WOHA: Deadline-Aware Map-Reduce Workflow Scheduling Framework over Hadoop Clusters

Shen Li *, Shaohan Hu *, Shiguang Wang *, Lu Su †, Tarek Abdelzaher *, Indranil Gupta*, Richard Pace ‡
* University of Illinois at Urbana-Champaign † State University of New York at Buffalo ‡ Yahoo! Inc.
{shenli3, shu17, swang83}@illinois.edu, lusu@buffalo.edu, {zaher, indy}@illinois.edu, rpace@yahoo-inc.com

***Abstract*—In this paper, we present WOHA, an efficient scheduling framework for deadline-aware Map-Reduce workflows. In data centers, complex backend data analysis often utilizes a workflow that contains tens or even hundreds of interdependent Map-Reduce jobs. Meeting deadlines of these workflows is usually of crucial importance to businesses (for example, workflows tightly linked to time-sensitive advertisement placement optimizations can directly affect revenue). Popular Map-Reduce implementations, such as Hadoop, deal with independent Map-Reduce jobs rather than workflows of jobs. In order to simplify the process of submitting workflows, solutions like Oozie emerge, which take a workflow configuration file as input and automatically submit its Hadoop jobs at the right time. The information separation that Hadoop only handles resource allocation and Oozie workflow topology, although preventing the Hadoop master node from getting involved with complex workflow analysis, may unnecessarily lengthen the workflow spans and thus cause more deadline misses. To address this problem and at the same time honor the efficiency of Hadoop master node, WOHA allows client nodes to locally generate scheduling plans which are later used as resource allocation hints by the master node. Under this framework design, we propose a novel scheduling algorithm that improves deadline satisfaction ratio by dynamically assigning priorities among workflows based on their progresses. We implement WOHA by extending Hadoop-1.2.1. Our experiments over an 80-server cluster show that WOHA manages to increase the deadline satisfaction ratio by 10% compared to state-of-the-art solutions, and scales up to tens of thousands of concurrently running workflows.**

***Index Terms*—MapReduce; Hadoop; Workflow; Scheduling; Deadline**

I. INTRODUCTION

This paper presents WOHA (WOrkflow over HAdoop), a holistic solution for deadline-aware workflow scheduling over Hadoop clusters. Hadoop [1] is a popular open source implementation of the Map-Reduce [2] framework that has attracted a great amount of interest from both industry and academia for the past few years. Researchers have already studied various Hadoop performance objectives including fairness [3, 4], throughput [5], fault tolerance [6], and energy efficiency [7, 8]. There have been studies on designing schedulers to meet the deadlines for Hadoop *jobs* [9, 10], but little effort has been spent on satisfying deadlines attached to Hadoop *workflows*. To the best of our knowledge, WOHA is the first system that addresses the problem of deadline-aware workflow scheduling over a Hadoop cluster.

Meeting deadlines of Hadoop workflows is an important problem in industry Hadoop clusters. Back-end big data analysis often involves complex workflows of multiple Hadoop jobs [11]. One example is generating statistics of user log information which will be later used for advertisement placement optimizations, personalized content recommendations, or user graph storage partitions. The site performance and the revenue for the company are directly affected by whether or not workflows can finish within a given amount of time.

Existing solutions for Hadoop workflow scheduling [12–14] rely on a standalone software (*e.g.* Oozie [12]) that resolves the workflow topology and submits each job to the Hadoop cluster when its input data is ready and its recurrence frequency is met. There is a clean separation between Oozie and Hadoop, in that Oozie only manages workflows and submits jobs to Hadoop, whereas Hadoop only manages resource allocation between jobs and executes them. This functionality decomposition makes life easier for programmers at development time, but could also create potential inefficiency that affects system performance at run time: The fact that only Oozie has workflow topology information and only Hadoop controls resource allocation makes it difficult for either side to employ any optimization for meeting workflow deadlines.

Our objective is to improve workflow deadline satisfaction ratios over Hadoop clusters. Deadline-aware workflow scheduling itself is an NP-hard problem [15]. Besides the difficulties in approaching the scheduling problem, the Hadoop architecture introduces two more challenges:

- The master node (scheduler) only bears simple scheduling algorithms. In Hadoop, single master node monitors the entire cluster, and is responsible for scheduling all tasks. Since it is the only node with the global view, the master node is naturally the best place to analyze and schedule workflows. However, as the single point of control, the master node receives lots of task assignment requests per second [16]. Each request needs to be answered within a very small amount of time, otherwise the master node will become a bottleneck for the Hadoop cluster. With these constraints, the master node is not able to carry out any complex runtime workflow analysis.

- The master node leaves little storage space for workflow related profiles. Monitoring and scheduling responsibilities already consume a considerable amount of memory on the master node. Consequently, one cannot compute comprehensive scheduling guidelines ahead of time and store it on the master node.

In order to tackle the above challenges, this paper introduces WOHA, a scalable framework with the emphasis on meeting workflow deadlines. WOHA offloads complex workflow analysis to clients[1] during workflow submissions, and shifts costly job processing to slave nodes upon job initialization. More specifically, with the workflow configuration file in hand, a client node queries for minimal cluster status information from the master node. Then, the client generates a scheduling plan locally, which the master node can use to prioritize jobs among multiple concurrent workflows. WOHA is implemented by adding 9,292 lines of code into Hadoop-1.2.1. Evaluation results confirm that WOHA outperforms the state-of-the-art solutions, and adds negligible overhead to the master node.

The remainder of this paper is organized as follows. Section II briefly introduces the workflow model we consider. Then we show WOHA design details in Section III. Section IV presents the progress based scheduling solution under WOHA framework. Implementation and evaluation are presented in Section V and Section VI, respectively. We survey the related work in Section VII. Section VIII concludes the paper.

II. Workflow Model

We consider a shared Hadoop cluster serving multiple user-defined Map-Reduce workflows. Workflows $\mathcal{W} = \{W_1, W_2, ..., W_l\}$ compete for slots on the Hadoop cluster. A workflow W_i is submitted at time S_i with deadline D_i, both are *not* known to the scheduler ahead of time. W_i contains a set of n_i jobs $\mathcal{J}_i = \{J_i^1, J_i^2, ..., J_i^{n_i}\}$, which we call *wjob*s. Job J_i^j contains m_i^j mappers and r_i^j reducers. Each job J_i^j has a prerequisite wjob set $\mathcal{P}_i^j \subset \mathcal{J}_i$. If $J_i^k \in \mathcal{P}_i^j$, J_i^j cannot start before J_i^k finishes, and J_i^k is called a predecessor of J_i^j. Define $\mathcal{P}_i$ as set $\{\mathcal{P}_i^1, \mathcal{P}_i^2, ..., \mathcal{P}_i^{n_i}\}$. With the above formulation, each workflow is characterized as $W_i = \{\mathcal{J}_i, \mathcal{P}_i, S_i, D_i\}$.

Existing Hadoop cluster usually consists of at least three components: client nodes, the master node, and worker nodes. Client nodes are portals that allow users to submit and monitor jobs without acquiring accesses of the master node and worker nodes, which protects the master and workers from mal-behaving users. The master node schedules jobs/tasks and monitors their status, and workers execute tasks. All three components are configured by the cluster administrator (not by users), and run a consistent set of Hadoop libraries. Users may only submit workflows through client nodes. The Hadoop cluster (clients, master, and workers) analyzes and schedules workflows without users involvement.

