Procedia Computer Science

Volume 29, 2014, Pages 523–533

ICCS 2014. 14th International Conference on Computational Science

Fault-Tolerant Workflow Scheduling Using Spot Instances on Clouds

Deepak Poola, Kotagiri Ramamohanarao, and Rajkumar Buyya

Cloud Computing and **D**istributed **S**ystems (CLOUDS) Laboratory
Department of Computing and Information Systems, The University of Melbourne, Australia
`deepakc@student.unimelb.edu.au,{kotagiri,rbuyya}@unimelb.edu.au`

Abstract

Scientific workflows are used to model applications of high throughput computation and complex large scale data analysis. In recent years, Cloud computing is fast evolving as the target platform for such applications among researchers. Furthermore, new pricing models have been pioneered by Cloud providers that allow users to provision resources and to use them in an efficient manner with significant cost reductions. In this paper, we propose a scheduling algorithm that schedules tasks on Cloud resources using two different pricing models (spot and on-demand instances) to reduce the cost of execution whilst meeting the workflow deadline. The proposed algorithm is fault tolerant against the premature termination of spot instances and also robust against performance variations of Cloud resources. Experimental results demonstrate that our heuristic reduces up to 70% execution cost as against using only on-demand instances.

Keywords: Workflows, Cloud, Fault-Tolerance, Scheduling, Spot Instances

1 Introduction

Cloud computing is a large-scale distributed computing paradigm offering computing resources (e.g., networks, servers, storage, applications and data) as a subscription based service. These resources are elastically scalable and highly available. They are dynamically provisioned and delivered in a transparent manner without manual intervention [14]. As a result, large number of organizations are moving towards it [10].

Cloud computing is increasingly used amidst researchers for scientific workflows to perform high throughput computing and data analysis [10]. 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 them easier to execute automatically, improving the application performance, and reducing the time required to obtain scientific results [4, 9].

Clouds are realizing the vision of utility computing by delivering computing resources as services. This is facilitating Cloud providers to evolve various business models around these

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.047

services. Most providers provision Cloud resources (e.g., Virtual Machines (VMs) instances) on a pay-as-you-go basis charging fixed set price per unit time. However, Amazon, one of the pioneers in this space, started selling idle or unused datacenter capacity as Spot Instances (SI) from around December 2009. The provider determines the price of a SI (spot price) based on the instance type and demand within the data center, among other parameters [6]. Spot price of a instance varies with time, it is different for different instance types. The price also varies between regions and availability zones. Here, 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 [1]. However, when the spot price becomes higher than the user bid, Amazon terminates the resources.

On-demand and SIs have the same configurations and characteristics. Nonetheless, SIs offers Cloud users reduction in costs of up to 60% for multiple applications like bag-of-tasks, web services and MapReduce workflows [12, 16]. Significant cost reductions are achieved due to lower QoS which make them less reliable and prone to out-of-bid failures. This introduces a new aspect of reliability into the SLAs and the existing trade-offs making it challenging for Cloud users [6].

Scientific workflows can benefit from SIs with an effective bidding and an efficient fault-tolerant mechanism. Such a mechanism can tolerate out-of-bid failures and reduce the cost immensely.

In this paper, we present a just-in-time 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 scheduling algorithm, for every ready task, evaluates the critical path and computes the slack time, which is the time difference between the deadline and the critical path time. The main motivation of the work is to exploit SIs to the extent possible within the slack time. As the slack time decreases due to failures or performance variations in the system, the algorithm adaptively switches to on-demand instances. The algorithm employs a bidding strategy and checkpointing to minimize cost and to comply with the deadline constraint. Checkpointing can tolerate instance failures and reduce execution cost, in spite of an inherent overhead [13] .

The **key contributions** of this paper are: 1) A just in-time scheduling heuristic that uses spot and on-demand resources to schedule workflow tasks in a robust manner. 2) An intelligent bidding strategy that minimizes cost.

