FarSpot: Optimizing Monetary Cost for HPC Applications in the Cloud Spot Market

Amelie Chi Zhou, Jianming Lao, Zhoubin Ke, Yi Wang and Rui Mao

Abstract—Recently, we have witnessed many HPC applications developed and hosted in the cloud, which can benefit from the elastic and diversified resources on the cloud, while on the other hand confronting high costs for executing the long-running HPC applications. Although public clouds such as Amazon EC2 offer spot instances with dynamic and usually low prices compared to on-demand ones, the spot prices can vary significantly and sometimes can even be more expensive than on-demand prices of the same type. Previous work on reducing the monetary cost for HPC applications using spot instances focused on designing fault tolerance techniques or selecting appropriate instance types/bid prices to make good usage of the low spot prices. However, with the recent update of spot pricing model on Amazon EC2, these work may become either inefficient or invalid. In this paper, we present FarSpot which is an optimization framework for HPC applications in the latest cloud spot market with the goal of minimizing application cost while ensuring performance constraints. FarSpot provides accurate long-term price prediction for a wide range of spot instance types using ensemble-based learning method. It further incorporates a cost-aware deadline assignment algorithm to distribute application deadline to each task according to spot price changes. With the assigned subdeadline of each task, FarSpot dynamically migrates tasks among spot instances to reduce execution cost. Evaluation results using real HPC benchmark show that 1) the prediction error of FarSpot is very low (below 3%), 2) FarSpot reduced the monetary cost by 32% on average compared to state-of-the-art algorithms, and 3) FarSpot satisfies the user-specified deadline constraints at all time.

Index Terms—Cloud computing, Spot market, Price prediction, Ensemble Models

1 Introduction

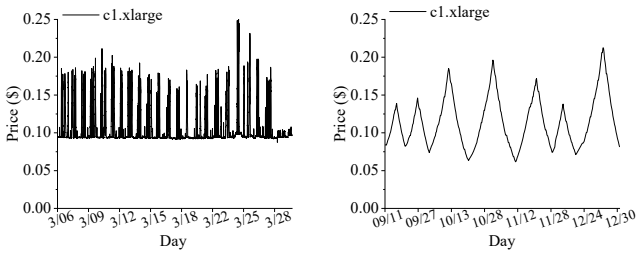
Recently, we have witnessed that many emerging high performance computing (HPC) or scientific computing applications are developed and hosted in the cloud [17]. As those applications are usually long running jobs and are costly in the cloud, monetary cost [26, 42] and performance [14, 15] are important optimization factors. Message Passing Interface (MPI) is the key programming paradigm for developing HPC and scientific applications. That motivates us to investigate whether and how we can reduce the monetary cost for MPI-based applications with performance constraint in the cloud.

Most public cloud providers such as Amazon Elastic Compute Cloud (EC2) offer a wide range of instance types. Users can pay for the instances with different pricing models, such as on-demand instances and spot instances. Spot instances have dynamic prices and the spot price can reach up to 90% discount compared to the on-demand prices of the same type. When using spot instances, users are supposed to offer a bid price. When the bid price is higher than the spot price, the chosen spot instance is successfully acquired. When the spot price increases and becomes higher than the bid price, a running spot instance will be stopped (i.e., an out-of-bid event is occurred). Thus, although spot instances can lead to lower cost, they can also cause execution failures due to unexpected out-of-bid events. This is especially devastating to MPI-based applications since the failure of one task leads to the failure of the entire job (assuming no fault tolerance technique is applied). Although one can set the bid price to be extremely high to avoid out-of-bid events, it could lead to high cost since spot prices may increase due to the high demand of the market and greatly exceed the on-demand prices occasionally [18].

Many existing studies [16, 27, 37, 39] have been working on the cost and performance optimizations for HPC applications in the spot market. Some of these studies focused on designing various fault tolerance techniques to reduce job failure rates while benefiting from the low price of spot instances. However, these techniques either recover failed job executions from checkpoints or from replications [16], both of which lead to additional cost. Another widely studied way is to dynamically migrate running tasks between instances to minimize task execution cost [18, 35, 36]. However, most of the existing studies make migration decisions according to near-future trade-offs between execution time and cost evaluated using the current spot prices as short-term predictions of future prices [18]. On the one hand, such simple model can easily lead to miss predictions and hence high cost. On the other hand, due to lack of long-term knowledge on spot prices, we may end up with too many migrations. A few studies have been working on making accurate price predictions and selecting appropriate bid prices to make good usage of spot instances [3]. However, it is hard to accurately predict spot prices which fluctuate frequently based on the short-term relationship between capacity demand and supply [36].

Recently, Amazon EC2 has updated its spot pricing model in December 2017. In the new model, spot prices vary according to the long-term relationship between resource demand and supply. Thus, the spot prices are less frequently

- *A. Zhou, J. Lao, Z. Ke and Y. Wang are with the College of Computer Science and Software Engineering, Shenzhen University. R. Mao is with Shenzhen Institute of Computing Sciences and Shenzhen University.*
- *Rui Mao is the corresponding author.*



(a) Old pricing model (b) New pricing model

Fig. 1. Spot price variations of c1.xlarge instances in: (a) ap-northeast-1c zone from Mar. 6^{th} to Mar. 31^{th}, 2017; and (b) us-west-1b zone from Sep. 11^{th} to Dec. 31^{th}, 2020.

updated and are more predictable. Figure 1 shows the spot price variations of the same instance type under the two generations of pricing models. Clearly the spot price history under the new pricing model has much less spikes compared to the old one. With the new pricing model, users do not need to submit any bid price and only pay for the spot price as it is. In this case, it is important to make accurate spot price predictions and select appropriate instance types to make the best usage of low spot prices. Existing spot price prediction models [2, 3, 8, 9, 40] are mostly based on the previous generation spot pricing model of Amazon EC2, which may not work well under the current spot market due to the different features of the two pricing models.