[1]A client executes Hadoop libraries to submit Hadoop jobs. We assume clients run on different machines from both the master node and slave nodes.

III. Workflow Scheduling Framework

This section presents the system design of WOHA. First, we identify potential expensive operations incurred by handling workflows, and describe solutions to mitigate the overhead. Then, a high level WOHA architecture is presented.

A. Design Philosophy

In Hadoop clusters, scheduling and monitoring already incur heavy overhead on the master node, and the master node may become the bottleneck if the the cluster becomes too large. Given this consideration, several design decisions have to be made to alleviate the overhead on the master node when introducing workflow scheduling mechanisms to the Hadoop architecture.

There are two expensive operations at the time of starting a Hadoop job: loading user-defined jar files, and initializing tasks. As a number of wjobs suddenly become available when a workflow is submitted, we must make sure that this process does not overload the master node. In the current Hadoop implementation, the client machine loads jar files locally, and hence adds no overhead to the master node. However, this approach no longer works for submitting wjobs. The reason is that some wjobs may depend on others' outputs, and hence are not ready at workflow submission time. Blocking the client session to wait for job activation signals from the master node is an undesired solution from the perspective of both user experience and system robustness. Fortunately, there are plenty of slave nodes in industry data centers. WOHA wraps the two expensive operations mentioned above into map tasks, and creates a map-only submitter job for each workflow. Each map task of the submitter is responsible for submitting one specific wjob. WOHA scheduler invokes submitter's map tasks in an on-demand manner (*i.e.*, a map task for J_i^j will not be invoked until all jobs in $\mathcal{P}_i^j$ have finished). In this way, WOHA shifts the overhead of loading jar files to slave nodes, and scatters the cost of initializing tasks over time.

The above design removes the bursty cost during workflow submissions. Besides that, processing workflow may also induce a huge amount of computational and storage resources. It is well known that the general workflow scheduling problem is NP-hard [15]. However, this does not mean WOHA needs to restrict users to use simple scheduling heuristics. Users should be free to choose algorithms as complex as they need. This creates a conflict. On one hand, in order to maintain scalability, the scheduling algorithm on the master node has to be fast. On the other hand, to

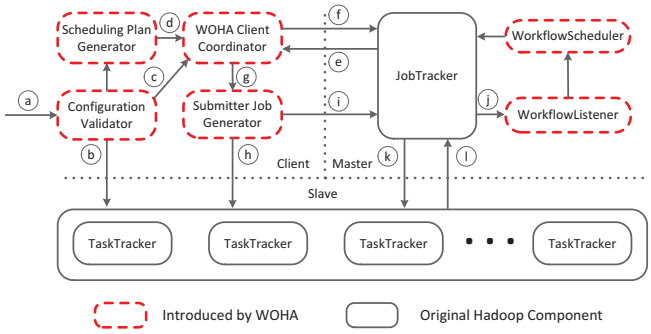

Fig. 1. WOHA Framework

preserve users' freedom in designing schedulers, they need to be allowed to use complex workflow analysis algorithms. To address this conflict, WOHA separates workflow analysis from the master node, and shifts it into the client node during workflow submission time. After the analysis, the client node generates and sends a workflow scheduling plan to the master node together with workflow configurations. The scheduler on the master node then follows the scheduling plan to prioritize jobs and workflows. A scheduling plan can be as simple as assigning a priority to the workflow, or as comprehensive as a total ordering of all tasks of all jobs inside the workflow along with the triggering signal to schedule it. Under this approach, users enjoy plenty of room to create intelligent schedulers. At the same time, the overhead of analyzing workflows is eliminated from the master node.

B. WOHA Architecture

WOHA enhances the existing Hadoop framework with workflow scheduling capabilities. In order to submit workflow W_i, users need to prepare an XML configuration file specifying input and output dataset paths, wjob jar file paths, wjob main class names, deadline, etc. WOHA takes this XML file, checks the existence and validity of jar files and input datasets, constructs prerequisite set $\mathcal{P}_i$ based on inputs and outputs of each wjob, submits wjobs on demand, and schedule wjobs to meet the deadline D_i in a best-effort manner. Fig. 1 shows the high-level system design of WOHA.

As illustrated in Fig. 1, WOHA takes the following steps to submit a workflow: (a) The user submits W_i's configuration XML file on WOHA client by executing *"hadoop dag [/path/to/W_i.xml]"*. (b) The Configuration Validator checks specified jar files and input datasets, and then copies them to HDFS if necessary. (c) The Configuration Validator passes W_i's configurations to the Scheduling Plan Generator and the Coordinator respectively. (d) The Scheduling Plan generator generates a scheduling plan for W_i locally based on $\mathcal{J}_i$, $\mathcal{P}_i$, and D_i (discussed in detail in Section IV). (e-f) The coordinator submits W_i's configuration to the JobTracker, and gets a unique workflow ID in return. (g) The Coordinator passes the workflow ID to the Submitter Job Generator. (h) The Submitter Job Generator creates a map-only Hadoop job (the submitter), and writes its input splits into HDFS. Each of the submitter's map task will submit one wjob in $\mathcal{J}_i$. (i) JobTracker receives the submitter job from the client side, and puts it under the internal data structure of W_i. (j) When some slots become available, the JobTracker will get that submitter job from the WorkflowScheduler if it is the time to schedule it. (k) The submitter job runs on slave nodes to prepare inputs for wjobs. (l) The submitter job submits wjobs to the JobTracker.

Same as Hadoop, the scheduling events in WOHA are triggered by heartbeat messages with slot idling information. JobTracker consults Workflow Scheduler for tasks to assign. The Workflow Scheduler first picks the workflow with the highest priority (Section IV explains how workflow priorities are assigned dynamically in detail), and then chooses the active job with the highest job priority within the workflow according to the scheduling plan. The job completion signal is also captured by WOHA, which is in turn used to activate its dependent jobs.

Users may replace the Scheduling Plan Generator module and the Workflow Scheduler module in WOHA with their own design and implementation according to their needs. When users have implemented those two modules, the substitution is as easy as modifying two lines of code in the workflow-scheduler.xml configuration file.

As we are the first to deal with deadline-aware Hadoop workflow scheduling, no existing solution directly fits into WOHA framework. To show benefits of the WOHA, we design and implement a novel progress based scheduling algorithm that covers the Scheduling Plan Generator and the Workflow Scheduler. This is also the default implementation of those two components in the current WOHA release.

IV. Progress Based Scheduling

WOHA architecture does not restrict how the Scheduling Plan Generator and Workflow Scheduler interact with each other to schedule workflows. Users have plenty of room to show their creativity. We present one novel solution that encloses job ordering and workflow progress requirement list ($\mathcal{F}_i$) in the scheduling plan to assist the Workflow Scheduler to schedule W_i at runtime. Intuitively, the progress requirement states that by ttd (time to deadline) time before the workflow's deadline, $\mathcal{F}_i(ttd)$ tasks from this workflow have to be scheduled. During scheduling, the Workflow Scheduler needs to prioritize both inter-workflow and intra-workflow jobs. We borrow existing solutions to intra-workflow cases (*e.g.*, Longest Path First, or Highest Level First), and focus on the inter-workflow prioritization.

