# WIRE: Resource-efficient Scaling with Online Prediction for DAG-based Workflows

Bing Xie[1*], Qiang Cao[2*], Mayuresh Kunjir[3*], Linli Wan[4*], Jeff Chase[5], Anirban Mandal[6], Mats Rynge[7]
[1]*Oak Ridge National Laboratory,* [2]*East China Normal University,* [3]*Qatar Computing Research Institute,*
[4]*Facebook,* [5]*Duke University,* [6]*Renaissance Computing Institute,* [7]*Information Sciences Institute*

***Abstract*—This paper introduces WIRE that manages resources for the DAG-based workflows on IaaS clouds. WIRE predicts and plans resources over the MAPE (Monitor-Analyze-Plan-Execute) loops to: 1) Estimate task performance with online data, 2) Conduct simulations to predict the upcoming loads based on online estimates and workflow DAGs, 3) Apply a resource-steering policy to size cloud instance pools for the maximal parallelism that is consistent with low cost. We implement WIRE on Pegasus WMS/HTCondor and evaluate its performance on the ExoGENI network cloud. The results show that WIRE attains low resource cost with the performance that is typically within a factor of two of optimal.**

***Keywords*-resource scaling; DAG-based workflows; machine learning; cloud computing**

## I. Introduction

This work discusses the cost-efficient auto-scaling of the rented virtual infrastructures—cloud resource instances such as IaaS VMs or containers—to run computational workflows. For our purposes, a workflow is a set of sequential *tasks* with a partial order specified in advance as a static DAG of the data flow dependencies among the tasks. For a workflow run, the workload is a number of tasks, with each task the unit of computation and resource consumption. When a group of tasks share the same executable and the same dependent predecessor tasks, the task group is considered as a *stage*. The elastic scaling for workflows is challenging because the available parallelism (width) of a workflow may vary dramatically as it runs. On the other hand, the DAG structure enables the prediction of resource demands *if* task runtimes are known.

A workflow scaling policy must address two competing objectives: performance and efficiency. To minimize the completion time (makespan), it must provision enough instances to harvest the parallelism available when the workflow is wide. But if it provisions instances too aggressively, then it leaves resources idle when the workflow is narrow. Idle instances are wasteful. Many previous workflow resource managers ( [1]–[4]) predict the loads for the *recurrent* workflows based on the measurements from *previous* runs, which ignores potential variations across runs (§II-B).

This paper presents a cost-efficient scaling system for DAG-based **W**orkflows on **I**aaS with **R**esource **E**fficiency (*WIRE*). The core of *WIRE* is a MAPE control loop (Monitor-Analyze-Plan-Execute) that builds machine learning models (online gradient descent method) over a continuous online stream of performance monitoring data, and uses those models to adjust the cloud resources for the workflow elastically as it executes. In each iteration, *WIRE* predicts the resource consumption of the future task executions and data transfers based on the best available information (§III-B1). It uses these predictions to estimate the upcoming load based on the workflow DAG structure and its current status. To avoid oscillations and stabilize the prediction results, *WIRE* introduces an online prediction policy (§III-B2) in consideration of both short- and long-term online information. Then it applies a *resource-steering* policy (§III-B2) to allocate or release cloud resources to obtain the shortest expected completion time that maintains the utilization of cloud resources above a given target level over any charging unit (defined in §III-A). Loosely, *WIRE* steers the workflow's cloud usage over time to obtain the "best bang for the buck."

*WIRE* is implemented for Pegasus WMS/HTCondor [5], [6] using the dynamic resource management APIs for the ExoGENI network cloud [7]. We evaluate *WIRE* on ExoGENI with a Pegasus epigenomics workflow and the emulated Pegasus workflows whose resource profiles match the measured Hadoop workflows (see Table I). The results show that *WIRE* takes advantage of the available parallelism with low overhead and low cost, even with many short tasks and/or imperfect predictions. Compared to the static executions provisioned for the peak task load, *WIRE* delivers 4.93×—14.66× of the lower resource cost while delivering performance within a factor of two for 83.75% of the runs.

## II. Background and Motivation

### A. Performance Variability: Within a Run

Table I summarizes the workflows used in our experiments. It shows that, for a workflow run, the number of the tasks of a stage may differ by three orders of magnitude; the average task execution time of a stage may vary from several seconds to several minutes. Moreover, for a stage in a run, different tasks process different inputs and may execute on different instances with varying network conditions. Thus the tasks in the same stage may exhibit different performance. This intra-stage performance variability (also called load skew) is widely observed [8]–[11].

*The authors conducted much of this research when they were with the Computer Science Department at Duke University.

| Workflow Name | Epigenomics | | TPC-H/TPCH-1 | | TPC-H/TPCH-6 | | PageRank/Intel Hibench | |
|---|---|---|---|---|---|---|---|---|
| Framework | Condor | | Hadoop | | Hadoop | | Hadoop | |
| Run | Genome S | Genome L | TPCH-1 S | TPCH-1 L | TPCH-6 S | TPCH-6 L | PageRank S | PageRank L |
| Data Size (Unit:GB) | 0.002 | 0.013 | 7.27 | 29.53 | 7.27 | 29.53 | 0.26 | 2.88 |
| Number of Stages | 8 | 8 | 4 | 4 | 2 | 2 | 12 | 12 |
| Aggregate Task Execution Time (Unit:Hour) | 1.433 | 13.895 | 0.402 | 5.22 | 0.162 | 1.136 | 0.661 | 5.415 |
| Total Number of Tasks | 405 | 4005 | 62 | 229 | 33 | 118 | 115 | 313 |
| Number of Tasks at a Stage | 1—100 | 1—1000 | 1—32 | 1—124 | 1—32 | 1—118 | 6—18 | 6—60 |
| Average Task Execution Time of a Stage (Unit:Second) | 1—54.88 | 1—57.57 | 2—13.24 | 1.05—14.89 | 2—7.3 | 3—8.43 | 5.28—21.5 | 26.61—166.18 |
| Types of Tasks | short/medium/long | short/medium/long | short/medium | short/medium | short | short | short/medium | medium/long |

Table I
**Example Workflows Used in the Experiments**

**Observation** ①. This variability comes from the varying parallelism in a run and the load skew in a stage, and can be derived from the DAGs and runtime performance information. It motivates us to build *WIRE* as a DAG-aware resource management system.

### *B. Performance Variability: Across Runs*

For a stage in a workflow, the task execution times may vary significantly across runs. This effect undermines the value of the approaches that predict task execution times using the data collected from the historic runs.

First, different runs may process different datasets. Ernest [12] reports that the runs with different datasets vary in application runtimes and benefit from different resource provisioning plans. Similarly, we observe from the Hadoop and Pegasus WMS/HTCondor workflows (Table I) that, for the same stage across different workflow runs, its task execution times are highly variable.

