Candidate RDD) < free memory then
32:          prefetch(Candidate RDD)
33:      end if
34:  end for
```

The DAG-aware scheduler traverses the DAG of the application and schedules tasks in parallel chains simultaneously to optimize the overall JCT. We developed our cache-aware algorithm on two layers. At the job layer, jobs on parallel chains have an inner relationship owing to the utilization of the cache management policy. An appropriate combination of job executions will increase the cache hit ratio and further improve the RU. The commonalities in the RDD access can be used to measure the correlations between jobs, because jobs with partially identical RDDs may perform similarly in the cache results.

Definition 2. *(RDD access similarity). For jobs i and j, the RDD access similarity (RAS) is defined as the sum of the RDD CUs, which are both scheduled in jobs i and j.*

According to the cache management policy, RDDs with various CUs have greater opportunities to be cached in the workflow. The RAS between jobs can be expressed by the sum of the same RDDs with various CUs for simplification. After traversing the DAG of the application, an initial schedule order is generated according to the data dependencies among the jobs, and the RAS of successor jobs relative to the entry job is calculated. The job with the highest RAS will be scheduled together with the entry job. As the application is executed, the RAS of the successor jobs is updated dynamically based on the running job in the work nodes.

The entry job and its successor job with the highest RAS are submitted simultaneously. In our solution, a heuristic algorithm based on stage dependencies and a cache management policy are selected for the running jobs.

At the stage layer of our cache-aware scheduling algorithm, the CU in LSF is used as the initial scheduling order. To reduce resource fragmentation, we propose the concept of schedule idle time (SIT), which is determined by the estimated JCT and cache hit ratio during execution.

Definition 3. *(Schedule idle time). The SIT of stage_i is defined as the longest delay time of stage_i without postponing the overall JCT.*

As mentioned previously, the CU of the RDDs is defined as the remaining completion time of the longest DAG components that are derived from the stages including these RDDs. The longest remaining completion time of the stage can be calculated in the same manner, and is represented as TR (stage). In our solution, the scheduling order is determined according to both the job dependencies and cache management policy. The RDDs that are cached in memory have a significant impact on the CU of the stages. The SIT of stage_i can be defined as follows:

$$SIT(stage_i) = JRT - \left(TR(stage_i) - \sum_{RDD_m \in stage_i} TC(RDD_m) \right), \tag{4}$$

where $JRT(stage_m)$ denotes the remaining estimated JCT starting from the running stages and $TC(RDD_m)$ denotes the estimated execution time of RDD_m that is cached in memory. Moreover, $SIT(stage_i)$ represents the possible delay time for stage_i without delaying the overall JCT. It also implies that the stage from another job could be scheduled before stage_i if its execution time is less than $SIT(stage_i)$. Algorithm 2 presents the pseudo-code for the cache-based DAG-aware task scheduling.

3.3. Spark Implementation

Architecture overview. Figure 3 presents an overview of the architecture as well as the interaction between the modules of the cache manager and DAG scheduler. Our implementation is composed of three pluggable modules and modifications in DAGScheduler: AppProfiler and CacheManager are deployed on the master node and CacheMonitor is deployed on each slave node. DAGScheduler is modified for the DAG-aware scheduling algorithm. The other modules, such as BlockManager EndpointMaster and BlockManager SlaveEndpoint, are original Spark components. The main APIs for our implementation are listed in Table 3.

Table 3. APIs of spark implementation.

API	Description
DAGProfile	AppProfiler analyze DAG and generates an initial scheduling order
updateCachePriority	CacheManager sends a new cache urgency file to CacheMonitor
updateRAS	DAGAnalyzer returns a new RAS index when receiving new DAGs
BlocksEviction	When the cache is full, data with a low cache urgency will be evicted
DataPrefetch	Prefetches specific blocks for use in the next stage

Algorithm 2 Cache-aware scheduling.

