2016 IEEE International Parallel and Distributed Processing Symposium Workshops

A Scheduling Algorithm for Hadoop MapReduce Workflows with Budget Constraints in the Heterogeneous Cloud

Andrew Wylie
School of Computer Science
Carleton University
Ottawa, Ontario K1S5B6
Email: andrew.dale.wylie@gmail.com

Wei Shi
Faculty of Business and IT
University of Ontario Institute of Technology
Oshawa, Ontario L1H7K4
Email: wei.shi@uoit.ca

Jean-Pierre Corriveau
School of Computer Science
Carleton University
Ottawa, Ontario K1S5B6
Email: jeanpier@scs.carleton.ca

Yang Wang
Shenzhen Institute of Advanced Technology
Chinese Academy of Science
Shenzhen, P. R. China
Email: yang.wang1@siat.ac.cn

***Abstract*—In recent years cloud services have gained much attention as a result of their availability, scalability, and low cost. One use of these services has been for the execution of scientific workflows as part of Big Data Analytics, which are employed in a diverse range of fields including astronomy, physics, seismology, and bioinformatics. There has been much research on heuristic scheduling algorithms for these workflows due to the problem's inherent complexity, however existing work has mainly considered execution in a utility grid environment using a generic distributed framework. For our research, we consider the ever-increasingly popular Apache Hadoop framework for scheduling workflow onto resources rented from cloud service providers. Contrary to other distributed frameworks, the Hadoop MapReduce model imposes a functional style onto application definition, and as such presents an interesting and unapproached challenge for workflow scheduling. Investigated in our work is budget-constrained workflow scheduling on the Hadoop MapReduce platform, wherein we devise both an optimal and a heuristic approach to minimize workflow makespan while satisfying a given budget constraint. We have implemented modifications to the Apache Hadoop framework to allow fully integrated workflow scheduling. These modifications are novel and have led to the completion of the first generic workflow scheduler fully integrated with the Apache Hadoop framework. Both the framework modifications and the proposed scheduler implementation have been extensively tested via execution on multiple workflow applications, which demonstrates the ability of our implementation to handle all possible workflow substructures. Results from our empirical studies further establish these facts.**

***Keywords*-Scientific Workflow; Cloud Service; Hadoop; MapReduce; Scheduling Algorithm.**

I. Introduction

In recent years, a great deal of data processing has moved to distributed processing platforms. This migration has been caused by several factors. One such factor is dataset size, which has been constantly growing as companies are able to gather more data relevant to their interests. One example is user engagement and usage patterns gathered by a game studio wanting to keep its subscribers. Other situations are not as novel, such as the tasks of website indexing or log processing. However, these tasks are still necessary, and therefore require innovation when older methods are unable to scale with dataset size. Another factor in the movement to distributed processing has been the low cost of commodity hardware brought on by the ubiquity of personal computers. Lastly, the emergence and adoption of distributed computing frameworks allow lower entrance barriers to distributed execution of programs.

As cloud computing is hailed as an emerging driving force of the next innovation wave, it has been forming the basis for many novel applications across a wide range of fields from business workflows, engineering activities to scientific computing. One of the applications is the *workflow computation* that in general consists of a set of data dependent jobs, forming a *directed acyclic graph* (DAG) to carry out a complex computational process. Generally, the computation of a worflow-based workload would require a sufficient amount of computing resources, and the execution of the jobs are scheduled to different servers for a cost-effective computing. Nowadays, Cloud Service Providers (CSPs) allow users to allocate (virtual) machines with different performance configurations according to "pay-as-you-go" billing model. Thus, the ability to provide resources-on-demand (aka the elasticity of resources), is unprecedented in the history of information technology. The cloud platforms offer a new paradigm so that users with different requirements can be fully satisfied at cost of difference expenses.

Many distributed computing frameworks have been developed in the last decade, though few have reached widespread adoption. The frameworks themselves generally allow the management and administration of executable programs that process data, often known as *applications* or *jobs*. Several frameworks also allow the execution of sequences of jobs known as *workflows*. Some commonly used frameworks include systems such as Apache Hadoop, CloudBATCH, DAGMAN/Condor/Pegasus, and VGE [1], [2], [3]. Among

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.30

these, the most popular is Apache Hadoop, as it has seen deployment onto clusters owned by companies such as Amazon, Google, Microsoft, and Yahoo!. Many complex data processing jobs have recently been implemented on distributed computing frameworks. For instance, Apache Hadoop is used for tasks such as social network mining, log processing, video analysis, image analysis, search indexing, recommendation systems, web indexing, and execution of large-scale scientific applications [4], [5]. As a result of the popularity of the Apache Hadoop framework, many scheduling algorithms have been proposed to optimize different aspects of job execution. The first of these algorithms are the Capacity and Fair schedulers developed by Yahoo! and Facebook, respectively [6]. These were then followed by research efforts dealing with issues such as data locality and node heterogeneity [5], [7], [4], [8].

Regardless of the framework, efficient scheduling is an important requirement. Schedulers themselves vary according to many properties. For instance, they can work to schedule individual jobs, sets of individual jobs (batch scheduling), or sets of jobs connected by dependencies. Additionally, schedulers can take into consideration constraints specified by the job's executor. These vary, though typically pertain to deadline or budget constraints. Deadline constraints specified to a scheduler instruct it to attempt to complete job execution within the specified time constraint, whereas budget constraints instruct it to complete job execution while satisfying a monetary constraint. Budget constraints are relatively new, and have been adopted as more users of distributed computing frameworks have decided to rent resources as opposed to purchasing them. This requirement for temporary resources has driven the emergence of Infrastructure as a Service (IaaS) platforms, which allow users to rent a number of different resources for a specific time period, each of which are priced proportionally to their processing power. This in turn has created a demand for budget-constrained scheduling. We believe that due to the widespread adoption of these products the requirement for efficient budget-constrained scheduling will only grow. Thus, this paper focuses on the creation of a scheduler for the Apache Hadoop framework that allows for the specification and use of budget constraints. As many types of resources are available from IaaS providers, our scheduler is also written to handle execution on a set of heterogeneous resources.