According to the scheduling plan, the scheduler picks the workflow that falls furthest from its progress requirement given the current ttd. Then, it chooses the highest-priority

job within that workflow to assign tasks. Scheduling proceeds in a work-conserving way. The intuition behind the progress based scheduling is as follows. Workflows call for different amount of resources during their runtime. For example, in order to unlock a long sequence of jobs, it may require large numbers of slots to finish some preceding jobs as soon as possible. After that, it needs little resource, as all remaining jobs may all contain a small number of tasks. That is to say, priorities of workflows need to be dynamically tuned. However, the master node does not have enough computational resource to analyze all workflows comprehensively to get a thorough understanding at runtime. Fortunately, the client node may generate some scheduling plans which the master node may efficiently follow.

Section IV-A elaborates details of the progress based Scheduling Plan Generator. Section IV-B presents an efficient scheduling algorithm for the Workflow Scheduler to carry out the scheduling plan.

A. Scheduling Plan Generator

We assume that each map task of J_i^j takes M_i^j time to finish, and each reduce task of J_i^j takes R_i^j time to finish. Estimations of task execution times can be acquired from logs of historical executions [17] or by using models based on task properties [9]. Developing accurate estimation algorithms is not the focus of this paper.

When receiving W_i's configuration XML file, the client consults the JobTracker about the maximum number of slots in the system (n). Then, with those two pieces of information, the client generates the scheduling plan locally by simulating the execution of W_i. The detailed algorithm is shown in Algorithm 1. Assuming W_i monopolizes the entire cluster (an improved version will be discussed later), the procedure GENERATEREQS makes scheduling decisions according to given job priorities and simulates job executions based on $\mathcal{J}_i$, and $\mathcal{P}_i$. The event queue E stores all pending system events following the ascending order of their occurrence time. Each event e in queue E consists of three fields: *time*, *type*, and *value*. The field $time$ indicates the absolute time event e fires. There are only two events in the simulation: *FREE* event indicates some slots are made available, and *ADD* event puts one or more jobs into the active job queue A where jobs are ordered according to their given priorities. The $value$ field encapsulates remaining parameters based on the type of the event. To simplify the presentation, we define $\mathcal{D}_i^j$ as the dependent wjob set of J_i^j, such that $J_i^k \in \mathcal{D}_i^j$ if and only if $J_i^j \in \mathcal{P}_i^k$. Let $\mathcal{D}_i$ denote $\{D_i^1, D_i^2, ..., D_i^{n_i}\}$.

Algorithm 1 takes workflow configurations W_i and resource cap n as input. It assumes jobs in $\mathcal{J}_i$ are already prioritized based on the user defined intra-workflow job prioritization algorithm. Line 6-36 is the main loop, simulating the execution of W_i on n slots. Events are processed in Line 9-13. Line 15-34 compute the number of mappers/reducers to be scheduled. Job J_i^j will be removed from the active queue A if all its m_i^j map tasks or r_i^j reduce tasks have been scheduled (Line 20-23). An *ADD* event of the same job J_i^j is inserted into $\mathcal{E}$ (Line 21) with m_i^j set to 0. Otherwise, if all r_i^j reduce tasks have been scheduled, an *ADD* event is inserted into E for J_i^k, such that $J_i^k \in \mathcal{D}_i^j$ and $|\mathcal{P}_i^k| = 0$. (Line 29-31). The last three lines convert the event occurrence time into the time-to-deadline (ttd). That is to say, the entry s in $\mathcal{F}_i$ indicates that at least $s.req$ tasks have to be scheduled by $s.ttd$ before its deadline D_i in order to meet D_i. In the worst case, the number of iterations in the loop on Line 6 is of the same order of the total number of tasks in the workflow W_i (*i.e.* $\sum_{j=1}^{n_i}\left(m_i^j + r_i^j\right)$). Since GENERATEREQS executes locally on the client side, the computational complexity is not a serious concern. Please note, due to data locality, resource contention from other workflows, error in execution time prediction, and many other factors, the progress requirement $\mathcal{F}_i$ may not faithfully represent the real execution trace of the W_i on the Hadoop cluster. The progress requirement is just a rough estimation

Algorithm 1 Generate Progress Requirements

Input: Workflow configuration W_i, amount of slots n.
Output: Scheduling Plan $\mathcal{F}_i$.

```
1: procedure GENERATEREQS(W_i, n)
2:     F_i ← ∅; E ← ∅                          /* Progress Requirements and Event queue */
3:     A ← initially active jobs in W                                  /* Active jobs */
4:     E ← E ∪ {(time : 0, type :FREE, value : n)}
5:     n ← 0; t ← 0
6:     while |E| > 0 do                        /* there is at least one pending event */
7:         e ← pick the event with earliest time from E
8:         E ← E \ {e}; t ← e.time
9:         if e.type = FREE then                                   /* process the event */
10:            n ← n + e.value
11:        else
12:            A ← A ∪ {e.value}
13:        end if
14:        if n > 0 then                       /* there are some available slots */
15:            J_i^j ← pick the job with highest priority from A
16:            if m_i^j > 0 then                                   /* in the map phase */
17:                maps ← min(m_i^j, n);
18:                F_i ← F ∪ {(ttd : t, req : maps)}
19:                n ← n − maps; m_i^j ← m_i^j − maps
20:                if m_i^j ≤ 0 then
21:                    E ← E ∪ {(time:t+M_i^j,type:ADD,value:J_i^j)}
22:                    A ← A \ {J_i^j}
23:                end if
24:            else                                                /* in the reduce phase */
25:                reduces ← min(r_i^j, n)
26:                F_i ← F_i ∪ {(ttd : t, req : reduces)}
27:                n ← n − reduces; r_i^j ← r_i^j − reduces
28:                if J.r ≤ 0 then
29:                    delete J_i^j from all P_i^k, where J_i^j ∈ P_i^k
30:                    D' ← {J_i^k | J_i^k ∈ D_i^j ∧ |D_i^k| = 0}
31:                    E ← E ∪ {(time:t+R_i^j, type:ADD, value:D')}
32:                    A ← A \ {J_i^j}
33:                end if
34:            end if
35:        end if
36:    end while
37:    for ∀s ∈ F_i do                         /* translate event occurrence time to ttd */
38:        s.ttd ← t − s.ttd
39:    end for
40:    return F_i
41: end procedure
```

of the workflow progress trace that helps the JobTracker to prioritize workflows.

The above algorithm translates the configuration of W_i into a deterministic progress requirement $\mathcal{F}_i$. Besides the workflow configuration W_i, the second parameter n (the resource consumption cap) also shapes $\mathcal{F}_i$. Without a meaningful resource cap, $\mathcal{F}_i$ may act too optimistically in the beginning and demand a huge amount of resources near the deadline D_i. Fig. 2 shows an example. The workflow topology is shown on the left. Three workflows (W_1, W_2, W_3) with the same topology are submitted at time 0. Their deadlines are $D_1 = 9$, $D_2 = 9$, and $D_3 = 50$ respectively. The cluster contains 3 map slots and 3 reduce slots. In Fig. 2 (a) where the resource cap is set to 6, each of W_1 and W_2 thinks itself as having exclusive access to the entire cluster, and thus it is able to meet the deadline by starting at time 6. Hence, all three workflows require 0 tasks to be scheduled in the first 5 time units, and each of them gets a fair share of cluster resources. Entering the 6th unit time, W_1 and W_3 start to enjoy higher prioritization over workflow 2. However, it is already too late. No matter how the scheduler behaves from then on, at least one of them will miss its deadline.