2 Background

A **Workflow** is represented as a Directed Acyclic Graph (DAG), $G = (T, E)$, where T is a set of nodes, $T = \{t_1, t_2, ..., t_n\}$, and each node represents a task. Here, 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 calculated based on the size of the data transferred and the Cloud datacenter internal network bandwidth. Additionally, each workflow task t_j also has a task length len_j 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. The deadline D is considered as a constraint, 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 ST is the submission time and $finish_{t_n}$ is finish time of the exit node the of the workflow.

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. 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. However, 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.

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 predecessors estimated time and the data transfer time calculated as per Equation 1 given by Topcuoglu et al. [15],

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

where, $pred(t_i)$ is all the parent nodes of t_i, 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_i to t_p. 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.

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 LTO and switches to on-demand instance later. LTO aids in choosing the right instance, to speed up or speed down and choose the apt pricing model. It is determined for every ready task and the scheduling decisions are made based on the current time t and the LTO. LTO at time t is difference between the deadline and the critical path ($LTO_t = D - CP_t$).

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).

3 System Model

The system architecture is presented in Figure 1. 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 *Dispatcher* analyses 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 Strategy: SIs are prone to out-of-bid failures and an efficient fault tolerant strategy is crucial for a deadline constraint workflow scheduling. Checkpointing is an effective fault tolerant mechanism [13] for spot markets, it takes a snapshot periodically and saves redundant computation in case of failure. It is especially useful in a SI scenario as we save

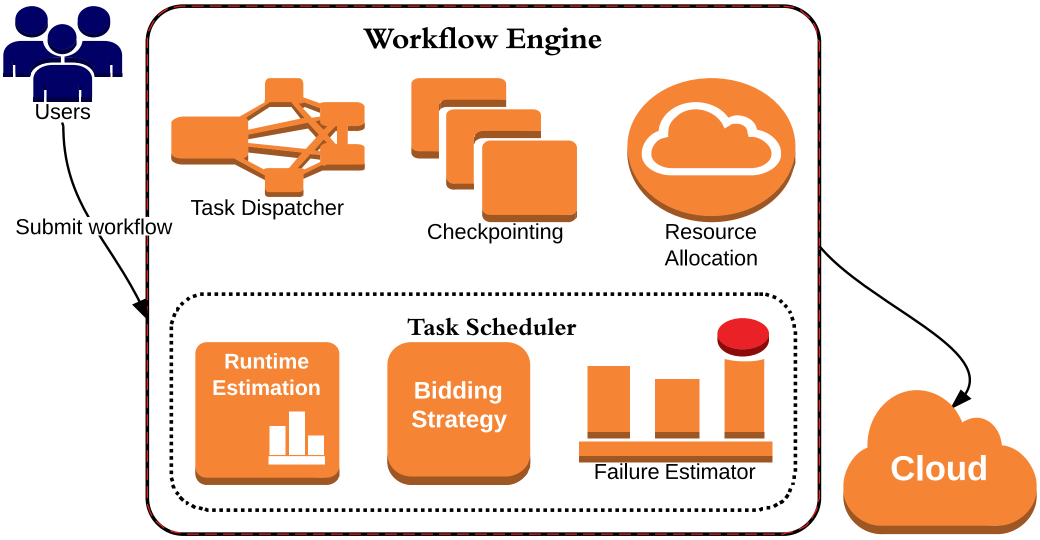

Figure 1: System Architecture.

partial computation in the event of failure and do not pay for that. We use checkpointing mechanism as a fault tolerant strategy. Checkpoints are taken periodically at a user defined frequency. Checkpointing overhead time is taken into account. However, the cost of storing checkpoints is not considered, as the price of storage service is negligible compared to cost of VMs [13]. Moreover, checkpointing can be done in parallel with the computation, so the time taken to transfer checkpointing data is ignored as it is insignificant.

Resource Allocation: Task scheduler chooses Cloud resource 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 scheduling algorithm are outlined in the next section.

