DAG-Aware Joint Task Scheduling and Cache Management in Spark Clusters

Yinggen Xu, Liu Liu, Zhijun Ding*
The Key Lab of Embedded System and Service Computing, Tongji University, Shanghai, China
*Email: dingzj@tongji.edu.cn

***Abstract*—Data dependency, often presented as directed acyclic graph (DAG), is a crucial application semantics for the performance of data analytic platforms such as Spark. Spark comes with two built-in schedulers, namely FIFO and Fair scheduler, which do not take advantage of data dependency structures. Recently proposed DAG-aware task scheduling approaches, notably GRAPHENE, have achieved significant performance improvements but paid little attention to cache management. The resulted data access patterns interact poorly with the built-in LRU caching, leading to significant cache misses and performance degradation. On the other hand, DAG-aware caching schemes, such as Most Reference Distance (MRD), are designed for FIFO scheduler instead of DAG-aware task schedulers.**

In this paper, we propose and develop a middleware *Dagon*, which leverages the complexity and heterogeneity of DAGs to jointly execute task scheduling and cache management. Dagon relies on three key mechanisms: DAG-aware task assignment that considers dependency structure and heterogeneous resource demands to reduce potential resource fragmentation, sensitivity-aware delay scheduling that prevents executors from long waiting for tasks insensitive to locality, and priority-aware caching that makes the cache eviction and prefetching decisions based on the stage priority determined by DAG-aware task assignment. We have implemented Dagon in Apache Spark. Evaluation on a testbed shows that Dagon improves the job completion time by up to 42% and CPU utilization by up to 46% respectively, compared to GRAPHENE plus MRD.

I. Introduction

Spark is a prevalent data analytics framework that generalizes MapReduce parallel and distributed data processing [22]. It uses an in-memory resilient distributed datasets (RDDs) [21] to bypass expensive disk and network I/O accesses. Many data analytics workloads inherently exhibit strong data dependencies. Such dependencies are usually presented as Directed Acyclic Graph (DAG). For instance, machine learning and query workloads usually have jobs exhibiting multiple levels of dependencies. However, Spark built-in schedulers, namely FIFO and Fair scheduler [7], and LRU caching are unaware of complexity and heterogeneity of application DAGs.

Many prior studies [10], [9], [6], [21], [14] have shown that application DAGs are usually constructed according to large and complex data dependencies. A median DAG can reach a depth of five with thousands of tasks [10]. The main source of heterogeneity in DAGs comes from the difference in task durations (ranging sub-seconds to hundreds of seconds), resource usages, and data locality sensitivity.

DAG-aware scheduling is not totally new. In Spark, DAGScheduler [21] examines the type of RDD dependencies to build a DAG of stages for job execution. Each stage cannot start running until its precedent stages are completed. At each scheduling step, the scheduler needs to select a stage and allocate resources to its tasks via a scheduling algorithm. However, the built-in scheduling algorithms, i.e., FIFO and Fair, often generate schedules for one DAG. Long-running chains of stages have no other stages to overlap with, which reduces task parallelism and results in resource fragmentation in executors. The classic critical path based scheduler [8] considers the complexity of DAGs by queuing stages in the order of the critical path length, but it ignores the different resource usages of tasks at each stage. A recently proposed DAG-aware scheduler, Graphene [10], considers both complexity and heterogeneity of DAGs. It accelerates job completion by focusing on long-running tasks and tasks with tough-to-pack resource demands (called troublesome tasks). However, as it pays little attention to cache management, the data access pattern resulted by Graphene is difficult to be exploited by the current DAG-aware caching policies [19], [14].

Once a stage is selected, Spark TaskSchedulerImpl launches its tasks into executors according to data locality using delay scheduling [20]. Based on the current wait time, the scheduler calculates the allowed data locality that the executors can launch the tasks. If an executor cannot launch a task with the allowed locality, it waits for the next scheduling period, allowing other executors to launch tasks. Some executors that have few local data accesses often have to wait and be idle, even though the tasks at this stage are insensitive to data locality. Thus, the resource wastage due to idle executors often outweighs the benefit of data locality achieved by delay scheduling. Timely removal of an executor without many local data accesses can avoid resource wastage. However, in Spark, once a container goes away, its executors need to stick around during the application lifetime to share cached data across multiple stages of application DAGs. Currently, there lacks research towards a stage-aware data locality strategy in prevalent data analytic platforms such as Spark.

In Spark, BlockManager within individual executors adopts the LRU policy for cache management. As LRU simply evicts the least recently used blocks, it is oblivious to the data access pattern due to DAG-aware scheduling. There are recent studies that aim to improve the cache hit ratio in DAG-aware environments. For example, LRC [19] evicts the cached data blocks with the smallest reference count. But they do not consider the time-spatial distribution of data references, which often causes inefficient use of the cache space since it may take a long time before a cached data block has its next access. Based on stage reference distance defined by stage ID, MRD [14] evicts the furthest data in the cache to be used while prefetching the nearest data that will be needed. However, the caching scheme is designed to follow the data access pattern caused by FIFO scheduler, and thus often makes erroneous cache decisions in case of DAG-aware task scheduling.

In this work, we aim for joint DAG-aware task scheduling and cache management. We propose and develop a middleware Dagon, which aims to improve resource utilization and application performance in Spark clusters. Dagon relies on three key designs that leverage DAG information, i.e., dependency structure and heterogeneous resource demands, various data-locality sensitivities of individual stages, and cache efficiency awareness to tackle the aforementioned scheduling and caching issues. We make the following contributions.

- We propose to exploit the different task resource demands of DAG stages and apply dynamic resource configuration to run tasks for each stage. It enables an executor to achieve the maximum throughput based on the actual resource demands of individual tasks.
- Motivated by the complex structure and heterogeneous resource demand in DAGs, we design a DAG-aware task assignment policy to reduce potential resource fragmentations. The resulted data access patterns are easy to be exploited by the cache management.
- We design a sensitivity-aware delay scheduling policy that takes into account the different data locality sensitivities of DAG stages to avoid unnecessary executor idling.
- We design a priority-aware cache management scheme that considers the time-spatial distribution of data references and exploits the data access patterns due to the DAG-aware task assignment.

We have implemented Dagon in Spark running on Apache YARN [16] cluster manager. We have evaluated its efficacy with applications from SparkBench suite [12]. Experimental results by running various applications show that Dagon reduces the job completion time by up to 42% while increases CPU utilization by 46% compared to GRAPHENE plus MRD. Dagon also can be tailored for other DAG-based data analytic platforms such as Apache Tez [1].

In the following, Section II gives motivational case studies. Section III elaborates the design and development of Dagon. Section IV presents the implementation details. Section V reports the experimental results. Related work is reviewed in section VI. We conclude the paper in Section VII.