Second, different runs may execute on different types of cloud resources. In this scenario, task execution times may vary from the previous runs since different instance types have different performance profiles (e.g., different bandwidth and capacity of memory and storage). For example, on Amazon EC2, different types of VM instances have different per-core network bandwidths [12]. Similarly, we observe from the ExoGENI [7] cloud sites that different types of VM instances have different per-core memory bandwidths.

Third, a task may take different execution times in different runs due to the contention or any other interference from the co-located loads in the runs. [13] reports that the interference from co-located applications may impact workflow runtimes significantly.

**Observation** ②**.** For a given workflow, its task execution times are highly variable across runs. The existing workflow-based resource management systems (e.g., Jockey [4], Apollo [3]) dealt with this variability by predicting the loads according to the job statistics collected from the previous runs. In contrast, *WIRE* predicts the upcoming loads with online information.

### *C. Predictive Control in WIRE*

Adaptation control and elastic resource management are longstanding problems. Our approach focuses specifically on the cost-effective elastic resource management for the computational workflows structured as task DAGs. We explain the structure of *WIRE* in terms of a MAPE (Monitor-Analyze-Plan-Execute) feedback loop [14], a widely used control model for autonomic and self-adaptive systems. In particular, the *WIRE* MAPE loop leverages the following workflow properties:

(1) *Workflow runs are managed and monitored.* In a workflow run, besides guarding the order of task executions, the workflow framework monitors the lifecycles of task executions. Like Hadoop, Spark and Pegasus WMS/HTCondor, the frameworks support counters, logs [6] and kickstarts to profile task executions for fault tolerance and user debugging. For each task, a framework collects its CPU time and start/end times, samples the memory usage over time, records input/output data sizes, etc. *WIRE* uses this information to predict task performance at runtime (§III-B1).

(2) *The load flows of a run are predictable.* A workflow run is executed according to its DAG that specifies the stage/task dependencies before the run starts. Jockey and other previous works simulate workflow runs based on the DAGs and task statistics from the previous runs to predict resource demands for the next run. In contrast, *WIRE* runs online workflow simulations to estimate the future resource demands (§III-B2) within a single execution, and iterates to refine the predictions as the task execution data becomes available.

(3) *Task executions are comparable.* For a stage in a run, the tasks share the same executable and stage dependencies. In *WIRE*, we predict task performance from the performance of the peer tasks from the same stages at runtime (§III-C).

## III. *WIRE*

### A. Overview

*WIRE* auto-scales the pool of cloud worker instances allocated to a workflow. Each worker instance is an IaaS VM or container instance with $l$ slots to run tasks. A task consumes a single slot of a worker instance for some period of occupancy to execute its computation and the associated I/O data transfer. At any time, *WIRE* hosts a single workflow run managed by a framework on a set of identically provisioned worker instances from a cloud site. We assume that the instances of a given type have identical performance, and that the cloud provider rents the instances of each type at some given price per fixed unit of time—a *charging unit* of length $u$.

From the view of *WIRE*, a workflow run executes over a sequence of equal-size time intervals, and each MAPE iteration plans the worker pool for one such interval. The pool changes only at the start of an interval. The lag time ($t$) is the time to institute a change to the worker pool; it is the maximum delay to launch or release (terminate) an instance in the underlying cloud system. Without loss of generality, we set the time between MAPE iterations to the lag time, and we presume that the change orders are timed so that they come into effect at the start of the next interval. Thus each MAPE iteration runs during the current interval to plan the pool for the start of the next interval, based on the measurement data collected from the current pool over the immediately preceding interval.

To address the high variability (**Observations ①, ②**), *WIRE* predicts the minimum (conservative) remaining slot occupancy time for each active task at the start of the target interval. For this purpose it builds a learning model for the tasks of a stage from online observations over peer tasks in the stage (§III-B1). In a stage, it presumes that the completed tasks are predictive of the others, and that the unstarted tasks are likely to run at least as long as the active (incomplete) tasks have already run. *WIRE* uses these conservative predictions to maintain a maximal set of instances that it expects to be highly utilized over at least the next charging unit.

### B. Architecture

Figure 1 depicts the *WIRE* architecture. In particular, *WIRE*'s MAPE loop runs as a daemon co-located on a slot with the framework master[1]. It consists of a *task predictor*, a *workflow simulator*, and a *cloud steering policy*, which

[1]For workflows across frameworks, a framework master runs as a process or a JVM instance on a slot of a cloud instance to manage a worker instance pool for workflow executions.

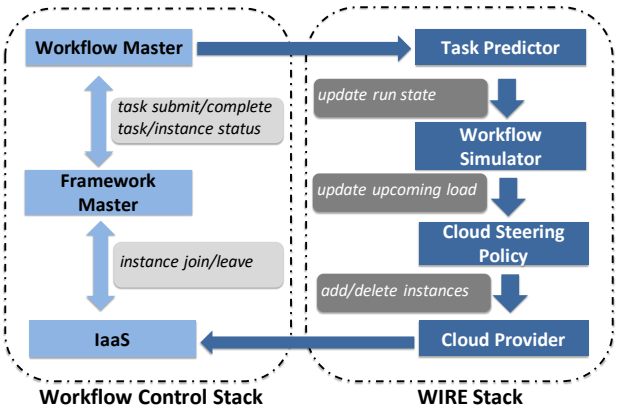


Figure 1. **WIRE Architecture**

in this paper is the resource-steering policy. Together they maintain a *run state* that tracks the worker instance pool and annotates the workflow DAG with the completed or predicted minimum execution times for a subset of tasks in the run, proceeding as a wavefront through the DAG as the workflow executes.

*1) Task Predictor:* At the start of an iteration, the task predictor harvests measurements from the previous interval by calling the framework-specific APIs. If a stage has incomplete and/or unstarted tasks, the task predictor collects the execution times (for completed tasks), run times (for running tasks), data transfer times (for running and completed tasks) and input data sizes (for all tasks). It uses this data to update the learning models to predict task occupancy times for the unstarted or incomplete tasks in each stage. Then it updates the run state with the prediction results.

A task occupies a slot for a period corresponding to the sum of its execution time and its data transfer times to read its input and write its output. The data transfer time is influenced by the size of the input dataset, data transfer patterns, transient interference (discussed in §II-B), and the resource configuration [12]. In this study, we focus on the data transfer of tasks on dynamic resources, where data transfer patterns, interference, and the number of instances vary over time. Accordingly, we presume a task's data transfer follows a memoryless distribution. We estimate the data transfer time for a task ($t_{i_{data}}$) according to the most recent observations: $\tilde{t}_{data}$, where $\tilde{t}_{data}$ represents the median of the data transfer times of the tasks between the $n-1$th and $n$th MAPE iterations.