An improvement

The above example demonstrates that, without the knowledge of other workflows, the Scheduling Plan Generator might underestimate resource contentions in the cluster, and hence miss critical execution opportunities. Unfortunately, it will incur heavy overhead on the master node to poll configurations and statuses of all other workflows when generating each scheduling plan. Moreover, often times, even the master node has no idea about what kind of workflows or jobs may come in the future. That is to say, the knowledge of other workflows is not available for the Scheduling Plan Generator when calculating the progress requirement $\mathcal{F}_i$ for W_i.

An intuitive solution is to apply a smaller resource cap when generating the scheduling plan, which enforces the generator to be less optimistic. The system has to carefully choose the value for the resource cap. Large values cannot prevent underestimation of the resource contentions, whereas small values lead to pessimistic behavior such that it may put W_i far behind its progress requirement at the submission time R_i. We propose to perform binary search using GENERATEREQS as a subroutine to find the minimum possible resource cap that does not violate its deadline D_i. As shown in Fig. 2 (b), if we enforce a resource cap of 2 on them, all three workflows meet their deadlines.

B. Efficient Scheduling Algorithm

To carry out scheduling plans, the Workflow Scheduler keeps track of the number of tasks that have been scheduled, which we call the *true progress* ρ_i. The scheduler then chooses the workflow with the most lags according to its progress requirement (*i.e.*, the largest $\mathcal{F}_i(ttd) - \rho_i$). The intuition is straightforward: try to keep every workflow following its progress requirement.

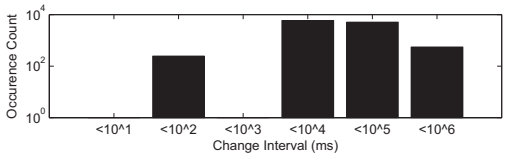

Fig. 3. Progress Requirement Change interval

However, the above scheduling algorithm brings another challenge. As the value ttd continuously changes, the progress requirement $\mathcal{F}_i(ttd)$ may also change. In order to get the correct ordering of all workflows, a naive solution might update all workflows' progress lags $(\mathcal{F}_i(ttd) - \rho_i)$ and then reorder them. It is not efficient enough to scale in large Hadoop clusters. In industry data centers, a huge number of jobs and workflows queue up in the Hadoop cluster, with a considerable amount of slots freeing up every second[16]. For example, Yahoo! Hadoop cluster spans over 42,000 nodes [18]. According to Yahoo! WebScope data [19], the master node may see a few thousand submissions within a minute during peak time. The algorithm needs to be invoked every time a heartbeat message mentions some slots are available, and needs to be applied to a large set of running jobs/workflows. The naive solution would thus take too much from the master node's scarce computational resources.

In order to speed up the workflow reordering algorithm, let us first identify opportunities that we can make use of. Although there will be one slot free-up every few milliseconds on average [16], the progress requirement does not change in this time scale. For example, Fig. 3 shows the intervals between two consecutive progress requirement changes. The scheduling plan is calculated based on Yahoo! data [19] using resource capped Highest Level First intra-workflow job prioritization algorithm. From the figure, we can see that all intervals incurred from this data set are larger than 10 milliseconds. More than 99% intervals are even larger than 10 seconds. That is to say, most of the time $\mathcal{F}_i(ttd)$ and $\mathcal{F}_i(ttd-\epsilon)$ will be the same value, where ϵ is the time difference between these two slot free-up events. The progress requirements list provides the information of the remaining time until next progress requirement change (ct). Hence, the scheduler can keep an ordered list of workflows based on ct, and only walks through the first few elements with ct smaller than ϵ.

After determining the right order of workflows, the scheduler picks the one with the highest priority (say W_i), schedules its tasks, calculates its new *true progress* ρ_i, updates its priority, and inserts it back into the workflow ordered list. Based on the semantics of this operation, only

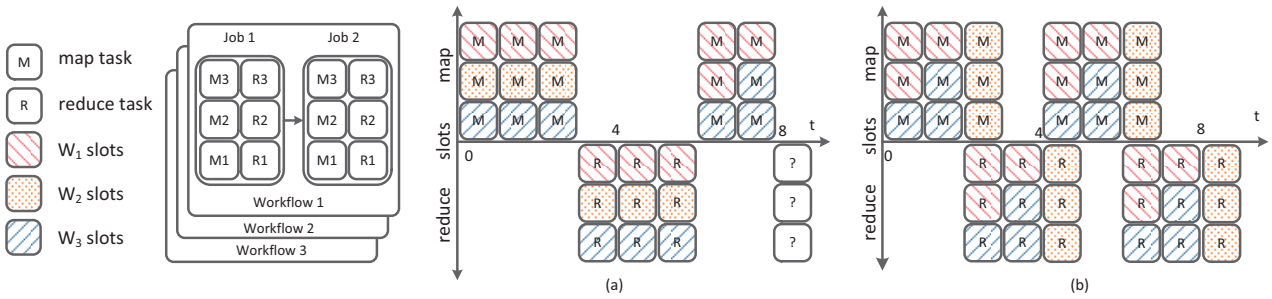

Fig. 2. Benefits of Resource Capped Scheduling Plan

the workflow with the highest priority needs to be deleted, and inserted again. Therefore, the scheduler calls for one efficient data structure to store the ct list, and another one to store the workflow priority list. Requirements on the two data structures are the same: fast deletion and insertion. Balanced Search Tree (BST) is a valid solution, but is not the best because BST does not make use of the fact that more than half of the deletions only happen to the head element of the list. Below, we present the *Double Skip List* (DSL) algorithm, that supports $\mathcal{O}(1)$ head deletions, $\mathcal{O}(\log n)$ arbitrary insertions, and $\mathcal{O}(\log n)$ arbitrary deletions.

The Double Skip List algorithm is inspired by the Skip List algorithm [20, 21]. It uses two correlated deterministic skip lists to serve the workflow ordering and updating purposes. Deterministic skip list is a data structure that allows worst-case $\mathcal{O}(\log n)$ searches, $\mathcal{O}(\log n)$ insertions, and $\mathcal{O}(\log n)$ deletions on an ordered sequence of elements. Please refer to the original paper [21] for more algorithm details.

Fig. 4 illustrates an example of the Double Skip List algorithm. Assume there are 8 workflows queuing in the scheduler. We specifically care about two properties of these workflows: 1) the time of the next progress requirement increase event (Next Event Time), and 2) their current priority value. The graph on the right side shows how those workflows are arranged into ct list and priority list based on their property values. When a slot free-up event occurs, the scheduler first iterates from the head of the ct list until it encounters the workflow whose next progress requirement change event has not fired yet. For each workflow visited during the iteration, the algorithm calculates its new progress requirement, and updates its position in the priority list accordingly. For example, suppose one slot frees up at time 3 with the DSL status shown in Fig. 4. Then, the algorithm behaves as follows: (a) The algorithm first accesses the head node of the ct list. (b) Following the link, it arrives at the node representing the same workflow in the priority list, and updates its priority according to the scheduling plan (say it is updated to 0). (c-g) Find the right location to insert the workflow with its new priority. Assume the workflow's next ct is 10. Steps (h-k) insert it into the right position in the ct list.