II. Motivational Case Studies

A. Case Studies and Challenges

DAG-blind Task Scheduling. We use the DAG shown in Figure 1 to illustrate the complexity and heterogeneity issues with the current Spark task assignment. Each box with a solid outline represents an RDD. The shaded rectangles are partitions, and they are in black if the data blocks are already cached in memory. The boxes with a dotted outline represent stages. The labels on the top of stages are their task resource demands and durations (represented as $\langle resource, duration \rangle$). For the sake of clarity, we assume that there is only one executor with 16 vCPUs available. Also, each task can start and finish its execution within the estimated duration given its required resource capacity.

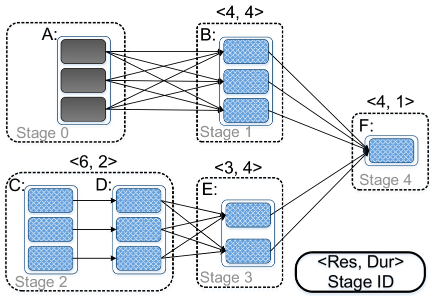
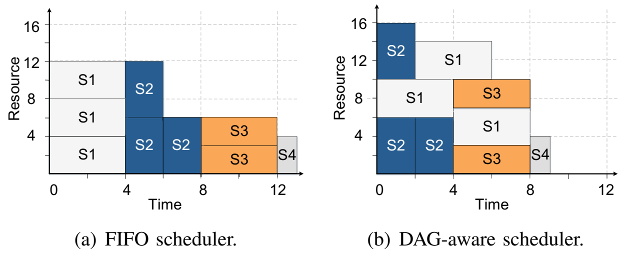


Fig. 1. A DAG with dependencies and various resource requirements.



(a) FIFO scheduler. (b) DAG-aware scheduler.

Fig. 2. Scheduling stages by two different schedulers.

Figure 2 shows the details of the resource scheduling diagram for an executor. Figure 2(a) shows that FIFO scheduler simply allocates 12 vCPUs to three tasks of stage 1 at time 0 according to the stage ID. However, its ignorance of heterogeneous resource demand, that is, each task of stage 2 requires 6 vCPUs instead of 4 vCPUs, causes 4 vCPUs wastage from time 0 to time 4. After time 4, there is no other work to overlap the long-running chain of stages (i.e., stage 2, stage 3, stage 4), which reduces task parallelism and results in severe resource fragmentation from time 4 to time 13. The cause is that FIFO scheduler picks tasks of the short-running chain of stages (i.e., stage 1, stage 4) at time 0, without considering the DAG structure. In contrast, a DAG-aware scheduler can exploit the heterogeneous demand and DAG structure to reduce resource fragmentation as shown in Figure 2(b). For example, it picks one task of stage 1 and two tasks of stage 2 at time 0, resulting in the full resource usage from time 0 to time 2. Since it schedules tasks of the long-running chain of stages with a higher priority, more tasks overlap from time 2 to time 8, which improves task parallelism and reduces resource fragmentation.

Stage-oblivious Locality Enhancement. To reveal the potential performance loss caused by delay scheduling, we conduct a case study on a cluster of seven machines (each with 16-core CPUs and 128GB RAM). For clarity, the number of replicas of

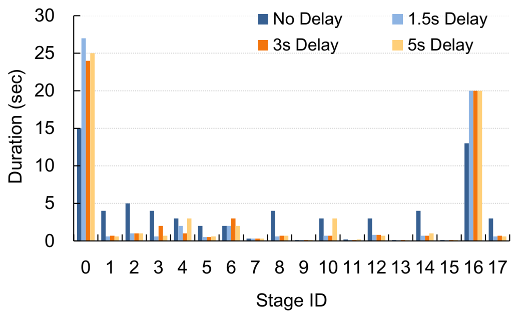


Fig. 3. Effect of four various locality wait times on the stage duration.

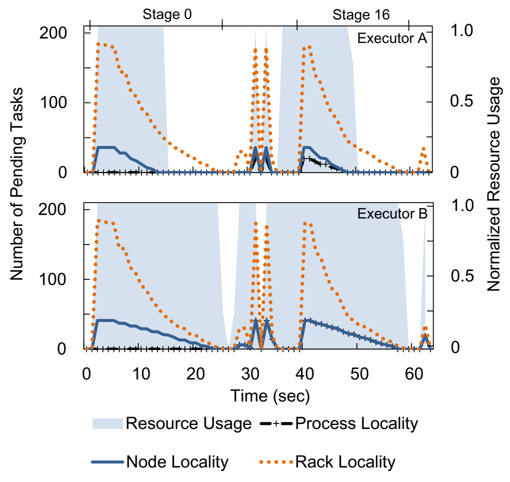


Fig. 4. Pending tasks and resource usage profiles for executors A and B.

HDFS is set to one. Each executor is configured with 4 cores and 32 GB memory. The rest of the experimental environments were configured identically to the default settings. A representative workload from SparkBench suite [12], i.e., *KMeans*, is executed by disabling and enabling delay scheduling.

We first set all *spark.locality.wait* parameters to 0 to disable delay scheduling policy. Figure 3 shows that the durations of stage 0 and 16 are 15 and 13 seconds, respectively. Each of other stages (1∼15 and 17) has a duration of about 3 seconds. We then enable delay scheduling by setting all *spark.locality.wait* parameters to the default value of 3s. The result shows that delay scheduling reduces the duration of stage 1∼15 and stage 17 from about 3 seconds to 0.7 seconds. However, it significantly increases the duration of stage 0 from 15 seconds to 27 seconds, and stage 16 from 13 seconds to 20 seconds. We also run the workload with 1.5s or 5s delay, delay scheduling increases the duration of stages 0 and 16 by 60%, compared to that without delay scheduling.

The results show that different stages have different data locality sensitivities. For stages 0 and 16, a task with rack locality achieves approximately the same performance compared with a task with node locality or process locality. Thus, although enabling delay scheduling can improve data locality for stages 0 and 16, it cannot improve the performance of the two stages. For stages 1∼15 and 17, a task with node locality or rack locality takes almost 15x more time to complete than a task with process locality. Enabling delay scheduling to improve data locality for these stages can significantly reduce their durations. We also note that in order to improve data locality, delay scheduling would make some executors be idle. Resource wastage due to idle executors caused by delay scheduling often outweighs the benefit of data locality enhancement when tasks of a stage are insensitive to locality level, e.g., stages 0 and 16 of *KMeans* workload.

To further illustrate executor idling due to delay scheduling, we show two executors' resource usage profiles corresponding to the default *spark.locality.wait* setting (i.e., 3s delay) in Figure 4. After the 1st second, stage 0 starts and task scheduler has about 40 pending tasks with node locality and 184 pending tasks with rack locality for executors A and B. After the 12th second, the task scheduler has 0 and 25 pending tasks with node locality for executors A and B, respectively. Thus, after the 12th second, the task scheduler still can launch the 25 pending tasks with node locality into executor B and refreshes the accumulated wait time to set node locality as the allowed level. As a result, delay scheduling makes executor A be idle from the 12th second to the 24th second since the scheduler has no pending task with node locality for executor A. The similar situation occurs in stage 16 (i.e., from the 39th to the 59th second). As stages 0 and 16 are insensitive to data locality, we propose to launch the pending tasks with rack locality into executor A to avoid expensive executor idling.