*2) Workflow Simulator:* The workflow simulator takes as input DAG, updated task states, estimated task occupancy times, and the current resource allotment (worker instance set). It simulates workflow execution over the next interval of length $t$. The output is the *upcoming load*: a set of tasks $Q_{task} = \{t_0, t_1, ..., t_i, ..., t_{n-1}\}$ that are expected to be active at the start of the target interval, where $t_i$ is the

predicted minimum remaining slot occupancy time for task $i$. It also records the sunk cost to run each active task up until the interval's start, i.e., the cost to restart the task at that time. The restart cost ($c_j$) for an instance $j$ is the maximum restart cost of all tasks running on slots of the instance.

*3) Cloud Steering Policy:* Given the upcoming load $Q_{task}$, the resource-steering policy estimates the ideal size $p$ for the worker pool at the start of the next execution interval. It adds a new instance when the models predict the load to be above a target threshold for at least the next charging unit. It avoids wasteful expenditures by releasing an instance when a charging unit is about to expire (which would incur a recharge cost) and there is not sufficient confidence that the workflow can continue to use it efficiently. When the policy decides to reduce the pool size, it also selects the instances to terminate to minimize task restart costs.

### C. Algorithms for Task Prediction

We introduce five heuristics to estimate the task occupancy at a stage according to different scenarios with the varying availability of the runtime information from the stage. These heuristics combine three design goals: (1) Predict task performance conservatively when little information is available at the start of a stage (Policy 1 and 2). (2) Use the median observations over a sequence of execution intervals (*moving median*) to address the longer-term and more-consistent trends of the task performance at each stage. (3) Compensate for the load skew among the tasks at a stage with an online gradient descent model (Policy 5).

For an incomplete/unstarted task $i$ at stage $S_j$, we predict its execution time ($t_i$) based on the data from peer tasks at $S_j$ by one of the five policies given below. In the policies, we take the median values of task execution times. Compared to the mean and the three-sigma rule [15] ), the median is more effective to capture "the middle performance" of skewed data distributions (e.g., Zipfian), which are widely observed in cloud loads [11], [16].

**Online Prediction Policies**

For an incomplete/unstarted task $i$ at stage $S_j$, its estimated minimum execution time is $t_i$:

(1) No task at $S_j$ starts: $t_i$=0.
(2) $S_j$ has running tasks and no completed task: conservatively presume that the running tasks are about to complete. Update $t_i = \tilde{t}_{run}$, where $\tilde{t}_{run}$ is the median run time of the running tasks at $S_j$.
(3) $S_j$ has completed tasks and task $i$ has not started (e.g., its input data is not available): update $t_i = \tilde{t}_{complete}$, where $\tilde{t}_{complete}$ is the median execution time of the completed tasks at $S_j$.
(4) $S_j$ has completed tasks, task $i$ is ready-to-run, its input data size is equivalent to the input size of a group of completed tasks ($L$) at $S_j$: update $t_i = \tilde{t}_L$, the median completion time of tasks in $L$.
(5) $S_j$ has completed tasks, task $i$ is ready-to-run, and its input data size is new to the input sizes of all completed tasks at $S_j$: estimate $t_i$ with an online gradient descent model built for $S_j$.

**Online gradient descent method** is a machine learning technique, introduced to train models with continuous data streams. *WIRE* adopts this method to address the dynamism of task completions at stage level. For each stage in a run, we build an online gradient descent model and update the models and their training sets over a sequence of intervals (MAPE loop iterations). We build the prediction problem as a linear system with the cost $O(N)$, $N$ is the number of tasks of a stage. In particular, we choose task input data size as the feature; for task $i$ at $S_j$ with input data size $d_i$, its expected execution time ($t_i$) in the $n$th interval is:

$$t_i = \alpha_{0_n} + \alpha_{1_n} \times d_i \quad (1)$$

We take $\alpha_{0_0}$=0, $\alpha_{1_0}$=0 as the initial state in the 1st loop. Algorithm 1 shows the process to update the model for stage $S_j$ in the $n$th interval. In the process, $\langle d_M, t_M \rangle$ is a data point in the training set, which represents a group of completed tasks ($M$) at $S_j$ with the same input data size $d_M$ and the expected execution time $t_M$, $t_M = \tilde{t}_M$. $\tilde{t}_M$ represents the median of the completed tasks in $M$. $\alpha_{0_{n-1}}$ and $\alpha_{1_{n-1}}$ are the coefficients derived from the algorithm in the $n-1$th interval.

**Algorithm 1** Prediction with online gradient descent method

**for Stage $S_j$ in the $n$th MAPE loop**
**Training Set:** $\langle d_1, t_1 \rangle, \langle d_2, t_2 \rangle, .., \langle d_X, t_X \rangle, .., \langle d_M, t_M \rangle$
**Input Parameters:** $\alpha_{0_{n-1}}, \alpha_{1_{n-1}}$
**Learning rate:** 0.1
$\alpha'_0 = 0$
$\alpha'_1 = 0$
**for** $\langle d_X, t_X \rangle$ in training set **do**
    $\alpha'_0 += -\frac{2}{M} \times (t_X - ((\alpha_{1_{n-1}} \times d_X) + \alpha_{0_{n-1}}))$
    $\alpha'_1 += -\frac{2}{M} \times d_X \times (t_X - ((\alpha_{1_{n-1}} \times d_X) + \alpha_{0_{n-1}}))$
**end for**
$\alpha_{0_n} = \alpha_{0_{n-1}} - 0.1 \times \alpha'_0$
$\alpha_{1_n} = \alpha_{1_{n-1}} - 0.1 \times \alpha'_1$
return $\alpha_{0_n}, \alpha_{1_n}$

It is clear that, for task performance prediction, when a stage has more completed tasks, the prediction results are more likely to be accurate. To obtain such stages for the predictor quickly, *WIRE* dispatches the first five ready-to-run tasks to fire in a stage with high priority. These tasks often run before the final tasks of predecessor stages or of other

**Algorithm 2** Resource-steering policy

1: **Upcoming Tasks:** $Q_{task} = \{t_0, t_1, ..., t_i, ..., t_{n-1}\}$
2: **Current Instances:** $P_{current} = \{s_0, s_1, ..., s_j, ..., s_{m-1}\}$
3: **Times to Next Charge:** $r_0$, $r_1$,...,$r_j$, ...,$r_{m-1}$
4: **Task Restart Cost:** $c_0$, $c_1$, ... , $c_j$, ..., $c_{m-1}$
5: **Charging Interval:** $u$
6: $p$ = resizePool() by Algorithm 3
7: **if** $p > m$ **then**
8: request $p - m$ new instances
9: **else if** $p < m$ **then**
10: **for** $s_j$ in $P_{current}$ **do**
11: **if** $r_j \leq t$ and $c_j \leq 0.2u$ **then**
12: terminate $s_j$, resubmit the running tasks on $s_j$
13: $P_{current}.remove(s_j)$
14: **end if**
15: **if** $size(P_{current}) == p$ **then**
16: break
17: **end if**
18: **end for**
19: **end if**