In this paper, we propose FarSpot - an optimization system for long-running HPC applications in the cloud spot market which aims at minimizing application cost while guaranteeing performance level requirements. There are two main design components in FarSpot, namely an ensemble-based predictor to accurately forecast spot price variations in near future and an instance migration strategy that can dynamically make cost-efficient migration decisions. Due to the better predictability of the current spot pricing model, our predictor can well capture the spot price variations. Evaluations using real HPC applications show that our ensemble-based predictor is highly efficient. It can reduce the training overhead to less than 1 minute and limit the average prediction error rate to below 3%. Based on the price prediction results, we further designed a cost-aware deadline assignment algorithm to distribute application deadline onto each task. Our migration strategy ensures that each task can be finished before its subdeadline with the minimum cost. According to our evaluations, the migration strategy of FarSpot can reduce the overall execution cost of HPC applications by 32% on average compared to state-of-the-art methods while ensuring performance constraints.

This paper makes the following contributions:

- **Accurate spot price prediction.** We carefully analyzed the latest spot pricing model of Amazon EC2 and constructed an ensemble-based model which combines Random Forest [6] and LightGBM [20] with dynamical weights for spot price prediction. Our ensemble-based predictor is able to capture the characteristics of spot price variations and make accurate predictions for a relatively long time period compared to existing models.
- **Cost optimization framework.** We design a cost optimization framework named FarSpot for HPC applications with deadline constraints. Using our spot price predictor, FarSpot incorporates a cost-aware deadline assignment method to optimally distribute application deadline over tasks and an instance migration strategy that dynamically migrates tasks to appropriate spot instance types to reduce the task execution cost.
- **Implementation and Evaluation.** We implement our methods using a simulator. We evaluate FarSpot using real HPC applications, including BT, SP and LU in NPB [31] benchmark. Experimental results indicate that our proposed method can reduce the monetary cost by 32% on average compared to state-of-the-art algorithms [35].

The rest of this paper is organized as follows. Section 2 presents the background and preliminaries. Section 3 shows our motivations and overall design. Section 4 introduces our VWH model, which is an ensemble-based predictor for spot prices. We then propose our deadline distribution policy in Section 5. We also elaborate our selection and migration policy in Section 6. We evaluate the proposed techniques in Section 7. We finally conclude this paper and discuss briefly on future research directions in Section 8.

2 BACKGROUND AND PRELIMINARY STUDIES

2.1 Cloud Spot Market

Many cloud providers offer multiple instance types with different capabilities to satisfy different user requirements. For example, Amazon EC2 offers a wide variety of instance types, such as compute optimized instances for high-performance computing applications and memory optimized instances for in-memory applications. These instance types are charged at different prices on per hour or per minute basis. Many studies have been proposed to optimize instance configurations for different applications to achieve a good trade-off between application performance and cost [17]. In addition to different instance types, most clouds also provide different pricing models to offer more flexibility to users with different requirements on quality of service (QoS). For example, Amazon EC2 provides five ways to pay for instances, such as on-demand and spot instances. On-demand instances provide reliable services to users and are the most commonly used instances among different pricing models. Spot instances are a special type of cloud instances which have dynamic prices adjusted based on long-term trends in supply and demand for spot instances.

Spot prices are usually much lower than on-demand prices of the same type, but can also exceed the on-demand prices occasionally. The spot pricing model of Amazon EC2 before 2018 was designed based on the *short-term* relationship between supply and demand and thus was prone to sudden spot price changes. For example, the spot price can suddenly go up from 10% to over ten times of the on-demand price when the demand for cloud resources increases. When requesting spot instances, users have to specify a bid price. Spot instances can be successfully acquired when the bid price is higher than the spot price and will be terminated when the spot price becomes higher than the bid price. This means that spot instances do not provide reliable services and many existing studies have been proposed to improve the reliability of spot instances and benefit from the low prices [35, 36].

Many existing studies focus on proposing fault tolerance techniques to handle unexpected spot failures. For example, Gong et al. [16] combined two common fault tolerance mechanisms to reduce the failure risks and cost of MPI applications using spot instances. Subramanya et al. [37] optimized the cost of running non-interactive batch jobs on instances by implementing fault-tolerance mechanisms at the systems level. On the other hand, accurate prediction of spot prices can also help users moderate the risk of spot failures thus increase the reliability of spot instances [21]. Some existing studies have proposed complicated models to predict the spot prices. For example, Xu et al. [40] and Baughman et al. [5] both leveraged the Long Short-Term Memory (LSTM) model to forecast spot prices. Chhetri et al. [8] employed time series decomposition-based forecasting method to decompose spot prices into time series components for precise prediction. Khandelwal et al. [21] used regression random forests to perform price prediction. Mishra et al. [29] utilized the probability between price transitions from the observed history for short-term price prediction. Despite the effectiveness of the above studies, most of them are outdated due to the update of spot pricing model.

Recently, Amazon EC2 has updated its spot pricing model in December 2017 [32]. The new pricing model does not require users submitting any bid price and the spot prices are only determined by supply and demand for Amazon EC2 spare capacity. We have analyzed the spot prices of 68 types of instances on Amazon EC2 for over three months (starting from Sept. 15^{th} to Dec. 25^{th}, 2020) and found that the spot price can range between 24%-120% of the on-demand price of the same type. This shows that the spot prices have become more stable and predictable after the model update. Existing studies that are designed based on the previous generation pricing model may no longer work for the updated pricing model.