Inefficient Cache Management. An efficient cache management scheme should consider the time-spatial distribution of data accesses. The LRU policy does not. We revisit the DAG shown in Figure 1 and FIFO scheduler shown in Figure 2(a). Table I (top) shows the data accesses due to FIFO scheduler, and the cached data blocks due to LRU caching (DAG-oblivious) and MRD caching (DAG-aware) [14].

In Table I, a letter with/without the underline denotes a cache hit/miss. For simplicity, we assume that each RDD block has a uniform size and the cache can store three blocks. At time 4, two tasks of stage 2 (S2) are launched, and blocks *C1* and *C2* are accessed. However, LRU caches three blocks of stage 1's output RDD B (i.e., *B1*, *B2* and *B3*) since they have the last write time, making erroneous cache decisions. In fact, we can see that RDD B is only needed in the computation of stage 4 (S4) at time 12, where both RDD B and RDD E are required and corresponding tasks can only be launched after RDD E has become available. From time 0 to time 12, LRU leads to 7 cache hits in total.

MRD caching considers the data access pattern caused by FIFO scheduler according to stage ID. It evicts the furthest data in the cache to be used, while prefetching ones that will soon be accessed. For example, after stage 1 has completed (i.e., time 4), MRD does not cache the recently used output RDD B, which is needed in stage 4. Considering that stage 2 will be scheduled by FIFO scheduler, it prefetches block *C1*, *C2* and *C3*, which results in three cache hits for stage 2. With

TABLE I
COMPARISON OF ACCESSED AND CACHED DATA BLOCKS.

Time	FIFO	Accessed Data	LRU	MRD
0	S1,S1,S1	A1,A2,A3	$\underline{A1}$,$\underline{A2}$,$\underline{A3}$	$\underline{A1}$,$\underline{A2}$,$\underline{A3}$
2				
4	S2,S2	C1,C2	B1,B2,B3	$\underline{C1}$,$\underline{C2}$,C3
6	S2	C3	C2,D1,D2	$\underline{C3}$,D1,D2
8	S3,S3	D1,D2,D3	$\underline{D2}$,C3,$\underline{D3}$	$\underline{D1}$,$\underline{D2}$,$\underline{D3}$
12	S4	B1,B2,B3,E1,E2	D3,$\underline{E1}$,$\underline{E2}$	$\underline{B1}$,$\underline{E1}$,$\underline{E2}$
Time	DAG-aware	Accessed Data	LRU	MRD
0	S1,S2,S2	A1,C1,C2	$\underline{A1}$,A2,A3	$\underline{A1}$,A2,A3
2	S1,S2	A2,C3	C2,D1,D2	C1,C2,$\underline{C3}$
4	S1,S3,S3	A3,D1,D2,D3	C3,B1,$\underline{D3}$	$\underline{D1}$,$\underline{D2}$,$\underline{D3}$
6				
8	S4	B1,B2,B3,E1,E2	$\underline{B3}$,$\underline{E1}$,$\underline{E2}$	$\underline{B1}$,$\underline{E1}$,$\underline{E2}$
12				

the knowledge of data access pattern caused by FIFO, MRD outperforms LRU, obtaining 12 cache hits in total.

Table I (bottom) shows the data accesses due to the DAG-aware scheduler, and the cached data blocks due to LRU and MRD caching schemes. We can see that both LRU and MRD perform poorly with a DAG-aware scheduler. In case of LRU, there are only five cache hits. We take a closer look at MRD since it is DAG-aware. From time 0 to time 2, MRD only obtains two cache hits for stages 1 and 2. Overall, MRD obtains 8 cache hits. The problem is that according to stage ID, MRD is unable to understand the data access pattern caused by DAG-aware task scheduling. The case study illustrates the need of a caching scheme that is designed to work jointly with a DAG-aware scheduler.

B. Opportunities

One major feature of Spark fits well to deal with the iterative and interactive applications due to its capability of sharing cached data across multiple stages of application DAGs. However, the current Spark task scheduling and caching often cause significant performance loss, because of DAG-blind task scheduling, stage-oblivious locality enhancement, and inefficient cache management. The root cause is due to its unawareness of the complexity and heterogeneity of DAGs.

Throughout the case studies, we have observed three opportunities, 1) task assignment should consider DAG information and focus on the long-running chains of stages to reduce potential resource fragmentation, 2) delay scheduling should avoid unnecessary executor idling when the tasks of some stages are insensitive to data locality, and 3) cache management should be designed to work jointly with DAG-aware task assignment. These motivate us to design joint task scheduling and cache management in Spark to improve the resource utilization and application performance.

III. DAGON DESIGN

We present the design of Dagon, a middleware for DAG-aware task scheduling and cache management in Spark. Dagon exploits the complex structure and heterogeneous resource demands in DAGs, enhances data locality in a flexible manner, and prioritizes cache management for DAG-aware task scheduling. It consists of three main components: DAG-aware task assignment, sensitivity-aware delay scheduling, and priority-aware cache replacement.

- **DAG-aware task assignment** dynamically allocates executor resources to each stage based on a priority-based heuristic. It overlaps the execution of long-running chains of stages to reduce resource fragmentation.
- **Sensitivity-aware delay scheduling** enhances task data locality based on locality sensitivities of individual stages to avoid low resource utilization due to executor idling.
- **Priority-aware cache replacement** makes the cache eviction and prefetching decisions based on the stage priority value determined by DAG-aware task assignment.

TABLE II
NOTATIONS IN THE DAGON'S DAG-AWARE TASK ASSIGNMENT.

S_i, F_i	Start and finish times of stage i
q_{it}	Resource quantity consumed by stage i at time t
d_i	Resource demand of each task in stage i
w_i	Number of resource-duration units needed for stage i
pv_i	Unprocessed workload of stage i and its successors

A. DAG-aware Task Assignment

The assignment problem is how to allocate the resource capacities of executors to each stage at runtime such that the dependencies between stages are satisfied, the resource capacities are not exceeded, and the job completion time is minimized. We formulate DAG-aware task assignment as an optimization problem, and then present a heuristic priority-based approach. The key notations are listed in Table II.