**Algorithm 3** Resizing the worker pool (instance set)

1: **Upcoming Tasks:** $Q_{task} = \{t_0, t_1, ..., t_i, ..., t_{n-1}\}$
2: **Charging Interval:** $u$
3: **Number of Task Slots Per Worker Instance:** $l$
4: **Output: Number of Planned Instances:** $p$=0
5: $T_{used}$=0, $slot_{used} = \emptyset$
6: **while** $size(Q_{task}) > 0$ **do**
7: **while** $size(slot_{used}) < l$ and $size(Q_{task}) > 0$ **do**
8: $task = Q_{task}.poll()$
9: $slot_{used}.add(task)$
10: **end while**
11: **if** $size(slot_{used}) == l$ **then**
12: $t_{min} = min(slot_{used})$
13: $T_{used} += t_{min}$
14: **if** $T_{used} \geq u$ **then**
15: $p$+=1
16: $T_{used}$=0, $slot_{used} = \emptyset$
17: **else**
18: **for** $t_c$ in $slot_{used}$ **do**
19: **if** $t_c == t_{min}$ **then**
20: $slot_{used}.remove(t_c)$
21: **else**
22: $t_c -= t_{min}$
23: **end if**
24: **end for**
25: **end if**
26: **end if**
27: **end while**
28: **if** $p = 0$ or $max(slot_{used}) > 0.2u$ **then**
29: $p += 1$
30: **end if**

stages that are active concurrently. This approach works well for online prediction and resource-efficient workflows that provides the performance data for more stages.

### *D. Algorithms for Cloud Workflow Steering*

Algorithm 2 summarizes the resource-steering auto-scaling policy. It uses Algorithm 3 to determine the ideal size of the worker pool (instance set) at the start of the next interval, then forms requests to grow or shrink the current pool to the target size as needed.

The input to Algorithm 3 is the upcoming load of $n$ tasks ($Q_{task}$) that are projected to be active at the start of the next interval, and their conservatively predicted minimum remaining occupancy times. Algorithm 3 assumes that $Q_{task}$ is non-empty, else it retains a minimal pool until the next control iteration, or the workflow terminates. The output of Algorithm 3 is a pool size $p$, the desired number of instances in the worker instance set for the next interval. It greedily assigns tasks to allocated instances until all upcoming tasks are consumed and all instances are fully utilized for at least the next charging unit $u$. If an instance receives a task that is predicted to complete before the charging unit expires, lines 17-23 update the instance state to the time of the task completion; the loop at line 6 then assigns another task until the instance is filled.

Algorithm 2 then compares $p$ to the size ($m$) of the current pool, and plans adjustments to the pool to take effect at the start of the next interval. If the pool should shrink ($p < m$), then it selects the current instances to release based on their times to next charge ($r_0, r_1, ..., r_j, ..., r_{m-1}$) and task restart costs. It releases an instance only if the charging unit expires before the start of the next execution interval ($r_j < t$) and the restart cost $c_i$ is below a target threshold. The target is arbitrarily chosen as $0.2u$ in the Algorithms and experiments, but is freely configurable. The restart cost $c_i$ is the maximum sunk cost (consumed slot occupancy time as of the start of the next interval) of any task assigned to a slot on instance $i$.

The elastic control policies in *WIRE* project the assignment of tasks to instance slots based on the expected scheduling algorithm (e.g., FIFO). The policy controller's predicted assignment of tasks to instances might differ from the true schedule selected by the framework master. The experiment results (§IV-E) show that the *WIRE* approach obtains high resource utilization across the sample workflows, datasets and resource charging units, suggesting that the effect of any drift from the predicted assignments is minor.

### *E. Discussion*

We consider a simple case to illustrate how the scaling algorithm works. Consider a simple workflow that executes a

sequence of stages and every task is a predecessor of all tasks in the next stage, if any, and that all tasks in a stage have the same run time $R$. Thus all tasks in each stage fire at the same time, and all tasks active at any given time are from the same stage. Without loss of generality, suppose further that the monitoring is continuous, control is instantaneous, and there is one slot per instance. In fact, the resource-steering policy does not depend on any of these assumptions.

Consider any stage in this linear workflow. Suppose it has $N$ tasks starting with an instance pool size $P$ and the charging unit $U$. Then after $U/N$ time units the algorithm predicts that the $N$ tasks of the stage will consume an entire instance-unit of resource. If the active tasks keep running, it launches a new instance after $(P+1)(U/N)$ time units and then every $U/N$ time units after that.

Suppose $R > U$. At time $U$, no task has terminated and the pool has $N$ instances. Then, after $R$ time units, an initial group of $P$ tasks completes. The algorithm then predicts (correctly) the completion time of all tasks in the stage. From this point it may shrink the pool as the tasks of the stage complete every $U/N$ time units. The algorithm no longer grows the pool, because nothing is known about the next stage until this stage completes. Then, the cycle repeats for the next stage.

The behavior is similar for $R \leq U$. At time $R$, $P$ tasks complete. At this point, there are $R/(U/N)$ running instances in the pool, and the scaling algorithm gains correct predictions for all tasks in the stage. It resizes the pool to fully use the remaining paid time of all running instances, and determines to launch or terminate instances based on their utilization. The algorithm continues until the last task of this stage completes. It then moves to the next stage.

For example, consider the case with $P = 1$ and $R = U - \epsilon$ ($R \leq U$). The first task completes just before the $N$th instance is launched, for a peak of $N - 1$ instances. Each instance expires and is released immediately after its task completes. The tasks of the next stage fire when the last task completes with $N = 1$, at the start of the next charging unit. All instances are fully utilized, and the stage completes within time $2R = 2U$. In this scenario, the algorithm has optimal efficiency—nothing is wasted—and performance is within a factor of two of optimal.

Alternatively, consider the case with $P = 1$ and $R = U+\epsilon$ ($R > U$). The $N$th instance is launched at time $R = U$, just before the first task completes. The controller renews each instance just before its task completes: it cannot release these instances because the sunk cost to restart a task is too high. The last task completes at time $2R+\epsilon$, firing the next stage with $P = N$ instances. After that, one instance comes up for renewal every $U/N$ time units. The controller releases these instances unless and until it has sufficient confidence that the demands of the next stage justify renewing them.

For these examples, the cost is the same as non-wasteful static provisioning. Performance is a factor of $N/2$ better than static provisioning with $P = 1$, but only half as good as it is with $P = N$. We use simulations to explore a wider range of parameter settings in §IV-A.