```
Input: DAG of application, initial schedule sequence init($App_s$)
$job_{entry}$: stage set of initial job in application
$job_{candidate}$: stage set of job suitable for parallel execution
$job_{running}$: stage set of running jobs
Pending_stage: stages pending for scheduling
1: $job_{entry}$ <- initial job of init($App_s$)
2:     for each job of $App_s$ do
3:         RAS_Table <- RAS($job_{entry}$, $job_i$)
4:     end for
5: $job_{candidate}$ <- job with highest RAS in RAS_Table
6: $job_{running}$ <- stages in $job_{entry}$ and $job_{candidate}$
//Schedule phase
7:     Pending_stage <- sort stages of $job_{running}$ by CU(stage) calculated in LSF
8:     Cache_Queue <- RDDs cached in memory
9:     for each stage i in $job_{entry}$ do
10:        $SIT(stage_i) = JRT - \left(TR(stage_i) - \sum_{RDD_m \in stage_i} TC(RDD_{cached})\right)$
11:        SIT_Table <- $SIT(stage_i)$
12:      end for
13:        for stage m in Pending_stage do
14:            if stage m is runnable then//Parent stages have been executed
15:                for each $RDD_{cached}$ in Cache_Queue do
16:                    if $RDD_{cached} \in stage_m$ then
17:            $TE(stage_m)$ <- $TE(stage_m) - TC(RDD_{cached})$
18:                    end if
19:                end for
20:                    end if
21:            If $SIT(stage_i) > TE(stage_m)$ then
22:                    Schedule_Queue() <- $stage_m$
23:                    Schedule_Queue() <- $stage_i$
24:                    update(Pending_stage)
25:                else
26:                    Schedule_Queue <- $stage_i$
27:            end if
28:            update(SIT_Table)
```

AppProfiler. AppProfiler derives the DAG of a job from the Spark application to prepare essential information for CacheManger and DAGScheduler. In recurring applications, it creates the entire DAG of the application by analyzing the DAG of the previous job. Subsequently, it analyzes the DAG and calculates RAS of each job. Finally, DAGAnalyer sends the DAG of the application to CacheManager and DAGScheduler.

DAGScheduler. DAGScheduler is responsible for dividing the DAG of the jobs into stages and setting an appropriate scheduling order for the stages. FIFO is the default scheduling policy. We modify DAGScheduler to adapt it to our DAG-aware scheduling algorithm.

CacheManager. CacheManager is one of the key components of this architecture. It reconstructs the DAG of the application according to the information that is received from AppProfiler. CacheManager computes the cache urgency for each RDD according to LSF cache management policy. Furthermore, with the information that is collected by CacheMonitor deployed on the slave nodes, CacheManager is also responsible for the eviction and prefetching algorithm of the RDDs at runtime.

CacheMonitor. CacheMonitor is deployed on the slave nodes in the cluster. It accesses various APIs and collects the necessary information from memory. Moreover, it carries out the RDD eviction strategy according to the instructions that are sent back from CacheManager.

Workflow. After submitting the application, the Spark driver creates a SparkContext within which AppProfiler and CacheManager are established. The other modules in Spark are also established. Subsequently, the driver informs the slave nodes to launch the Spark executors, and then deploys CacheManager and BlockManager. After establishing the connection between the driver and executor, AppProfiler derives the DAG from the application and generates an initial scheduling order according to the dependencies between jobs. Thereafter, it sends the essential information, including initial scheduling order and RAS of each job, to DAGScheduler and CacheManager. CacheManager reconstructs the DAG and recomputes the cache urgencies of the RDDs. In combination with the running information received from CacheMonitor and the new cache urgencies, CacheManager sends the eviction and prefetching strategy of the current stages to BlockManager MasterEndpoint. DAGScheduler receives the cache stats and stage stats of the executor nodes, following which it calculates the scheduling order for the jobs and stages according to the cache-aware scheduling algorithm. DAGScheduler assigns tasks according to the optimized scheduling sequence.