1) Problem Formulation: Consider a job DAG defined as follows. A set of stages V has to be scheduled, and a set of dependency relationships E between stages need to be preserved. In each scheduling period t, a stage $i \in V$ consumes a variable quantity of the resource, q_{it}. The problem is then to determine, for each stage $i \in V$, its start time $S_i \geqslant 0$, its finish time $F_i > S_i$ and its resource allocation profile $q_i = \{q_{it}\}$ $(i \in V;\ t = 0, 1, \cdots, T)$. These decision variable values should minimize the overall job completion time, i.e., $max\{F_i : i \in V\}$, while satisfying the following constraints.

First, as each stage cannot be started until its parent stages are finished, we must preserve the dependency relationships. Let P_i denote the set of parent stages of stage i. For any $p \in P_i$, the following constraint guarantees that the stage i is started after the stage p is finished.

$$S_i \geqslant F_p \quad (p \in P_i) \tag{1}$$

Then, let w_i be the workload of stage i. The stage workload is measured by the total resource requirement of its tasks in terms of the number of resource-duration units (vCPU-minute). One vCPU-minute stands for one vCPU used for one minute. Consider the execution of stage 1 in Figure 1. It consists of three tasks. Each task requires $\langle 4\,vCPUs, 4\,minutes \rangle$.

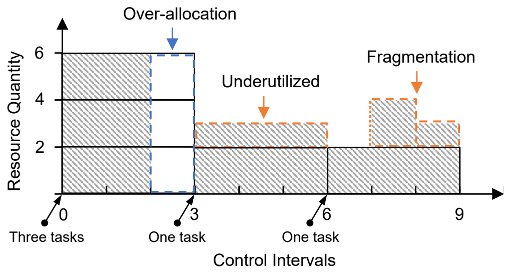


Fig. 5. Example of a resource allocation profile for stage i, $q_i = \{q_{i,0-1} = 6, q_{i,2} = 0, q_{i,3-5} = 3, q_{i,6} = 2, q_{i,7} = 4, q_{i,8} = 3\}$. We portray q_i in the gray area. Consider the case that stage i consists of 5 tasks and each task requires $\langle 2\,vCPUs, 3\,minutes \rangle$. To run stage i, we assign its first three, the fourth and fifth tasks at periods 0, 3 and 6, respectively. Two scenarios occur. Case 1) Resource over-allocation and resource fragmentation occur during period [2, 3] and period [7, 9], respectively, due to the notable fluctuation in the quantity of resource. Case 2) Since $q_{i,3-5}$ mod $2\,vCPUs = 1\,(\neq 0)$, 1 vCPU is un-utilized during period [3, 6].

Algorithm 1 DAG-aware Priority-based Task Assignment.

```
1: $Executors = \langle Exec_1, \cdots, Exec_m \rangle$        // executors available
2: $TaskSet_i$; $d_i$; $pv_i$; $SuccessorSet_i$ $\quad (i \in V)$        // stage $i$'s setting
3: repeat
4: if Any free resource is available on $Executors$ then
5:     Sort stages in $SQ$ according to $pv_i$ in $DESC$ order;
6:     for each stage $i$ in $SQ$ do
7:         $(task, exec) = DelayScheduling\,(TaskSet_i, d_i, Executors)$;
8:         if $task \neq$ null then
9:             $pv_i = w_i + \sum_{j \in SuccessorSet_i} w_j$;    // update $i$'s priority
10:            $exec.freeCPUs - d_i$;    // update $exec$'s free resource
11:            assign $task$ to executor $exec$;
12:            break the loop;
13:        end if
14:    end for
15: end if
16: until All stages completed.
```

The stage workload is 48 vCPUs-minutes. The workload w_1 can be processed in 4 minutes by 12 vCPUs, or in 8 minutes by 8 vCPUs in four minutes and 4 vCPUs on additional four minutes, or in 12 minutes by 4 vCPUs. Generally, for any workload w_i, it must be processed within the execution of stage i by acquiring sufficient amount of resource. The requirement can be represented by the constraints

$$\sum_{t=S_i}^{F_i-1} q_{it} \geqslant w_i \quad and \quad \sum_{t=0}^{S_i-1} q_{it} = \sum_{t=F_i}^{T} q_{it} = 0 \quad (i \in V) \qquad (2)$$

where T is the length of control horizon. Due to the limited available resource, the total quantity of resource allocated to all stages in each period t cannot exceed the resource capacity of executors, RC. That is,

$$\sum_{i \in V} q_{it} \leqslant RC \quad (t = 0, 1, \cdots, T). \qquad (3)$$

Note that some of the resource allocation profiles $q_i = \{q_{it}\}\,(i \in V; t = S_i, S_i+1, \cdots, F_i-1)$ may have notable fluctuations in resource availability, which can harm job performance. Figure 5 illustrates such a case (i.e., case 1). Thus, we add the following constraint.

$$\frac{q_{i,t-1} - q_{it}}{q_{i,t-1}} \leqslant r \quad and \quad q_{it} = q_{i,t+\Delta t} \quad (i \in V;\, t = S_i, S_i+1, \cdots, F_i - 1 : q_{i,t-1} \neq q_{it};\, \Delta t = 1, 2, \cdots, l-1) \qquad (4)$$

where $r \in [0, 1]$ denotes the maximum rate of resource change per scheduling period and $l \in Z_{>0}$ denotes the minimum change interval. The first part in Eq. (4) avoids the sudden large reduction in resource availability. The second part is used for the continuity of constant resource quantities.

The formulation of the optimization problem for the DAG-aware task assignment is summarized as follows:

$$Minimize \quad max\{F_i : i \in V\} \qquad (5)$$

Subject to

$$Eqs.\ (1), (2), (3)\ and\ (4)$$
$$q_{it} \bmod d_i = 0 \quad (i \in V;\, t = 0, 1, \cdots, T)$$

where d_i is the resource demand of each task in stage i, and q_{it} mod $d_i = 0$ makes sure that the resources allocated to stage i in scheduling period t can be fully utilized by its tasks (referring to case 2 in Figure 5 for a counterexample).

2) Priority-based Approach: The formulated DAG-aware resource allocation is a generalization of the resource-constrained project scheduling problem [13] and hence belongs to the class of NP-hard problems. Existing algorithms cannot yield a near-optimal solution in a time acceptable to Spark framework, especially with the increase in the number of stages. In addition, since Spark is often deployed in a multi-tenant cluster, the resource capacity available to an application, i.e., variable RC in Eq. (3), often changes during runtime. It means that the optimization model needs to be resolved multiple times. To this end, we present a heuristic priority-based algorithm for efficient DAG-aware resource allocation.

The key idea of the heuristic priority-based approach is to use the constantly updated priority value of each stage to make task assignment decisions. We define the priority value pv_i of stage i as the currently unprocessed workload of stage i and all its successor stages, i.e.,

$$pv_i = w_i + \sum_{j \in SuccessorSet_i} w_j \qquad (6)$$

where $SuccessorSet_i$ denotes the set of stages that cannot be started before stage i is finished. Thus, the ready-to-execute stages with a higher priority value are more likely to be on the long-running chains of stages.