Runtime Estimation: To determine the runtime of a workflow task on a particular instance type, we use Downey's analytical model [5]. 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 et al. [3] 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 2. This estimation is used while evaluating the bid value and also while scheduling the task.

$$FP_{bid_t} = OBT_{bid_t}/HT \tag{2}$$

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 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 datacenter is free. The datacenter 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 datacenter capacity.

4 Proposed Approach

4.1 Scheduling Algorithm

The proposed just in-time scheduling algorithm maps ready tasks submitted by the task dispatcher onto Cloud resources. It selects a suitable instance type based on the deadline constraint and the *LTO*. The algorithm along with a suitable instance type also selects an apt pricing model to minimize the overall cost. The outline of the algorithm is given in Algorithm 1. Mapping workflow tasks onto heterogeneous instance types with different pricing models is a well known NP-complete problem [7]. Hence, we propose a heuristic to address the same.

The crux of the algorithm is to map tasks that arrive before the *LTO* to SIs and those that arrive after the *LTO* to on-demand instances. In this approach, a single SI type is used. This instance has lowest cost. The rationale behind this is to minimize the overall execution cost. On the other hand, multiple types of on-demand instances are used. This helps to speed up and slow down execution.

Initially, *CP* and *LTO* are computed before the workflow execution. They are recomputed for all ready tasks during execution. Whilst recomputing the *CP* time, if there are any running tasks in the critical path, the time left for their execution is only accounted. This reflects a realistic *CP* time at that point, giving the algorithm a strong approximation of the time left for the completion of the workflow.

Run time of a particular task vary with different instance types. Similarly, the critical path also varies depending on the instance type used to estimate the same. Henceforth, the *LTO* also varies accordingly. The scheduling decision changes depending on the instance type used to estimate the critical path. We have developed two algorithms keeping this aspect in consideration, namely Conservative and Aggressive.

Conservative algorithm: it estimates the *CP* and *LTO* on the lowest cost instance type. The *CP* estimated in this approach is usually the longest. Hence, it uses SIs only when the deadlines are relaxed. Under tight and moderate deadlines, it does not generate enough slack time to utilize SIs and therefore maps tasks predominantly to on-demand instances. It is conservative in approach and utilizes SIs in a cautious manner only under relaxed deadline making it more robust.

Aggressive algorithm: it estimates the *CP* and *LTO* on a highest cost instance type. Here, the *CP* is smaller than the Conservative algorithm. This approach generates more slack time than the Conservative algorithm and therefore uses SIs even with a strict deadline. This offers significant reduction in cost under moderately relaxed deadline. Under relaxed deadline both algorithms perform similarly. When the market is volatile inducing failures, this approach has less slack time. Hence, it has to opt on-demand instances that are expensive, increasing the overall cost. The performance of these two algorithms is investigated in the evaluation section.

Algorithm 1 outlines the generic heuristic, which is common to both Conservative and the Aggressive algorithms. When a new task is ready to be mapped, the algorithm through the

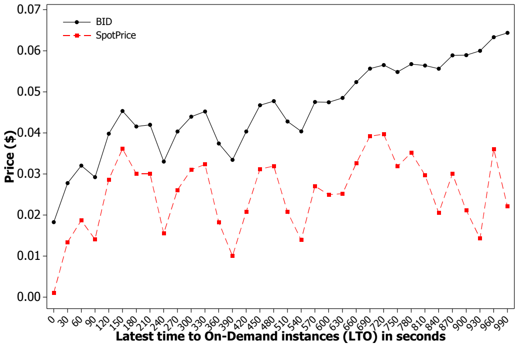

Figure 2: Generation of bid value through Intelligent Bidding Strategy.

method *FindFreeSlot* tries to pick empty slots among the existing running instances. If there is no free slot it searches for a running instance that will be free before the task's latest start time. Latest start time is the latest time a task can start its execution such that the whole workflow can finish within the deadline. Finding such free slots reduces cost as the algorithm avoids creating new instances for every task. This also saves time as the initiation time for starting a new instance is avoided. Additionally, the algorithm creates a new instance when there are no existing instances available before the latest start time of the task.