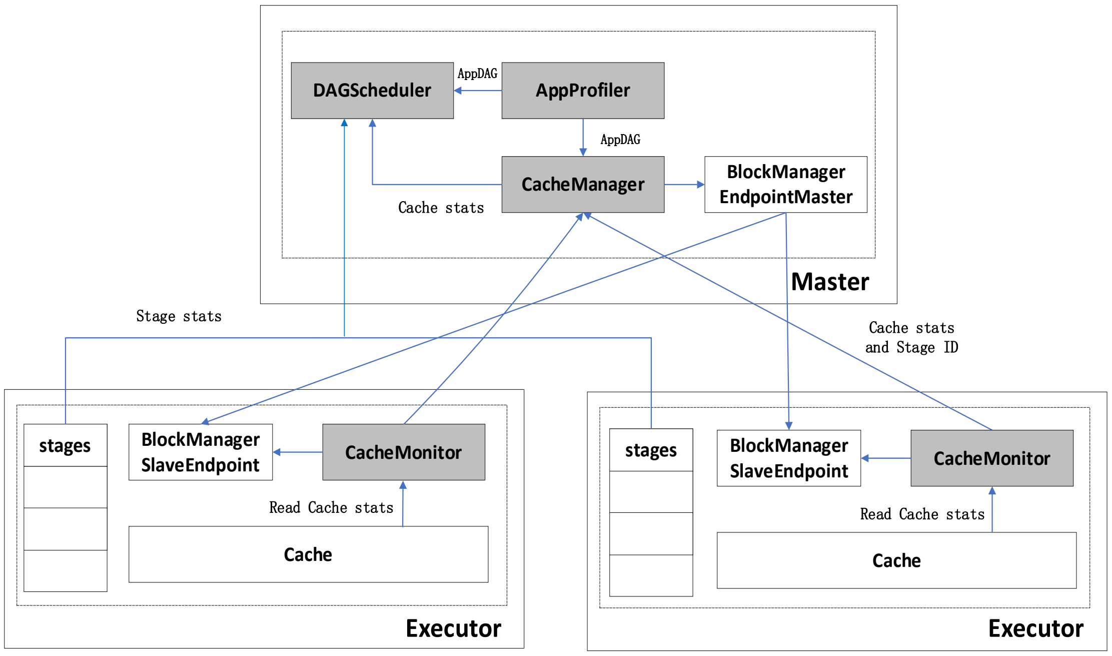


Figure 3. Overall system architecture of Spark with LSF cache management and DAG scheduling. Our modules are highlighted as shaded boxes.

Communication overhead. Our solution introduces a slight communication overhead in Spark. While CacheManager reconstructs the DAG and determines the cache priority for each RDD, the cache priority file is sent to each slave node and can be maintained locally during the workflow. CacheManager only updates the cache priority when necessary through heartbeats between the master and slaves. In particular, CacheManager should inform the slave nodes to update their initial RDD caching priority when AppProfiler receives the DAG of a new job. Thus, the overhead caused by communication can be neglected during this workflow.

4. Evaluations

We evaluated the performance of our cache-aware scheduling algorithm using typical benchmarks.

4.1. Experimental Environment

Our experimental platform was composed of several virtualized machines in two high-performance blade servers, each with 32 cores and 64 GB of memory. The main tests were conducted in this virtual environment with nine nodes, which consisted of one master node and eight slave nodes. The master node obtained a better configuration to satisfy the computing demand for the cache policies. All nodes were deployed using Spark 2.4.0 and Hadoop 2.8.0. The datasets were generated using SparkBench. The workloads and input data sizes are listed in Table 4.

Table 4. Workloads and data input sizes.

Workloads	Data Sizes
KMeans	6 GB
PageRank	9 GB
Connected Component	8 GB
PregelOperation	10 GB
SVD++	8.3 GB

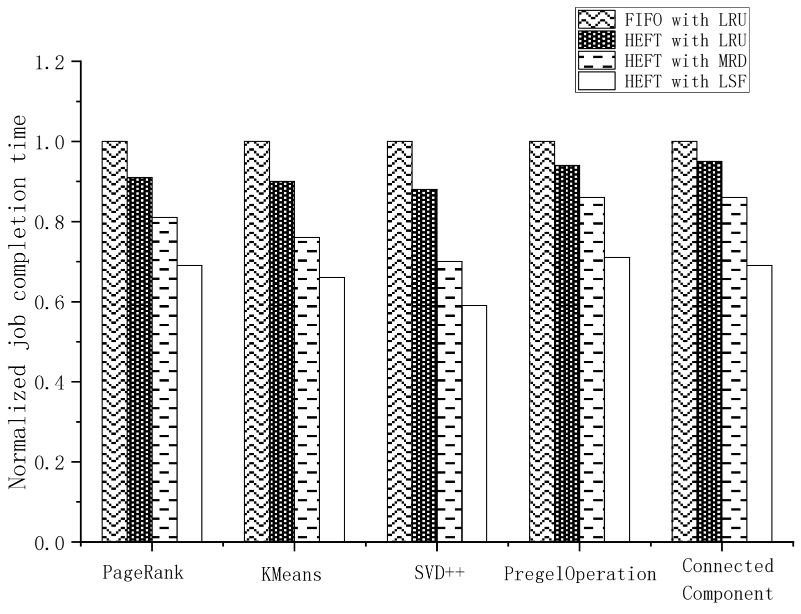
4.2. Performance of LSF

The master was configured with eight cores and 8 GB of memory, whereas the slave node was configured with four cores and 4 GB of memory. We implemented two typical DAG-aware scheduling algorithms in our cluster to evaluate the performance of our cache management policy LSF. The benchmarks were selected from SparkBench. Each DAG-aware task scheduling algorithm was implemented with three different cache management policies: LRU, MRD, and LSF. LRU is the default cache management policy in Spark, whereas MRD evicts the furthest data block and prefetches the nearest one according to the FIFO task scheduler. The results are presented in Figure 4.