Algorithm 1 shows the pseudo-code of the priority-based DAG-aware task assignment. At each step, it selects the stage with the highest priority value among the stages that are ready to execute (lines 5 $\sim$ 6). For a given stage, if delay scheduling assigns tasks to an executor, the stage's priority value and the executor's available resource will be updated (lines 8 $\sim$ 11). Here, we consider the general case where a stage can have tasks with heterogeneous resource demands (denoted as d_i in

TABLE III
DAG-AWARE TASK ASSIGNMENT FOR THE EXAMPLE IN FIGURE 1.

Schedule	Stage 1		Stage 2		Free CPUs	Step
	w_1	pv_1	w_2	pv_2		
	48	52	36	**64**	16	
Stage 2	48	**52**	24	52	10	1
Stage 1	32	36	24	**52**	6	2
Stage 2	32	36	12	40	0	3
	32	36	12	**40**	12	
Stage 2	32	36	0	28	6	4

Algorithm 2 Locality Sensitivity-aware Delay Scheduling.

```
Input: $TaskSet_i$; $d_i$; $Executors$;
Output: ($task$, $exec$);
1: $allowedLocality = TaskSet_i.getAllowedLocalityLevel(curTime)$;
2: Sort stage $i$'s valid locality levels in $LQ$ in $ASC$ order;
3: for each $exec$ in $Executors$ do
4:     for each $locality$ in $LQ$ do
5:         if $TaskSet_i$ has a pending $task$ with $locality$ on $exec$ and $exec.freeCPUs \geq d_i$ then
6:             if $locality \leq allowedLocality$ or $task.finishTime < TaskSet_i.earliestCompletionTime$ then
7:                 return ($task$, $exec$);
8:             else
9:                 break the loop;
10:            end if
11:        end if
12:    end for
13: end for
```

line 10). By dynamically allocating executor resources to each stage according to the priority order, Dagon prioritizes the long-running chains of stages to overlap their executions.

Consider the example DAG in Figure 1. Table III illustrates some steps in Dagon DAG-aware task assignment process for the DAG. Initially, the ready-to-execute stages include stage 1 and stage 2. The priority value of stage 2 (pv_2) is $sum(w_2 = 36, w_3 = 24, w_4 = 4)$ that equals to 64, which is larger than that of stage 1, i.e., $sum(w_1 = 48, w_4 = 4)$ that equals to 52. As a result, Dagon selects stage 2 to run a task, then the workload of stage 2 w_2 becomes 24 and the priority value pv_2 becomes 52. For the next steps, Dagon always selects the stage with the higher priority value if there are available CPUs. When stage 2 has no unprocessed workload ($w_2 = 0$) at step 4, the ready-to-execute stages will be stage 1 and stage 3. The process continues until all stages are completed. Finally, we can observe that the task assignment obtained by the heuristic priority-based algorithm is the same as that in Figure 2(b).

B. Sensitivity-aware delay scheduling

The data locality problem occurs because delay scheduling lets an executor be idle when the executor is unable to launch a task with the required locality. We address it through a simple sensitivity-aware technique. The key idea of the approach is to use idle executors to launch tasks at a lower locality level than it is required. However, launching tasks with a lower locality can cause a significant delay in the duration of some stages that are sensitive to locality. Thus, we only assign low-locality tasks that do not affect the earliest completion time of the stage to the idle executors. We estimate the earliest completion time of stage i, ect_i, based on its number of pending tasks ptn_i, the current task parallelism tp_i and the average duration of tasks $\overline{td_i}$. It is represented as

$$ect_i = \lceil ptn_i / tp_i \rceil * \overline{td_i}. \quad (7)$$

Algorithm 2 shows the pseudo-code for the sensitivity-aware delay scheduling in Dagon. In line 1, the stage's maximum allowed locality level for launching tasks is calculated according to native delay scheduling, based on the current wait time. It then sorts the valid locality levels of the stage in ascending order. For each executor, it searches the stage's pending tasks in the executor in order of the locality level and selects one when task resource demand can be satisfied. If the locality of the selected task is not higher than the allowed locality or this task will finish earlier than the earliest completion time of the stage, it accepts the task. Otherwise, it skips the current executor and processes the next one. Note that the the finish time of a pending task (line 6) is estimated as the average duration of the finished tasks with the same locality level. Finally, it replaces the native delay scheduling in Algorithm 1 (line 7) with the sensitivity-aware delay scheduling.

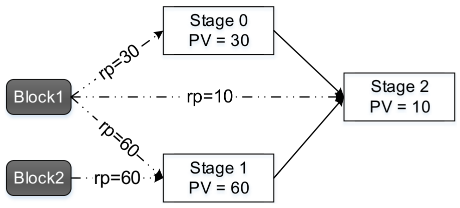


Fig. 6. Example of reference priority for two data blocks.

C. Priority-aware cache replacement

We present Least Reference Priority (LRP) caching to work with DAG-aware task assignment. It makes the cache eviction and prefetching decisions based on the stage priority value calculated from Eq. (6). The reference priority is defined as:
Definition 1 (Reference Priority). *For each data block, the reference priority is defined as the priority value of each stage that uses the data block.*

LRP tracks the reference priority of each data block. As a stage is completed, LRP deletes the corresponding reference priority for its data blocks, and uses the next highest one. Figure 6 shows an example of reference priority for two data blocks. Block 1 has three reference priorities 30, 10 and 60. If stage 1 is completed, the reference priority of 60 will be deleted for both block 1 and block 2, and thence the highest reference priority of block 1 becomes 30, and that of block 2 becomes zero. LRP always evicts the data block whose reference priority is the lowest, and prefetches the data block whose reference priority is the highest.

LRP has two desirable properties. First, it is suitable for use with the DAG-aware priority-based task assignment. Intuitively, the higher the priority value of a stage is, the larger is the reference priority of its needed data and thus more likely its

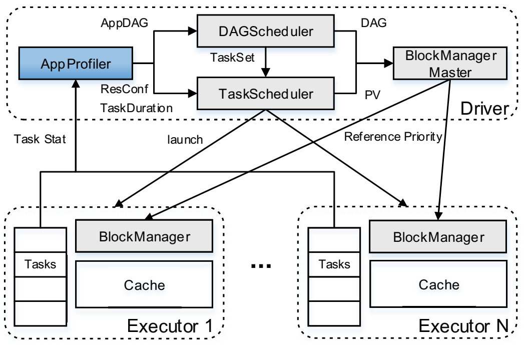


Fig. 7. Dagon architecture (in blue a new component; in gray a modification.)

needed data is cached when the stage is scheduled. Second, it makes efficient use of cache space. Dagon always schedules the stage with the highest priority value, which means that the data with a larger reference priority is the recently needed one, and those with a smaller reference priority will be used later in the future. Thus, according to the reference priority, LRP can proactively delete inactive data (i.e., with zero reference priority) and evict the furthest data to be used. Also, it prefetches the recently needed data when the available cache space exceeds a certain threshold.