Algorithm 2 presents the detailed pseudo code. For workflow W_h, $W_h.t$ denotes the absolute time of next progress requirement change event, $W_h.i$ denotes the corresponding index of that event in the progress requirement, $W_h.p$ is its inter-workflow priority, and $\mathcal{F}_h$ represents its progress requirement. For progress requirement $\mathcal{F}_h$, $\mathcal{F}_h[i].ttd$ is the time to deadline of its i's element, and $\mathcal{F}_h[i].req$ the corresponding progress requirement. Line 4-19 iterates over the ct list. If the current head workflow in ct has fired one or more progress requirement change events before the current time t, its position in both ct list and priority list will be updated accordingly in line 6-15.

Algorithm 2 Double Skip List

Input: The current double skip list $\mathcal{L}$, current time t.
Output: The task to be scheduled.

```
1: procedure AssignTask(L, t)
2:     L^ct ← the ct list of L
3:     L^pri ← the priority list of L
4:     while True do                                   /* update ordering of workflows */
5:         pick the head workflow W_h from L^ct.
6:         if W_h.t ≤ t then                           /* W_h's progress requirement changes */
7:             L^ttd ← L^ttd \ {W_h}
8:             while D_h - F_h[W_h.i].ttd ≤ t do       /* go to the latest change */
9:                 W_h.i ← W_h.i + 1
10:            end while
11:            W_h.t ← D_h - F_h[W_h.i].ttd             /* next change time */
12:            L^ct ← L^ct ∪ {W_h}                      /* insert back to ct list */
13:            L^pri ← L^pri \ {W_h}
14:            W_h.p ← F_h[W_h.i - 1].req - ρ_h         /* inter-workflow priority */
15:            L^pri ← L^pri ∪ {W_h}
16:        else
17:            break
18:        end if
19:    end while
20:    pick the head workflow W_h from L^pri
21:    L^pri ← L^pri \ {W_h}
22:    ρ_h ← ρ_h + 1; W_h.p ← W_h.p − 1
23:    L^pri ← L^pri ∪ {W_h}
24:    return next task of W_h
25: end procedure
```

Complexity Analysis: Let n^f denote the number of slots free-up per unit time in the cluster, and n^w denote the number of workflows queuing on the master node. Assume, on average, a progress requirement change event fires every l seconds for one workflow. Then, the AssignTask method will be called every $\frac{1}{n^f}$ second. During this time period, there are at most $\min\{\frac{n^w}{n^f l}, n^w\} \leq \frac{n^w}{n^f l}$ workflows firing progress requirement change event, which is the upper bound

Workflow ID	Next Event Time	Current Priority
1	6	39
2	27	-3
3	1	22
4	5	-17
5	15	31
6	11	13
7	20	2
8	7	-19

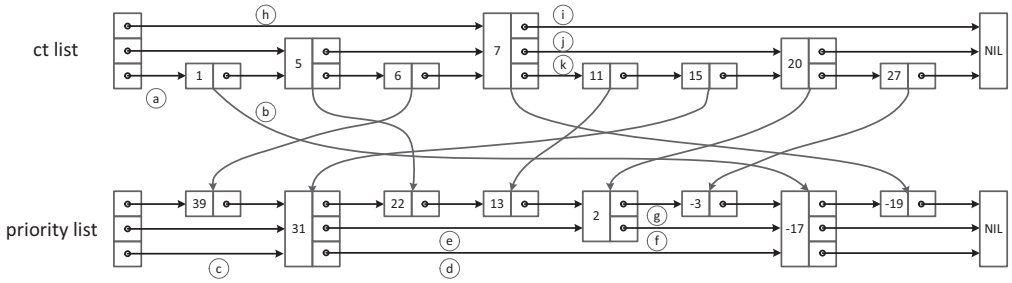

Fig. 4. Double Skip List Example

of the number of iterations occurs on Line $4-19$. Each iteration contains an access operation from the head of the ct list (A^h), a pair of delete and insert operations at arbitrary places of the priority list ($D^a + I^a$), a delete operation from the head of the ct list (D^h), and an insert operation into an arbitrary position of the ct list (I^a). After that, the algorithm schedules the workflow with the highest priority, which includes a delete operation from the head of the priority list (D^h) and an insert operation into an arbitrary position of the priority list. Altogether, the computational complexity of one AssignTask call is:

$$\begin{aligned} & \frac{n^w}{n^f l}\left(A^h + D^a + I^a + D^h + I^a\right) + D^h + I^a \\ = \; & \frac{n^w}{n^f l}\left(2\mathcal{O}(1) + 3\mathcal{O}(\log n^w)\right) + \mathcal{O}(1) + \mathcal{O}(\log n^w) \\ = \; & \frac{n^w + 1}{n^f l}\mathcal{O}(\log n^w) \end{aligned}$$

The total scheduling overhead is the complexity of one AssignTask invocation times the number of invocation in one second, which is $\frac{1}{l}\mathcal{O}(n^w \log n^w)$. According to the statistics shown in Fig. 3, l is larger than 10 seconds. Therefore, the asymptotic scheduling overhead is $\mathcal{O}(n^w \log n^w)$. Therefore, the scheduler scales up to tens of thousands of queuing workflows. In contrast, the naive algorithm causes $\mathcal{O}(n^w \log n^w)$ overhead for each AssignTask call, and hence results in $\mathcal{O}(n^f n^w \log n^w)$ computational complexity.

V. Implementation

A. Experiments Setup

We implemented WOHA by adding 9,292 lines of code into Hadoop-1.2.1. The revision included adding the org.apache.hadoop.mapred.workflow package as well as modifying JobTracker, JobInProgress, TaskTrackerManager, JobConf, JobStatus, etc. The experiments were carried out on the UIUC Green Server Farm cluster [22–25]. The cluster consists of 40 Dell PowerEdge R210 and 40 Dell PowerEdge R620 servers. Each server was configured to run 2 map slots and 1 reduce slot.

Evaluations are based on both synthetic data and job traces from Yahoo! WebScope [19]. The data trace contains the detailed information of more than 4000 jobs on 2012 March 7th. Fig. 5 (a) and Fig. 6 (a) show the cdf of task execution duration and task number respectively: the former shows that most mappers finish between 10s to 100s, while more than half of the reducers take more than 100s and about 10% reducers even take more than 1000s; the latter illustrates that about 30% jobs have more than 100 mappers, while more than 60% jobs have less than 10 reducers. In Fig. 5 (b), we calculate the average mapper duration and the average reducer duration within each job, then plot the ratio of two durations. Fig. 6 (b) shows the ratio of the mapper number over the reducer number in the same job. The statistics meet the common belief that mappers usually outnumber reducers, while reducers take much longer to finish. These statistics acted as guidelines when we generated synthetic jobs.