Some studies have noticed the difference between the two generation pricing models and came up with various new spot prediction models. For instance, Chittora and Gupta [9] have proposed a 2-layer stacked LSTM model for spot price prediction, which is prone to over-fit when using larger datasets. Al-Theiabat et al. [2] also utilized LSTM model to perform spot price prediction, but they can only ensure the accuracy of the price prediction over a relatively short period of time (e.g., the next four hours). In fact, our evaluation shows that the LSTM-based method is prone to over-estimate the peak values or under-estimate the trough values of spot prices, leading to an unfavorable forecast. For example, when using LSTM to predict the long-term spot prices (e.g., 24 hours), the relative error can reach up to 85%.

In summary, although the spot market has been well explored by many existing studies, it is necessary to revisit existing mechanisms to better predict spot prices and make good usage of spot instances to reduce cost.

2.2 Ensemble Learning Methods

Machine learning is one of the most effective methods in the field of data analysis [4]. It can discover hidden patterns or features of data through historical learning and data trends [34], so as to make accurate predictions for the future. When it comes to spot prices, existing studies have leveraged various machine learning algorithms to predict the price of spot instances [1, 41]. Xu et al. [40] utilized LSTM network to predict spot prices in 5-minutes and 1-hour time windows for big data analytics workloads with complex dataflow execution graphs, and thus may not be suitable for long running MPI applications. Alkharif et al. [3] proposed a SARIMA-based approach to predict spot prices. The SARIMA model requires to calculate the autocorrelation and partial autocorrelation functions to determine model parameters, which incurs extra overhead for the price prediction. Therefore, using a single model for spot price prediction may not be able to achieve satisfactory results.

In statistics and machine learning, ensemble methods use multiple learning algorithms to obtain better prediction performance than any component learning algorithm alone [33]. Ensemble methods are usually used to combine several base predictors built with different learning algorithms. Below are three popular ways to combine base predictors:

- **Voting.** This method builds multiple models separately and uses simple statistics (e.g., the mean) to combine the output of different models.
- **Bagging.** This method builds multiple models from different subsets of the training data and then combines these models using voting.
- **Boosting.** This method builds multiple models, each of which learns to correct the prediction errors from a prior model in the sequence of models.

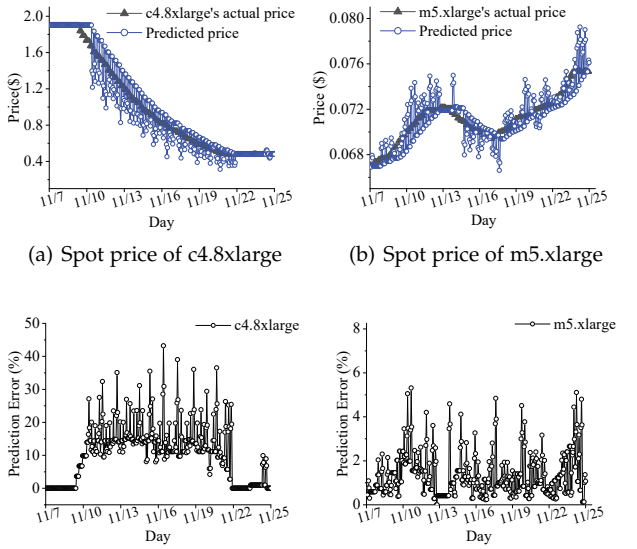
The goal of ensemble methods is to make up for the shortcomings of each individual learning algorithm, thereby improving the generality and robustness of predictors.

The ensemble model has made great contributions in many previous studies. For example, Liu et al. [23] proposed a hybrid forecasting model using multi-resolution ensemble method to carry out wind speed forecasting. Liu et al. [24] combined four base SVM regressors as an ensemble to estimate free lime content in cement clinkers. Liu et al. [25] came up with a hybrid ensemble with time series decomposition algorithm for electrical energy consumption forecasting. Gao et al. [13] designed an adaptive ensemble learning model by combining several base classifiers for intrusion detection. Despite the abundant studies on ensemble learning methods, few has considered using the method for spot price prediction.

3 Motivation and Design Overview

3.1 Motivation

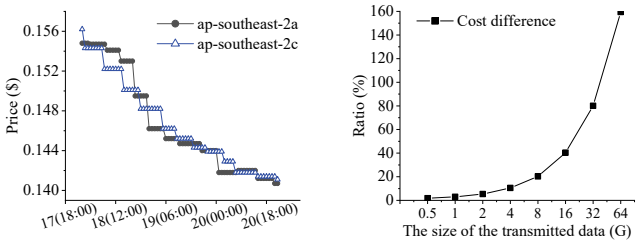
Spot instances provide elastic computing services at low cost. However, with the dynamic change of spot prices, services provided by spot instances may suffer from sudden interruption due to out-of-bid events [38]. When an out-of-bid event occurs, additional time and cost are wasted on the killed executions. On the other hand, one can set the maximum price that he or she is willing to pay to be extremely high to avoid out-of-bid events. However, this may lead to high cost as spot prices can rise up to even higher than on-demand prices. To improve the reliability and reduce the cost of spot instances, it is important to accurately predict spot price changes and migrate the computing from high price instances to low price ones when necessary. However, this task is non-trivial due to the following challenges.



(a) Spot price of c4.8xlarge (b) Spot price of m5.xlarge

(c) Prediction error of c4.8xlarge (d) Prediction error of m5.xlarge

Fig. 2. 24-hour spot price prediction using linear regression for two instance types in eu-west-2c from Nov 7^{th}, 2020 to Nov 25^{th}, 2020.



(a) (b)

Fig. 3. (a) Spot price variation of c5.2xlarge in ap-southeast-2a and ap-southeast-2c zones from Oct 17^{th} to Oct 20^{th}, 2020. (b) Ratio of the cost of a simple migration policy over the cost of no migration.