IV. System Implementation

Figure 7 shows the overall architecture of Dagon. From the history log or online statistics, the new component named *AppProfiler* learns the application DAG and estimates the task duration and resource demand for each stage. When a user runs a workload for the first time, it submits the workload with a small dataset to obtain the profile and then re-submits it with the full dataset. The modifications for Spark to support the features of Dagon include *DAGScheduler*, *TaskScheduler*, *BlockManagerMaster*, and *BlockManager*.

Scheduling workflow. When a user deploys an application with spark-submit script, Spark launches the driver program on the master node. The driver creates a *SparkContext*, through which AppProfiler is instantiated. During the stage execution, AppProfiler periodically collects data (e.g., task resource usage and finish event) from executors, and passes re-estimated resource configuration and task duration to TaskScheduler. The driver also instantiates DAGScheduler to build a DAG of stages, TaskScheduler to schedule stages in the priority order, and BlockManagerMaster to update reference priority profile. For each stage, TaskScheduler finds tasks via sensitivity-aware delay scheduling, and launches them into executors using the estimated resource configuration. Spark driver starts its executors on the slave nodes, which leads to the distributed deployment of BlockManager across a cluster.

Eviction and prefetching workflow. BlockManagerMaster implements the main logic of LRP caching. It maintains and updates the reference priority profile based on DAG that is obtained from AppProfiler or built by DAGScheduler, and the stage priority value provided by TaskScheduler. Once the profile is updated, it sends the updated profile to BlockManager in the corresponding nodes. BlockManager evicts data blocks in two ways accordingly to the reference priority profile. First, from time to time, it proactively checks the data blocks in the cache and evicts those blocks with a reference priority of zero to free up the cache space. Second, when the free cache space is insufficient to accommodate a new data block, it selects the data block that has the smallest reference priority for eviction. BlockManager also keeps track of the reference priority of the evicted data. When the free cache space reaches a certain threshold, it prefetches the in-disk data block whose reference priority is the largest. Such a prefetch operation effectively overlaps the disk access time with computation time. After an eviction/prefetch event occurs, BlockManager reports the status of the data block to BlockManagerMaster.

Implementation details. We deployed Spark 2.2.0 equipped with Dagon on Hadoop Yarn 2.7.1, where an executor is implemented using Linux containers (LXC) based on cgroup. We added a new function `trackContainer()` to Yarn `LinuxContainerExecutor`. It reads the CPU usage and throttled time of tasks from `cgroup/cpuacct/cpuacct.usage` and `cgroup/cpu/cpu.stat`, respectively. These statistics are periodically communicated to *AppProfiler* and used to estimate the task resource demand for each stage. To prevent a long tail task due to high parallelism or low locality from prolonging the stage execution, we modified speculative execution. For a long tail task, it launches a speculative task to an executor that has free resource close to the input data.

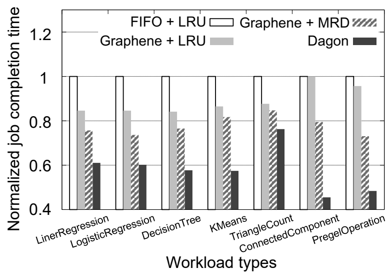
V. Evaluation

A. Testbed Setup

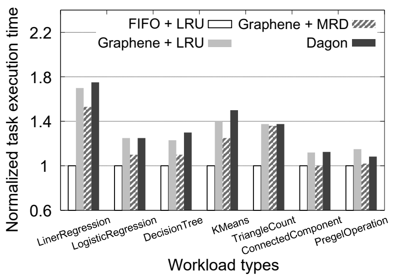
We evaluate Dagon on a physical cluster composed of 12 Sugon I620-G20 (16-core CPUs and 128 GB RAM) and 8 ThinkServer RD650 (20-core CPUs and 128 GB RAM). Each server has a 6TB hard disk, and is connected with 10 Gbps Ethernet. Dagon is implemented in Hadoop Yarn 2.7.1 and Spark 2.2.0. Two servers are configured as `ResourceManager` and `NameNode`. The rest 18 servers run as the slave nodes for HDFS storage and Spark job execution. Based on the previous study [15], we configure each Spark executor with 4 cores and 8GB memory.

For comparison, we implement a recently proposed dependency-aware scheduler Graphene [10], as well as reference-distance based cache management scheme M-RD [14]. Graphene reduces resource fragmentation and improves job completion time by first scheduling troublesome tasks and then scheduling the remaining tasks. MRD always evicts and prefetches data blocks for the furthest and nearest stages that will be scheduled by FIFO scheduler, respectively. Thus, there is an incoherency between the data access pattern and data caching when applying MRD caching under Graphene scheduler (denoted as Graphene + MRD).

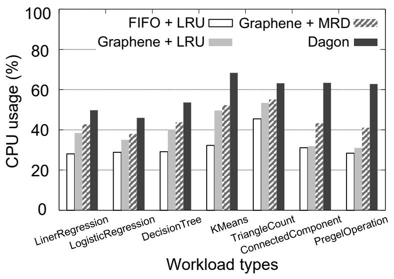
We use a set of representative workloads from SparkBench suite [12]. The workloads are grouped into three categories



(a) Job completion time.

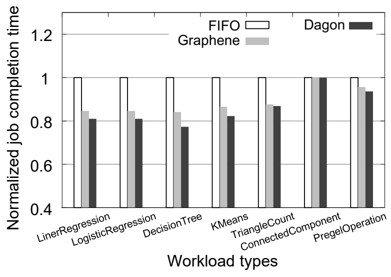


(b) Task execution time.

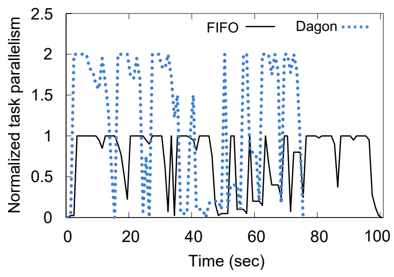


(c) CPU utilization.

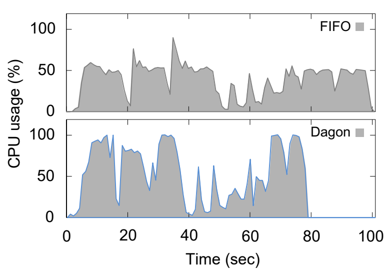
Fig. 8. Comparison of Dagon with representative approaches in the job completion time and cluster resource utilization.



(a) The job completion time.



(b) Task parallelism of *DecisionTree* workload.



(c) CPU utilization of *DecisionTree* workload.