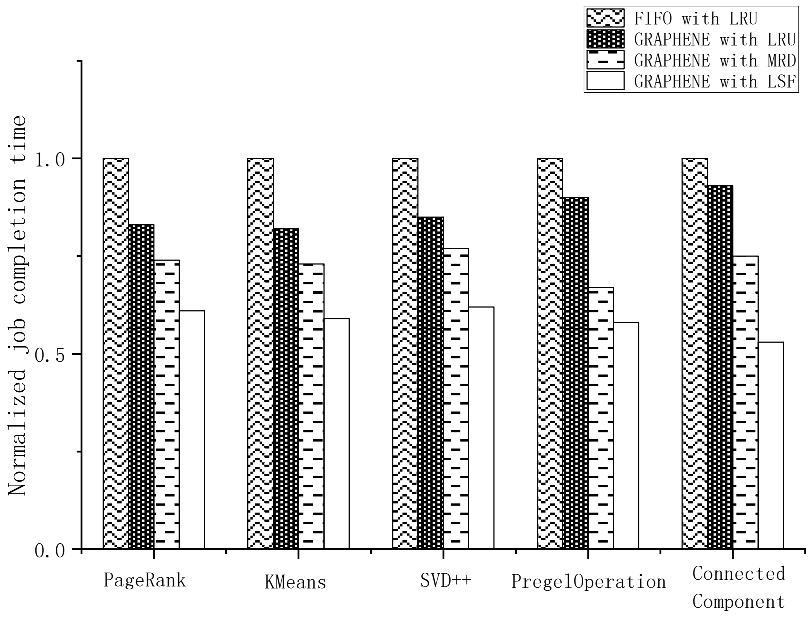
LSF exhibited a significant decrease in the benchmark runtime by increasing the hit ratio of the cache in both DAG-aware task scheduling algorithms. We used a combination of the default scheduler FIFO and default cache management policy LRU as a baseline in Spark. It is clear that the DAG-aware task scheduling algorithm exhibited a performance advantage compared to the default scheduler. HEFT with LRU and GRAPHENE with LRU reduced the JCT by an average of 9% and 15%, respectively, compared to the default FIFO with LRU. As illustrated in Figure 4, HEFT and GRAPHENE with the MRD policy implemented outperformed the same scheduling algorithms with LRU. This was mainly a result of the prefetching method. MRD always prefetches data blocks with the nearest distance in the FIFO scheduler. Although an inconsistency existed between the scheduling order of the DAG-aware task scheduling algorithms and prefetch priority of MRD, the prefetch policy exhibited its advantage in increasing the hit ratio of the cache. HEFT with LSF and GRAPHENE with LSF demonstrated significant performance advantages in our results. GRAPHENE with LSF reduced the job completion time by up to 19% compared to GRAPHENE with MRD in ConnectedComponent. Compared to the default FIFO with LRU, the improvement reached 39%. LSF exhibited better interaction with DAG-aware task scheduling algorithms than MRD and LRU. The hit ratio of the cache during the execution is depicted in Figure 5.

