Enhancing Reliability of Workflow Execution Using Task Replication and Spot Instances

DEEPAK POOLA, KOTAGIRI RAMAMOHANARAO, and RAJKUMAR BUYYA,
The University of Melbourne

Cloud environments offer low-cost computing resources as a subscription-based service. These resources are elastically scalable and dynamically provisioned. Furthermore, cloud providers have also pioneered new pricing models like spot instances that are cost-effective. As a result, scientific workflows are increasingly adopting cloud computing. However, spot instances are terminated when the market price exceeds the users bid price. Likewise, cloud is not a utopian environment. Failures are inevitable in such large complex distributed systems. It is also well studied that cloud resources experience fluctuations in the delivered performance. These challenges make fault tolerance an important criterion in workflow scheduling. This article presents an adaptive, just-in-time scheduling algorithm for scientific workflows. This algorithm judiciously uses both spot and on-demand instances to reduce cost and provide fault tolerance. The proposed scheduling algorithm also consolidates resources to further minimize execution time and cost. Extensive simulations show that the proposed heuristics are fault tolerant and are effective, especially under short deadlines, providing robust schedules with minimal makespan and cost.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Algorithms, Reliability, Performance, Experimentation

Additional Key Words and Phrases: Fault tolerance, workflows, cloud, scheduling, spot instances, task duplication, task retry

ACM Reference Format:
Deepak Poola, Kotagiri Ramamohanarao, and Rajkumar Buyya. 2016. Enhancing reliability of workflow execution using task replication and spot instances. ACM Trans. Auton. Adapt. Syst. 10, 4, Article 30 (February 2016), 21 pages.
DOI: http://dx.doi.org/10.1145/2815624

1. INTRODUCTION

Scientific workflows are used to compose and execute computational tasks that are control and/or data dependent. They are used to perform high throughput computing and data analysis [Lifka et al. 2013]. Numerous disciplines use scientific workflows to perform large scale complex analyses. Workflows enable scientists to easily define computational components, data, and their dependencies in a declarative way. This makes the workflows easier to execute automatically, improving the application performance,

This work is partially supported by Linkage and Discovery Project grants supported by ARC (Australian Research Council).
Authors' addresses: D. Poola, K. Ramamohanarao, and R. Buyya, Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Parkville, VIC 3010, Australia; email: deepakc@student.unimelb.edu.au, {kotagiri, rbuyya}@unimelb.edu.au.

DOI: http://dx.doi.org/10.1145/2815624

and reducing the time required to obtain scientific results [Deelman et al. 2013; Juve and Deelman 2010].

Scientific workflows are increasingly moving toward cloud computing, which offers low-cost computing resources (e.g., networks, servers, and storage) as a subscription-based service. These resources are elastically scalable, dynamically provisioned, and delivered in a transparent manner without manual intervention [Sun et al. 2013]. As a result, a large number of organizations are rapidly adopting cloud computing [Lifka et al. 2013].

Although scheduling scientific workflows on cloud will immensely reduce cost and makespan, cloud computing, like any other distributed system, is also prone to resource failures. These failures are generally due to software faults, hardware faults, errors in network, data staging issues, failures due to virtualization, disk errors, power issues, and many others. These failures from a workflow application perspective can be classified into (1) task failures, (2) Virtual Machine failures, and (3) workflow level failures [Hwang and Kesselman 2003]. Nonetheless, failures are inevitable whilst running a complex application like workflows consisting of thousands of tasks.

Further, cloud resources also experience performance variations because of resource sharing, consolidation, and migration, among other factors. Performance variation of cloud resources affects the overall execution time (i.e., makespan) of the workflow. It further increases the difficulty to estimate the task execution time accurately. Dejun et al. [2010] show that the behavior of multiple "identical" resources vary in performance while serving exactly the same workload. A performance variation of 4% to 16% is observed when cloud resources share network and disk I/O [Armbrust et al. 2010].

Clouds are realizing the vision of utility computing by delivering computing resources as services. The demand for cloud computing is facilitating cloud providers to evolve various business models around these services. Most providers provision cloud resources (e.g., Virtual Machines (VMs) instances) on a pay-as-you-go basis (similar to On-Demand instances) charging fixed prices per time unit. However, Amazon, one of the pioneers in this space, started selling idle or unused data center capacity through bidding in an auction-like market as Spot Instances (SI) since December 2009. On-demand and SIs have the same configurations and characteristics. Nonetheless, SIs offers cloud users a reduction in costs of up to 70% for multiple applications like bag-of-tasks, web services, and MapReduce workflows [Poola et al. 2014; Ostermann and Prodan 2012; Voorsluys et al. 2011]. Significant cost reductions are achieved due to lower Quality of Service (QoS), which makes SIs less reliable and prone to out-of-bid failures. This introduces a new aspect of reliability into the Service Level Agreements (SLAs) and the existing trade-offs making it challenging for cloud users [Javadi et al. 2011].

These challenges emphasize the necessity for an effective fault-tolerant and robust workflow scheduling algorithm to mitigate resource failures and performance variations. Scientific workflows can also benefit from SIs with an effective bidding and an efficient fault-tolerant mechanism. Such a mechanism can also tolerate out-of-bid failures and further reduce the cost immensely.

Therefore, in this article, we present a just-in-time, fault-tolerant and adaptive scheduling heuristic. It uses spot and on-demand instances to schedule workflow tasks. It minimizes the execution cost of the workflow and at the same time provides a robust schedule that satisfies the deadline constraint.

The **key contributions** of this article are (1) a just-in-time scheduling heuristic that uses spot and on-demand resources to schedule workflow tasks in a robust manner and (2) a replication strategy for cloud environments that utilizes different pricing models offered by clouds.

2. RELATED WORKS

Cloud resources experience failures and performance variations that demand fault-tolerance in a schedule. Studies [Dejun et al. 2010; Ostermann et al. 2010] have shown that performance of VMs in a cloud environment exhibits variability, and it varies for different instance types, different availability zone, different data centers, and different times of the day. Ming and Humphrey [2012] has shown that there is significant variation in VM startup and varies with size, operating system, and type of instance. They also show that up to 8% of VMs fail while they are acquired. Failures in a distributed system are inevitable and they occur at multiple sources. Failures occur in any of the following levels: hardware, operating system, middleware, network, storage, and task or at the user level. Some of the most common reasons for failure are low memory or disk space, network congestion, unavailability of input files at the right moment, nonresponding services, errors in file staging, authentication, uncaught exception, missing libraries, task crashes, and many more [Plankensteiner et al. 2009]. Li et al. [2010] emphasize the need for fault tolerance in workflow applications on a cloud environment. Prominent fault-tolerant techniques that can mitigate failures are retry, alternate resource, checkpointing, and replication [Yu and Buyya 2005; Chen and Yang 2007]. In essence, redundancy is fundamental in providing fault tolerance and it is mainly in two forms: space and time [Gärtner 1999].

Redundancy in space is one of the widely used mechanisms for providing fault tolerance. Redundancy in space is achieved by providing duplication or replication of resources. There are broadly two variants in this approach: task duplication and data replication.

Task duplication creates replicas of tasks. Replication of tasks can be done concurrently [Cirne et al. 2007], where all the replicas of a particular task start executing simultaneously. When tasks are replicated concurrently, the child tasks start its execution depending on the schedule type.

Schedules are of two types: first, where the child task starts only when all the replicas have finished execution [Benoit et al. 2008]. In the other schedule type, the child tasks start execution as soon as one of the replicas finishes execution [Cirne et al. 2007].

Replication of task is also done in a backup mode, where the replicated task is activated when the primary tasks fail [Mosse et al. 1994]. This technique is similar to retry or redundancy in time. However, here they employ a backup overloading technique, which schedules the backups for multiple tasks in the same time period to effectively utilize the processor time.

Duplication is employed to achieve multiple objectives, the most common being fault tolerance [Benoit et al. 2008; Kandaswamy et al. 2008; Yang et al. 2009; Hashimoto et al. 2002]. When one task fails, the redundant task helps in completion of the execution. Additionally, algorithms also employ data duplication where data is replicated and prestaged, thereby moving data near computation especially in data intensive workflows to improve performance and reliability [Chervenak et al. 2007]. Furthermore, estimating task execution time a priori in a distributed environment is arduous. Replicas are used to circumvent this issue using the result of the earliest completed replica. This minimizes the schedule length to achieve hard deadlines [Darbha and Agrawal 1994; Ranaweera and Agrawal 2000; Dogan and Ozguner 2002; Tang et al. 2010], as it is effective in handling performance variations [Cirne et al. 2007]. Calheiros and Buyya [2013] replicated tasks in idle time slots to reduce the schedule length. These replicas also increase resource utilization without any extra cost.