Fig. 9. Impact of Dagon's priority-based task assignment on the job completion time, task parallelism, and CPU utilization.

based on their resource consumption characteristics. It includes three CPU-intensive (*LinearRegression*, *LogisticRegression* and *DecisionTree*), two mixed (*KMeans* and *TriangleCount*), and two I/O intensive (*ConnectedComponent* and *PregelOperation*) workloads. We compare results of Dagon to the Graphene + MRD approach using these workloads.

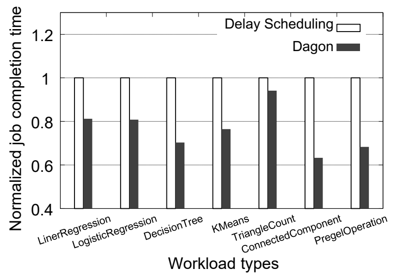
B. Effectiveness of Dagon

We firstly compare the job completion time and cluster resource utilization achieved by FIFO + LRU, Graphene + LRU, Graphene + MRD, and Dagon. We use the stock Spark (i.e., FIFO + LRU) as the baseline. Figure 8(a) shows that Graphene + LRU reduces the job completion time of CPU-intensive and mixed workloads by about 15% compared to FIFO + LRU. It is relatively less effective for two I/O intensive workloads (i.e., *ConnectedComponent* and *PregelOperation*) as Graphene's Spark implementation is a CPU-only scheduling algorithm. Since MRD can effectively mitigate the I/O bottleneck for I/O intensive workloads, we observe that Graphene + MRD outperforms Graphene + LRU by 23% for *ConnectedComponent* and *PregelOperation* in the job completion time. Graphene + MRD improves performance even more for other workloads. The results show that Dagon improves the average job completion time of all workloads by 42%, 31%, and 20% compared to the stock Spark, Graphene + LRU, and Graphene + MRD, respectively. For *ConnectedComponent* workload, Dagon improves the job completion time by 42% compared to Graphene + MRD. There are two reasons for the performance improvement. First, Dagon prioritizes the long-running chains of stages to reduce resource fragmentation while keeping the data needed by these stages in memory. Graphene + MRD lacks the coherency between its task scheduling and caching, which results in performance loss due to cache misses. Second, Dagon launches low-locality tasks to idle executors based on the data locality sensitivities of stages, and thus speeds up the overall job completion.

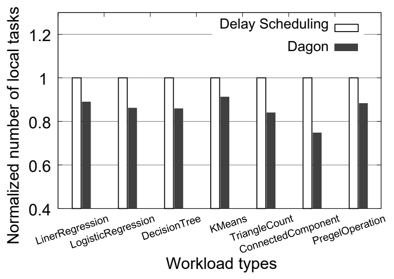
Figure 8(b) shows that both Dagon and Graphene (+ LRU/MRD) lead to longer task execution times at some stages because of DAG-aware task scheduling. However, this provides more opportunities for improving task parallelism and reducing resource fragmentation. Compared to Graphene + MRD, Dagon slightly increases the task execution time by about 10%. The reason is that Dagon launches low-locality tasks to idle executors, which may cause longer task execution time but also results in higher resource utilization. Figure 8(c) demonstrates Dagon increases the CPU utilization by 26%, 18%, and 13% compared with stock Spark, Graphene + LRU, and Graphene + MRD, respectively. Note that for I/O intensive workloads, e.g., *ConnectedComponent*, Dagon achieves 46% higher utilization than Graphene + MRD does. Frequent disk I/Os often cause low CPU utilization. Due to its effective caching, Dagon improves the CPU utilization significantly for I/O intensive workloads.

C. Benefit of DAG-aware task scheduling in Dagon

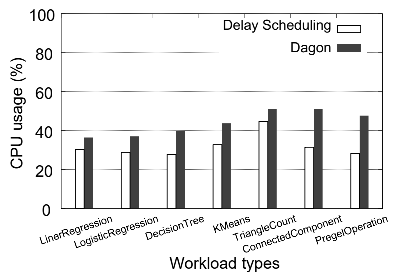
We investigate the impacts of Dagon priority-based task assignment and sensitivity-aware delay scheduling on the job completion time, task parallelism, and CPU utilization. We use Spark default task scheduler (FIFO) and delay scheduling for comparison. Caching is disabled in the experiments.



(a) Job completion time.



(b) Data locality.

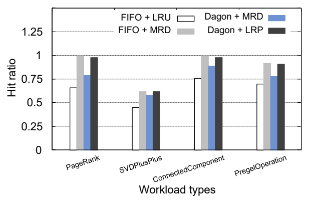


(c) CPU utilization.

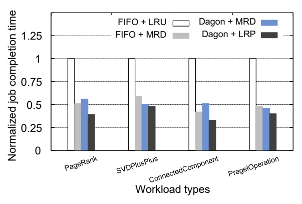
Fig. 10. Impact of sensitivity-aware delay scheduling on the job completion time, data locality and resource utilization.

Priority-based task assignment. Figure 9(a) compares the job completion time achieved by FIFO scheduler, Graphene, and priority-based task assignment of Dagon. It shows that Dagon slightly outperforms Graphene in the job completion time. Graphene assumes that all available resources can be used to launch tasks on time, but the assumption indeed does not hold because delay scheduling often makes some executors be idle. The results also reveal that the improvement in the job completion time also varies because of different workload characteristics. For CPU intensive workloads, i.e., *LinearRegression*, *LogisticRegression* and *DecisionTree*, Dagon outperforms FIFO by 19%, 19% and 23%, respectively. For mixed workloads, i.e., *KMeans* and *TriangleCount*, Dagon outperforms FIFO by 18% and 13%, respectively. However, Dagon is less effective for I/O intensive workloads, i.e., *ConnectedComponent* and *PregelOperation*. The cause is that Spark allows workloads to specify only their resource demands on CPU (number of cores) for a task, regardless of I/O demands and resource availability. Thus, while Dagon and Graphene help to reduce CPU fragmentation, for I/O intensive workloads they may overuse I/O resources and cause unexpected contention, canceling out the benefit. Figures 9(b) and 9(c) show the task parallelism and CPU utilization of *DecisionTree* are effectively improved when applying priority-based task assignment of Dagon. Dagon achieves about 20% improvement in the job completion time compared to FIFO.

Sensitivity-aware delay scheduling. Figure 10 compares the job completion time, data locality, and cluster resource utilization achieved by delay scheduling and sensitivity-aware delay scheduling of Dagon, respectively. Figure 10(a) shows that Dagon outperforms delay scheduling by 24% in the average job completion time. For different workloads, performance improvements are different due to the workload characteristics. In general, the higher the proportion of stages insensitive to data locality in a workload, the larger is the room for improvement by Dagon. Figure 10(b) shows that the number of high-locality tasks of stages insensitive to locality is reduced by 14% when applying sensitivity-aware delay scheduling of Dagon. This prevents some executors from wasting resources due to unnecessary waiting for high-locality tasks. As shown in Figure 10(c), Dagon increases the average CPU utilization by 12% compared to delay scheduling.