### F. Implementation

We modified 2151 lines of Python code in Pegasus for the task predictor (1058 lines), resource-steering policy (87 lines), workflow simulator (779 lines), and the cloud-side messaging protocol (227 lines). We add 94 lines of C++ code in Condor to prioritize the first five ready-to-run tasks for each stage at run time. We add/modify 706 lines of Java code for the ExoGENI client to support the messaging protocol.

## IV. Experiments

This section evaluates the performance and efficiency of *WIRE*. We first explore the performance bounds using the simulation on a range of linear workflows to gain the intuition about the algorithm and its limitations (§IV-A). We then present the experimental results with the real and synthetic workflows running on the ExoGENI cloud with elastic provisioning under the *WIRE* prototype. We describe the experiment setting (§IV-B) and test workflows and configurations (§IV-C), and then evaluate the prediction accuracy (§IV-D), resource cost (§IV-E), and overhead (§IV-F) relative to the purely reactive approaches and static provisioning.

### A. Simulation with Linear Workflows

To illustrate the behaviors and boundaries of the scaling algorithm, we evaluate its performance under various load and resource settings by simulation for the class of simple linear workflows discussed in Section III-E. Specifically, for a single stage, we analyze the algorithm's resource usage (cost) and stage completion time (speed) for varying $N$, $R$ and $U$. We compare the results to the stage's best possible performance on the two metrics. A stage achieves its optimal resource usage $NR/U$ when the tasks of the stage execute sequentially on a fixed-size pool with a single instance ($P = 1$). A stage achieves its optimal completion time $R$ when all tasks of the stage execute in parallel on a fixed-size pool with $N$ instances ($P = N$).

We perform simulations on the scaling algorithm for two cases, $R > U$ and $R \leq U$. Figures 2 and 3 present the simulation results for those two cases respectively. In each case, we analyze the performance on three workload scales: small-scale (10 tasks), medium-scale (100 tasks) and large-scale (1000 tasks). We vary $R/U$ (in Figure 2) and $U/R$ (in Figure 3).

Figure 2 suggests that, for $R > U$ and across load scales, the performance of our scaling algorithm is bounded to $1.33\times$ and $1.67\times$ of the optimal performance for resource usage and completion time, respectively. Moreover, with the growth of $R/U$ , the algorithm tends to deliver better performance and approximates the optimal performance for both

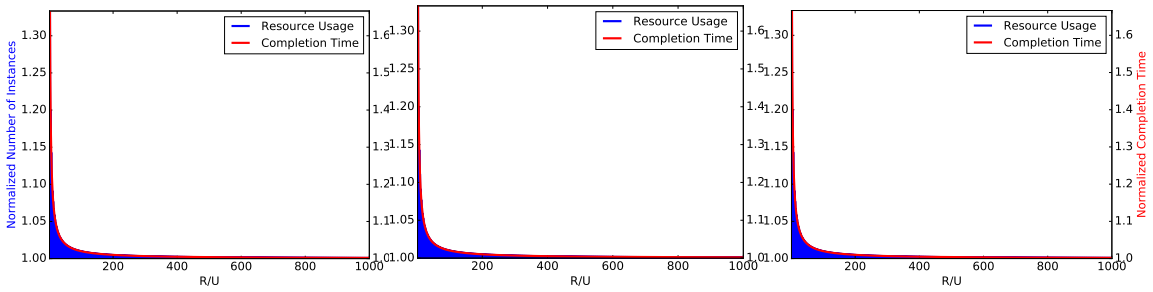


Figure 2. **Performance of Resource Steering Policy** for the use cases with $R > U$. From left to right, the 3 subfigures present the results of $N = 10$, $N = 100$, and $N = 1000$, respectively. In each subfigure, $R/U$ grows along the x-axis. The two y-axis represent the ratios of the policy's resource usage (left) and completion time (right) to the best performance. Here, $N$ is the number of tasks at a stage, $R$ is the run time of the $N$ tasks, and $U$ is the charging unit.

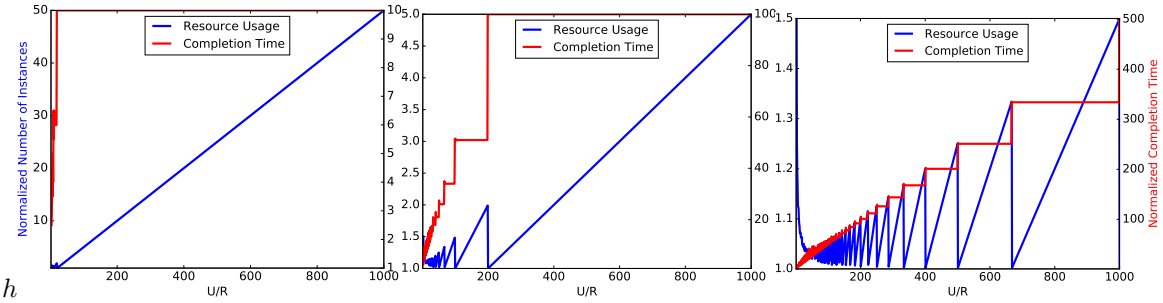


Figure 3. **Performance of Resource Steering Policy** for the use cases with $R \leq U$. From left to right, the 3 subfigures present the results of $N = 10$, $N = 100$, and $N = 1000$, respectively. In each subfigure, $U/R$ grows along the x-axis. The two y-axis represent the ratios of the policy's resource usage (left) and completion time (right) to the best performance. Here, $N$ is the number of tasks at a stage, $R$ is the run time of the $N$ tasks, and $U$ is the charging unit.

resource usage and completion time at the extreme cases ($R/U \geq 400$). It suggests that the scaling algorithm delivers good performance with low resource cost for $R > U$, at least for this idealized group of workflow scenarios. Moreover, it is possible to modulate the aggressiveness of the heuristic to obtain a selected balance of cost and speed, e.g., by modulating the target utilization level.

However, the heuristic is far less effective when the charging interval is long relative to task runtimes, when the agility of elastic resource allocation is inherently limited. Figure 3 shows that, for $R \leq U$, when $U/R$ varies in 1 — 1000, the scaling algorithm may deviate widely from optimal behavior along either metric, depending on the specific scenario.

### B. Experiment Setup

We execute experiments on the ExoGENI network cloud [7]. An experiment is on an ExoGENI site and has 1 — 12 worker instances (the max number of the worker instances a site can provide). An instance is an XOXLarge ExoGENI VM instance and can host up to four concurrent tasks at a time. We consider the resource charging unit ($u$) as: 1, 15, 30, 60 minutes according to the conventional numbers in production use. In our experiments, the VM instantiation time is $\sim$ 3 minutes (the lag time), which we also use as the time-intervals between MAPE loops in a run.