3.1.1 Prediction Models

Traditional time series forecasting methods, such as linear regression [28], cannot accurately capture the changes in spot prices. As an example, we utilize linear regression to make 24-hour spot price predictions for c4.8xlarge and m5.xlarge instances in eu-west-2c region from Nov 7^{th} to Nov 25^{th}, 2020. As shown in Figure 2, the actual spot price of c4.8xlarge dropped sharply from Nov 8^{th}, while the price of m5.xlarge fluctuates frequently with low variance. When using linear regression to predict the spot price of c4.8xlarge, the error rate goes up to 45%, while the prediction error for m5.xlarge is much lower (less than 6%). This is because traditional time series forecasting methods highly rely on stationary data sets, while spot prices are non-stationary [1].

Ensemble learning algorithms can make up for the weaknesses of single learning algorithms and thus make better prediction results. Random Forest (RF) [6] and LightGBM [20] are two prevalent ensemble methods that have been proven effective on a wide range of predictive modeling problems. Specifically, RF is a bagging-based method that constructs less correlated sub-trees to make combined decision. Thus RF has the advantage of reducing the variance of decisions and tackling the over-fitting issue. LightGBM is a boosting-based method, in which "weak" models are built sequentially and try to fix the training errors of the predecessors. This characteristic notifies that boosting can reduce bias, but it is prone to be over-fitting without careful consideration in tuning the hyper-parameters. As a result, RF and LightGBM are complementary to each other and could work better for spot price prediction if combined together. In order to further limit the total errors by aggregating the predictions from these two models, we propose a dynamically weighted ensemble. We also adopt a SVM classifier to adjust weights dynamically since it is easily adaptable to multi-class problems and is relatively memory efficient.

3.1.2 Migration Policy

Existing studies have been using migration technique to hop between spot instances to reduce the job execution cost [35]. In the migration process, the instance on the original host will be stopped. Then all memory pages from the source will be copied to the destination, and the instance will be resumed again on the destination host. It is clear that there is downtime between stopping the source instance and resuming the destination instance, thus incurring cost and performance overhead. For this reason, merely chasing the lowest spot price may not lead to the lowest job execution cost. For example, consider using c5.2xlarge instances from two different zones for job execution. Figure 3(a) shows the spot price variation in the two zones. A simple migration policy is to always migrate the current instance to the zone with the lowest price. As prices in the two zones change frequently, the frequency of migration is high and thus bringing additional cost. When the size of the memory footprint increases from 512MB to 64GB, the migration cost increases and the execution cost optimized by the simple migration policy increases from 2% to 1.6x of that of no migration. Thus, we need to carefully design the migration policy to reduce execution cost. Also, many existing studies have proposed various methods to estimate the execution time of HPC applications on different platforms [7, 30], which gives us the benefit of accurately estimating the cost and gain when migrating HPC tasks between spot instances.

3.2 Design Overview

Motivated by the above challenges, we need to design a task scheduling algorithm which migrates tasks among different spot instance types and availability zones to achieve low execution cost while satisfying the deadline constraint of the job. To achieve this goal, we first need to accurately predict the spot price variations of different instance types to make the best migration decisions. In this paper, we propose FarSpot, which has two main components. The first component is an ensemble-based predictor to predict spot prices in the near future. Considering the complementary features of RF and lightGBM algorithms, our predictor combines the two algorithms as an ensemble using weighted voting by a Support Vector Machine (SVM) classifier. The second component is a controller, which makes migration decisions dynamically for tasks according to spot price prediction results, system information (such as resource usage) and execution requirements (such as task deadlines). Note that we propose a cost-aware deadline distribution algorithm to assign the subdeadline to each task.

Figure 4 shows the overall framework of FarSpot. Specifically, the Predictor collects historical spot price traces from

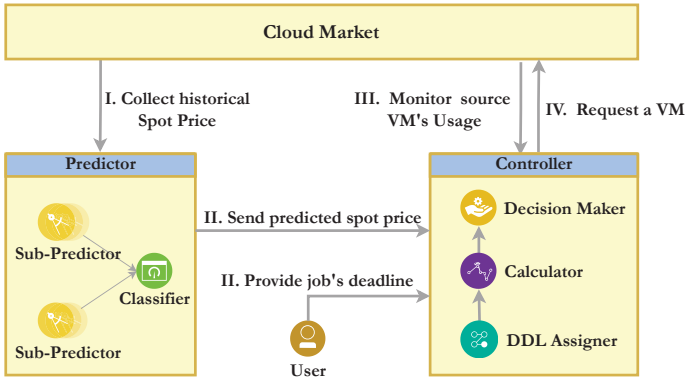


Fig. 4. FarSpot Overview.

the Cloud Market, and predicts spot price changes by combining the prediction results of the two base predictors (i.e., RF and LightGBM) using SVM classifier. The predicted spot price is sent to the Controller to make scheduling decisions. For each running task, the Controller first computes the subdeadline for it; it then monitors the usage of the source instance of the task and decides whether to migrate the task considering the predicted spot price changes and the deadline requirement. If a migration decision is made, the Controller will send a request to the Cloud Market to launch a new instance and migrate the running task accordingly.

In the following three sections, we elaborate the details of the Predictor and Controller components of FarSpot.

4 Ensemble-based Predictor for Spot Prices

In this section, a variable-weight hybrid (VWH) model is proposed by combining a LightGBM and a RF model, where the weights of the two models are dynamically adjusted depending on the relative errors between the actual and predicted spot prices using a SVM classifier. We collect real spot price traces from Amazon EC2 to train our model.