For demonstration purpose, we constructed a small scale (33 jobs) workflow topology as shown in Fig. 7. Three workflows with the same topology (Fig. 7) are submitted at time 0, 5 minute, and 10 minute respectively. Their relative deadlines ($D_i - R_i$) are set to 80 minutes, 70 minutes, and 60 minutes. That is to say, the third workflow has the latest release time, and earliest deadline. In order to show WOHA's performance in real-world cases, larger workflow topologies (180 jobs) from Yahoo! cluster were also used in evaluations.

B. Porting Existing Schedulers

Since WOHA is the first system that schedules workflow over Hadoop cluster with deadline considerations, there is no directly comparable counterparts. Therefore, we also ported several state-of-the-art Hadoop job scheduling behaviors onto workflows by implementing a WOHA Workflow Scheduler for each of the following three policies.

Oozie with FIFO job scheduler (FIFO): This is the default behavior when putting Oozie and Hadoop together. If all predecessors are completed, Oozie will submit the job to Hadoop JobTracker. The default JobQueueTaskScheduler of Hadoop maintains an ordered list of jobs according to their submission times. To assign a task to an idle slot, the scheduler goes through the ordered list one by one until it finds an available task to schedule. Oozie offers a rich set of configuration options, including the maximum concurrently running jobs within the same workflow, workflow recurrence frequency, etc. However, those are all static configurations set independently with the scheduling progress.

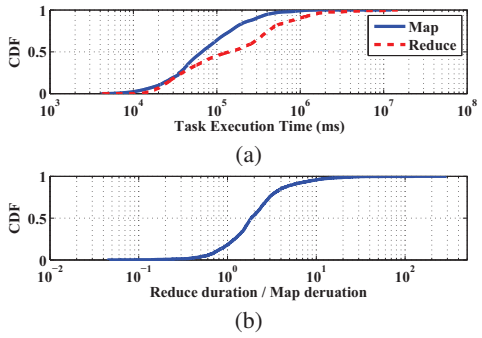

Fig. 5. Task Length

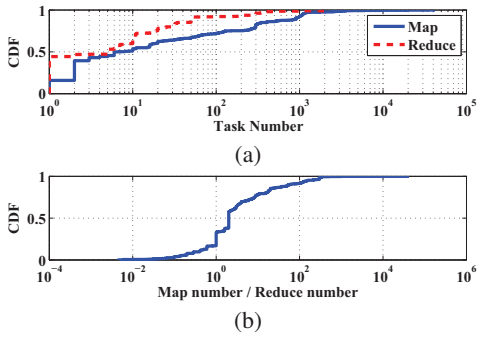

Fig. 6. Task Number

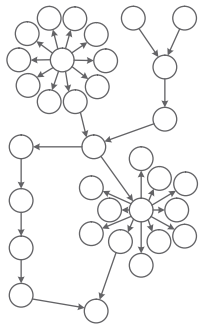

Fig. 7. Workflow Topology

Oozie with Fair job scheduler (Fair): This workflow scheduler mimics the behavior of Facebook FairScheduler, under which all running jobs evenly share the resources of the Hadoop cluster in a work conserving way.

Earliest Deadline First job scheduler (EDF): Earliest Deadline First scheduler is one of the most famous schedulers from the realtime community [26]. It assigns the highest priority to the job with the earliest deadline. Verma *et al.* [10] first introduced the EDF scheduler to Hadoop job scheduling. We ported EDF by assigning highest priority to the workflow with the earliest deadline.

C. Job Priorities

Progress-based scheduling takes job priorities within each workflow as input, and generates the scheduling plan accordingly. There are a large number of existing job prioritizing algorithms for workflows. In the evaluation, we cover three of them.

Highest Level First (HLF): With HLF, jobs are arranged into levels, those with no dependents are assigned to level 0. For jobs on level i, all its dependents must be on levels smaller than i, and at least one of its dependent is on level $i-1$. Then, the job at higher level gets higher priority. Ties are broken by using their job IDs in the workflow. The intuition behind HLF is that, jobs at higher levels have longer sequences of subsequent jobs following it. It assumes longer sequence of jobs takes longer time to finish, and hence deserves higher priority.

Longest Path First (LPF): LPF is an improved solution compared to HLF. Besides the number of jobs on the path, it also considers the length of each job. In the evaluation, the job length is defined as the sum of estimated map task execution time and estimated reduce task execution time.

Maximum Parallelism First (MPF): Under MPF, the job with maximum dependents gets the highest priority. During the scheduling procedure, a workflow has to satisfy two conditions to schedule a task on the available slot: 1) it currently owns the highest workflow priority, and 2) it has available tasks to schedule. MPF aims at reducing the chance of violating the second condition.

VI. Evaluation

In this section, we evaluate WOHA with both synthetic data and real-world traces from Yahoo!. Synthetic data are used to demonstrate detailed scheduler characteristics and behaviors.

A. Meet Deadlines

Let us first take a look at WOHA's performance at meeting deadlines, as this is one of the major objectives of this paper. The experiment employs the workflow topology illustrated in Fig. 7. Three workflows are submitted consecutively with 5 minute time intervals. Their relative deadlines are set to 80 minutes, 70 minutes, and 60 minutes respectively (*i.e.*, workflows with larger release time have to meet earlier deadline). Experiments are carried out on a 32-slave Hadoop cluster. Each slave offers two map slots, and one reduce slot. This small scale setup is also used later to explain the behavior of different schedulers in Section VI-C.

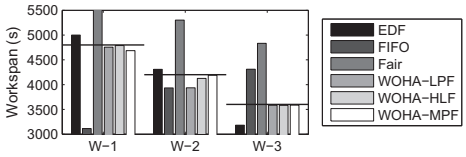

Fig. 11. Synthetic Workflow Workspan - 32 slaves

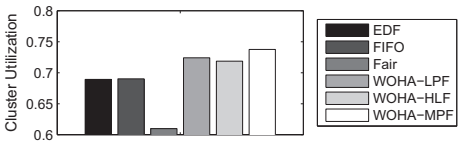

Fig. 12. Cluster Utilization with 3 recurrence

Fig. 11 shows workspans of three workflows under six different schedulers. Among the schedulers, EDF, FIFO, and Fair are ported from existing solutions on scheduling Hadoop jobs, while the other three combine WOHA with workflow scheduling algorithms from the realtime community. In the figure, the X axis marks workflow IDs, and the Y axis shows their workspan values. Fair scheduler behaves the worst compared to others, as each workflow

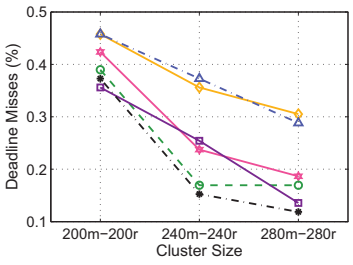

Fig. 8. Deadline Violation Ratio

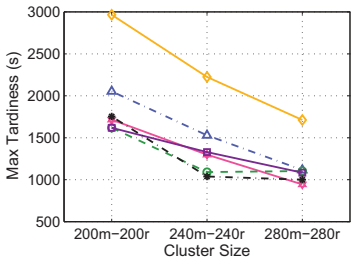

Fig. 9. Max Tardiness

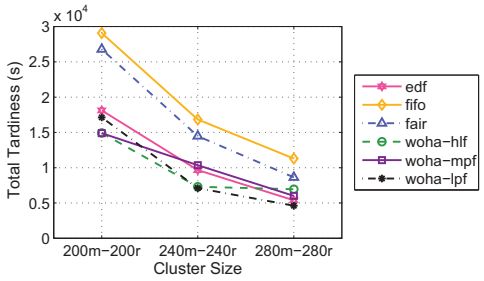

Fig. 10. Total Tardiness

gets fair share of the cluster resource regardless of their deadlines. The EDF scheduler favors the third workflow, since it has the earliest deadline. As a result, W-3 finishes far before its deadline, while the other two workflows violate their deadlines. FIFO completes W-1 20 minutes ahead of the requirement. However, it creates huge tardiness on the third workflow. In contrast, all three experiments using the WOHA framework satisfy their deadlines. As a side benefit, WOHA also increases the cluster utilization as shown in Fig. 12.