Task duplication is achieved by replicating tasks in either idle cycles [Calheiros and Buyya 2013] of the resources or exclusively on new resources. Some schedules use a hybrid approach replicating tasks in both idle cycles and new resources. Idle cycles are

those slots in the resource usage time period where the resources are unused by the application. Schedules that replicate in these idle cycles profile resources to find unused time slots and replicate tasks in those slots. This approach achieves benefits of task duplication and simultaneously minimizes monetary costs. In most cases, these idle slots might not be sufficient to achieve the needed objective. Hence, many algorithms place their task replicas on new resources. These algorithms trade off resource costs to their objectives.

There is a significant body of work in this area encompassing platforms like cluster, grids, and clouds [Darbha and Agrawal 1994; Benoit et al. 2008; Kandaswamy et al. 2008; Yang et al. 2009; Hashimoto et al. 2002; Ranaweera and Agrawal 2000; Dogan and Ozguner 2002; Tang et al. 2010; Brandic et al. 2009]. Resources considered can either be bounded or unbounded depending on the platform and the technique. Algorithms with bounded resources consider a limited set of resources. Similarly, an unlimited number of resources are assumed in an unbounded system environment. Resource types used can either be homogeneous or heterogeneous in nature. Darbha and Agrawal [1994] is one of the early works, which presents an enhanced Search and Duplication Based Scheduling algorithm (SDBS) that takes into account the variable task execution time. They consider a distributed system with homogeneous resources and assume an unbounded number of processors in their system.

Resubmission and task redundancy are the most prominent fault-tolerant strategies amongst workflow management systems [Plankensteiner et al. 2009]. They resolve most failures mentioned previously in a distributed environment like the cloud. The works by Benoit et al. [2008], Ranaweera and Agrawal [2000], and Calheiros and Buyya [2013] are among the closest to our work.

Benoit et al. [2008] proposed an active replication heuristic for heterogeneous processors, which are bounded. Alternatively, Ranaweera and Agrawal [2000] present a replication based scheduling, which uses processor idle time. Their objective is to minimize makespan of workflow in a cluster environment. Similarly, Calheiros and Buyya [2013] developed a heuristic for the cloud environment that uses processor idle time for replication. These replicas also increase resource utilization without any extra cost.

In contrast, we employ both redundancy in space and time. We use task replication and task retry to achieve fault tolerance, to minimize makespan, and also maximize resource utilization, contrary to Benoit et al. [2008], Calheiros and Buyya [2013], and Ranaweera and Agrawal [2000], who use only task replication. The combination of task replication and task retry, saves both time and cost. Calheiros and Buyya [2013] and Ranaweera and Agrawal [2000] replicated tasks only in idle time slots, whereas our proposed algorithm replicates tasks both on idle slots as well as on new resources providing higher tolerance to failures. The novelty of this approach is that it uses spot instances in a judicious manner to save cost and provide fault tolerance. To the best of our knowledge, this approach has not been used earlier. Additionally, our proposed system model unlike Benoit et al. [2008], uses an unbounded number of processors, which are heterogeneous in character.

3. BACKGROUND

In this section, we define important terminologies, concepts, and metrics that will be further referred to in the text.

Workflow is represented by a Directed Acyclic Graph (DAG), $G = (T, E)$, where T is a set of nodes, $T = \{t_1, t_2, \ldots, t_n\}$, and each node represents a task. Additionally, E represents a set of edges between tasks, which are control and/or data dependencies. Each workflow is bounded by a user-defined deadline D. We also account for data transfer times between tasks. The data transfer time between two tasks is computed based on the size of the data transferred and the cloud data center internal network

bandwidth. Additionally, each workflow task t_i also has a task length len_i given in Million Instructions, which is used to estimate the task execution time.

Makespan, M, is the total elapsed time required to execute the entire workflow. Deadline D is considered as a constraint measured relative to the workflow Submission Time, ST, where makespan should not be more than the deadline ($M \leqslant D$). The makespan of the workflow is computed as $M = finish_{t_n} - ST$, where $finish_{t_n}$ is the finish time of the exit node of the workflow.

Critical Path, CP, is the longest path from the start node to the exit node of the workflow. Critical path determines the makespan of a workflow. The critical path is evaluated in a breadth-first manner calculating the weights of each node. The node weight is the maximum of the predecessor's estimated execution time and data transfer time calculated as per Equation (1),

$$weight(t_i) = w_i + \max_{t_p \in pred(t_i)} \{weight(t_p) + c_{p,i}\}, \tag{1}$$

where $pred(t_i)$ is all the parent nodes of t_i, and w_i is the execution time of node t_i on an instance type chosen by the algorithm. $c_{p,i}$ is the data transfer time from node t_p to t_i. The maximum weight among the exit nodes is the critical path time. When a node completes execution its weight and data transfer time to all its child nodes is made zero, and the critical path is recomputed.

Essentially Critical Tasks ($ESCT$). It is important to define the notion of Earliest Finish Time (EFT) and the Latest Finish Time (LFT) to explain $ESCTs$. To explain the concept of EFT we introduce Earliest Start Time (EST), which is the earliest time a task can start, given by Equation (2) [Abrishami et al. 2013],

$$\begin{aligned} &EST(t_{start}) = 0, \\ &EST(t_i) = \max_{t_p \in pred(t_i)} \{EST(t_p) + MT(t_p) + c_{p,i}\}, \end{aligned} \tag{2}$$

where $MT(t_p)$ is the Minimum Execution Time of t_p on any instance type. This leads to the definition of Earliest Finish Time (EFT), which is the earliest a task can finish its execution and is determined by Equation (3) [Abrishami et al. 2013].

$$EFT(t_i) = EST(t_i) + MT(t_i). \tag{3}$$

Finally, Latest Finish Time (LFT) is the latest time a task has to finish execution so that the deadline constraint is not violated. It is described by Equation (4) [Abrishami et al. 2013].

$$\begin{aligned} &LFT(t_{exit}) = D, \\ &LFT(t_i) = \min_{t_s \in succ(t_i)} \{LFT(t_s) - MT(t_s) - c_{i,s}\}, \end{aligned} \tag{4}$$

where $succ(t_i)$ is all the children nodes of t_i.

Hitherto, Essentially Critical Tasks are the tasks that have no slack time to finish their execution, that is, if the $EFT(t_i) \geq LFT(t_i)$, then the task is an $ESCT$. In other words, $ESCT$ is not just a task on the critical path but a task that does not have any slack time and must finish by their EFT; this is shown diagrammatically in Figure 1(b). The algorithms schedule $ESCTs$ on instances that offer low execution time to avoid $ESCTs$ further in the workflow execution.

Latest Time to On-Demand, LTO is the latest time the algorithm has to switch to on-demand instances to satisfy the deadline constraint. The algorithm exploits the spot market before the time flag LTO and switches to on-demand instance later. LTO aids in choosing the right instance, to speed up or slow down and choose the apt pricing model. It is determined for every ready task and the scheduling decisions are made

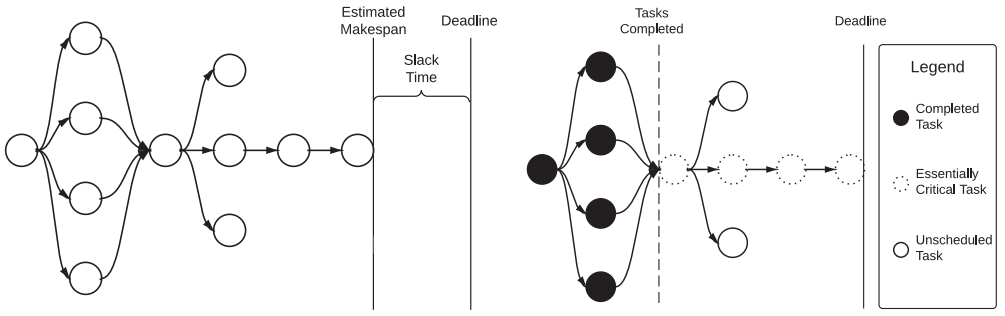

Fig. 1. Figure (a) shows a workflow at time t_0, where there is enough slack time. Under such situation the tasks are scheduled onto spot instances. Figure (b) shows a workflow at time t_1, where there is no slack time. It also shows some completed tasks. Under such situation, Essentially Critical Tasks (ESCTs) are scheduled onto on-demand instances and replicated on spot instances. Other tasks with slack time are scheduled on spot instances.