As mentioned above, LSF works in different ways in recurring applications and non-recurring applications. In non-recurring applications, LSF recomputes the cache urgencies of RDDs in each single job because the data dependency and runtime information of these jobs are usually not available a priori. We evaluated the performance of LSF in recurring mode and non-recurring mode. As shown in Figure 6, GRAPHENE with LSF suffered about 14% performance loss in non-recurring mode on average compared to its performance in recurring mode. We attribute the performance loss to the fact that workloads of succeeding stages in future jobs, whose DAG is yet available in non-recurring mode, directly affect the computation of cache urgencies. Therefore, the cache priority calculated in non-recurring mode may not be accurate. Studies show that a large portion of cluster workloads are recurring applications [22]. In most instances, LSF works in recurring mode to achieve better performance.

LSF significantly increased the hit ratio of the cache in the jobs scheduled by GRAPHENE. The reference distance in the MRD was calculated according to the FIFO scheduling order. Thus, MRD performed effectively in the default mode but suffered a significant performance loss in the DAG-aware task scheduling algorithm. Thus, LSF exhibited its efficiency in DAG-aware task scheduling algorithms, and its advantages will be expanded in memory-constrained situations.



(**a**)



(**b**)

Figure 4. Performance comparison with existing cache policies in DAG-aware task scheduling. (**a**) performance comparison with LRU and MRD in HEFT; (**b**) performance comparison with LRU and MRD in GRAPHENE.

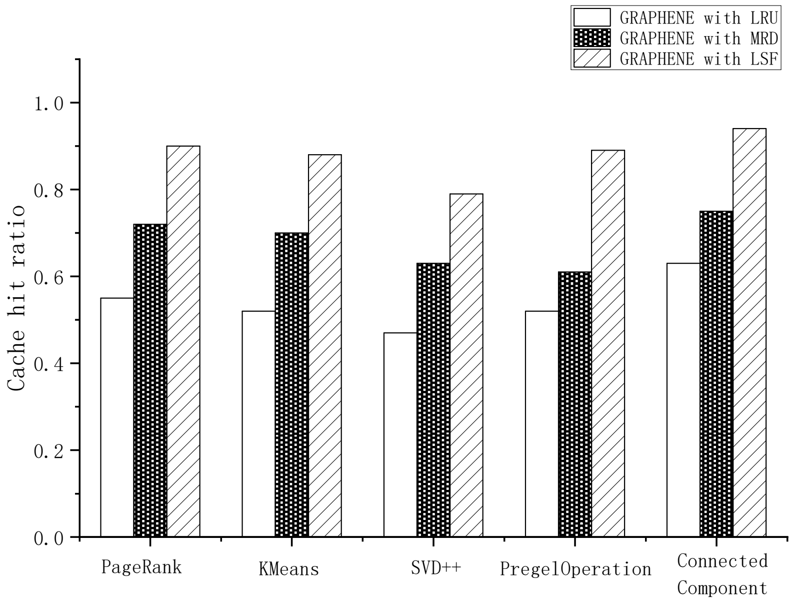


Figure 5. Hit ratio of cache during execution.

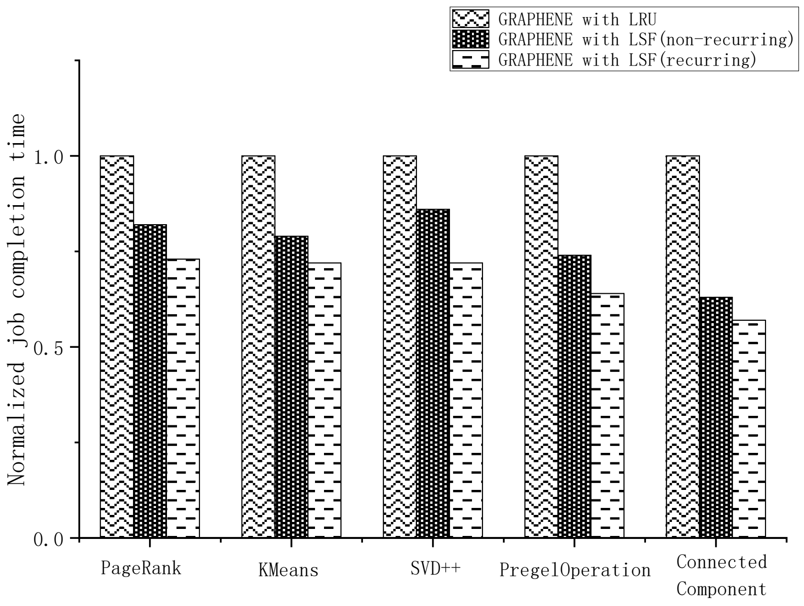


Figure 6. Performance comparison between recurring mode and non-recurring mode of LSF.