### C. Test Workflows and Configurations

*1) Sample workflows:* We examine *WIRE*'s performance on Hadoop and Pegasus WMS/HTCondor workflows with four samples, including a computational workflow (Epigenomics) [17], two database benchmarks from TPC-H [18], and a big data benchmark from Intel HiBench [19].

(1) *Epigenomics* is a scientific workflow that simulates various genome sequencing operations. It is used by the USC Epigenome Center to process the data of DNA methylation and histone modification.

(2) *TPC-H* is a benchmark suite designed to address ad-hoc queries and concurrent data modifications. It is representative for the database load in production clusters. We examine *WIRE* on TPCH-1 and TPCH-6.

(3) *Intel Hibench* is a big data benchmark suite that measures workflow frameworks and tools with main-stream loads in cloud computing. We measure *WIRE* with PageRank.

Table I presents the sample workflows in detail. For each workflow, we experiment on two datasets.

*2) Workflow transformation.:* To process and measure the Hadoop runs on Pegasus WMS/HTCondor, we implement *task emulator* and transform Hadoop DAGs to Pegasus

WMS/HTCondor DAGs. In a run, task emulator behaves as if it runs task executables. It reads the performance records of Hadoop tasks and consumes the amount of resources according to the records. Moreover, we modify Hadoop (v2.7.2) to collect the task dependencies for each run and repeat them in the corresponding Pegasus WMS/HTCondor DAGs used in the experiments.

*3) Workflow runs:* We run the sample workflows with four types of resource management settings: 1) Static settings with 12 VM instances. Relative to Section IV-B, these settings host workflows with the maximum number of worker instances. We call the sample runs on these settings *full-site runs*. 2) Elastic settings managed by *WIRE*. We call the runs on these settings *wire runs.* 3) Elastic settings ruled by the active tasks. At run time, the capacities of these settings vary according to the number of idle/running tasks. We call the runs on these settings *pure-reactive runs*. 4) Elastic settings ruled by the active tasks and the resource steering policy. At run time, we predict the load according to the number of idle/running tasks and add/delete resources according to the resource steering policy (§III-B2). We call the runs on these settings *reactive-conserving runs*.

For a setting hosting a wire, pure-reactive, or reactive-conserving run, we monitor and vary the resource setting on every 3 minutes (§III-B2 and §IV-B). For all workflow runs on all 4 types of the settings, we repeat each run on a setting for 3—7 times.

### D. Task Performance Prediction

The next section evaluates the online prediction policies (§III-B1). We focus on the stages with two or more tasks (overall 45 such stages in Table I) and measure the prediction results for these stages with Policies 3), 4) and 5). We leave the efficiency analysis for all 5 policies in Section IV-E.

To address the policy performance on different task orders (§III-B1), for a stage in a run on a setting [2], we predict task execution times with 5 randomly-chosen task orders. Plus the 3 — 7 repetitions (§IV-C) for a run on a setting, for each of the target 45 stages, we report 225 — 315 prediction results.

To evaluate the policy effectiveness on different task execution times, we categorize the 45 stages into 3 types based on the average task execution time ($\bar{\mu}$) of a stage, including short-task stages ($\bar{\mu} \leq$10 seconds), medium-task stages (10 $< \bar{\mu} \leq$30 seconds), and long-task stages ($\bar{\mu} >$30 seconds). In summary, the 45 stages include 16 short stages, 18 medium stages and 11 long stages respectively; across workflows, datasets, experiment settings and task orders, we report the prediction results of 4030, 3900 and 2420 short, medium and long stages respectively.

Figure 4 presents the CDFs of the prediction errors for Epigenomics, TPCH-1, TPCH-6 and PageRank. For short and medium tasks, an execution prediction error of even a few seconds can result in a large difference in resource scheduling. Therefore, we report *true error* [3] for short and medium stages, and *relative true error* [3] for long stages.

In summary, the 4 sample workflows present a good coverage on task execution times in production runs. As shown in Figure 4, the prediction results are highly accurate: 1) For a task, the average prediction error is $\leq 0.1$ and $\leq 2.15$ seconds for short and medium tasks, $\leq 13.1\%$ for long tasks. 2) for a stage, on average 93.18% and 79.4% of tasks at the stage report $\leq 1$ second prediction error for short and medium stages, 83.19% of tasks at the stage report under 15% prediction error for long stages.

Moreover, the prediction results also report the task-order issue (§III-B1). It shows that, for a stage across task orders, 29 out of 34 ($16+18$) short and medium stages report $\leq 1.8$ seconds error difference; 8 out of 11 long stages report $\leq 15.2\%$ error difference. We look into the outlier stages (e.g., a short stage in PageRank S) and find that, the parallelism within these stages are low (5 — 17 tasks on a stage). At the same time, we also notice that these outlier stages present low prediction accuracy with some task orders. We conclude that: 1) Online prediction may provide imperfect estimates with insufficient observations. 2) relative to the low cost of wire runs (§IV-E), it suggests that *WIRE* can achieve low cost with this imperfection.

### E. Resource Cost and Execution Time

This section evaluates *WIRE* on resource cost and efficiency. We compare *WIRE* with the other settings (defined in §IV-C) on both resource cost and end-to-end performance.

Figures 5 and 6 report the results for *resource cost* and *relative execution time* respectively. For resource cost, we report the number of resource charging units used to complete a run for a worklfow on a dataset. For relative execution time, we first consider that, all execution times of a workflow on a dataset are comparable, then normalize the times across settings and resource charging units to the best performance.

The results suggest that: 1) Wire runs report the lowest resource cost in most cases among the reported settings. In summary, for the same workflow with the same dataset, the resource cost on the other policies is $0.93\times$ — $14.66\times$ of the cost of wire runs. Specifically, reactive conserving spends $0.93\times$ , $0.94\times$ and $0.96\times$ of charging units to wire runs do on Epigenomics L, TPCH-1 L, and TPCH-6 L, all for the charging unit 1 minute. Relative to their execution times in Figure 6 where *WIRE* delivers shorter execution times than resource-conserving policy does, we conclude that, for small charging units *WIRE* prioritizes application execution times over cost. In general, *WIRE* can effectively consume

[2] A setting is 1 out of 16 experiment settings ($4 \times 4$) across full-site, pure-reactive, reactive-conserving, wire and 4 resource charging units.

[3] For a task with execution time ($t$) and estimated time $t'$, the true error is: $t' - t$; the relative true error is: $\frac{t'-t}{t}$.