Table I. List of Acronyms

Acronyms	Description
ESCT	Essentially Critical Tasks
LTO	Latest Time to On-Demand
EFT	Earliest Finish Time
EST	Earliest Start Time
LFT	Latest Finish Time
MT	Minimum Execution Time
EIT	Expected Idle Time of an instance
GT	Gratis Time of an instance
MET	End time of an Instance
ECT	Estimated Completion Time of a task
MST	Max Start Time of the task
CPT	Critical Path Time of the remaining workflow
ERT	Estimated task runtime
TCT	Task Completion Time

based on the current time t and the *LTO*. *LTO* at time t is the difference between the deadline and the critical path ($LTO_t = D - CP_t$), where CP_t is the critical path at time t for the remaining workflow. Additionally, Table I lists all the acronyms used in this article.

Pricing models. In our model, we adapt two types of instances from the Amazon model, which vary in their pricing structure. The two pricing models considered are (1) *On-Demand instance*: the user pays by the hour based on the instance type; and (2) *Spot Instance*: users bid for the instance and it is made available as long as their bid is higher than the spot price. Spot prices change dynamically and it can change during the instance runtime. The price of a SI (spot price) is determined by the provider based on the instance type and demand within the data center, among other parameters [Javadi et al. 2011]. The spot price of an instance varies with time and it is different for different instance types. The price also varies between regions and availability zones. The users participate in an auction-like market and bid a maximum price they are willing to pay for SIs. The user is oblivious to the number of bidders and their bid prices. The user is provided the resource/instance whenever their bid is higher than or equal to the spot price [amz 2009]. However, when the spot price becomes higher than the user bid,

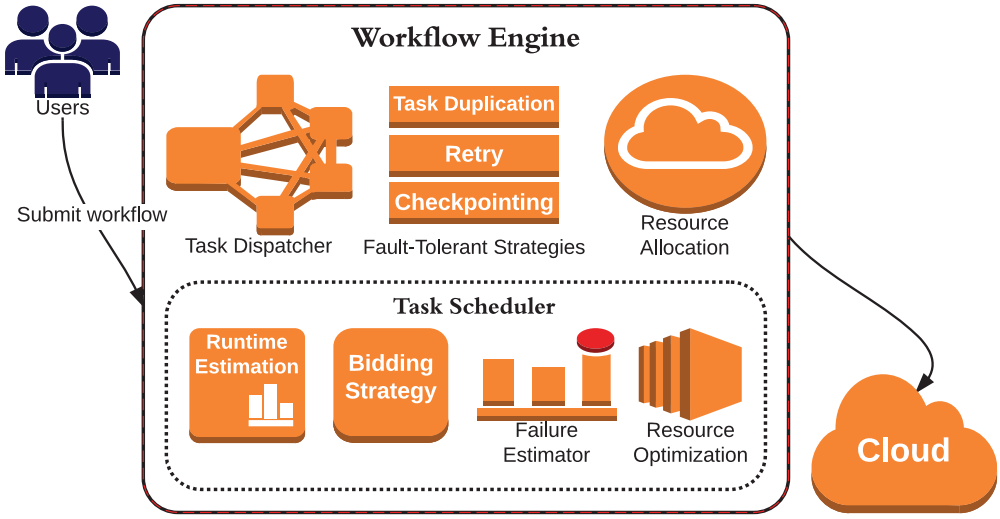

Fig. 2. System architecture of fault-tolerant workflow management system.

Amazon terminates the resources. Users do not pay the bid price, they pay the spot price that was applicable at the start time of the instance. Users are not charged for the partial hour when terminated by the provider. Nevertheless, when the user terminates the instance, they have to pay for the full hour.

Total cost, C, is the sum of the cost of all the instances used for the workflow execution, based on their instance type and pricing model. The cost of each instance is calculated as per the Amazon model. If the instance is an on-demand instance, the on-demand price of that instance is used. If the instance is spot, the spot price of the instance is used to calculate the cost. All partial hours are rounded to full hours for both spot and on-demand instances (e.g., 5.1 hours is rounded to 6 hours).

Two metrics are used in this article to measure the **robustness** of a schedule. The first metric is *failure probability*, R_p, which is the likelihood of the workflow to fail before the given deadline [Shi et al. 2006], which can be formulated as follows:

$$R_p = (TotalRun - FailedRun)/(TotalRun), \tag{5}$$

where $TotalRun$ is the number of times the experiment was conducted and $FailedRun$ is the number of times the constraint, $finish_{t_n} \leq D$, was violated. This equation is based on the methodology offered by Dastjerdi and Buyya [2012].

The second metric is the *tolerance time*, R_t, which is the amount of time a workflow can be delayed without violating the deadline constraint. This provides an intuitive measurement of robustness given the same schedule and resource to task mapping, expressing the amount of uncertainties it can further withstand.

$$R_t = D - finish_{t_n}. \tag{6}$$

Replication factor is the ratio of the total number of replicas created to the number of workflow tasks. This gives an estimate about the number of replicas created for a workflow with a known number of tasks.

4. SYSTEM MODEL

The system architecture of our fault-tolerant workflow management system is presented in Figure 2. The *workflow engine* acts as a middle layer between the user application and the cloud. Users submit a workflow application into the engine, which

schedules the workflow tasks, provides fault tolerance mechanism, and allocates resources in a transparent manner.

The *task dispatcher* analyzes the data and/or control dependencies between the tasks and submits the ready tasks to the task scheduler. Ready tasks are those tasks whose predecessor tasks have completed their execution and have received all input files, and are prepared to be scheduled.

Fault-Tolerant Strategies. In this article, resource failures are considered, which refer to anomalies. These anomalies are software, hardware, or any other faults in the environment causing disruption in task execution. Failures also arise from out-of-bid events among SIs. As a workflow engine with limited knowledge of the application, replication in time and space are employed to address these failures. Failures in any distributed system including cloud is shown to follow a Weibull distribution [Javadi et al. 2012; Yigitbasi et al. 2010; Iosup et al. 2007; Tang et al. 2014; Kondo et al. 2010; Litke et al. 2007; Plankensteiner et al. 2009]. Weibull distibution is used in this article to model resource failures similar to the one used in Javadi et al. [2012]. To mitigate such failures, fault-tolerant strategies such as task duplication, retry, and checkpointing can be employed. Checkpointing saves states of a running process periodically to a reliable storage. These saved states are called *checkpoints*. The process will be restarted from its last checkpoint or the saved state after a failure.

In this article, we employ task duplication and retry as fault-tolerant mechanisms. Task duplication and resubmission are the most prominent mechanisms used to mitigate failures in most worklfow management systems [Plankensteiner et al. 2009]. Here task duplication is used when the deadline is short and resubmission is employed when the deadline is relaxed. These mechanisms are effectively used in a dynamic fashion based on the deadline constraint.

Resource Allocation. The task scheduler chooses the instance type and also the pricing model (e.g., spot or on-demand). This module allocates the appropriate resource as chosen by the task scheduler.

The *task scheduler* employs a scheduling algorithm to find a suitable cloud resource for every task. The details of the proposed scheduling algorithm are outlined in the next section. This proposed fault-tolerant scheduling algorithm is a generic algorithm that can be used by workflow management system scheduler, provided it supports cloud resources and can provision spot instances.

Runtime Estimation. To determine the runtime of a workflow task on a particular instance type, we use Downey's analytical model [Downey 1997]. Downey's model requires a task's average parallelism A, coefficient of variance of parallelism σ, the task length, and the number of cores of the target instance type to estimate the runtime. We have used the model of Cirne and Berman [2001] for generating the values of A and σ for each task.This model has been shown to capture the behavior of moldable jobs in parallel production environments. With the use of these two models the task's runtime is estimated on different instance types.

Failure estimator estimates the Failure Probability (FP) of a particular bid price (bid_t) based on the spot price history. The history price of one month prior to the start of the execution and the spot prices until the point of estimation is used. The failure probability estimator analyzes the spot price history for the bid value in consideration, for which the total time of the spot price history, HT, and the total out of bid time, OBT_{bid_t}, for the bid bid_t is measured. The total out of bid time is the aggregated time in history when the spot price was higher than the bid bid_t. These two factors are used to estimate the probability of failure as shown in Equation (7). This estimation is used while evaluating the bid value and also while scheduling the task.

$$FP_{bid_t} = OBT_{bid_t}/HT. \tag{7}$$

Bidding strategy evaluates the bid price that can be used in bidding for a spot instance. The bidding strategy used in this article is explained in detail in Section 5.