As workflow scheduling is not implemented in the Apache Hadoop framework, several workflow engines exist to provide this functionality; the three main ones being Oozie, Azkaban, and Luigi [9]. These schedulers do have several shortcomings however, especially when considering the need for budget-constrained scheduling seen in recent years. Most importantly, none of these schedulers allow for constraints to be defined, causing them to be unrelated to the main goal of our work. Secondly, the workflow engines all handle the executed workflow themselves, while passing individual jobs to Hadoop for execution. As a result, any possible optimizations available through scheduling the jobs as a single unit are lost. Furthermore, workflow engines do not determine the method of scheduling used by Hadoop. As such, the scheduler used by Hadoop could unknowingly decrease the efficiency of workflow execution.

Along with the lack of feature-rich workflow engines for Hadoop, no budget-constrained workflow schedulers for Hadoop have been proposed in the literature. This is perhaps a result of the complexity of workflow scheduling, as optimal scheduling is an NP-complete problem, and is additionally non-approximable [10], [11], [12], [13], [14], [15], [16]. Specifically, our main objective of workflow scheduling is to minimize computational execution time of the abstract workflow DAG generated from jobs and their constituent tasks. To achieve this objective, the tasks represented as nodes in the DAG must be mapped to resources that satisfy these constraints. It is this mapping problem that is in general NP-complete, and thus also the reason we propose a greedy heuristic.

Considering that many scientific applications require workflow scheduling which considers both execution time and cost, a concrete implementation of budget-constrained scheduling would prove extremely useful [17]. Such an implementation would also be practical for both users of the distributed framework, and for IaaS providers. For instance, users of cloud services would be given peace of mind through assurance of a particular cost for the work they require. Along with this, they would also be able to maintain control over the total cost for workflow execution. Providers of IaaS systems would also benefit through more efficient resource use, as they would be able to serve more customers, and thus outcompete competitors through superior economies of scale. It is for these reasons that we propose in this work both modifications to the Apache Hadoop framework to allow integrated generic workflow scheduling, and introduce a novel budget-constrained workflow scheduling algorithm for the Apache Hadoop framework in an IaaS cloud environment.

II. Problem Formulation

A. Environment Setup and Assumptions

Our assumed execution environment consists of a number of virtual machines rented from an IaaS provider. As we rent the resources, we can assume that the cluster is not shared with other users, and that there are no other programs running that would monopolize resources. Different types of virtual machines can be rented based on our needs, allowing for creation of a heterogeneous set of resources for computation. Of course, the provider charges different service rates for these provisioned machines, which are proportional to their attributes (such as processing power and amount of memory) [15]. We also note that even though

some providers dynamically alter service rates based on market demands, we assume a static pay-per-use rate during scheduling as it is the most common pricing model [18].

We consider an environment wherein the Apache Hadoop framework is run on the rented machines, and focus on the use of MapReduce for workflow scheduling. As mentioned previously, workflow scheduling on Hadoop is not much studied, with only a few approaches considered [19], [20], [21], [22]. Concerning these studies, only [22] considers scheduling in a budget-constrained environment. In their research however, the authors also constrict the scheduled workflows to a simple fork & join structure. Following from the research in [22], we also consider budget-constrained workflow scheduling, though assume no artificial limitations on input workflows.

As we consider general workflows which comprise a set of interdependent jobs, we can represent workflows as DAGs. For our problem, we define a DAG as a directed graph comprising a single connected component that contains no cycles. DAGs consist of a set of vertices (nodes) V along with a set of edges E, and can be formally defined as a tuple $G = (V, E)$. When a DAG represents a workflow, each vertex $v_i \in V$, $1 \leq i \leq |V|$ represents a job in the workflow. Edges are denoted by $e(i, j)$, and correspond to the path from v_i to v_j. This path represents either a control or data dependency constraint which requires v_j finish execution before v_i begins execution. Using this notation v_i is called a *parent* of v_j, and v_j a *child* of v_i. When edges and vertices in the DAG are given weightings, they represent the communication cost and processing cost, respectively. Costs are usually representative of time, though they may also be considered as monetary cost if the environment takes into account budget constraints during scheduling.

Mainly due to the fact that Hadoop abstracts the task of data organization throughout the cluster nodes, we do not consider the cost or time of data transmission in our model. However, as the resources rented from an IaaS provider are supplied with root access, we can assume complete control over the resources. This assumption also allows us to assume control over the configuration of the Apache Hadoop framework, and as such we can configure the number of map and reduce slots provided by different resources. Additionally, control over the Hadoop configuration allows the number of map tasks created for a job to be selected. The number of virtual machines available to rent from the IaaS provider is also assumed to be configurable, and only limited by the given budget constraints. Therefore, we can assume that machines (slots) are never competed for by more than a single task.

Our algorithm assumes scheduling of a single workflow at a time, each of which has access to all resources provided by the Hadoop cluster. For our tests we mainly execute the SIPHT workflow, as shown in Figure 1. To produce a schedule for the input workflow, we assume that dependency

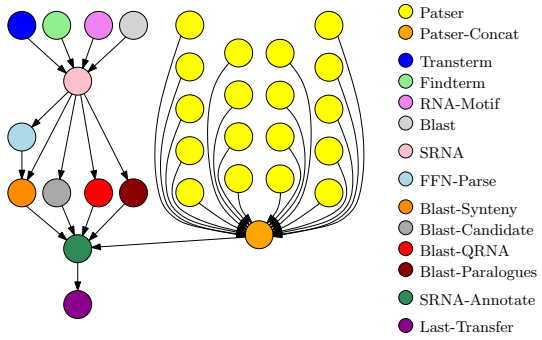

Figure 1: A simplified SIPHT workflow. The job type is represented by node colour in the workflow DAG.

information is provided at the beginning of execution along with all required data files and individual jobs. Individual task processing times are also assumed to be known (and can be calculated by various methods), where tasks split from the same job are generally assumed to be homogeneous in both execution time and resource utilization. Task processing times are modelled along with their cost for each machine type, with the information stored in a table.