SIs offer the compute instance at a much lower price. These are terminated prematurely if the bid price goes below the spot price. The failure of SIs is governed by the bid price. Hence, an intelligently calculated bid price reduces the risks of failures. The bid price is provided by the one of the bidding strategies, which is explained later. If the bid price is higher than the on-demand price, the algorithm chooses on-demand instances as they offer higher QoS, as shown in line 15-16. Additionally, the bid price fluctuates with the spot price. Therefore, the algorithm makes sure the bid price is higher than the previous bid price, if not the previous bid price is used. The algorithm also estimates the failure probability of a bid price based on the spot price history (line 17-19). Failure probability of the current bid price is estimated by the failure estimator as explained earlier. If the failure probability is higher than a user defined threshold, the algorithm chooses on-demand instance instead of SI. Lines 14-19 of the Algorithm 1 show that, while creating a SI, it also evaluates the risk propositions and bid intelligently. SI with the calculated bid price is instantiated by the resource provisioner.

The other important aspect of the algorithm is choosing the right instance type. When the algorithm chooses SIs, it selects the cheapest instance type to minimize the cost. However, while choosing on-demand instances the algorithm has to select a cost-effective instance type to satisfy the deadline constraint. The $FindSuitableInstances$ method in Line 20 computes the critical path time for all instance types and creates a list of instance types whose critical path time satisfy the deadline constraint. The algorithm further tries to find an already running instance of type contained in the list to assign to the task. If no suitable instance type is found, the $FindCostPerfEffectiveVM$ method estimates the critical path time for the each instance type. It then 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. Further, through the resource provisioner the selected instance type is instantiated.

The *time complexity* for calculating the critical path and re-computing 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 1: Schedule(t)

```
input : task $t_i$
$vms$ ← all VMs currently in the pool;
$types$ ← available instance types;
$estimates$ ← compute estimated runtime of task $t_i$ on each $type \in types$;
Recompute $CP$ and $LTO$.
$timeLeft = LTO - currentTime$
if $timeLeft > 0$ then
    $decision$ ← FindFreeSpace($t_i$, $vms$, PriceModel.ANY);
    if $decision.allocated = true$ return $decision$;
    if $decision.allocated = false$ then
        $decision$ ← FindRunningVM($t_i$, $vms$, PriceModel.ANY);
        if $decision.allocated = true$ return $decision$;
$timeLeft = timeLeft - vmInitTime$
if $timeLeft > 0$ then
    $bid$ ← EstimateBidPrice($t_i$, $type$);
    if $bid > on\text{-}demand\ price$ then
        Map to on-demand instance and return $decision$.
    $failProb$ ← EstimateFailureProbability($bid$);
    if $failProb < threshold$ then
        Map to spot instance and return $decision$;
$InstanceList$ ← FindSuitableInstances($CP$, $D$)
$decision$ ← FindFreeSpace($t_i$, $InstanceList$, PriceModel.ONDEMAND);
if $decision.allocated = true$ return $decision$;
if $decision.allocated = false$ then
    $decision$ ← FindRunningVM($t_i$, $InstanceList$, PriceModel.ONDEMAND);
    if $decision.allocated = true$ return $decision$;
// If no running instance is found from $InstanceList$ return $decision$ ←
FindCostPerfEffectiveVM($t_i$, $InstanceList$);
```

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 $O(n^2)$.

4.2 Bidding Strategies

Three bidding strategies are presented here, which are used by the scheduling algorithm to obtain a bid price whilst instantiating a SI.