4.1 Basics of LightGBM and RF Models

4.1.1 LightGBM

LightGBM is an open-source Gradient Boosting Decision Tree (GBDT) algorithm proposed by Microsoft. It adopts the leaf-wise strategy to grow trees and perform splits by calculating the gain of variance. The abstract process of LightGBM algorithm can be described as below:

1) Creating and initializing n decision trees with training samples whose weight is $\frac{1}{n}$;
2) Training each weak predictor $f_i(x)$;
3) Calculating and updating the weight of each weak predictor w_i;
4) Constructing final strong predictor $F(x)$ as below:

$$F(x) = w_1 f_1(x) + w_2 f_2(x) + \cdots + w_n f_n(x) \quad (1)$$

Overall, LightGBM is highly efficient in both memory consumption and training speed [20]. This merit allows us to easily retrain our model if cloud providers such as Amazon updated their spot price model again in the future.

4.1.2 Random Forest (RF)

RF is a classification and regression algorithm utilizing bagging method to build numerous decision trees from different training data subsets. For regression tasks such as price forecasting, the mean prediction of individual trees is outputted. For example, given the sample dataset x, RF constructs and grow k trees. After each tree T_i is grown, RF returns its price prediction $F^k_{rf}(x)$ as below:

$$F^k_{rf}(x) = \frac{1}{k}\sum_{i=1}^{k} T_i(x) \quad (2)$$

Although a well-tuned LightGBM is likely to outstrip the RF model, tuning the parameters for LightGBM is by no means easy. In contrast, we can get satisfactory results with RF without caring much about parameters. Also, it is robust in most cases since the RF model does not overfit easily [10].

4.2 Combining LightGBM and RF Models

Due to the complementary feature of LightGBM and RF discussed earlier, we propose to combine the prediction results of the two models to make better spot price forecast. Specifically, we use a SVM-based multi-class classifier to dynamically combine the two models according to the measurement of the combination results. We adopt RE (Relative Error) as the standard metric to measure the accuracy of the results.

$$RE^i_m(t) = \frac{|P^i_m(t) - P^i_a(t)|}{P^i_a(t)} \quad (3)$$

where $RE^i_m(t)$ is the RE of the forecast result from model m on instance i at time t, and $P^i_m(t)$ and $P^i_a(t)$ are the predicted and actual spot price on instance i at time t. Our goal is to reduce RE as much as possible.

Our SVM-based multi-class classifier dynamically assign different weights to each model according to the RE measures. In order to train our classifier, we generate training data from collected spot traces using sliding windows of size W. Each data point represents the spot price in one hour and thus each window contains the spot price history in the past W hours (i.e., $t-W, \ldots, t-1$ for time window t). We label each training data window according to the ratio of the RE of the LightGBM's forecast results to that of the RF model's. That is:

$$R(t) = \frac{|RE_{RF}(t) - RE_{LG}(t)|}{min(RE_{RF}(t), RE_{LG}(t))} \quad (4)$$

where $R(t)$ is the ratio of time t, $RE_{RF}(t)$ and $RE_{LG}(t)$ are the RE of the forecast results of LightGBM and RF of time t, respectively. Therefore, we can finish the data labeling using the criteria listed in Table 1. Furthermore, we train the SVM using the data set generated above. We divide the data set into two parts, namely training set (50%) and evaluation set (50%). Also, we set the regularization parameter to 0.8, whereby the SVM has an average prediction error of 12.5% on the input data whose $R(t)$ is higher than 0.01.

Finally, we combine the time series forecast results across the LightGBM and the RF model with variable weights derived from the SVM. The final forecast result of our VWH model can be calculated as below:

$$P_f(t) = C(t) \times P_{LG}(t) + (1 - C(t)) \times P_{RF}(t) \quad (5)$$

where $C(t)$ is the output of SVM using training data window at time t. $P_{LG}(t)$ and $P_{RF}(t)$ are the forecast results for time t using LightGBM and RF models, respectively.

Here is an example to illustrate the process. When $RE_{RF}(t)$ is 8% and $RE_{LG}(t)$ is 10%, that is, RF is inferior to LightGBM in conducting price prediction for time t,

TABLE 1
Weights Selection for the Combination of Forecasts

RE	R(t)	Label
$RE_{RF}(t) < RE_{LG}(t)$	> 0.03	1
	> 0.02 & $\leqslant 0.03$	2
	> 0.01 & $\leqslant 0.02$	3
	$\leqslant 0.01$	4
$RE_{RF}(t) = RE_{LG}(t)$	\	5
$RE_{RF}(t) > RE_{LG}(t)$	$\leqslant 0.01$	6
	> 0.01 & $\leqslant 0.02$	7
	> 0.02 & $\leqslant 0.03$	8
	> 0.03	9

$R(t)$ can be calculated using Equation 4 to be 0.25. We then label this training data window as Label 1 according to the criterion, notifying that RF is dominant in making price forecast for time t. Specifically, the output of SVM for time t is calculated as $C(t) = Label \times 0.1 = 0.1$, and the final forecast result of our VWH model is calculated as $P_f(t) = 0.1 \times P_{LG}(t) + 0.9 \times P_{RF}(t)$.

5 The Deadline Distribution Policy

Provided with the user specified deadline of the job, we propose our deadline distribution policy to determine the subdeadline of each task. As the application deadline is allocated to individual tasks, FarSpot can ensure that each task is able to finish execution before its subdeadline, thus satisfying the user given deadline.