B. Formulation

Similar to the algorithms proposed in [22], we consider makespan minimization through reduction of task execution time. In particular, tasks that lie on the critical path(s) of the workflow DAG have their execution time reduced through rescheduling on faster, more expensive machines. To accomplish this, we consider the workflow as consisting of k *stages*, where a single stage S_s is defined as the set of all map (or reduce) tasks in a single job: $S_s = \{\tau_{s1}, \tau_{s2}, \ldots, \tau_{sn_s}\}$, where $0 < s \leq k$, and n_s tasks exist in stage S_s. We also denote the total number of tasks in the workflow as n_τ. Decomposition of the workflow in this manner is permitted by data-flow constraints caused by the framework. Specifically, since all map tasks of a job J_j must complete before any reduce tasks of J_j can begin, and all reduce tasks of a job J_j must complete before the map tasks of any successor of J_j can begin, we can group these tasks together with regards to execution time and priority.

As the cost and execution time of tasks in a workflow are determined by the machine they are scheduled on, we need to account for this in our model. For each task, we store this information in a *time-price* table, as shown in Table I. The table contains the time (t) and price (p) information with regards to all $M_u \mid 0 < u \leq n_m$ available machine/resource types for a specific stage S_s and task $\tau_{s\tau}$. For ease of notation, the table has task execution times sorted in increasing order and task cost in decreasing order.

C. Stage Optimization

Since our algorithm is driven by the provided budget constraint, we can use its value to select valid machine types

Time-Price Table

$t^1_{s\tau}$	$t^2_{s\tau}$	$\cdots$	$t^{n_m}_{s\tau}$
$p^1_{s\tau}$	$p^2_{s\tau}$	$\cdots$	$p^{n_m}_{s\tau}$

Table I: The time-price table for a single task $\tau_{s\tau}$ contains time and price information for each M_u , $0 < u \leq n_m$.

for a particular task. For instance, if given a budget $B_{s\tau}$ for a task $\tau_{s\tau}$, valid machines $u \in M_u$ are those in the time-price table whose cost is less than the budget: $p^u_{s\tau} < B_{s\tau}$. Using the same table, we can compute the shortest time to finish a task when given a budget by selecting the most expensive machine that costs less than the budget. We can compute this time as

$$T_{s\tau}(B_{s\tau}) = t^u_{s\tau} \qquad | \; p^{u-1}_{s\tau} > B_{s\tau} > p^{u+1}_{s\tau}. \tag{1}$$

Using this information, we can determine how to compute the execution time of a single stage S_s with respect to a given budget B_s. As all tasks in a stage must finish for the stage to complete, and since all tasks are independent within that stage, we can define the stage makespan as the maximum execution time of all tasks in the stage:

$$T_s(B_s) = \max_{0<\tau\leq n_s} \{T_{s\tau}(B_{s\tau})\}. \tag{2}$$

Consequently, to decrease the stage execution time we simply reschedule the task with maximum execution time.

D. Workflow Optimization

To minimize the workflow makespan we need to determine a method to calculate the makespan value with respect to the budget constraint B. Given that the workflow makespan is equivalent to the execution time of the longest path, we can find which stages to optimize by determining the DAG's critical path. Since our workflow is arbitrary however, we must also consider that there may exist multiple critical paths. To find a single critical path we first modify the DAG to contain a single entry and exit node, after which a single-source longest-path algorithm can be run between these nodes to find the critical path. Overall, all steps in the process have at most linear computational complexity with respect to the input workflow DAG size. Throughout this paper, we consider our workflow $G = (V, E)$ to consist of $|V| = k$ stages (nodes) and $|E|$ dependencies (edges).

We begin by modifying the arbitrary workflow DAG G to contain only a single entry and exit node. To do this we first find all entry and exit nodes in the workflow DAG. After locating these nodes, all existing entry (exit, respectively) nodes are then connected to a single new zero-cost entry (exit, respectively) node in the DAG. As stated in [13], [23], [24], [25], [15], [14], adding these new nodes does not affect schedule length. Finding all entry and exit nodes takes $|V|$ time, as all nodes must be visited to determine existence of dependencies or successors. Adding the new nodes and edges takes $O(1)+O(|E|)$ time in the worst case (assuming adjacency list storage).

Generally, graph edges are required to be weighted for execution of a shortest or longest path-finding algorithm. However in our case, for each edge $e(u, v),\ u, v \in V$ we can simply use the weight of v as the edge weight $e_w(u, v) = T_v(B_v)$. As an edge stores the incoming (from) and outgoing (to) vertices, this retrieval takes $O(1)$ time for the lookup of the vertex along with its attribute.

Theorem 1: Let $G = (V, E)$ be a node-weighted DAG, and let the weight for any node $u \in V$ be defined as u_w. Also assume that G contains a single entry node s and a single exit node t, where $s_w = 0$ and $t_w = 0$. The results of a deterministic shortest-path algorithm SP on G using node weights is equivalent to the results of the same algorithm run on G using edge weights after setting each edge weight $e(u, v),\ u, v \in V$ to v_w.

Proof: Consider an execution of SP on $G = (V, E)$ from s to t using node weights. As SP is deterministic, the same shortest path is returned for any run of SP on G, even if there exist multiple shortest paths in G. We denote the returned shortest path as P, and its weight as P_w. Since the shortest path is from s to t, both weights $s_w = 0$ and $t_w = 0$ are included in P_w. Therefore, $P_w = \sum_{u\in P} u_w = \sum_{u\in P\setminus\{s,t\}} u_w$.