The next set of experiments apply the same set of schedulers to data traces from Yahoo!. The data set contains 180 jobs arranged into 61 workflows, among which 15 contains only single job. The largest workflow contains only 12 jobs. There are much larger ones running in their cluster for user log analysis purposes. However, those workflow configurations are not available to authors of this paper. Small scale workflows actually bias against WOHA, since the topologies are too simple to show benefits of applying smart workflow analysis. In the evaluation, we remove workflows containing only single job to even the bias to some degree. Fig. 8 shows the ratio of deadline violation number over the total number of workflows. The X axis shows the cluster size, where "200m-200r" means the cluster contains 200 map slots and 200 reduce slots. The two widely used Hadoop job schedulers (FIFO, and Fair) behave terribly in meeting deadlines. When the cluster contains more resources, EDF and WOHA schedulers result in similar performance. However, when the amount of resource shrinks, WOHA with HLF and LPF start to outperform the EDF scheduler. When the cluster offers even less resources, their performance starts to merge again. This behavior agrees with the intuition. If the cluster offers more than adequate resources, all active jobs get enough resources to execute their tasks. On the other hand, if the cluster contains a very limited amount of slots, it will be impossible for workflows to meet their deadlines. Hence, the largest differences are incurred when the amount of resources are less than adequate but more than scarce.

Fig. 9 and Fig. 10 present maximum tardiness and total tardiness of all workflows. Although EDF violates more deadlines, its total tardiness is very close to WOHA schedulers' outcomes. Sometimes ("280m-280r"), EDF's total tardiness is even less. Please note, reduce the tardiness is not an objective of this paper. Evaluations on tardiness again show that EDF does not allocate tardiness clever enough to meet more deadlines, which agrees with results shown in Fig. 11.

B. Scalability

The second major concern is the scalability of WOHA. It has to guarantee that new components added to Hadoop do not restrict the deployment size of the system. In this section, we specifically evaluate the scheduler throughput and the progress plan size of WOHA.

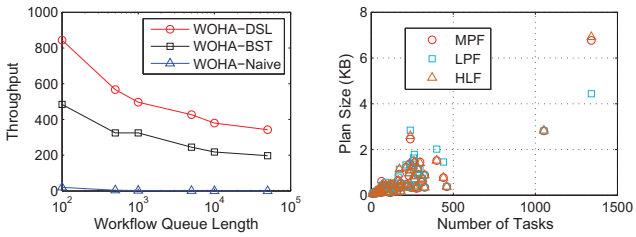

Fig. 13. Cluster Utilization with 3 recurrence

Fig. 13 (a) shows the number of ASSIGNTASK calls that can be made within one second (throughput) against different running workflow queue lengths. The naive solution that updates all workflows priority and reorders accordingly them cannot finish 2 invocations when the queue size increases to 10,000. In Section IV-B, in order to honor the considerable amount of head element access, we designed the Double Skip List (DSL) algorithm, rather than combining two Binary Search Trees (BST). Fig. 13 (a) clearly shows the benefits.

Fig. 13 (b) illustrates the size of scheduling plans when using the progress-based scheduling. This is an important metric, as the scheduling plan consumes both network and memory resources on the master node. It has to be small enough, even when the workflow is large. We feed Yahoo! data and different job prioritization algorithms into the progress requirement list generation algorithm as shown in Algorithm 1. Even when the workflow contains more

than 1400 tasks, the scheduling plan only grows to 7 KB. Under most cases, it stays within 2 KB. Hence, we can conclude that the scheduling plan approach adds negligible communication and storage overhead to the master node.

C. Scheduler Behaviors

Showing only the workspan information offers little for understanding the behavior of different schedulers. In this section, we dive into the detailed slot allocation under different workflow schedulers and analyze the cause of scheduling outcomes. This information will help people to design better schedulers for Hadoop workflows.

Fig. 14 - 19 illustrate slot allocation status of the six experiments used in Fig. 11. The workflow with earlier release time is shaded with darker color. We keep the scale small for demonstration purpose. Rectangles emphasize those spots that convey the characteristics of schedulers. With the FIFO scheduler as shown in Fig. 14, W_1 and W_2 win every resource contention over W_3. As shown in red rectangles, even if W_3 only requires little resource to activate more dependent jobs, it has to wait until W_1 and W_3 finish all their computation requirements. In contrast, in Fig. 19, WOHA with MPF scheduling plan makes room for W_3 during slot contentions if W_3 falls too far from its plan, which activates W_3's subsequent jobs in time and significantly reduces its workspan. Same behavior happens to EDF in the reversed manner. As highlighted on the reduce slot allocation graph of EDF, W_1 and W_1 have only small resource demand around the 900 second. Finishing them in time will fill up the big gap from 900 second to 1600 second on the map graph. However, EDF's pure deadline-based priority assignment prevents that from happening. The FairScheduler is used in Facebook's Hadoop cluster with the purpose of finishing small jobs faster. Nevertheless, it is a terrible scheduler in terms of meeting deadlines. Fair sharing resources just prevents every workflow from finishing in time. Different from the above three scheduler, WOHA dynamically assigns priorities to workflows based on their progresses. No workflow dominates others during the entire execution. Each workflow only takes adequate resources to keep up with their scheduling plan. Fig. 17-19 also show that the scheduling plan itself significantly affects the resource allocation among workflows. An interesting future direction will be to study what is the best we can do under WOHA framework to achieve different workflow scheduling objectives.

VII. Related Work

As WOHA is the first framework that schedules Hadoop workflows with deadlines, there is no directly comparable counterparts. However, plenty of efforts have been spent on Hadoop job scheduling and multi-core workflow scheduling.

Along with broad deployments of Hadoop, many solutions are proposed to improve its scheduling performance. Delay scheduling [4] allows tasks to wait for a certain amount of time, which increases the chance of encountering a slot with the task data locally stored on its hard drive. Chang *et al.* model MapReduce job scheduling as an optimization problem, and design an approximation algorithm within a factor of 3 compared to the optimal solution. EDF-based scheduling algorithms for Hadoop have also been studied in [10], where simulations confirm that simple scheduling heuristics considerably improve the performance in terms of meeting job deadlines. However, all those solutions apply to job scheduling rather than workflow scheduling. A workflow may contain large numbers of Hadoop jobs with complex interdependent relationships. Above Hadoop job schedulers help little to prioritize workflows or jobs within the same workflow. Oozie emerges [12–14] as a generic hadoop workflow management tool. It submits workflow's jobs at the right time based on predefined dependency topologies and recurrence requirements. Nevertheless, Oozie does not share any workflow configuration information with the Hadoop cluster. From the Hadoop's perspective, all submissions made by Oozie are independent, which may lead to terrible deadline violation ratios.