5.1 Elemental definition

To begin with, for each task t_i, we specify its Earliest Start Time $EST(t_i)$ and Latest Finish Time $LFT(t_i)$ as the earliest time when t_i starts its execution and the latest time when t_i finishes its execution, respectively. Its Minimum Execution Time $MET(t_i)$ is defined as follow:

$$MET(t_i) = \min_{v \in V} ET(t_i, v) \tag{6}$$

where V is the set of instance types, and $ET(t_i, v)$ denotes the total execution time for t_i on instance v. After having the user specified deadline for the whole job $Deadline$, we then can initialize $EST(t_i)$ and $LFT(t_i)$ as follows:

$$EST(t_i) = \begin{cases} 0, & \text{if } t_i \text{ is the first task} \\ EST(t_{i-1}) + MET(t_{i-1}), & \text{otherwise} \end{cases} \tag{7}$$

$$LFT(t_i) = \begin{cases} EST(t_i) + MET(t_i), & \text{if } t_i \text{ is the last task} \\ LFT(t_{i+1}) - MET(t_{i+1}), & \text{otherwise} \end{cases} \tag{8}$$

5.2 The Subdeadline Assigning Algorithm

In our deadline distribution policy, we attempt to find the cheapest distribution plan whereby each task can finish execution before its latest finish time. Thus we propose the subdeadline assigning algorithm as shown in Algorithm 1.

This algorithm traverses the set of tasks starting from the first task to the last one, at each step trying to increase the subdeadline of the current task by a unit of time (line 6-8) and then recompute the cheapest schedule for all tasks (line 9-17). If extending the subdeadline of $task_i$ brings about the lowest execution cost, the algorithm considers this schedule as the best; it then updates all the relative attributes for each task according to the schedule (line 23, 24). However, if there is no saving by increasing the subdeadline for any task, that is, no best schedule currently, the algorithm either increases the subdeadline by a unit of time for all tasks or extends the subdeadline of the last task (line 25-32).

Algorithm 1 Subdeadline Assigning Algorithm

Input: $P_a(V)$ and $P_f(V)$: The set of actual and predicted spot prices, where V is the set of instance types; $EST(T)$ and $LFT(T)$: The set of EST and LFT for all tasks computed by Eq. 7 and Eq. 8, where T is the set of tasks; $MET(T)$: The MET for all tasks; $Deadline$: The user specified deadline;
Output: $SubD$: The set of subdeadlines for each task;

1: set $subD(task_i) \leftarrow EST(task_i) + MET(task_i)$ for all tasks;
2: $D \leftarrow subD(task_n)$;
3: **while** $D < Deadline$ **do**
4: $best \leftarrow null$;
5: $min \leftarrow \infty$;
6: **for all** $task_i \in T$ **do**
7: $D \leftarrow D + a\ unite\ of\ time$;
8: $subD(task_i) \leftarrow subD(task_i) + a\ unite\ of\ time$;
9: **for all** $task_j \in T$ **do**
10: $v \leftarrow$ the VM that has the lowest cost executing $task_j$;
11: $cost_j \leftarrow 0$;
12: **for** $m = 1$ to $ET(task_j, V_v)$ **do**
13: **if** $EST(task_j) + m$ is in the current hour **then**
14: $cost_j \leftarrow cost_j + P_a^{EST(task_j)}(V_v)$;
15: **else**
16: $cost_j \leftarrow cost_j + P_f^{EST(task_j)+m}(V_v)$;
17: update EST and LFT for all tasks;
18: $cost \leftarrow \sum_{k=1}^{t} cost_k$;
19: **if** $cost < min$ **then**
20: $min \leftarrow cost$;
21: set this schedule as $best$;
22: revoke the changes of D, $subD$, EST, and LFT for all tasks;
23: **if** $best$ **then**
24: update D, $subD$, EST, and LFT for all tasks based on $best$;
25: **else if** $D + n \leqslant Deadline$ **then**
26: set $subD \leftarrow subD + a\ unite\ of\ time$ for all tasks;
27: update EST and LFT for all tasks;
28: $D \leftarrow D + n\ unites\ of\ time$;
29: **else**
30: $subD(task_n) \leftarrow subD(task_n) + Deadline - D$;
31: update $EST(task_n)$ and $LFT(task_n)$;
32: $D \leftarrow Deadline$;
33: **return** $subD$;

6 Selection and Migration Policy

In this section, we introduce the selection policy that FarSpot adopts to choose the initial instances for tasks. We then propose a migration policy that decides when and where to migrate a running instance to a new one based on predicted spot prices. The objective of the migration is to reduce job execution cost while satisfying the deadline constraint.

6.1 Initial Instance Selection

Algorithm 2 shows the algorithm adopted by FarSpot to select initial instances for tasks. Specifically, FarSpot first initializes the minimum spot price recorder min and the selected instance type parameter c (line 1). It then traverses the set V to find out which instance type can satisfy the deadline assigned to the task and has the lowest predicted cost (lines 2-7). Finally, FarSpot requests a spot instance of this given type as the initial container for the task to run.

When requesting a spot instance, users can optionally specify a maximum price that they are willing to pay.

Algorithm 2 Initial Instance Selection Algorithm

Input: $P_a(V)$ and $P_f(V)$: The set of actual and predicted spot prices, where V is the set of instance types; $task$: The task for scheduling; ct: The current time;
Output: c: Selected instance type;

```
Initialize the minimum spot price min ← ∞ and c ← −1;
for i = 1 to n do
    if ET(task, V_i) < subD(task) then
        cost ← 0;
        for j = 1 to ET(task, V_i) do
            if ct + j is in the current hour then
                cost ← cost + P_a^{ct}(V_i);
            else
                cost ← cost + P_f^{ct+j}(V_i);
        if cost < min then
            min ← cost;
            c ← i;
return c
```

Requested spot instances are charged at the spot prices, and will be terminated/stopped/hibernate when the spot price exceeds the specified maximum price. As FarSpot aims at migrating tasks among spot instances to reach low cost, we set the maximum price to very high to guarantee uninterrupted access to the resources. In our implementation, we set the maximum price to 10x of the On-Demand price of the same type.