Consider now an execution of SP on G from s to t using edge weights, where the weight of $e(u, v) = v_w \ \forall \ u, v \in V$. We denote the returned shortest path as P', and its weight by P'_w. Since there does not exist an edge $e(u, s) \mid u \in V$, s_w is not included in P'_w. This not an issue, as s_w also does not contribute any weight to P_w. In P', there must be an edge $e(u, t) \mid u \in V$, as the shortest path runs from s to t. However, as the weight of $e(u, t) = 0$, its inclusion does not contribute to P'_w. For any other edge $e(u, v) \mid u, v \in V$ where $e(u, v) \in P'$, the algorithm must have traversed the path from $u \rightsquigarrow v$, and as such visits both $e(u, v)$ and v. Note that as the graph is a DAG, no other edge $e(x, v) \mid x, v \in V$ can be selected for inclusion in P' after the selection of $e(u, v)$. This ensures that v_w is not considered more than once via the weights of any edges $e(x, v)$.

Therefore, as selection of an edge $e(u, v)$ represents the traversal from $u \rightsquigarrow v$, consideration of either the assigned edge weight of $e(u, v)$ or the node weight v_w is equivalent. ■

At this point we can consider G equivalent to an edge-weighted DAG. The next step taken is to run a topological sort on G, which orders nodes such that each node's dependencies occur before the node itself appears in the ordering. A topological ordering can be found in $O(|V| + |E|)$ time using a modified DFS, and as such has complexity linear in the size of the DAG.

To compute the path weight information a single-source longest-path algorithm is employed, which uses the DAG's topological ordering to run in linear time. The algorithm

works by iteratively updating longest path information as it visits new nodes, with the topological ordering allowing all dependencies of a node to have path weight information computed before the node itself. Due to this, each node must only be visited once. Path weight information is updated for a node using the *relax* function, which throughout the algorithm is executed $|V|$ times, overall visiting $|E|$ edges. Therefore the total time taken to relax all edges is $O(|V| + |E|)$, while initialization takes linear time for the topological sort, and $|V|$ time for variable initialization.

To prove a linear running time for the algorithm, we consider an argument by contradiction regarding the number of calls to *relax*. Assume that there exists an execution of our algorithm in which there are more than $|V|$ calls to the *relax* function. As a result, there must exist a node $v \in V$ that is relaxed more than once. Since nodes are relaxed to update their weight according to maximum path distance among their dependencies, this must mean that v's distance from the entry node s must have been updated more than once due to one of its dependencies having its weighting updated. Otherwise, v would already have a correct weight, and would not have needed to be updated again after the first relaxation. This implies that v was relaxed before one of its dependencies. However, as the topological sort returns nodes in an ordering where all dependencies occur before their successors, it is impossible for any dependency of v to be relaxed after v. As such, v must have a correct weighting after its first relaxation, and therefore is only relaxed once.

After path weight information is computed, we can now determine which stages lie on the critical path(s). Beginning from the exit node, or *sink*, we traverse back along the critical paths using a modified BFS. More specifically, edges traversed are selected by considering only the node(s) of maximum value among all predecessors. These traversed nodes (stages) are added as they are found to a set of critical stages, which are returned when the function completes. The algorithm runs in $O(|V|+|E|)$ time in the worst case; when all nodes lie on the critical path(s), this forces all vertices to be visited by traversal along all edges. For instance, consider the nodes which can be added to the *vertices* set. These are selected as the predecessors of the current nodes in *vertices*. Since the graph is acyclic, no node can be added to *vertices* more than once. Similarly, since only incoming edges to a node are traversed, no edge can be traversed more than once. Therefore, this portion of the algorithm has a linear time complexity of $O(|V| + |E|)$.

III. Greedy Scheduling Algorithm Description and Theoretical Analysis

In this section, we introduce a heuristic that iteratively reschedules tasks from stages on the workflow's critical path(s). We begin by considering the applicability of the globally optimal algorithm proposed in [22] to our modified problem. Following this, we also considered several other possible methods for computation of an optimal scheduling. For the heuristic, selection of tasks for rescheduling is based on a utility value that accounts for changes in both the workflow cost and makespan caused by the reschedule. The principal goal for our algorithms is to efficiently distribute budget over the individual stages (and tasks), such that the makespan of the workflow is minimized while the total cost is within the given budget constraint. For the input workflow DAG G, task execution information is recorded as attributes on tasks themselves. As such, we denote the list of tasks in the workflow as $G.\tau$. Indexing into this list allows the price, time, and machine attributes to be set and retrieved. For instance, price information for a task Γ (or an index i) can be set by the statement $G.\tau_{\Gamma}.p = price$ (or $G.\tau_i.p = price$). Additionally, we also assume that the $G.\tau_{exit}$ attribute contains the workflow's exit stage. As we add the exit stage to the workflow ourselves, this attribute is trivial to define. Also used in the algorithms is the time-price table, denoted by TP. The table is accessed to select either a p or t function for execution, which return the price or time, respectively. Similar to the original time-price table introduced in Section II-B, the p and t functions require as input a task τ and a machine M_u.

Scheduling begins with an initial assignment of all tasks to the least expensive resource type and is followed by rescheduling of select stages on the critical path, based on prospective changes to makespan and cost. The initial assignment simply ensures that the given budget is valid for the input workflow, along with establishing a base configuration to build from. After this step the algorithm proceeds in an iterative manner, selecting a stage on the critical path to have its slowest task rescheduled. This rescheduling has the potential to alter the workflow's critical paths, and as such recomputation of the critical path is necessary. Stage selection is based on the relative improvement of schedule makespan with respect to cost increase, compared against other eligible stages (those on the critical path). As mentioned the rescheduling is done iteratively, and executes until there is either insufficient budget left for rescheduling, or until there are no tasks left that can be rescheduled. The pseudocode for this algorithm is shown in Algorithm 1, with an explanation forthwith.