(a) The cache hit ratio.



(b) The job completion time.

Fig. 11. Comparison of caching policies under different schedulers.

D. Benefit of LRP cache policy

We conduct experiments to compare MRD [14] and LRP (of Dagon) caching under two scheduling modes, i.e., FIFO and Dagon. We use LRU + FIFO as the baseline. For a fair comparison, we use the same experimental environment and workloads as in a previous study [14]. Figure 11 compares the cache hit ratio and job completion time achieved by four different combinations. Figure 11(a) shows that although MRD outperforms LRU by 24% in the cache hit ratio under FIFO scheduler, it performs poorly with Dagon. The reason is that MRD is designed for FIFO scheduler, thus it does not accurately cache and prefetch the data mostly needed by Dagon during workload execution. On the other hand, LRP achieves 11% higher cache hit ratio than MRD does under Dagon. As a result, in Figure 11(b) we can see that Dagon + LRP outperforms Dagon + MRD by 18% in the job completion time for *ConnectedComponent* workload, but also it improves job performance for all workloads. We also observe that Dagon only has marginal performance improvement compared to FIFO when using MRD. This is due to the fact that all four workloads are I/O intensive and FIFO benefits more from MRD caching because of their coordination. In summary, we can conclude that the LRP cache policy is more effective for Dagon's DAG-aware task scheduling.

VI. Related Work

DAG-aware scheduler: CARBYNE [9] and A-scheduler [4], [5] aim to optimize inter-DAG job scheduling for DAG-structured computations. In CARBYNE, the short-term left-over resources of jobs are rescheduled to improve job performance and cluster utilization. A-scheduler is an adaptive approach for Spark Streaming that schedules jobs using different policies based on their dependencies and automatically adjusts job parallelism and resource shares based on workload characteristics. For intra-DAG task scheduling, heuristic methods [11], [15] consider the complex intra-job dependencies so as to speed up job completion but they assume homogeneous demands. GRAPHENE [10] builds task schedules offline by placing the troublesome tasks into a virtual resource-time space and then places the remaining task subsets. It focuses on multi-resource packing and data dependency, but not the coherency between its data access pattern and data caching.

Delay scheduling and data locality: Many prior studies have shown that enhancing task data locality is an effective approach for job performance improvement [2], [3]. Delay scheduling [20] significantly improves MapReduce performance by slightly relaxing job fairness to enhance task data locality. However, it is unaware of data locality sensitivities of DAG stages in Spark. Dawn [17] proposes a dependency-aware network-adaptive scheduler that aggregates and co-locates the data and tasks of dependent jobs to improve data locality and job performance. DelayStage [15] applies delay scheduling to optimize the launch time of parallel stages so as to avoid significant fluctuation of resource utilization. Dagon differs from these efforts in that it uses idle executors due to delay scheduling to launch tasks at a lower locality level, based on data locality sensitivities of different DAG stages.

Cache management: LRU is the de facto caching policy used in Spark and Tez. However, it is oblivious to the data access pattern in DAG jobs. MemTune [18] adjusts memory partitions for task execution and data storage at runtime in Spark. It evicts and prefetches data blocks according to local dependencies on runnable tasks. LRC [19] keeps track of the count of references to each data block and evicts the cached data blocks with the smallest reference count. MRD [14] examines data access pattern caused by FIFO scheduler and evicts the furthest data to be used while prefetching the nearest ones that will be needed. But none of the existing policies consider the data access pattern caused by DAG-aware task scheduling, and thus often make erroneous cache decisions.

VII. Conclusions

This works aim to leverage the complexity and heterogeneity of DAGs to jointly execute task scheduling and caching. It proposes and develops Dagon, a middleware for joint DAG-aware task scheduling and cache management. Dagon consists of three key mechanisms, 1) DAG-aware task assignment that considers dependency structure and heterogeneous resource demands to reduce resource fragmentation, 2) sensitivity-aware delay scheduling that prevents executors from wasting resources due to unnecessary waiting for high-locality tasks, and 3) priority-aware cache management that makes the cache eviction and prefetching decisions based on the stage priority determined by DAG-aware task assignment. Dagon is implemented in Spark. Experiments with benchmark applications show that Dagon significantly improves job performance and cluster utilization, compared to the prevalent approaches.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1704102, the National Natural Science Foundation of China under Grant No. 61672381, and the Fundamental Research Funds for the Central Universities under Grant 22120180508. The corresponding author is Prof. Zhijun Ding.

References

[1] Apache Tez. http://tez.apache.org/.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters. In *Proc. of USENIX ATC*, 2014.

[3] X. Bu, J. Rao, and C.-Z. Xu. Interference and locality-aware task scheduling for MapReduce applications in virtual clusters. In *Proc. of ACM HPDC*, 2013.

[4] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milljicic. Adaptive scheduling of parallel jobs in Spark streaming. In *Proc. of IEEE INFOCOM*, 2017.

[5] D. Cheng, X. Zhou, Y. Wang, and C. Jiang. Adaptive scheduling parallel jobs with dynamic batching in Spark streaming. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2672-2685, 2018.

[6] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, 2015.

[7] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proc. of USENIX NSDI*, 2011.

[8] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[9] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proc. of USENIX OSDI*, 2016.

[10] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. of USENIX OSDI*, 2016.

[11] Z. Hu, D. Li, Y. Zhang, D. Guo, and Z. Li. Branch scheduling: DAG-aware scheduling for speeding up data-parallel jobs. In *Proc. of ACM IWQoS*, 2019.

[12] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform Spark. In *Proc. of ACM CF*, 2015.

[13] A. Naber and R. Kolisch. Mip models for resource-constrained project scheduling with flexible resource profiles. *European Journal of Operational Research*, 239(2):335–348, 2014.

[14] T. B. Perez, X. Zhou, and D. Cheng. Reference-distance eviction and prefetching for cache management in spark. In *Proc. of ICPP*, 2018.

[15] W. Shao, F. Xu, L. Chen, H. Zheng, and F. Liu. Stage delay scheduling: Speeding up DAG-style data analytics jobs with resource interleaving. In *Proc. of ICPP*, 2019.

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, et al. Apache Hadoop Yarn: Yet another resource negotiator. In *Proc. of ACM SoCC*, 2013.

[17] S. Wang, W. Chen, X. Zhou, L. Zhang, and Y. Wang. Dependency-aware network adaptive scheduling of data-intensive parallel jobs. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):515–529, 2019.

[18] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *Proc. of IEEE IPDPS*, 2016.

[19] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief. LRC: Dependency-aware cache management for data analytics clusters. In *Proc. of IEEE INFOCOM*, 2017.

[20] 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 *Proc. of ACM EuroSys*, 2010.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of USENIX NSDI*, 2012.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of USENIX HotCloud*, 2010.