Workflow scheduling on multi-core systems also attracts increasing interests from the real-time community. Saifullah *et al.* [27] first arrange workflow into stages, and then propose a job decomposition method that breaks each parallel job into a set of sequential jobs. If the input workflow is schedulable under the optimal algorithm, their solution is guaranteed to satisfy the deadline with 4X (speedup bound) speed processors. Baruah *et al.* [28] further prove that EDF can achieve 2X speedup bound for recurrent workflows. Delay Composition Algebra [29] deals with a special case of the workflow model where each job has at most one predecessor and at most one dependent. The solution utilizes a set of operators to transform distributed real-time task systems into single-resource ones that preserve schedulability properties of original systems. All those solutions require global knowledge when processing workflows. Unfortunately, the Hadoop master node, as the single entity with the global view, suffers from a very limited amount of computational and storage resources. Hence, above solutions are not applicable to the Hadoop architecture.

VIII. Conclusion

This paper presents the design, implementation, and evaluation of WOHA, a framework specifically designed to schedule Hadoop workflows with deadlines. The Hadoop architecture introduces unique challenges to deadline-aware workflow scheduling. Due to the limited amount of resources, the node with global knowledge (the master node) is not able to carry out any expensive workflow analysis algorithms, while all remaining nodes in the system only has incomplete information of the cluster status. To tackle

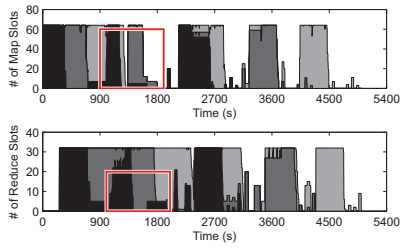

Fig. 14. FIFO

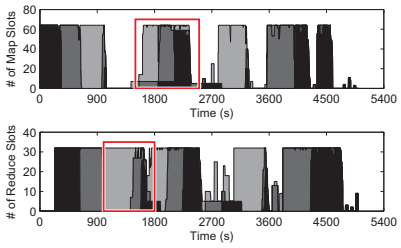

Fig. 15. EDF

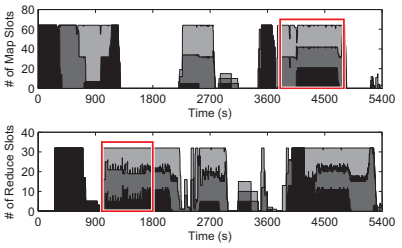

Fig. 16. Fair

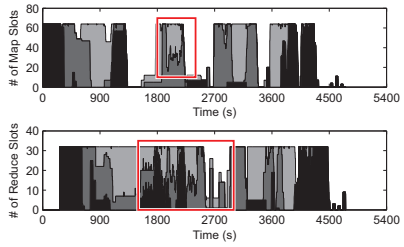

Fig. 17. WOHA-LPF

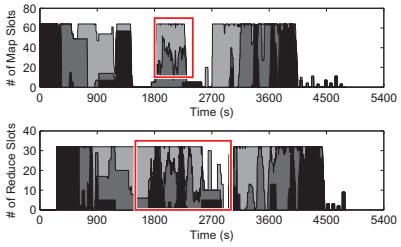

Fig. 18. WOHA-HLF

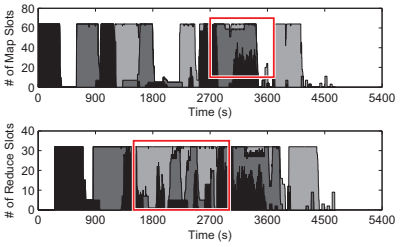

Fig. 19. WOHA-MPF

this problem, WOHA allows client node to locally generate scheduling plans of workflows. Those plans are later sent to the master node together with workflow configuration information. On the master node, efficient algorithms are designed to effectively follow received scheduling plans. Our evaluations confirm that WOHA outperforms existing Hadoop job schedulers in terms of meeting workflow deadlines, and scales up to tens of thousands of concurrent workflows.

ACKNOWLEDGEMENT

This work was sponsored in part by the National Science Foundation under grants CNS 13-20209, CNS 13-02563, CNS 10-35736 and CNS 09-58314. We are also thankful to Yahoo! Inc for access to Webscope data and for Yahoo! reviewers' comments on this work.

REFERENCES

[1] "Apache hadoop," http://hadoop.apache.org/, November 2013.
[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, Jan. 2008.
[3] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *USENIX NSDI*, 2011.
[4] 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 *ACM EuroSys*, 2010.
[5] 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 *USENIX NSDI*, 2012.
[6] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "Making cloud intermediate data fault-tolerant," in *ACM SoCC*, 2010.
[7] S. Li, T. Abdelzaher, and M. Yuan, "Tapa: Temperature aware power allocation in data center with map-reduce," in *IEEE IGCC*, 2011.
[8] I. n. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "Greenhadoop: Leveraging green energy in data-processing frameworks," in *ACM EuroSys*, 2012.
[9] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *IEEE CLOUDCOM 2010*.
[10] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Campbell, "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle," in *IEEE NOMS*, 2012.
[11] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *ACM SIGMOD*, 2008.
[12] "Apache oozie," http://oozie.apache.org/, November 2013.
[13] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur, "Oozie: towards a scalable workflow management system for hadoop," in *ACM SWEET*, 2012.
[14] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang, "Nova: Continuous pig/hadoop workflows," in *ACM SIGMOD*, 2011.
[15] H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*. Prentice-Hall, Inc., 1994.
[16] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *ACM SOCC*, 2013.
[17] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *USENIX NSDI*, 2012.
[18] "Which big data company has the worlds biggest hadoop cluster?" http://www.hadoopwizard.com/, September 2008.
[19] "Yahoo! webscope," http://webscope.sandbox.yahoo.com/, November 2013.
[20] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, Jun. 1990.
[21] J. I. Munro, T. Papadakis, and R. Sedgewick, "Deterministic skip lists," in *ACM-SIAM SODA*, 1992.
[22] "Uiuc green server farm," http://greendatacenters.web.engr.illinois.edu/, November 2013.
[23] S. Li, S. Wang, F. Yang, S. Hu, F. Saremi, and T. Abdelzaher, "Proteus: Power proportional memory cache cluster in data centers," in *IEEE ICDCS*, 2013.
[24] S. Li, H. Le, N. Pham, J. Heo, and T. Abdelzaher, "Joint optimization of computing and cooling energy: Analytic model and a machine room case study," in *IEEE ICDCS*, 2012.
[25] S. Li, S. Wang, T. Abdelzaher, M. Kihl, and A. Robertsson, "Temperature aware power allocation: An optimization framework and case studies," *Sustainable Computing: Informatics and Systems*, 2012.
[26] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, 1973.
[27] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *IEEE RTSS*, 2011.
[28] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *IEEE RTSS*, 2012.
[29] P. Jayachandran and T. Abdelzaher, "Delay composition algebra: A reduction-based schedulability algebra for distributed real-time systems," in *IEEE RTSS*, 2008.