To direct stage selection, we denote the *utility* of a task τ in stage s as $v_{s\tau}$, calculated by

$$v_{s\tau} = \frac{\min\left\{(t^u_{s\tau} - t^{u-1}_{s\tau}), (t^u_{s\tau} - t^{u'}_{s\tau'})\right\}}{p^{u-1}_{s\tau} - p^u_{s\tau}} \tag{3}$$

if there exists more than one task in a stage. Otherwise, *utility* is given by

$$v_{s\tau} = \frac{t^u_{s\tau} - t^{u-1}_{s\tau}}{p^{u-1}_{s\tau} - p^u_{s\tau}}. \tag{4}$$

In Equation 3, the slowest task τ is assumed to be currently assigned to the machine u, whereas the second-slowest task

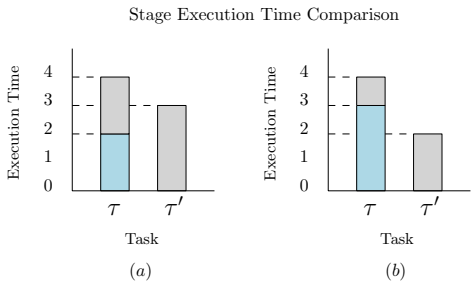

Figure 2: Task execution times before rescheduling are shown in grey, with shorter, rescheduled execution times shown in blue. In (a), rescheduling τ allows τ' to become the slowest task. However it is possible, as in (b), for τ to continue to be the slowest task in a stage.

τ' is assigned to u'. Also recall that the time-price table is sorted, with times in increasing order and cost in decreasing order. The meaning of this equation is visualized in Figure 2, which shows a bar chart representing the makespan of tasks in a single stage. For example, consider the difference between part (a) and (b). In (a), rescheduling the slowest task causes the bottleneck to change to the second-slowest task. However in (b), rescheduling the slowest task does not change which task is limiting the stage makespan. It is because of these two separate cases that the minimum execution time decrease between the slowest and second-slowest tasks is used by the equation, as this allows realization of the actual time speedup caused by rescheduling.

Algorithm 1 accepts as arguments the workflow DAG G, set of machines M, time-price table TP, and a budget B. It begins by performing an initial scheduling in the for loop on Line 3. This initialization assigns the least expensive machine type to each task in the workflow, along with updating the time and price information for the task. As this task-resource mapping is being performed, the schedule cost is also recorded so that it can be reconciled against the input budget B later in the algorithm.

The main while loop begins on Line 13, where it repeatedly reschedules tasks until no budget remains. Within the main execution loop several functions are first called to configure the graph weightings, and then to calculate the critical stages. The first of these functions updates stage weights to be the maximum weight of their composite tasks. In addition to updating stage weight information, the *UPDATE_STAGE_TIMES* function is also modified in this case to record for each stage the slowest task and second-slowest task. The following two functions serve to calculate the critical stages, first by determining makespan information, and then using the computed information to ascertain which stages lie on the critical path(s). Following computation of the critical stages, utility values are computed in the for loop beginning on Line 18. In this for loop the slowest and second-slowest tasks are retrieved for each stage, and then are used in conjunction with the time-price table to calculate each stage's utility.

Algorithm 1 A greedy scheduling algorithm.

```
1:  procedure SCHEDULE_HEURISTIC(G, M, TP, B)
2:      cost = 0
3:      for i ∈ G.n_τ do                                              ▷ O(n_τ)
4:          G.τ_i.m = M_{n_m − 1}
5:          G.τ_i.p = TP.p(G.τ_i, G.τ_i.m)
6:          G.τ_i.t = TP.t(G.τ_i, G.τ_i.m)
7:          cost ≥ G.τ_i.t
8:      end for
9:      B ≤ cost
10:     if B < 0 then
11:         return None
12:     end if
13:     while B ≥ 0 do                                                ▷ O(n_τ × n_m)
14:         UPDATE_STAGE_TIMES(G)                                     ▷ O(|V| + |E| + n_τ)
15:         distances = CALCULATE_CRITICAL(G)                         ▷ O(|V| + |E|)
16:         S_critical = GET_CRITICAL_STAGES(G, G.τ_exit, distances)  ▷ O(|V| + |E|)
17:         v = ∅
18:         for S_s ∈ S_critical do                                   ▷ O(|V|)
19:             (Γ, γ) = (S_s.slowest, S_s.second_slowest)
20:             X = (TP.t(Γ, G.τ_Γ.m) − TP.t(Γ, G.τ_Γ.m − 1))
21:             Y = (TP.t(Γ, G.τ_Γ.m) − TP.t(γ, G.τ_γ.m))
22:             v[Γ] = min{X,Y} / (TP.p(Γ,G.τ_Γ.m−1) − TP.p(Γ,G.τ_Γ.m))
23:         end for
24:         while v ≠ ∅ do                                            ▷ O(|V| log |V|)
25:             Γ = max_{τ∈v.keys()}{v[τ]}
26:             new_price = TP.p(Γ, G.τ_Γ.m − 1)
27:             old_price = TP.p(Γ, G.τ_Γ.m)
28:             if B < (new_price − old_price) then
29:                 v.remove(Γ)
30:             else
31:                 G.τ_Γ.m ≤ 1
32:                 G.τ_Γ.p = TP.p(Γ, G.τ_Γ.m)
33:                 G.τ_Γ.t = TP.t(Γ, G.τ_Γ.m)
34:                 B ≤ (new_price − old_price)
35:                 break
36:             end if
37:         end while
38:         if v = ∅ then
39:             return G
40:         end if
41:     end while
42: end procedure
```

At this point the while loop on Line 24 is entered, where the critical stages' utility values are iterated through, as orderd by descending utility values. For each utility value the corresponding task Γ is retrieved and analyzed. Since the utility value only ensures that the most cost-efficient task is selected, it is still possible that the magnitude of the cost is greater than allowed. Thus, the change in price caused by rescheduling is computed and compared against the available budget. If the budget is insufficient then the task is discarded while the algorithm continues on to the next task. Otherwise, we reschedule the task onto a more powerful machine, update the related attributes, and break out of the inner loop to allow critical path information to be recomputed. The algorithm exits based upon the condition on Line 38; if no critical stages can be rescheduled then the workflow makespan cannot be decreased any further, and so the algorithm exits.