1. **Intelligent Bidding Strategy** this strategy takes into account the current spot price (p_{spot}), on-demand price (p_{OD}), failure probability (FP) of the previous bid price, LTO, the current time (CT), α and β. α, as seen in Equation 3, dictates how much higher the bid value must be above the current spot price. β determines how fast the bid value reaches the 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 according to Equation 3 given below. 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. Lower the value of α, higher is the value of the bid w.r.t the spot price. Figure 2 shows the working on this bidding strategy with spot price varying with time, it also shows that the bid value steeps up towards the end to reach closer to the on-demand price. This 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 value of β, the faster the bid reaches closer to on-demand price. The bidding strategy considers all these factors and calculates a bid value in accordance to the situation.

$$\gamma = (-\alpha(LTO - CT))/FP$$

$$bid = \mathrm{e}^{\gamma} * p_{OD} + (1 - \mathrm{e}^{\gamma} * (\beta * p_{OD} + (1 - \beta) * p_{spot})) \tag{3}$$

2. **On-Demand Bidding Strategy** uses the on-demand price as the bid price.

3. **Naive Bidding Strategy:** uses the current spot price as the bid price for the instance.

5 Performance Evaluation

5.1 Simulation Setup

CloudSim [2] was used 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: Large LIGO workflow with size of 1000 tasks was considered, its characteristics is explained in detail by Juve et al. [8]. 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 datacenter is considered. The VMs/Cloud resources are modeled similar to Amazon EC2 instances. We have considered 9 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 m1.small for SI. The prices of on-demand instances are adapted from the Linux based instances of Amazon EC2 US West region (North California availability zone). The spot price history is taken from the same region from the period of July 2013 - October 2013. The spot price for this period has a mean of \$0.05475 with a standard deviation of 0.239 and a minimum of \$0.007 and a maximum of \$3. In this period, the spot market has around 445 peaks exceeding the on-demand price, making it highly volatile and a suitable time period for testing our methods. A charging period of 60 minutes is considered. A boot/startup time of 100 seconds is considered for each instance [11].

Baseline Algorithms: We developed six baseline algorithms to compare our heuristics and bidding strategy. We developed a full *on-demand baseline algorithm (ODB)* which works similar to our conservative algorithm but maps tasks only to on-demand instances. Similarly, we developed a full *spot baseline algorithm (SPB)*, which uses only SIs with a naive bidding strategy. Additionally, a *conservative with on-demand bidding strategy (CODB)*, *conservative with naive bidding strategy (CNB)*, *aggressive with on-demand bidding strategy (AODB)* and *aggressive with naive bidding strategy (ANB)* are also presented.

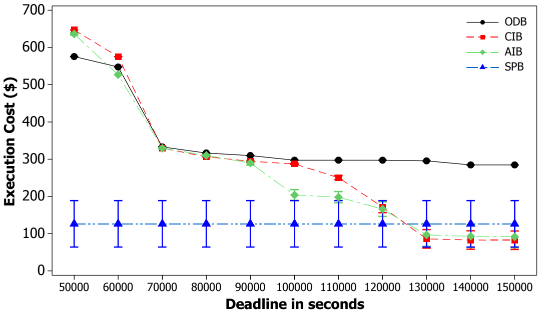

Figure 3: Mean execution cost of algorithms with varying deadline (with 95% confidence interval).

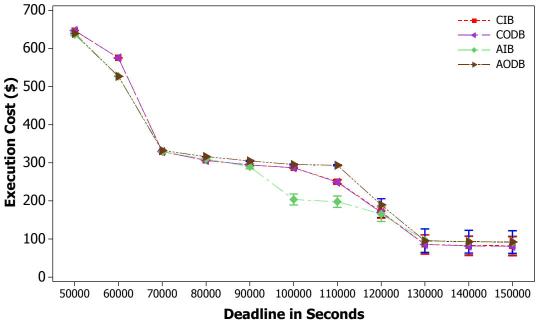

Figure 4: Mean Execution Cost of bidding strategies with varying deadline (with 95% confidence interval).

5.2 Analysis and Results

In this section, we discuss the execution cost incurred by our algorithms, effect of bidding strategies on execution cost, and also the effect of checkpointing on our model. Here, the performance of the algorithms *Conservative with intelligent bidding (CIB)* and *Aggressive with intelligent bidding (AIB)* is investigated against the baseline algorithms. Each experiment runs for 30 times, on each run we randomly change the execution start time in the spot trace, to experience the effect of different price changes. The average value of these 30 runs is reported. Additionally, a sensitivity analysis for the Intelligent Bidding Strategy parameters α and β was performed. Values 0.0005 and 0.9 for α and β respectively gave the best results, which are used in the following experiments. Failure threshold parameter value was set to 1 in these experiments, to demonstrate the working of the algorithm and the bidding strategy.