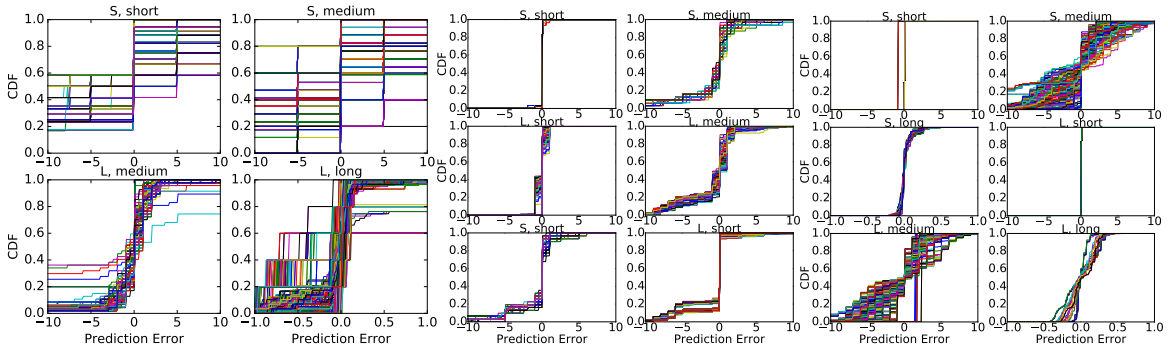


Figure 4. **CDFs of Task-performance Prediction Error** for the 4 sample workflows. A subfigure presents the results of the short, or medium or long stages of a workflow with a dataset; a line in a subfigure presents the CDF of the task-performance prediction errors of a stage from a specific experiment setting (§IV-C), with a specific resource charging unit and on a specific task-order (§III-B1 and §IV-D). Across subfigures, the x-axis represent s true error[3] in $[-10, 10]$ seconds for short and medium stages, relative true error[3] in $[-1, 1]$ for long stages respectively.

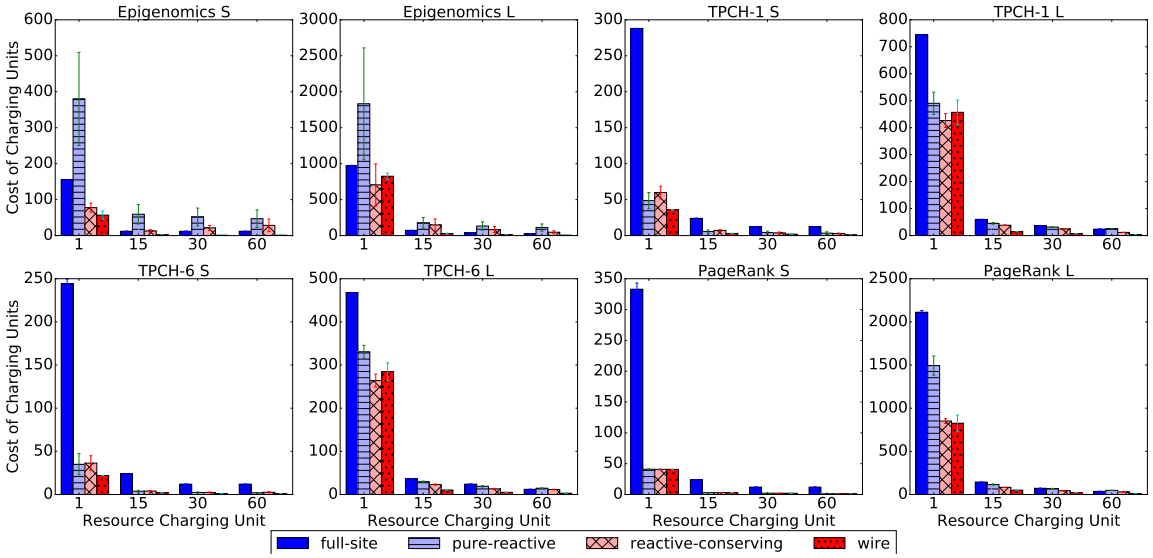


Figure 5. **Resource Cost**. Each subfigure reports the number of resource instances used of a workflow on a dataset across settings and resource charging units (§IV-C). Across subfigures, we report the mean and std of resource cost, the x-axis represents the 4 charging units: from 1 minute to 60 minutes.

IaaS-cloud resources. 2) *WIRE* presents low performance degradation. Figure 6 shows that, wire runs report 1.02× — 3.57× slowdown; for 1-minute resource charging unit, its slowdown ranges in 1.02× — 1.65×. Compared to the best-performance runs (full-site runs), wire runs deliver 4.93× — 14.66× lower resource cost. It suggests that, *WIRE* obtains reasonably good performance with low resource cost, specifically for small resource charging units. 3) the *WIRE* solution is robust to imperfect prediction. For all sample workflows, there exist stages with 1—6 tasks, at which the prediction accuracy is more likely to be low. Moreover, for PageRank S, the imperfect prediction occurs on 41.2% of all stages. Relative to the lowest cost of wire runs across the sample workflows, we conclude that, *WIRE* can capture and apply the observed performance variations within a stage agilely, which is sufficient to attain low cost even with imperfect prediction.

### F. Overhead

We evaluate overall 127 wire runs (§IV-C) with 4 types of resource charging units (§IV-B). For all such runs *WIRE* uses ≤16KB memory, and consumes 0.011% — 0.49% of the aggregate task execution time (given in Table I) in each run. We conclude that the overhead of *WIRE* is low for both memory usage and time cost.

## V. Related Work

*Resource management on private clusters*. Mesos [20], Omega [21], Borg [22] and Yarn [23] are cluster schedulers that allocate private cloud resources for workflows across frameworks. DRF [24] and [25] address the resource fairness issue in shared private clusters. Paragon [26] and Quasar [27] investigate the job co-location issue: they use