6.2 Migration Policy

FarSpot dynamically migrates tasks from the current running instances to a more cost-efficient one if any. Specifically, FarSpot monitors the spot price variations of different instance types and periodically predicts spot price changes in near future using the latest collected price traces. In our implementation, we set the period to one hour. On detecting a spot price change, FarSpot estimates the migration cost and benefit for different tasks and makes migration decisions accordingly.

6.2.1 Migration cost

FarSpot adopts the stop-and-copy migration strategy which contains three major steps: 1) stop the task execution in the original instance, 2) copy and transfer all memory pages from the original instance to the new instance, and 3) resume task execution on the new instance. Once the memory copying process is complete and the destination machine receives a consistent image, the original VM is suspended and the new VM will take over all its services [12, 22].

The migration process incurs additional time and cost on the data movement. Assume the memory size of the source VM c is M_c, and the network bandwidth between the source VM and target VM i is B_{ci}. Then the memory data transfer time can be calculated as $H_{ci} = M_c/B_{ci}$. Due to the stop-and-copy strategy, both VM c and i are running during the data transfer while the task execution is stopped. Thus, the migration cost from VM c to a new VM i at time t can be calculated as below:

$$G_i(t) = \sum_{k=t}^{t+H_{ci}+O_i} (P_f^c(k) + P_f^i(k)) \tag{9}$$

where $P_f^c(t)$ and $P_f^i(t)$ represent the predicted spot price for the source VM c and the target VM i, respectively, at the billing cycle time t. The term $H_{ci} + O_i$ denotes the total amount of billing cycles, including the migration time H_{ci} and the boot time of the target VM O_i.

6.2.2 Anticipated saving

FarSpot performs an instance migration only when there's potential of cost saving. The cost saving mainly comes from running the unfinished task execution on a cheaper spot instance. We estimate the remaining execution time of a task on a VM i as below:

$$RT_W^i = (E_c - RT_A^c) \times E_c/E_i \tag{10}$$

where RT_A^c is the time that the task has run on instance c. E_c and E_i are the total time needed for the task to finish execution on VM c and i, respectively. We can estimate the execution time of tasks on all instance types with offline profiling.

As a result, FarSpot estimates the anticipated saving of the migration in time t as below:

$$S_i(t) = \sum_{k=t}^{t+RT_W^c} P_f^c(k) - \sum_{k=t}^{t+RT_W^i} P_f^i(k) - G_i(t) \tag{11}$$

where the first and second terms represent the remaining execution cost of the task without and with the migration, respectively. The anticipated saving equals to the spot instance rental saving deducted by the migration cost.

After obtaining the anticipated saving, FarSpot can make migration decisions based on a simple rule that FarSpot only hops to a new VM if the saving is positive. Among all candidate instance types j where $RT_W^j \leqslant Deadline$ and $S_j > 0$, FarSpot selects the one with the maximum anticipated saving as the destination VM. After finishing all the selection process, FarSpot performs the stop-and-copy migration between the source VM and destination VM.

7 Evaluation

This section presents our experimental results on evaluating the proposed framework. Overall, we conduct two major sets of experiments. Specifically, we first assess the prediction accuracy of our VWH model to evaluate how accurately FarSpot predicts the spot price for different instance types in Section 7.3. Further, we use real applications to study the cost optimization results of our approach in comparison with the state-of-the-art approaches in Section 7.4. Finally, we also experimentally assess the impact of different parameters with sensitivity studies in Section 7.5.

7.1 Implementation Details

Fisrt, Amazon provides historical spot price traces via AWS CLI for a period of up to 90 days before a request is made. We have utilized its API endpoint to collect publicly-available EC2 spot price traces from the ap-southeast-2 and eu-west-2 region for over 3 months, starting from September 15^{th}, 2020 to December 25^{th}, 2020. The dataset contains the historical spot price traces for 68 instance types with distinct prices among different AZs or regions. Specifically, spot price histories are collected every hour and the price trace for each instance type is an array with 2445 price data.

As for our VWH model, we build the LightGBM and RF models using the interfaces in Scikit-learn[1] and train them individually using the spot price traces for three instance types from the dataset (i.e., m5ad.xlarge in ap-southeast-2a, c5ad.12xlarge in ap-southeast-2b, and c5.large in ap-southeast-2c). We also set the number of lagged dependent

1. https://scikit-learn.org/stable/index.html

TABLE 2
The regions and instance types involved in the prediction experiment

I#	Region	Instance
I1	ap-southeast-2b	c5ad.12xlarge
I2	eu-west-2c	c4.8xlarge
I3	ap-southeast-2c	c5ad.12xlarge
I4	ap-southeast-2a	m5.8xlarge
I5	ap-southeast-2b	c5.large
I6	ap-southeast-2a	c5.2xlarge
I7	ap-southeast-2c	c5d.xlarge
I8	ap-southeast-2c	c5.2xlarge

variables to 4, that is, the predictors for the forecast result of the i^{th} hour will be x_{i-1}, x_{i-2}, x_{i-3}, and x_{i-4}, where x_{i-1}, x_{i-2}, x_{i-3}, and x_{i-4} are the actual spot prices of the past four hours. Furthermore, we divide each spot price trace in the dataset mentioned above into two parts: training set (80 percent) and test set (20 percent). We evaluate the accuracy of our models using the test set after training.

The training overhead of these two models is well controlled since it only takes within 1 minute to train the models on our commodity server. Also, our models can be easily fine-tuned by the incremental learning without retraining all over again when inputting the lately collected spot price traces. In order to evaluate the overall performance of the VWH model, we conduct the following prediction experiments with the instance types described in Table 2.

As for our simulator, we simulate a cloud environment where applications run within the ap-southeast-2 and eu-west-2 region with totally 20 compute optimized spot instance types, including c5ad.12xlarge in ap-southeast-2c, c5.9xlarge in eu-west-2a, and c5ad.8xlarge in ap-southeast-2b. For all instance types, we set the network bandwidth of them according to the data provided by Amazon (e.g., 10Gbps for c5.9xlarge).