In our experiments, the deadline varies from strict to moderate to relaxed. A strict deadline being one where high performance instances are needed to complete the execution (e.g. deadlines 50000-80000 seconds in Figures 3, 4 and 5). A moderate deadline is met using a combination of low and high performance instances (e.g. deadlines between 90000-120000). Lastly, a relaxed deadline can be achieved using slow performance instances (e.g. deadlines above 130000).

The monetary cost incurred by our algorithms can be observed in Figure 3. AIB and CIB perform similar to on-demand baseline algorithm with strict and relaxed deadline. AIB algorithm starts using SIs under moderately relaxed deadline giving 28.8% reduction in costs in comparison to ODB and 13.7% w.r.t CIB algorithm. When the deadline is lenient, AIB reduces cost as large as 67.5% w.r.t ODB. On the other hand, the CIB uses SIs more cautiously. CIB offers 16.6% lower cost in comparison to the ODB algorithm when the deadline is moderately relaxed. However, when the deadline is relaxed, it generates saving as high as 71% compared to ODB algorithm. CIB and AIB predominantly use on-demand instances when the deadline is strict. Therefore, have higher costs with lower deadline violations. They also perform better under relaxed deadline as compared to SPB. This is because they use an efficient bidding strategy and use SIs only when its price is lower than the on-demand price. Thenceforth, the costs of CIB and AIB under relaxed deadline are 25.8% and 33.7% lower than SPB respectively.

The effectiveness of our bidding strategy is presented in Figure 4. Our bidding strategy is compared against the on-demand bidding strategy, which bids the on-demand price of the instance. Figure 4 shows that Conservative algorithm performs similarly with both the bidding strategies. However, the aggressive algorithm performs better under intelligent bidding strategy,

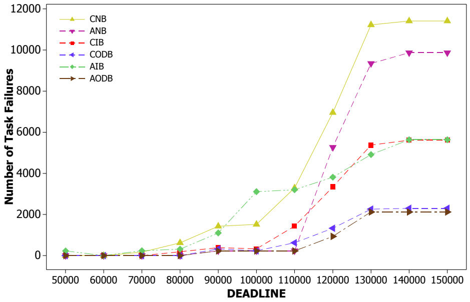

Figure 5: Mean of task failures due to bidding strategies.

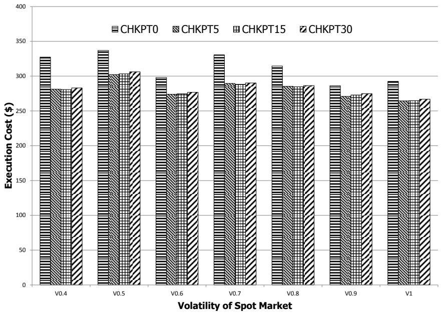

Figure 6: Effect of checkpointing on execution cost.

especially with moderate deadlines. AIB saves 20.3% cost as against AODB. AIB is able to reduce cost as it bids low initially, and since it has enough slack time it is able to tolerate out-of-bid failures. Additionally, checkpointing also saves redundant computing reducing the makespan. Even though the task failures for AIB are higher than AODB as shown in Figure 5, it does not violate the deadline. Moreover, it reduces costs due to its dynamic bidding strategy.

Figure 5 shows the number of failures for conservative and aggressive algorithms under different bidding strategies. It can be observed that naive bidding strategy has the highest failures. However, as the algorithm is adaptive, the impact of failures is not reflected on the execution time. As the figure shows, failures under strict and moderate deadlines are low as the slack time is less. Failures are high under relaxed deadline as the slack time is high. Experimental results show that there is no deadline violation and the algorithm is able to withstand failures irrespective of the bidding strategy.