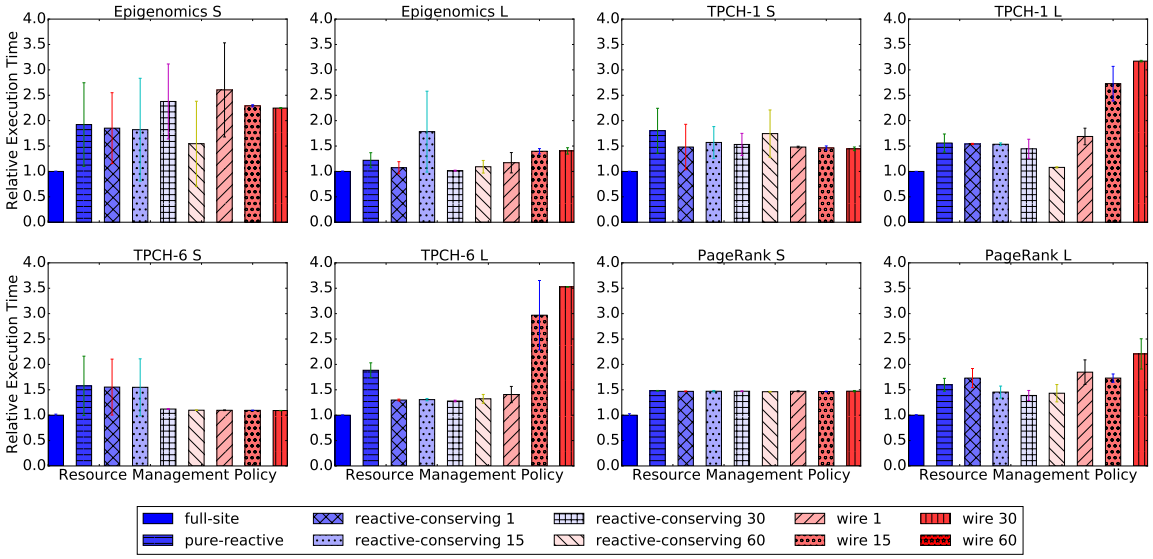


Figure 6. **Relative Execution Time**. Each subfigure reports the relative execution times of a workflow on a dataset across settings and resource charging units (§IV-C). Across subfigures, we report the mean and std of relative execution times; each x value represents a policy on a charging unit, e.g., 'wire 1' meaning the wire runs with charging unit 1 minute, 'resource-conserving 15' meaning the resource-conserving runs with charging unit 15 minutes.

machine learning techniques to classify workloads and further use it to determine their best locations. Compared to the studies in this category, *WIRE* addresses elastic resource management for many-task workloads with high resource utilization in an IaaS scenario, in which the task scheduler runs a *single* workflow on an elastic pool of dedicated (virtual) worker instances that cost money: the goal is to obtain the fastest runtime consistent with bounding waste to a target level as the parallelism of the workflow varies. More recently, Khorasani et. al. [28] proposed to adapt the number of executors between the CPU- and I/O- bound phases in Spark-based applications. Compared to *WIRE*, they also employed MAPE feedback loops to manage resource adaptation with online information, but tune the thread pool accordingly without the prediction about the future load.

*Managing dynamic resources on IaaS*. Researchers developed auto-scalers [29]–[31] to manage elastic virtual infrastructures for cloud-based web services and database applications. They use analytical models to predict load and build policies to add/delete VM instances for the instantaneous resource needs. Fox [32] serves as a mediator between cloud providers and autoscalers. Iosup et al. [33] also identified the performance variations on Amazon AWS and Google App Engine. In particular, they derive performance indicators (e.g. response time) from performance traces and relative statistics (quartiles, mean, one derivative) from the cloud provider viewpoint. Compared to these works, we address elasticity control for the case of task-DAG workflows, whose resource needs vary as the workflow executes according to a declared workflow DAG structure that is known to the auto-scaler. Ilyushkin et al. [34] takes an experimental approach to evaluate the performance of auto-scalers for workflows. *WIRE* is distinct from those approaches in that *WIRE* uses online measurements to predict task performance, and plans resource adaptation for the efficient use of resources based on resource charging unit.

## VI. Conclusion

This paper introduces *WIRE* that maintains workflow states and predicts near-term resource demand in a wavefront ahead of the workflow execution. It uses online predictions to proactively adapt an elastic worker pool. We show that this approach enables elastic resource allocation on a *resource-steering* heuristic that grows the pool for faster execution while bounding waste and cost.

## Acknowledgment

We thank Mert Cevik for his help on arranging the experiments on the ExoGENI cloud. This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. This research used the Pegasus Workflow Management Software funded by the National Science Foundation under grant #1664162. Experiments in the paper used the ExoGENI testbed supported by the National Science Foundation's GENI initiative. Work was also supported by the NSF CC-NIE ADAMANT grant (Award #1245926).

## REFERENCES


[1] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI'10*, 2010.

[2] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao, "Morpheus: Towards automated SLOs for enterprise clusters." in *OSDI'16*, 2016.

[3] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing." in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[4] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *EuroSys'12*, 2012.

[5] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, 2005.

[6] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience," *Concurrency and computation: practice and experience*, 2005.

[7] I. Baldine, Y. Xin, A. Mandal, P. Ruth, C. Heerman, and J. Chase, "ExoGENI: a multi-domain infrastructure-as-a-service testbed," in *TridentCom'12*, 2012.

[8] W. Chen, R. F. da Silva, E. Deelman, and R. Sakellariou, "Using imbalance metrics to optimize task clustering in scientific workflow executions," *Future Generation Computer Systems*, vol. 46, pp. 69–84, 2014.

[9] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing reducer skew in MapReduce workloads using progressive sampling," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC'12)*, San Jose, CA, 2012, pp. 16–24.

[10] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman, "Upper and lower bounds on the cost of a Map-Reduce computation," in *VLDB'13*, 2013.

[11] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in MapReduce applications," in *SIGMOD'12*, 2012.

[12] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *NSDI'16*, 2016.

[13] B. Farley, V. Varadarajan, K. Bowers, A. Juels, T. Ristenpart, and M. M. Swift, "More for your money: exploiting performance heterogeneity in public clouds," in *SoCC'12*, 2012.

[14] IBM, "An architectural blueprint for autonomic computing," *IBM White Paper*, 2006.

[15] F. Pukelsheim, "The three sigma rule," *The American Statistician*, 1994.

[16] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *NSDI'15*, 2015.

[17] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

[18] TPC, "TPC-H Benchmark," http://www.tpc.org/tpch/.

[19] intel-hadoop, "HiBench," https://github.com/intel-hadoop/HiBench.

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *NSDI'11*, 2011.

[21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *EuroSys'08*, 2013.

[22] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys'15*, 2015.

[23] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O. Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: yet another resource negotiator," in *SOCC'13*, 2013.

[24] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: fair allocation of multiple resource types." in *NSDI'11*, 2011.

[25] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multiresource allocation: fairness-efficiency tradeoffs in a unifying framework," *TON*, 2013.

[26] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *ASPLOS'13*, 2013.

[27] ——, "Quasar: resource-efficient and QoS-aware cluster management," in *ASPLOS'14*, 2014.

[28] S. O. Khorasani, J. S. Rellermeyer, and D. Epema, "Self-adaptive executors for big data processing," in *Middleware'19*, 2019.

[29] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications." in *ICAC'14*, 2014.

[30] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: elastic distributed resource scaling for Infrastructure-as-a-Service," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*, San Jose, CA, 2013, pp. 69–82.

[31] A. V. Papadopoulos, A. Ali-Eldin, K.-E. Årzén, J. Tordsson, and E. Elmroth, "Peas: a performance evaluation framework for auto-scaling strategies in cloud applications," *TOMPECS*, 2016.

[32] V. Lesch, A. Bauer, N. Herbst, and S. Kounev, "FOX: Cost-awareness for autonomic resource management in public clouds," in *ICPE'18*, 2018.

[33] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *CCGrid'11*, 2011.

[34] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. Bauer, A. V. Papadopoulos, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscalers for complex workflows," *ToMPECS'18*, 2018.