Theorem 2: The running time of Algorithm 1 is $O(n_\tau + (n_\tau \times n_m) \times (|V| \log |V| + |V| + |E| + n_\tau))$.

Proof: The algorithm begins with an initialization loop on Line 3. The loop runs once for each of the n_τ tasks in the workflow DAG $G = (V, E)$. In each iteration it does a constant amount of work to initialize the machine type, price, and time of each task. As well, an initial cost is calculated. As a constant amount of work is done in each of the n_τ

loop iterations, the initialization stage takes $O(n_\tau)$ time.

Next, we consider the execution time of the main while loop on Line 13. As a bound on the running time, we know that the maximum number of times the loop can run is related to the number of times rescheduling can occur, as this is also the maximum number of times that the budget can be updated. Specifically, since each of the n_τ tasks can be rescheduled $n_m - 1$ times, the loop must execute less than $n_\tau \times (n_m - 1)$ times. Therefore, the outer loop executes at most $O(n_\tau \times n_m)$ times.

The first computation within the outer while loop updates the stage times. This is accomplished by the *UPDATE_STAGE_TIMES* function, whose run time was derived as $O(|V| + |E| + n_\tau)$. In this instance however, it has been modified to record both the slowest and second slowest tasks of each stage. To find the these tasks for a particular stage, the algorithm needs to iterate over all tasks belonging to the stage. However, since all tasks are already visited for computation of the stage's execution time, this addition only adds a constant amount of work for each task visited, leaving the total execution time unmodified. As such, the run time of the algorithm is $O(|V| + |E| + n_\tau)$.

After the stage times are updated the critical path information is computed and then retrieved. As shown in subsection II-D, both algorithms take $O(|V| + |E|)$ time.

To place an upper bound on the inner for loop, we realize that as the workflow DAG is arbitrary there is no guarantee on the number of stages which comprise the critical path(s). As a result, all stages can in fact lie on the critical path, causing the loop to execute $|V|$ times in the worst case. For each iteration of this loop, stage utilities are computed in constant time. Therefore, the inner for loop takes at most $O(|V|)$ time to execute in the worst case.

The inner while loop can execute at most $|V|$ times, assuming that for each stage the cost for rescheduling is found to be larger than the available budget. Within the loop at each iteration, we first retrieve the task with maximum utility, and then compare its current and new execution prices using information from the time-price table. The lookup of information from the time-price table along with all subsequent operations take $O(1)$ time. To obtain tasks ordered by their utilities, we use a priority queue structure to allow retrieval in $O(1)$ time, albeit with an overhead of $\log |V|$ time for initialization in the worst case. However, this is much more efficient than a linear search, which would take $O(|V|)$ time per iteration. Therefore, the execution time of the inner while loop is at most $O(|V| \log |V|)$.

Overall, the algorithm's time is therefore $O(n_\tau + (n_\tau \times n_m) \times (|V| \log |V| + |V| + |E| + n_\tau))$. Simplifying this, we can both remove the non-dominant terms and assume that the number of machines n_m is bounded by a small integer constant (4 in our experiments) to give a resultant time of $O(n_\tau \times (|V| \log |V| + |E| + n_\tau))$. ■

IV. Implementation

Implementation of the proposed algorithm was carried out in Hadoop version 1.2.1. It includes the addition of several packages, with around 8500 lines of code added in total. This included creation of the $org.apache.hadoop.mapred.workflow$, $org.apache.hadoop.mapred.workflow.schedulers$, and $org.apache.hadoop.mapred.workflow.scheduling$ packages. Several existing classes were also modified to allow workflow execution. In this case, the main modifications were made to the $org.apache.hadoop.mapred.core.util.RunJar$, $org.apache.hadoop.mapred.JobTracker$, and $org.apache.hadoop.mapred.JobClient$ classes. For testing of the implementation, classes were also added to the new packages $org.apache.hadoop.workflow.examples$ and $org.apache.hadoop.workflow.examples.jobs$. The changes made along with the method of execution are briefly reviewed in this section.

We consider the result of a single heartbeat message sent from a *TaskTracker* node to the *JobTracker*. These heartbeat messages are sent repeatedly after framework initialization, and allow regular communication with the *JobTracker* in order to synchronize status information, and for task assignment. While handling a message, the *JobTracker* delegates task assignment to the current task scheduler. The scheduler itself operates alongside the *JobTracker* on the server node, and is constructed via reflection from a configuration property during startup. When initialized the task scheduler creates listeners, which it then adds to the *JobTracker*. It is these listeners that the *JobTracker* and task scheduler use to control scheduling, as they notified when jobs are added, changed, or removed. As a result, when the *JobTracker* calls the scheduler to assign tasks to the *TaskTracker* the currently submitted jobs can be retrieved from a listener and used to determine which and how many tasks should be executed.

The implementation of workflow scheduling attempts to take advantage of the current job scheduling features as much as possible. As a result, it follows the same basic pattern as job scheduling. Modifications include a customized listener which listens for both job events and workflow events (addition, modification, removal), and the scheduler itself which launches executable workflow jobs as well as assigning tasks to querying *TaskTracker*s. Due to the complexity of workflow scheduling, our assumptions of known data, and the constraint requirements, much of the scheduling decisions are delegated to an additional *SchedulingPlan* class. The class is instantiated on the client machine during workflow submission, where it generates the plan based on cluster properties. The plan is then passed to the scheduler (via the *JobTracker*) during workflow submission, and used to decide how to run both workflow jobs and their composite tasks.

V. Empirical Evaluation

A. Test Setup

In the related literature there exist many different test configurations proposed for validation of modifications to the Hadoop framework. The authors of [5] utilize a cluster of 800 virtual machines on Amazon EC2 (spread over 160 dedicated physical machines). At the other end of the spectrum, [26] employs a 30-node cluster on Amazon EC2, with nodes split evenly between three different machine types. Lastly, the authors of [27] propose the most convincing configuration. In their work, three different configurations are tested on Amazon EC2. Each configuration varies in both the total number of machines (68, 97, and 99) and the composition (between two different machine types). Comparing their configuration to a 2010 survey outlining the average Hadoop cluster size as 66 nodes, this seems to be an excellent cluster size [28].