Figure 6 demonstrates the effectiveness of checkpointing. Here, checkpointing with four different frequencies is used for different volatilities of the spot market. The volatility of the spot market is varied by changing the scale of the inter price time i.e., the time between two spot prices. Time between two consequent price change events is reduced, making the price changes more frequent. This in effect compresses the spot market to a smaller time interval. This makes the peaks in the spot market more frequent increasing the risk of pre-emptions. Four different frequencies of checkpointing are used: no checkpointing (CHKPT0), every 5 minutes (CHKPT5), every 15 minutes (CHKPT15) and 30 minutes (CHKPT30). It can be observed that when there is no checkpointing, the cost of execution is 9-14% higher. CHKPT5 gives better reduction in costs than CHKPT15 and CHKPT30. It can be observed that the execution cost between the CHKPT5, CHKPT15 and CHKPT30 are comparable without significant difference. This can be attributed to low spot prices, the price history we have considered has 82.7% of price changes below \$0.01. Therefore, when the average spot price is higher, we will observe a significant difference. Under the spot market considered, CHKPT30 is better as the overhead is lower than CHKPT5, CHKPT15.

6 Conclusions and Future Work

In this paper, two scheduling heuristics that map workflow tasks onto spot and on-demand instance are presented. They minimize the execution cost. They are shown to be robust and fault-tolerant towards out-of-bid failures and performance variations of Cloud instances. A bidding strategy that bids in accordance to the workflow requirements to minimize the cost is also presented. This work also demonstrates the use of checkpointing and offers cost savings up to 14%. Simulation results show that cost reductions of upto 70% are achieved under relaxed deadlines, when SIs are used.

In the current work, slack time is exploited to schedule SIs. In our future work, we like to consider task replication using SIs to provide fault-tolerance. We are interested to investigate efficient failure prediction methods for SIs. Further, we wish to extend this work to use multiple SI types and investigate its effect on cost and performance.

References

[1] Amazon EC2 Spot Instances. `http://aws.amazon.com/ec2/spot-instances/`, 2009. [Online; accessed 23-October-2013].

[2] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A.F. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[3] W. Cirne and F. Berman. A model for moldable supercomputer jobs. In *Proceedings of 15th Int'l Parallel and Distributed Processing Symp.*, 2001.

[4] E. Deelman, G. Juve, M. Rynge, J. Voeckler, and G. Berriman. Comparing futuregrid, amazon ec2, and open science grid for scientific workflows. Number 99, 2013.

[5] A. B. Downey. *A model for speedup of parallel programs.* University of California, Berkeley, Computer Science Division, 1997.

[6] B. Javadi, R.K. Thulasiram, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. In *4th IEEE Int'l Conf. on Utility and Cloud Computing*, 2011.

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

[8] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3), 2013.

[9] G. Juve and Ewa. Deelman. Scientific workflows and clouds. *Crossroads*, 16(3):14–18, 2010.

[10] D. Lifka, I. Foster, S. Mehringer, M. Parashar, P. Redfern, C. Stewart, and S. Tuecke. Xsede cloud survey report. Technical report, National Science Foundation, USA, 2013.

[11] M. Ming and M. Humphrey. A performance study on the vm startup time in the cloud. In *IEEE 5th International Conference on Cloud Computing*, 2012.

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

[13] Y. Sangho, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *3rd IEEE Int'l Conf. on Cloud Computing*, 2010.

[14] D. Sun, G. Chang, C. Miao, and X. Wang. Analyzing, modeling and evaluating dynamic adaptive fault tolerance strategies in cloud computing environments. *J. Supercomputing*, 66(1), 2013.

[15] H. Topcuoglu, S. Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *8th Proc. of Heterogeneous Computing Workshop*, 1999.

[16] W. Voorsluys, S. Garg, and R. Buyya. Provisioning spot market cloud resources to create cost-effective virtual clusters. In *Algorithms and Architectures for Parallel Processing*, volume 7016. Springer, 2011.