4.3. Performance of Cache-Aware Scheduling

We evaluated our cache-aware task scheduling algorithm with the LSF cache management policy using the same benchmarks as above in Spark. Cache-aware algorithms dynamically update scheduling priorities based on both the resource demands and cache status. Compared to the current DAG-aware task scheduling algorithms, our approach was more sensitive to the available resources of the cluster and exhibited improved resource utilization. The results are presented in Figure 7.

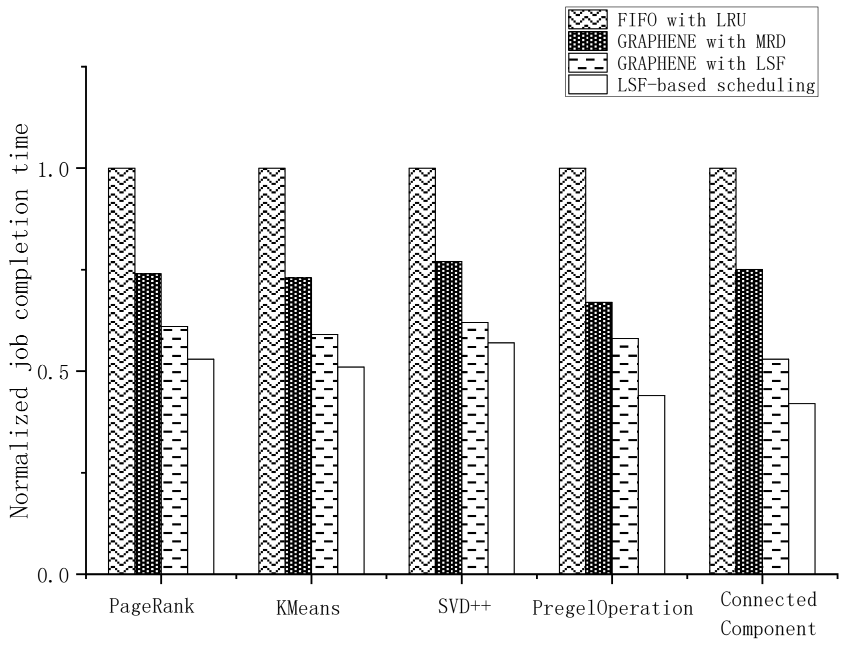


Figure 7. Performance comparison between GRAPHENE and cache-aware scheduling algorithm.

As illustrated in Figure 7, the cache-aware task scheduling algorithm with LSF outperformed FIFO with LRU and GRAPHENE with MRD by an average of 48% and 23%, respectively. Moreover, it achieved a performance improvement of approximately 12% compared to GRAPHENE with LSF. Particularly on the CPU-intensive benchmarks, such as PregelOperation and ConnectedComponent, the cache-aware task scheduling algorithm reduced the JCT by up to 15% compared to GRAPHENE with LSF. Cache-aware scheduling calculates the remaining resources and estimated JCT that are dynamically associated with the cache management policy. It postpones certain stages with negligible overheads to reduce resource fragmentation, which improves the overall JCT. The CPU utilization during the execution is depicted in Figure 8. Our cache-aware task scheduling algorithm made full use of the SIT and achieved better performance in reducing the CPU fragmentation.