Using these figures, we decided on a cluster size of 81 nodes, all located in one Amazon EC2 region for the purpose of minimizing data transfer times. The size was selected due to the figures proposed in [28], along with the cluster configuration used in [27]. Another factor in the decision of overall cluster size is the fact that many large companies are continuing to grow their Hadoop clusters, and as a result the average as of 2010 has likely also increased.

With regards to machine types, Amazon EC2 offers many machines suitable for different computational situations [29]. As an overview, they provide instances optimized for any of compute, memory, Graphics Processing Unit (GPU), or storage scenarios. They also offer instances suitable for general-purpose computation, which are the types that we employ in our tests. Altogether, we utilize four different machine types within the EC2 *m3* family: *m3.medium*, *m3.large*, *m3.xlarge*, and *m3.2xlarge*.

We base our selected machine distribution on the insight that composition in a production cluster is often not balanced; as machines become obsolete or otherwise are decommissioned, they are replaced by newer machines which have more computational power. With this in mind, we propose a test configuration comprising 30 *m3.medium* nodes, 25 *m3.large* nodes, 21 *m3.xlarge* nodes, and 5 *m3.2xlarge* nodes. A single node of type *m3.xlarge* is used as the master (*JobTracker*), while the remaining nodes are retained as slaves to take on the role of *TaskTracker*s.

B. Workflow Configuration & Job Definition

Apart from configuration of the cluster environment, the other main testing decision is which workflow to test with. Originally we selected both the SIPHT and LIGO workflows. The aforementioned workflows were mainly selected for two reasons. First, they both contain all workflow substructures. Second, both workflows are sufficiently large at 31 jobs for SIPHT and 40 jobs for LIGO. In addition to these properties, SIPHT was constructed to use two separate input directories, and the LIGO workflow is actually defined as two DAGs contained in a single graph. These properties together cover all edge cases to be tested with the workflow scheduling modifications made to the Hadoop framework. In this paper, we report testing of our modifications to the Hadoop framework and the proposed scheduler on the final 81-node cluster were accomplished with the SIPHT workflow.

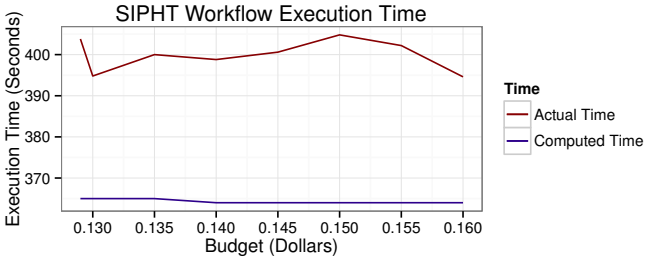

Figure 3: The actual and computed time for execution of the SIPHT workflow are shown according to different budget amounts. The proposed greedy scheduler was used for execution on our modified Hadoop framework.

C. Workflow Scheduling Experiments

We first collected task times for use in the time-price table on a per-machine-type basis for each machine type utilized in the 81-node test cluster, as calculated from statistically averaged historical data. We then executed the greedy budget-constrained workflow scheduler on the SIPHT workflow using our modified Hadoop framework. This execution was run 5 times for 8 budget values within the range \$0.129 to \$0.16. The budgets were selected such that the range covered from an infeasible amount (budget is less than workflow cost when using only the least expensive machine type) up to an amount larger than the highest cost selected by the scheduler (all tasks assigned to the most expensive machine type). Within these boundaries, additional values were selected at even intervals between the boundary values. For each run the computed execution cost and time were recorded, along with the actual time and machine type mapping. The metric logging code used during task time collection was also used during testing, and along with the machine type mapping allowed us to compute the actual cost of workflow execution. The values for these runs were then averaged to a single value for each budget value, the results of which are displayed in Figure 3 and Figure 4.

Figure 3 shows both the execution time computed by the greedy scheduler and the actual execution time as recorded during testing. From these two figures, we observe that the actual execution time was generally 35 seconds above

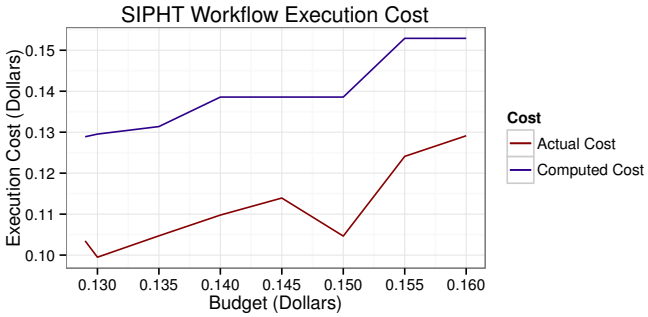

Figure 4: The actual and computed cost for execution of the SIPHT workflow are shown according to different budget amounts. The proposed greedy scheduler was used for execution on our modified Hadoop framework.

the computed execution time. As mentioned previously, and as shown through earlier testing in Section V-B, the data transfer times experienced during execution are not insignificant. Specifically, they will vary depending upon where the data is placed in relation to the node selected for task execution. Since these data transfer times are not considered by the greedy scheduler, it is easy to realize this as the main source of the disparity.

Figure 4 similarly shows both the execution time computed by the greedy scheduler and the actual cost as computed from testing. As expected, both cost values increase as the budget increases, while still remaining below the budget amount. However, the actual cost is generally $0.03 below the computed cost. The most likely reason is rounding errors seen with float values at the higher precision required for these computations. As for the local minimum appearing when the budget equals $0.15, we suspect from comparison with Figure 3 that the change in budget allowed for a larger number of less expensive machines to be selected (or a smaller number of more expensive machines), causing a tradeoff between execution time and cost to occur. In reality, workflows are generally larger in size and comprise jobs with longer execution times. As such, less precision is required, and the difference between computed cost and actual computed cost seen in these experiments will not manifest.