Resource optimization method ensures that resource utilization is maximized. It tries to minimize the number of resources used for a workflow execution by assigning tasks on already running resources to save cost and also maximize utilization.

The **problem** we address in this work is to find a mapping of workflow tasks onto heterogeneous VM types, using a mixture of on-demand and SIs such that the cost of workflow execution is minimized within the deadline. The schedule should also be robust against resource failures including premature termination of SIs and performance variations of the resources.

Assumptions: Data transfer cost between VMs are considered to be zero, as in most public clouds, data transfer inside a cloud data center is free. The data center is assumed to have sufficient resources, avoiding VM rejections due to resource contention. This is not a prohibitive assumption as the resources required are much smaller than the data center capacity.

5. PROPOSED APPROACHES

Replication is the most widely used mechanism for enhancing availability and reliability of services. Replication can be done either in time (task resubmission) or space (task duplication). The rationale behind task replication with n number of replicas is that it can tolerate (n-1) failures without affecting the makespan of the workflow. The downside of task duplication is consumption of extra resources. Task resubmission or retry is an effective fault-tolerant strategy where tasks are resubmitted onto a new resource only when resources fail, hence it is cost effective although it increases the makespan of the workflow.

In this article, the proposed heuristic employs both these fault-tolerant mechanisms. When the deadline is short, it employs task duplication and as the deadlines become lenient it employs task retry to mitigate failures. The working of this heuristic is depicted in Figure 1. The proposed heuristics are detailed in the next subsection.

5.1. Heuristics

Scheduling workflow tasks onto heterogeneous VMs is an NP-Complete problem [Johnson and Garey 1979]. Hence, we propose an adaptive, just-in-time heuristic. The task dispatcher dispatches ready tasks to the scheduler. It monitors the execution of tasks and resubmits the task if it fails, or submits the child task when all its parent tasks have completed execution. The scheduler maps these ready tasks onto the best suitable resource, such that cost and makespan is minimized and the schedule is fault tolerant. We detail the working of the proposed heuristics in this section.

Once the scheduler receives a task from the task dispatcher, it estimates the critical path. The critical path will potentially be different for every instance type used to estimate it. Therefore, after a task completes its execution, its critical path weight is made zero and for every ready task the critical path time is recomputed. Based on the deadline and the estimated critical path time, the time flag LTO is computed. The difference between LTO and the current time dictates the type of resource and the pricing model that will be selected.

The heuristic acts based on the position of the time flag LTO with respect to the current time. We explain the heuristics in four possible scenarios. Scenarios 1 and 2 are when LTO is ahead of the current time connoting sufficient slack time to complete workflow execution before the deadline. Under such circumstances, tasks are mapped to spot instances. Scenario 1 illustrates the task mapping onto running instances to consolidate resource usage reducing cost and time. In scenario 2 tasks are mapped onto new spot instances, when no suitable running instance was found. On the other hand,

ALGORITHM 1: FindFreeSlot(t, $InstanceList$, P)

```
   input: task t, InstanceList, PriceModel P
   output: Suitable VM
 1 types ← available instance types;
 2 estimates ← compute estimated runtime of task t_i on each type ∈ types;
 3 minComplTime ← MaxValue;
 4 for ∀v ∈ InstanceList do
 5     if P = ANY or v.pricemodel = P then
 6         ERT ← estimates(t_v);
 7         GT ← MET − EIT;
 8         ECT ← D − CPT − ERT;
 9         if EIT ≤ MST and ERT ≤ GT then
10             TCT ← EIT + ERT;
11             if TCT < ECT and TCT < minComplTime then
12                 minComplTime ← TCT;
13                 suitableVM ← v;
14 return suitableVM;
```

in scenarios 3 and 4, when the LTO is before the current time, then the algorithm has to choose an expensive and high performing machine to speed up the execution to meet the deadline. Here, tasks are duplicated to provide fault tolerance as there is no slack time. Replication is done on spot instances to save cost. Hitherto, fault tolerance is achieved by replication. Tasks are either replicated in time or in space based on the deadline, LTO, and the current time.

5.1.1. Scenario 1: Mapping Task on Already Running Spot Instances. First, let us consider the case in which the LTO is conveniently ahead of the current time. In such a case, the algorithm first tries to map the tasks onto spot instances as they are cheap and even if they fail due to out-of-bid events, there is enough slack time to rerun them. Before mapping onto spot instances the heuristic searches for free slots among the resources already in use. If no free slot is found, the scheduler searches for resources, which are running and can finish within the task's latest finish time. The latest finish time is the time beyond which if any delay occurs it will violate the workflow deadline.

Free slots are unused idle time periods in instances before the end of their charged time period. Algorithm 1 describes the methodology of finding these free slots. This method explores only among the specified instances of a particular price model stated by the function call. Here, for every instance in use, the time it will become idle is estimated, that is, Expected Idle Time (EIT). Further, Gratis Time (GT) is computed, which is the difference between EIT and the end time, MET, until which the machine is leased. This gives an estimate of the available idle time in that resource. Furthermore, Estimated Completion Time (ECT) and Max Start Time (MST) for the task is estimated. ECT is computed as shown in line 8 of Algorithm 1, where ERT is the Estimated task Runtime on that instance type and CPT is the Critical Path Time, which is the time taken on the slowest instance. ECT is a virtual task deadline indicating that the task has to finish within this time to avoid any delay. Hence, the task has to complete its execution before ECT. If the conditions $EIT \leq MST$ and $ERT < GT$ are met, then task completion time (TCT) is computed as shown in line 10. Finally, the suitable instance is selected if the TCT on that instance is less than the ECT and its TCT is the minimum among all considered instances.

In the event when no free slots are found, the algorithm finds a suitable running instance that can be used instead of starting a new instance. The rationale being that

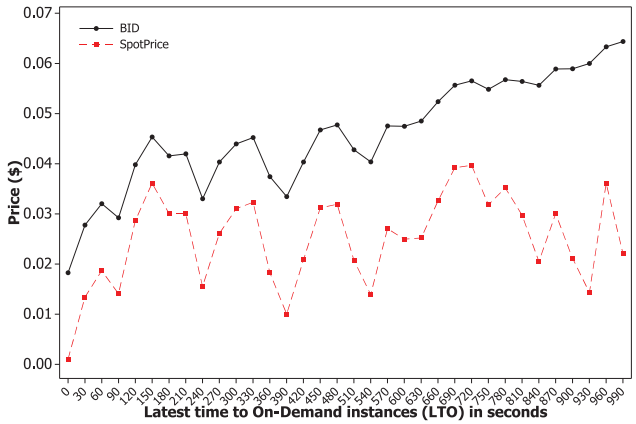

Fig. 3. Generation of bid value through Intelligent Bidding Strategy.

such instances are readily available, saving boot time. The method *FindRunningVM* is very similar to the method *FindFreeSlot*, the prominent difference being in line 9 of Algorithm 1, where the condition $ERT \leq GT$ is omitted. In other words, the algorithm does not validate if the estimated task runtime is less than or equal to the gratis time.