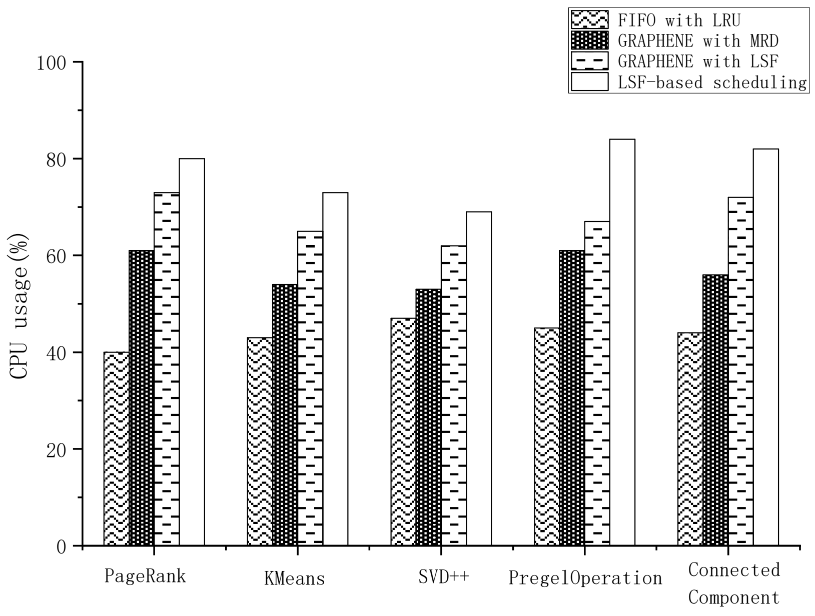


Figure 8. CPU utilization of GRAPHENE and LSF-based scheduling.

5. Conclusions

We have presented a cache management policy with data eviction and prefetching in Spark, known as LSF. LSF calculates the caching and prefetching priority of the RDDs according to their unprocessed workloads and significance in parallel scheduling, which are key factors in DAG-aware scheduling algorithms. Furthermore, we proposed a cache-aware task scheduling algorithm based on LSF to reduce the resource fragmentation in computing. We implement LSF and cache-aware scheduling algorithm as a pluggable memory manager in Spark 2.4. The proposed approaches are evaluated in a Spark cluster by typical benchmarks of parallel computing. With different evaluation methods, the results show that LSF outperforms present cache management policies in DAG-aware task scheduling approach and the LSF-based scheduling algorithm further improves the job completion time in Spark. Further study will explore the impact of unified memory management [23] in Spark, a flexible allocation method of storage and execution memory will improve the efficiency of cache management policy and task scheduling algorithms.

Author Contributions: Conceptualization, Y.Z. and J.D.; methodology, Y.Z., J.D. and H.L.; software and visualization, Y.Z. and Y.L.; validation, Y.Z. and H.L.; investigation, Y.Z. and J.D.; resources, Y.L. and J.W.; writing—original draft preparation, Y.Z., J.D. and H.L.; writing—review and editing, Y.Z., J.D., H.L. and J.W.; supervision, J.D. and H.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China under Grant 61100029.

Data Availability Statement: The data presented in this study are available upon reasonable request from the corresponding author.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

1. Wang, J.; Balazinska, M. Elastic Memory Management for Cloud Data Analytics. In Proceedings of the 2017 USENIX Annual Technical Conference, Santa Clara, CA, USA, 12–14 July 2017; pp. 745–758.
2. Nasu, A.; Yoneo, K.; Okita, M.; Ino, F. Transparent In-memory Cache Management in Apache Spark based on Post-Mortem Analysis. In Proceedings of the 2019 IEEE International Conference on Big Data, Los Angeles, CA, USA, 9–12 December 2019; pp. 3388–3396.
3. Sun, G.; Li, F.H.; Jiang, W.D. Brief Talk about Big Data Graph Analysis and Visualization. *J. Big Data* **2019**, *1*, 25–38. [CrossRef]
4. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster Computing with Working Sets. *HotCloud* **2010**, *10*, 95–101.
5. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Stoica, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, San Jose, CA, USA, 25–27 April 2012; pp. 15–28.
6. Bathie, G.; Marchal, L.; Robert, Y.; Thibault, S. Revisiting Dynamic DAG Scheduling under Memory Constraints for Shared-memory Platforms. In Proceedings of the 2020 IEEE international Parallel and Distributed Processing Symposium Workshops, New Orleans, LA, USA, 18–22 May 2020; pp. 597–606.
7. Liu, J.; Shen, H. Dependency-Aware and Resource-Efficient Scheduling for Heterogeneous Jobs in Clouds. In Proceedings of the 2016 IEEE International Conference on Cloud Computing Technology and Science, Luxembourg, 12–15 December 2016; pp. 110–117.