VI. Conclusion

In recent years cloud services - such as those supplied by IaaS providers - have been gaining traction, mainly due to their availability, scalability, and low cost. This increase in the use of cloud services, combined with the proliferation of distributed computing frameworks and widespread adoption of Apache Hadoop in particular has allowed more users to take advantage of the benefits of distributed computing. However, the combination of Apache Hadoop with execution of applications on rented infrastructure reveals several important, unimplemented features.

To address these issues, we have implemented modifications to the Apache Hadoop framework to allow fully integrated workflow scheduling. Additionally, we have developed and tested a greedy budget-constrained scheduling algorithm against several workflows, as well as developing both a progress-based and an optimal brute-force algorithm. These modifications are novel and have led to the completion of the first generic workflow scheduler fully integrated with the Apache Hadoop framework. Moreover, our greedy budget-constrained algorithm is the first scheduler for Apache Hadoop that both deals with budget constraints and executes workflows. Both the framework modifications and the greedy scheduler implementation have been extensively tested via execution on multiple workflow applications, which demonstrates the ability of our implementation to handle all possible workflow substructures. Results from our empirical studies establish these facts, given that the greedy scheduler both executes correctly, and produces an expected makespan and cost according to various budget constraint values.

Acknowledgment

The authors gratefully acknowledge financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) under Grant No. RGPIN-2015-05390.

References

[1] C. Zhang and H. De Sterck, "Cloudbatch: A batch job queuing system on clouds with hadoop and hbase," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, November 2010, pp. 368–375.

[2] "Dagman," http://research.cs.wisc.edu/htcondor/dagman/dagman.html, [Accessed 2014-11-10].

[3] S. Benkner, I. Brandic, G. Engelbrecht, and R. Schmidt, "Vge - a service-oriented grid environment for on-demand supercomputing," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, November 2004, pp. 11–18.

[4] J. Tan, X. Meng, and L. Zhang, "Coupling task progress for mapreduce resource-aware scheduling," in *INFOCOM, 2013 Proceedings IEEE*, April 2013, pp. 1618–1626.

[5] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744

[6] B. Rao and L. Reddy, "Survey on improved scheduling in hadoop mapreduce in cloud environments," *CoRR*, vol. abs/1207.0780, 2012. [Online]. Available: http://arxiv.org/abs/1207.0780

[7] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–9.

[8] B. Kapil and S. Kamath, "Resource aware scheduling in hadoop for heterogeneous workloads based on load estimation," in *Computing, Communications and Networking Technologies (ICCCNT),2013 Fourth International Conference on*, July 2013, pp. 1–5.

[9] "Workflow Engines for Hadoop," http://www.slideshare.net/jcrobak/data-engineermeetup-201309, [Accessed 2015-08-19].

[10] H. Prajapati and V. Shah, "Advance reservation based dag application scheduling simulator for grid environment," *International Journal of Computer Applications*, vol. 61, no. 7, pp. 45–51, January 2013, published by Foundation of Computer Science, New York, USA.

[11] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, pp. 219–237, 2005.

[12] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 171–200, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10723-005-9010-8

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

[14] S. Abrishami, M. Naghibzadeh, and D. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 158–169, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.future.2012.05.004

[15] H. Arabnejad and J. Barbosa, "A budget constrained scheduling algorithm for workflow applications," *Journal of Grid Computing*, pp. 1–15, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10723-014-9294-7

[16] X. Lin and C. Wu, "On scientific workflow scheduling in clouds under budget constraint," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 90–99.

[17] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi, "Characterization of scientific workflows," in *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, Nov. 2008, pp. 1–10.

[18] C. N. Höfer and G. Karagiannis, "Cloud computing services: taxonomy and comparison," *Journal of Internet Services and Applications*, vol. 2, no. 2, pp. 81–94, 2011. [Online]. Available: http://dx.doi.org/10.1007/s13174-011-0027-x

[19] Z. Tang, M. Liu, K. Li, and Y. Xu, "A mapreduce-enabled scientific workflow framework with optimization scheduling algorithm," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, Dec 2012, pp. 599–604.

[20] Y. Mao, W. Wu, H. Zhang, and L. Luo, "Greenpipe: A hadoop based workflow system on energy-efficient clouds," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 2211–2219.

[21] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, "Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, June 2014, pp. 93–103.

[22] Y. Wang and W. Shi, "Budget-driven scheduling algorithms for batches of mapreduce jobs in heterogeneous clouds," *Cloud Computing, IEEE Transactions on*, vol. 2, no. 3, pp. 306–319, July 2014.

[23] J. Yu, R. Buyya, and K. Ramamohanarao, "Workflow scheduling algorithms for grid computing," in *Metaheuristics for Scheduling in Distributed Computing Environments*, ser. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2008, vol. 146, pp. 173–214. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69277-5_7

[24] L. Zeng, B. Veeravalli, and X. Li, "Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud," in *Proceedings of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, ser. AINA '12, 2012, pp. 534–541.

[25] H. Arabnejad and J. Barbosa, "Budget constrained scheduling strategies for on-line workflow applications," in *Computational Science and Its Applications – ICCSA 2014*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8584, pp. 532–545. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09153-2_40

[26] Z. Zhang, L. Cherkasova, and B. Loo, "Performance modeling of mapreduce jobs in heterogeneous cloud environments," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, ser. CLOUD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 839–846. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2013.107

[27] ——, "Exploiting cloud heterogeneity for optimized cost/performance mapreduce processing," in *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*, ser. CloudDP '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:6. [Online]. Available: http://doi.acm.org/10.1145/2592784.2592785

[28] A. Verma, L. Cherkasova, and R. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998637

[29] "Amazon EC2 Instances," http://aws.amazon.com/ec2/instance-types/, [Accessed 2015-10-17].