5.1.2. Scenario 2: Mapping Task on a New Spot Instance. When no running instance has either a free slot or is capable of honoring the deadline, the heuristic investigates further and checks whether there is sufficient time to run on a new spot instance. If so, the bid price is estimated using a bidding strategy, then the failure probability for that bid price is estimated. Here, **Intelligent Bidding Strategy** is used to estimate bid prices, which was proposed in Poola et al. [2014]. This strategy takes into account the current spot price (p_{spot}), on-demand price (p_{OD}), *LTO*, *FP* of the previous bid price, the Current Time (*CT*), α, and β. α, as seen in Equation (8), dictates how much higher the bid value must be above the current spot price. The lower the value of α, the higher is the value of the bid with respect to the spot price. β determines how fast the bid value reaches the on-demand price. The increase in bid price closer to the on-demand price as the *CT* reaches closer to the *LTO* is attributed to the parameter β. The higher the value of β, the faster the bid reaches closer to on-demand price. *FP* of the previous bid is used as a feedback to the current bid price; the current bid price varies in accordance to the *FP* adding intelligence to the bidding strategy. The bid price is calculated as per Equation (8). The bid value increases gradually with the workflow execution and as the *CT* moves closer to the *LTO*. The bid starts around the initial spot price and ends closer to the on-demand price. The rationale of increasing the bid price is to lower the risk of out-of-bid events as the execution nears the *LTO* making sure that the deadline constraint is not violated. Figure 3 shows the bid price (BID) generated by the bidding strategy as explained earlier for the spot price at different *LTO* times. It also shows that the bid value steeps up toward the end to reach closer to the on-demand price. The bidding strategy considers all these factors and calculates a bid value in accordance to the situation.

$$bid = e^{\gamma} * p_{OD} + (1 - e^{\gamma} * (\beta * p_{OD} + (1 - \beta) * p_{spot})), \quad (8)$$
$$where, \gamma = (-\alpha(LTO - CT))/FP.$$

5.1.3. Scenario 3: Mapping Task to an On-Demand Instance. Let us now examine the case where LTO is behind the current time, that is, there is no slack time, or when the estimated bid price is higher than the on-demand price, or the failure probability is higher than the threshold. In such cases the algorithm tries to find a suitable

ALGORITHM 2: Schedule(t)

```
   input: task t_i
   output: DecisionList
   /* Variable Initializations                                              */
1  vms ← all VMs currently in the pool;
2  types ← available instance types;
3  estimates ← compute estimated runtime of task t_i on each type ∈ types;
4  decisionList ← null;
5  Recompute CP and LTO.
6  timeLeft = LTO − currentTime
7  if timeLeft > 0 then      // If there is sufficient slack time, then find a running
   instance
8      decision ← FindFreeSlot(t_i, vms, PriceModel.ANY);
9      if decision.allocated = true then decisionList.add(decision);
10     if decision.allocated = false then
11         decision ← FindRunningVM(t_i, vms, PriceModel.ANY);
12         if decision.allocated = true then decisionList.add(decision);
13 timeLeft = timeLeft − vmInitTime
14 if timeLeft > 0 then  // Initialize a new spot instance as no running instance was
   found
15     bid ← EstimateBidPrice(t_i, type);
16     if bid > on-demand price then
17         Map to on-demand instance and decisionList.add(decision).
18     failProb ← EstimateFailureProbability(bid);
19     if failProb < threshold then
20         Map to spot instance and decisionList.add(decision);
21 InstanceList ← FindSuitableInstances(CP, D) ;     // Find Instance types that can
   honor the deadline
   /* Finding on-demand instances as sufficient slack time is not available  */
22 decision ← FindFreeSpace(t_i, InstanceList, PriceModel.ONDEMAND);
23 if decision.allocated = true then decisionList.add(decision);
24 if decision.allocated = false then
25     decision ← FindRunningVM(t_i, InstanceList, PriceModel.ONDEMAND);
26     if decision.allocated = true then decisionList.add(decision);
27 decision ← FindCostPerfEffectiveVM(t_i, InstanceList);     // Finding an appropriate
   new on-demand instance
28 compute EFT and LFT for task t_i
29 if Number of Replicas of t_i ≤ 1 then      /* Task Duplication under short deadline */
30     if EFT+VMinitTime ≥ LFT then
31         unusedInstance ← instances not used to map replicas of T_i
32         repDecision ← FindFreeSpace(t_i, unusedInstance, PriceModel.ANY);
33         if repDecision.allocated = true then decisionList.add(repDecision);
34         if repDecision.allocated = false then
35             repDecision ← FindRunningVM(t_i, unusedInstance, PriceModel.ANY);
36             if repDecision.allocated = true then decisionList.add(repDecision);
37         if null = repDecision then
38             InstanceList ← FindSuitableInstances(CP, D)
39             InstanceType ← FindCostPerfEffectiveVM(t_i, InstanceList);
40             bid ← EstimateBidPrice(t_i, InstanceType);
41             if bid > on-demand price then
42                 Map to on-demand instance and decisionList.add(repDecision).
43             Map to spot instance and decisionList.add(repDecision);
44 return decisionList;
```

ALGORITHM 3: FindSuitableInstances(estimates)

```
   input: estimates
   output: Eligible Instance List
 1 types ← available instance types;
 2 InstanceList ← null
 3 for ∀i ∈ InstanceTypes do
 4 |   CPTasks ← computeCPTasks(i);
 5 |   prevTask ← null;
 6 |   CPTime ← 0;
 7 |   for ∀t ∈ CPTasks do
 8 |   |   if prevTask ≠ null then
 9 |   |   |   edgeTime ← edgeTime(prevTask, t);
10 |   |   CPTime + = estimates(t) + edgeTime;
11 |   |   prevTime ← t;
12 |   totalCPTime = CPTime + VMInitTime;
13 |   if totalCPTime ≤ D − currenttime then
14 |   |   InstanceList.add(i);
15 return InstanceList;
```

on-demand instance, as on-demand instances have higher QoS guarantees. Before finding an instance, a list of suitable instance types that can honor the deadline are found as shown in Algorithm 3.

In this method, **Find Suitable Instances**, critical tasks are determined for every instance type. Ideally, the critical path can vary for different instance types and so do the tasks on it. In other words, tasks have different runtimes on different instance types and therefore, critical path will also change based on the instance type used to estimate. Hence, this method evaluates the critical path per instance type and maintains a list of critical tasks per instance type. This computation is done initially, and when a task finishes execution, the task dispatcher checks whether the task was critical and if it was the critical paths for those instance types (i.e., where the completed task was critical) are recomputed. This increases efficiency and avoids computing critical paths for every task mapping. Once the critical path tasks are computed, lines 7–12 adds the task runtime, the transfer time for all the tasks on the critical path. Finally, the total critical path time is computed and if this is less than the remaining deadline, then the instance type is added into the eligible instance list.

This instance list is a list of instance types that can comply with the deadline constraint. Akin to scenario 1, the algorithm first tries to find a free slot among the instance list; if no free slot is found, then an instance is found among the running instances that can execute the task without delaying the deadline. If no instances are found, then *FindCostPerfEffectiveVM* method calculates the cost of the estimated critical path times with their respective on-demand prices. The instance that can execute with the lowest cost is selected. The algorithm does not select an instance type with lowest price; it selects an instance whose price to performance ratio is the lowest.

5.1.4. Scenario 4: Task Duplication Under Short Deadline. Critical tasks are replicated to provide fault tolerance when the deadline is short. We propose two variants of this heuristic. The two heuristics are very similar, with the difference being the tasks they replicate.

Essential Critical Path Task Replication (ECPTR) heuristic: Algorithm 2 details its working. Here, when the LTO has passed the current time, and the task has no slack time, then a replica is created. In other words, all ESCTs are replicated. Similar

to the scenarios presented before, first free slots are found among the instances that have not been used to map the replicas of the task considered. If no free slots are found, then a running instance is found that can be used to the map the task without violating the deadline. When neither free slots nor running instances are found, the heuristic maps the replica onto a spot instance. The type of spot instance is decided by methods *FindSuitableInstances* and *FindCostPerfEffectiveVM* as shown in lines 38 and 39. Finally, using the bidding strategy a bid price is estimated for the spot price and if this bid price is less than the on-demand price, then a spot instance is instantiated mapping the replica task.

Furthermore, when a resource fails, the tasks on the resource are resubmitted as ready tasks to the scheduler. In such case, the task duplication is done only when the number of replicas of the task is zero or one as shown in line 29. In other words, we do not have more than one replica at any particular time for a given task. When the execution of the task finishes, all the replicas are terminated, so that the resources can be freed to accommodate other tasks.

The other heuristic is **Critical Task Replication (CTR**). Here, all the critical tasks are replicated, that is, once the LTO has moved past the current time then all tasks are replicated. The replicated tasks are mapped to spot instances to minimize cost. This heuristic is very similar to Algorithm 2, the only difference being, the validation in line 30 is omitted in this heuristic. In other words, under short deadline all tasks will be replicated, although only one replica will be created for a given task at any time.

5.2. Time Complexity

The time complexity for calculating the critical path and recomputing the same for all ready tasks is $O(n^2)$ in the worst case, where n is the number of tasks. The complexity of algorithm for finding a suitable instance for every task is $O(n)$. The complexity of finding the suitable instance depends on the number of instances considered, which is negligible. Hence, the asymptotic time complexity of the algorithm is $O(n^2)$.

6. PERFORMANCE EVALUATION

6.1. Simulation Setup

We used CloudSim [Calheiros et al. 2011] to simulate the cloud environment. It was extended to support workflow applications. It was also extended to model the Amazon spot market. It uses Amazon spot market traces to simulate spot prices.