8. Li, Z.; Zhang, Y.; Zhao, Y.; Li, D. Efficient Semantic-Aware Coflow Scheduling for Data-Parallel Jobs. In Proceedings of the 2016 IEEE International Conference on Cluster Computing, Taipei, Taiwan, 12–16 September 2016; pp. 154–155.
9. Wang, S.; Chen, W.; Zhou, L.; Zhang, L.; Wang, Y. Dependency-Aware Network Adaptive Scheduling of Data-Intensive Parallel Jobs. *IEEE Trans. TPDS* **2018**, *30*, 515–529. [CrossRef]
10. Shao, W.; Xu, F.; Chen, L.; Zheng, H.; Liu, F. Stage Delay Scheduling: Speeding up DAG-style Data Analytics Jobs with Resource Interleaving. In Proceedings of the 48th International Conference on Parallel Processing, Kyoto, Japan, 5–8 August 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1–11.
11. Grandl, R.; Chowdhury, M.; Akella, A.; Ananthanarayanan, G. Altruistic Scheduling in Multi-Resource Clusters. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, Savannah, GA, USA, 2–4 November 2016; pp. 65–80.

12. Topcuoglu, H.; Hariri, S.; Wu, M.Y. Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. TPDS* **2002**, *13*, 260–274. [CrossRef]
13. Grandl, R.; Kandula, S.; Rao, S.; Akella, A.; Kulkarni, J. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, Savannah, GA, USA, 2–4 November 2016; pp. 81–97.
14. Hu, Z.; Li, D.; Zhang, Y.; Guo, D.; Li, Z. Branch Scheduling: DAG-Aware Scheduling for Speeding up Data-Parallel Jobs. In Proceedings of the International Symposium on Quality of Service, Phoenix, AZ, USA, 24–25 June 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1–10.
15. Dessokey, M.; Saif, S.M.; Salem, S.; Saad, E.; Eldeeb, H. Memory Management Approaches in Apache Spark: A Review. In Proceedings of the International Conference on Advanced Intelligent Systems and Informatics, Cairo, Egypt, 19–21 October 2020; pp. 394–403.
16. Yu, Y.H.; Wang, W.; Zhang, J.; Letaief, K. LRC: Dependency-Aware Cache Management Policy for Spark. In Proceedings of the 2017 IEEE Conference on Computer Communications, Atlanta, GA, USA, 1–4 May 2017; pp. 1–9.
17. Wang, B.; Tang, J.; Zhang, R.; Ding, W.; Qi, D. LCRC: A Dependency-Aware Cache Management Policy for Spark. In Proceedings of the Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communication, Melbourne, Australia, 11–13 December 2018; pp. 956–963.
18. Perez, T.B.; Zhou, X.; Cheng, D. Reference-distance Eviction and Prefetching for Cache Management in Spark. In Proceedings of the 47th International Conference on Parallel Processing, Eugene, OR, USA, 13–16 August 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 1–10.
19. Li, M.; Tan, J.; Wang, Y.; Zhang, L.; Salapura, V. SparkBench: A Comprehensive Benchmarking Suite for In-memory Data Analytic Platform Spark. In Proceedings of the 12th ACM International Conference on Computing Frontiers, Ischia, Italy, 18–21 May 2015; Association for Computing Machinery: New York, NY, USA, 2015; pp. 1–8.
20. Xu, L.; Li, M.; Zhang, L.; Butt, A.R.; Wang, Y.; Hu, Z.Z. MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium, Chicago, IL, USA, 23–27 May 2016; pp. 383–392.
21. Bittencourt, L.F.; Sakellariou, R.; Madeira, E.R.M. DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, Italy, 17–19 February 2010; pp. 27–34.
22. Ferguson, A.D.; Bodik, P.; Kandula, S.; Boutin, E.; Fonseca, R. Jockey: Guaranteed job latency in data parallel clusters. In Proceedings of the 7th ACM European Conference on Computer Systems, Bern, Switzerland, 10–13 April 2012; Association for Computing Machinery: New York, NY, USA, 2012; pp. 99–112.
23. Zhu, Z.; Shen, Q.; Yang, Y.; Wu, Z. MCS: Memory Constraint Strategy for Unified Memory Manager in Spark. In Proceedings of the 23rd International Conference on Parallel and Distributed Systems, Shenzhen, China, 15–17 December 2017; pp. 437–444.