Application Modeling: The Laser Interferometer Gravitational Wave Observatory (LIGO) workflow with size of 1,000 tasks was considered. Its characteristics are explained in detail by Juve et al. [2013]. This workflow covers all the basic components such as pipeline, data aggregation, data distribution, and data redistribution.

Resource Modeling: A cloud model with a single data center is considered. The VMs/cloud resources are modeled similar to Amazon EC2 instances. We have considered nine instance types (m1.small, m1.medium, m1.large, m1.xlarge, m3.xlarge, m3.2xlarge, m2.xlarge, m2.2xlarge, m2.4xlarge) for on-demand instances and SIs. The prices of on-demand instances are adapted from the Linux based instances of Amazon EC2 US West region (North California Availability Zone (AZ)). The spot price history is taken from the same region from the period of June 2014–September 2014. The characteristics of the spot prices for all instances are given in the Table II. Here, we have reported the average spot price, standard deviation (St Dev), and minimum and maximum spot prices for the period considered and also the peaks, which is the number of times the spot price was higher than its on-demand price. A charging period of 60 minutes is considered. A boot/startup time of 100 seconds is considered for each instance [Ming and Humphrey 2012].

Table II. Spot Instance Characteristics for US West Region (North California AZ)

Instance	Average	St Dev	Max	Min	Peaks
m1.small	0.063283	0.055774	0.24	0.0071	570
m1.medium	0.008131	7.30E-05	0.0085	0.008	0
m1.large	0.0163	5.79E-04	0.025	0.016	0
m1.xlarge	0.229384	0.478334	1.92	0.0322	132
m2.xlarge	0.042649	0.116727	1.07	0.0161	18
m2.2xlarge	0.089227	0.204482	2.45	0.0321	48
m2.4xlarge	0.115971	0.160712	2	0.0645	2
m3.xlarge	1.37942	1.763904	6	0.5	167
m3.2xlarge	1.495879	0.544046	2	0.064	64

Failure Modeling: Failures are modeled by Weibull distribution similar to many other prominent works [Javadi et al. 2012; Yigitbasi et al. 2010; Iosup et al. 2007; Tang et al. 2014; Kondo et al. 2010; Litke et al. 2007; Plankensteiner et al. 2009]. We assume these resources to be fail-stop processors, implying that after a failure the resource does not become available again. Further, the failures are considered to be independent. The distribution models the time to failure for a particular resource. The parameters of the Weibull distribution are modeled similar to the parameters used in Javadi et al. [2012].

Baseline Algorithms: We compare our proposed heuristics, ECPTR and CTR (detailed in Section 5) with two baseline approaches:

(1) *Conservative with Intelligent Bidding (CIB)*: This approach was proposed in Poola et al. [2014], which also uses spot instances to reduce cost. It uses retry as a fault-tolerant mechanism, whereas the proposed solution in this article uses both retry and replication. This is one of the few works that use spot instances for workflow scheduling for fault tolerance, hence we use this as a baseline for our work.

(2) *Essential Critical Path Task Replication without Resource Maximization Algorithm (ECPTRRM)*: We introduced this approach, which is similar to ECPTR without resource utilization maximization. This algorithm performs similarly to ECPTR but does not place tasks on the same resource maximizing its usage. ECPTRRM demonstrates the effect of maximizing resource utilization on makespan and cost.

6.2. Results

In this section, we analyze the performance of our heuristics. We investigate them on the parameters of fault tolerance, makespan, cost, and resource utilization. Each experiment was run 100 times, with each run starting at a different point in the trace. In other words, for each run we choose a date and time randomly between the start date and end date of the spot market trace to create randomness in the spot prices. We have run for three different combinations of deadline: (1) short deadline (35,000–45000), (2) moderate (47,500–57,500), and (3) relaxed deadline (60,000–70,000) as shown in Figures 4–10. Next, we present results and their analysis on the performance of our heuristics with regard to different parameters.

6.2.1. Fault Tolerance. Providing fault-tolerant schedules is the main objective of this proposed algorithm. Figures 4 and 5 show the performance of our heuristics with respect to two metrics, failure probability and tolerance time. The details of the metrics were mentioned earlier in Section 3.

The failure probabilities of the different algorithms are shown in Figure 4. It can be observed that under relaxed deadline, all algorithms have substantially lower failure probabilities. When the deadline is short, it can be observed that the heuristics CTR

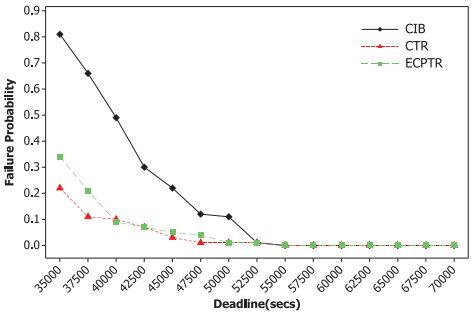

Fig. 4. Failure probability of algorithms with varying deadline.

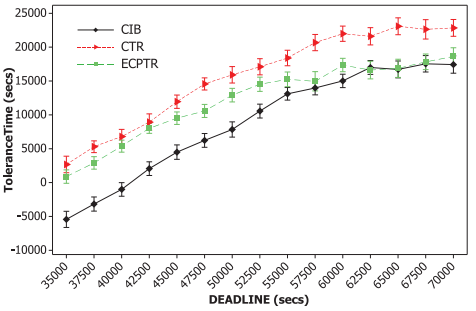

Fig. 5. Tolerance time of algorithms with varying deadline (with 95% confidence interval).

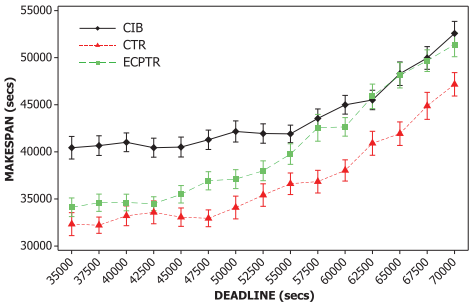

Fig. 6. Mean makespan of the proposed algorithms against the baseline with varying deadlines (with 95% confidence interval).

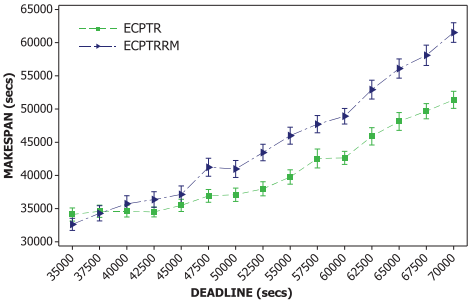

Fig. 7. Showing the effect of resource consolidation on makespan for ECPTR heuristic (with 95% confidence interval).

and ECPTR perform considerably better than the baseline CIB. Failure probabilities of CTR and ECPTR are lower than CIB by 79.75% and 72.36%, respectively, on average under short deadlines. This is a strong indication that our heuristics have a high probability of success in spite of failures in the environment.

Figure 5 depicts the results with respect to the tolerance time. It can be observed that the baseline algorithm CIB has negative tolerance time under short deadlines. This implies that CIB algorithm is not fault tolerant under short deadline, whereas on the other hand, both the proposed heuristics CTR and ECPTR have significantly positive tolerance time. This suggests that they are fault tolerant and can withstand more failures and performance variations given the same schedule. It is also evident from the figure that ECPTR has a higher tolerance time than CTR and this difference becomes larger as the deadline becomes relaxed. Additionally, the tolerance time of CIB and ECPTR become similar as the deadline is relaxed. This is due to the fact that CIB and ECPTR perform similarly to each other under relaxed deadlines.

6.2.2. Effect on Makespan. Makespan is the other important objective that we consider. We attempt to minimize it especially when the deadline is short. Figures 6 and 7 are presented, showing the performance of our algorithms with respect to makespan. The working of our algorithms with respect to the baseline algorithm is depicted in Figure 6. Additionally, the effect of resource maximization on makespan is shown in Figure 7.

The proposed heuristics in comparison with the baseline generates a schedule with makespan lower than the baseline. Figure 6 shows the results with a 95% confidence

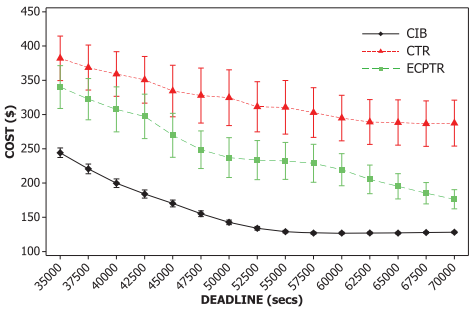

Fig. 8. Mean execution cost of the proposed algorithms against the baseline with varying deadline (with 95% confidence interval).

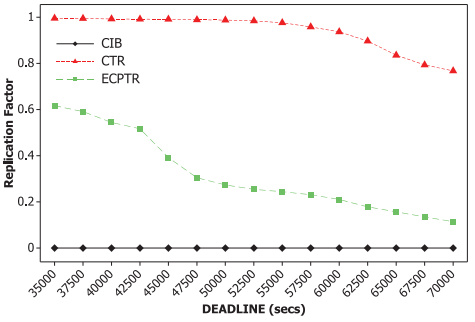

Fig. 9. Replication factor for the algorithms with varying deadline.

interval. It can be observed from the figure that CTR outperforms both CIB and ECPTR, which is essentially because CTR generates more replicas. Task duplication helps generate effective schedules with lower makespan in spite of failures and performance variations.

ECPTR has makespan lower than CIB by 14.47%, 25.14%, and 32.17%, respectively, when the deadline is short, moderate, and relaxed, respectively. Similarly, CTR has a makespan lower by 43.43%, 56.44%, and 55.95% under short, moderate, and relaxed deadlines, respectively, against CIB algorithm. Furthermore, schedules generated by CTR are 23.93% lower in makespan than ECPTR schedules.

Figure 7 shows the results when resource consolidation is done. The effect on makespan is minimal when the deadline is short, the reason being compaction of resource is not possible to a significant extent due to deadline constraints. Whereas, under moderate and relaxed deadlines, ECPTR generates schedules with 11.37% and 14.24% lower makespan, respectively, when compared to ECPTRRM. This reinforces that maximizing resource utilization reduces makespan. It reduces data transfer time, and the boot time needed to initialize new instance, by mapping two or more tasks onto the same resource.

These experiments solidify the fact that task duplication and maximizing resource utilization help lower the makespan significantly.

6.2.3. Effect on Cost. Execution cost is the third objective we strive to minimize. The proposed algorithms use a mixture of spot and on-demand instances to reduce cost. Here, spot instances are used for replication when the deadline is tight and spot instances are used as the primary resource when there is sufficient slack time. This dynamic approach makes the best use of the available pricing models to significantly reduce cost. Cost savings when using spot instances are quantified in the article [Poola et al. 2014].

However, in Figure 8, the costs of algorithms ECPTR and CTR are higher than the baseline algorithm. This increase is attributed to the replicas these algorithms create. It can be further noticed that the cost of ECPTR is higher than CTR, as ECPTR generates more replicas than CTR. Figure 9 shows the number of replicas created by each algorithm for different deadlines. It can also be observed that as the deadline becomes more relaxed, the number of replicas is also reduced and this is more significant for ECPTR heuristic. Similarly, the cost difference between CIB and ECPTR is also reduced as the deadline becomes relaxed.

Apart from the efficient use of pricing models, effective resource usage also reduces costs considerably. Figure 10 testifies to this; it can be observed here that the cost of

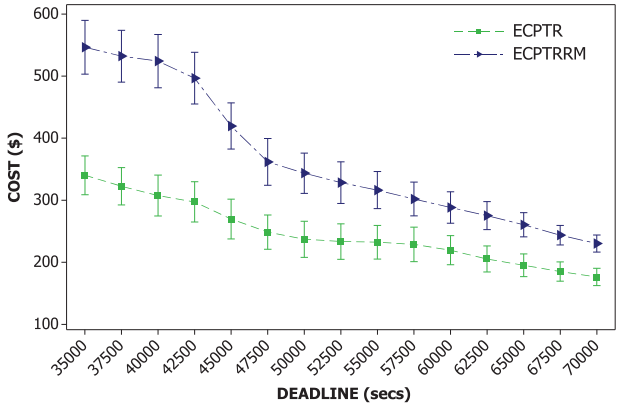

Fig. 10. Showing the effect of resource consolidation on cost for ECPTR heuristic (with 95% confidence interval).

ECPTR is much lower than the ECPTRRM algorithm. When tasks are packed into a single resource the costs can be reduced significantly. Figure 10 shows that when the deadline is short, execution cost of ECPTR is 38.86% lower than ECPTRRM and it is 24.34% lower under relaxed deadlines.

7. CONCLUSIONS AND FUTURE DIRECTIONS

Cloud computing offers low-cost computing services as a subscription based service, which are elastically scalable, and dynamically provisioned. Additionally, it also offers attractive pricing models like on-demand and spot instances, because of which scientific workflow management systems are rapidly moving toward it.

However, cloud environments are prone to failures and performance variations among resources. Failures are traditionally mitigated using replication and this increases the execution cost and time. Whereas with the innovative pricing models cloud offers, the cost for providing fault tolerance can be drastically reduced. In this article, we have proposed two just-in-time adaptive workflow scheduling heuristics for clouds. These heuristics use on-demand and spot instances to provide fault-tolerant schedules whilst minimizing time and cost. They are fault-tolerant against performance variations, out-of-bid failures, and resource failures. Extensive simulations have shown that the proposed heuristics generate schedules with significantly lower failure probabilities. The makespan of these schedules are also much lower than the baseline algorithm. These heuristics are also shown to maximize resource utilization. These experiments establish that pricing models offered by cloud providers can be used to reduce costs and makespan and at the same time offer robust and resilient schedules.

As future work, we will extend this work for multiple data centers considering data transfer cost between different regions. The prices of resources change across data centers. Hence, we will develop a task placement strategy considering the varying prices between regions and also the associated data transfer costs. This model can be further extended for a multicloud environment, extending the number of pricing models.

ACKNOWLEDGMENTS

The authors would like to thank Rodrigo Calheiros, Amir Vahid, Maria Alejandra Rodriguez, Adel Nadjaran Toosi, and Nikolay Grozev for their constructive feedback toward improving this work.

REFERENCES

Amazon 2009. Amazon EC2 Spot Instances. Retrieved November 24, 2014 from http://aws.amazon.com/ec2/purchasing-options/spot-instances/.

S. Abrishami, M. Naghibzadeh, and D. H. J. Epema. 2013. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems* 29, 1 (2013), 158–169.

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. 2010. A view of cloud computing. *ACM Communications* 53, 4 (April 2010), 50–58. DOI:http://dx.doi.org/10.1145/1721654.1721672

A. Benoit, M. Hakem, and Y. Robert. 2008. Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. 1–8. DOI:http://dx.doi.org/10.1109/IPDPS.2008.4536133

I. Brandic, D. Music, and S. Dustdar. 2009. Service mediation and negotiation bootstrapping as first achievements towards self-adaptable grid and cloud services. In *Proceedings of the 6th International Conference Industry Session on Grids Meets Autonomic Computing (GMAC'09)*. ACM, New York, NY, 1–8. DOI:http://dx.doi.org/10.1145/1555301.1555302

R. N. Calheiros and R. Buyya. 2013. Meeting deadlines of scientific workflows in public clouds with tasks replication. *IEEE Transactions on Parallel and Distributed Systems* 99 (2013), 1–1. DOI:http://dx.doi.org/10.1109/TPDS.2013.238

R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. 2011. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience* 41, 1 (2011), 23–50. DOI:http://dx.doi.org/10.1002/spe.995

J. Chen and Y. Yang. 2007. Adaptive selection of necessary and sufficient checkpoints for dynamic verification of temporal constraints in grid workflow systems. *ACM Transactions on Autonomous and Adaptive Systems* 2, 2, Article 6 (June 2007). DOI:http://dx.doi.org/10.1145/1242060.1242063

A. Chervenak, E. Deelman, M. Livny, M. Su, R. Schuler, S. Bharathi, G. Mehta, and K. Vahi. 2007. Data placement for scientific applications in distributed environments. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (GRID'07)*. IEEE Computer Society, Washington, DC, 267–274. DOI:http://dx.doi.org/10.1109/GRID.2007.4354142

W. Cirne and F. Berman. 2001. A model for moldable supercomputer jobs. In *Proceedings of the 15th International Symposium of Parallel and Distributed Processing*. DOI:http://dx.doi.org/10.1109/IPDPS.2001.925004

W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Goes, and W. Voorsluys. 2007. On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Computing* 33, 3 (2007), 213–234. DOI:http://dx.doi.org/10.1016/j.parco.2007.01.002

S. Darbha and D. P. Agrawal. 1994. A task duplication based optimal scheduling algorithm for variable execution time tasks. In *Proceedings of the International Conference on Parallel Processing (ICPP 1994)*, Vol. 2. 52–56. DOI:http://dx.doi.org/10.1109/ICPP.1994.47

A. V. Dastjerdi and R. Buyya. 2012. An autonomous reliability-aware negotiation strategy for cloud computing environments. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*. IEEE, 284–291.

E. Deelman, G. Juve, M. Rynge, J. Voeckler, and G. Berriman. 2013. Comparing FutureGrid, Amazon EC2, and Open Science grid for scientific workflows. *Computing in Science Engineering* 15, 4 (July 2013), 20–29. DOI:http://dx.doi.org/10.1109/MCSE.2013.44

J. Dejun, G. Pierre, and C. H. Chi. 2010. EC2 performance analysis for resource provisioning of service-oriented applications. In *Workshop on Service-Oriented Computing. ICSOC/ServiceWave 2009*. Springer, 197–207.

A. Dogan and F. Ozguner. 2002. LDBS: A duplication based scheduling algorithm for heterogeneous computing systems. In *Proceedings of the International Conference on Parallel Processing, 2002*. 352–359. DOI:http://dx.doi.org/10.1109/ICPP.2002.1040891

A. B. Downey. 1997. *A Model for Speedup of Parallel Programs*. University of California, Berkeley, Computer Science Division.

F. C. Gärtner. 1999. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *Computing Surveys* 31, 1 (March 1999), 1–26. DOI:http://dx.doi.org/10.1145/311531.311532

K. Hashimoto, T. Tsuchiya, and T. Kikuno. 2002. Effective scheduling of duplicated tasks for fault tolerance in multiprocessor systems. *IEICE Transactions on Information and Systems* 85, 3 (2002), 525–534.

S. Hwang and C. Kesselman. 2003. Grid workflow: A flexible failure handling framework for the grid. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, 2003*. IEEE, 126–137.

A. Iosup, M. Jan, O. Sonmez, and D. Epema. 2007. On the dynamic resource availability in grids. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (GRID'07)*. IEEE Computer Society, Washington, DC, 26–33. DOI:http://dx.doi.org/10.1109/GRID.2007.4354112

B. Javadi, J. Abawajy, and R. Buyya. 2012. Failure-aware resource provisioning for hybrid cloud infrastructure. *Journal of Parallel and Distributed Computing* 72, 10 (2012), 1318–1331. DOI:http://dx.doi.org/10.1016/j.jpdc.2012.06.012

B. Javadi, R .K. Thulasiram, and R. Buyya. 2011. Statistical modeling of spot instance prices in public cloud environments. In *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing*. DOI:http://dx.doi.org/10.1109/UCC.2011.37

D. S. Johnson and M. R. Garey. 1979. *Computers and Intractability—-A Guide to the Theory of NP-Completeness*. W. H. Freeman.

G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. 2013. Characterizing and profiling scientific workflows. *Future Generation Computer Systems* 29, 3 (2013). DOI:http://dx.doi.org/10.1016/j.future.2012.08.015

G. Juve and E. Deelman. 2010. Scientific workflows and clouds. *Crossroads* 16, 3 (2010), 14–18. http://dl.acm.org/citation.cfm?id=1734166

G. Kandaswamy, A. Mandal, and D. A. Reed. 2008. Fault tolerance and recovery of scientific workflows on computational grids. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'08)*. 777–782. DOI:http://dx.doi.org/10.1109/CCGRID.2008.79

D. Kondo, B. Javadi, A Iosup, and D. Epema. 2010. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *Proceedings of the IEEE 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*. 398–407. DOI:http://dx.doi.org/10.1109/CCGRID.2010.71

J. Li, M. Humphrey, Y. Cheah, Y. Ryu, D. Agarwal, K. Jackson, and C. van Ingen. 2010. Fault tolerance and scaling in e-science cloud applications: Observations from the continuing development of MODIS-Azure. In *Proceedings of the IEEE 6th International Conference on e-Science (e-Science'10)*. 246–253. DOI:http://dx.doi.org/10.1109/eScience.2010.47

D. Lifka, I. Foster, S. Mehringer, M. Parashar, P. Redfern, C. Stewart, and S. Tuecke. 2013. *XSEDE Cloud Survey Report*. Technical Report. National Science Foundation, USA. https://www.ideals.illinois.edu/handle/2142/45766/.

A. Litke, D. Skoutas, K. Tserpes, and K. Varvarigou. 2007. Efficient task replication and management for adaptive fault tolerance in Mobile Grid environments. *Future Generation Computer Systems* 23, 2 (2007), 163–178. DOI:http://dx.doi.org/10.1016/j.future.2006.04.014

M. Ming and M. Humphrey. 2012. A performance study on the VM startup time in the cloud. In *Proceedings of the IEEE 5th International Conference on Cloud Computing*. DOI:http://dx.doi.org/10.1109/CLOUD.2012.103

D. Mosse, R. Melhem, and S. Ghosh. 1994. Analysis of a fault-tolerant multiprocessor scheduling algorithm. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers*, 16–25. DOI:http://dx.doi.org/10.1109/FTCS.1994.315661

S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. 2010. A performance analysis of EC2 cloud computing services for scientific computing. *Cloud Computing* (2010), 115–131.

S. Ostermann and R. Prodan. 2012. Impact of variable priced cloud resources on scientific workflow scheduling. In *Parallel Processing Euro-Par*. Vol. 7484. Springer.

K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertesz, and P. Kacsuk. 2009. Fault detection, prevention and recovery in current grid workflow systems. In *Grid and Services Evolution*. Springer US, 1–13. DOI:http://dx.doi.org/10.1007/978-0-387-85966-8_9

D. Poola, K. Ramamohanarao, and R. Buyya. 2014. Fault-tolerant workflow scheduling using spot instances on clouds. *In Proceedings of the International Conference on Computational Science in the Procedia Computer Science, 2014.* 29 (2014), 523–533. DOI:http://dx.doi.org/10.1016/j.procs.2014.05.047

S. Ranaweera and D. P. Agrawal. 2000. A task duplication based scheduling algorithm for heterogeneous systems. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing, 2000. IPDPS 2000.* 445–450. DOI:http://dx.doi.org/10.1109/IPDPS.2000.846020

Z. Shi, E. Jeannot, and J. J. Dongarra. 2006. Robust task scheduling in non-deterministic heterogeneous computing systems. In *Proceedings of the IEEE International Conference on Cluster Computing, 2006*. IEEE, 1–10.

D. Sun, G. Chang, C. Miao, and X. Wang. 2013. Analyzing, modeling and evaluating dynamic adaptive fault tolerance strategies in cloud computing environments. *Journal of Supercomputing* 66, 1 (2013). DOI:http://dx.doi.org/10.1007/s11227-013-0898-7

X. Tang, K. Li, and G. Liao. 2014. An effective reliability-driven technique of allocating tasks on heterogeneous cluster systems. *Cluster Computing* (2014), 1–13. DOI:http://dx.doi.org/10.1007/s10586-014-0372-1

X. Tang, K. Li, G. Liao, and R. Li. 2010. List scheduling with duplication for heterogeneous computing systems. *Journal of Parallel and Distributed Computing* 70, 4 (2010), 323–329. DOI:http://dx.doi.org/10.1016/j.jpdc.2010.01.003

W. Voorsluys, S. Garg, and R. Buyya. 2011. Provisioning spot market cloud resources to create cost-effective virtual clusters. In *Algorithms and Architectures for Parallel Processing*, Vol. 7016. Springer. DOI:http://dx.doi.org/10.1007/978-3-642-24650-0_34

Z. Yang, A. Mandal, C. Koelbel, and K. Cooper. 2009. Combined fault tolerance and scheduling techniques for workflow applications on computational grids. In *Proceedings of the IEEE 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'09).* 244–251. DOI:http://dx.doi.org/10.1109/CCGRID.2009.59

N. Yigitbasi, M. Gallet, D. Kondo, A Iosup, and D. Epema. 2010. Analysis and modeling of time-correlated failures in large-scale distributed systems. In *11th IEEE/ACM International Conference on Grid Computing (GRID), 2010.* 65–72. DOI:http://dx.doi.org/10.1109/GRID.2010.5697961

J. Yu and R. Buyya. 2005. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing* 3, 3 (2005), 171–200.

Received December 2014; revised April 2015; accepted August 2015