We adopt existing models to estimate task execution time on different instance types [7]. Specifically, we first run workflows on our local machine and collect the execution time traces. Assuming that the performance of each task is a linear function of its host VM's number of vCPU, we can estimate the execution time of tasks on different types of instances using the profiling traces and the hardware configurations of the instances according to the estimation model [7]. For example, if an application is running on an instance with 16 vCPUs, it will have a 4x slowdown after being migrated to the one with 4 vCPUs. Also, we presuppose Amazon EC2's standard billing interval as 1 minute.

7.2 Experimental setup

7.2.1 Applications

In order to measure the monetary cost optimization of long-running HPC applications on spot instances, we apply our model to NAS Parallel Benchmarks (NPB [31]) kernels version 3.3.1. We select three pseudo applications, including BT (Block Tridiagonal solver), SP (Scalar Penta-diagonal solver), and LU (Lowerupper Gauss-Seidel solver). The default problem size is CLASS C. We run each of these applications multiple times (150 to 250 times) in a back-to-back manner sequentially to extend to large scale computing.

7.2.2 Comparisons

We first evaluate the prediction accuracy of the VWH model in FarSpot by comparing with the LSTM model. Then,

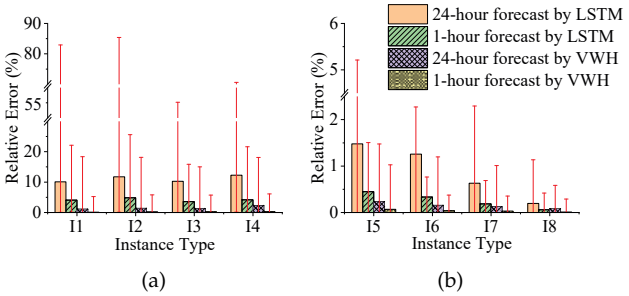


Fig. 5. Comparison of LSTM and VWH for different instance.

we assess the cost-efficiency of FarSpot by comparing the following four approaches.

- **On-demand**. This approach, as our **Baseline**, simulates the monetary cost optimization with only on-demand instances. We select the on-demand instance with the best performance (minimal execution time). In the following experiments, we set the *loose deadline*, *moderate deadline*, and *tight deadline* as five times, three times, and twice of the runtime of Baseline, respectively. We set the deadline to loose by default.
- **HotSpot**. Shastri and Irwin [35] proposed a resource container that dynamically selects and self-migrates to new VMs as spot prices change. It does not perform any price prediction, but makes decision based on the historical price combining with sophisticated migration policy instead.
- **LSTMSpot**. The Long Short-Term Memory (LSTM [19]) is one of the state-of-the-art models for time series forecasting. In this approach, we replace our VWH model with LSTM model instead. Specifically, it leverages LSTM to forecast the spot prices but distributes deadline and makes migration decision based on FarSpot's deadline distribution policy and migration mechanism, respectively.
- **FarSpot**. This approach is our proposed framework, which is a prediction-based algorithm for tasks scheduling. It makes migrations with full consideration of the predictability of spot prices and the migration trade-off.

7.2.3 Configuration

As for simulations, we randomly select a start point in the trace and leverage the approaches above to execute the applications. We keep on calculating the monetary cost and runtime while executing the applications. We repeat the simulation for 150 times and then calculate the average execution results for evaluations on all applications.

7.3 Prediction accuracy of FarSpot

In this part, we compare our VWH model with LSTM to illustrate its effectiveness. We study the prediction error rates for 1-hour prediction and 24-hour prediction of the two models above respectively. Note that we first sort our testing instance types in descending order according to the variance of their actual spot prices (i.e., the greater the variance is, the greater the spot price fluctuates). Then, we present the experimental results for the top 4 (I1-I4) and last 4 (I5-I8) instance types. Table 2 shows the regions and name of these 8 instance types.

Fig. 5 shows the accuracy of LSTM and VWH respectively in predicting the price of the instances mentioned

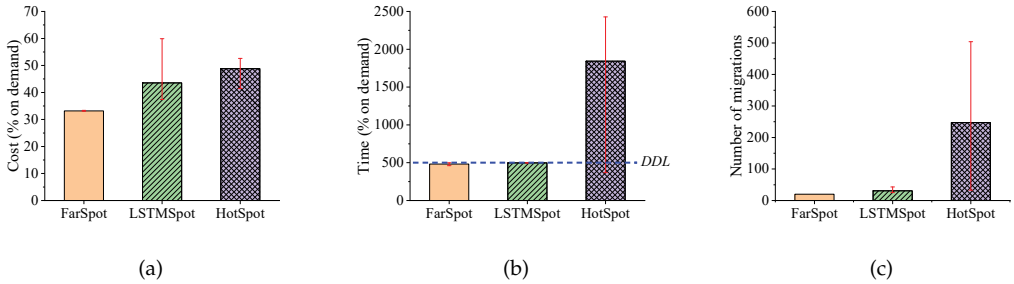


Fig. 6. Comparison of the average (a) monetary cost, (b) execution time and (c) number of migrations when using HotSpot, LSTMSpot, and FarSpot under the loose deadline. The error bars represent the maximum and minimum of each metric.

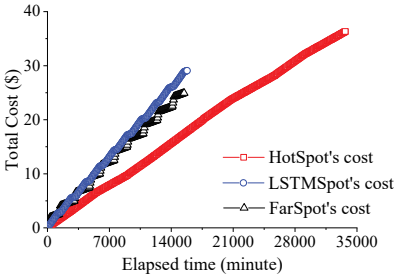


Fig. 7. Detailed execution results of one simulation under loose deadline.

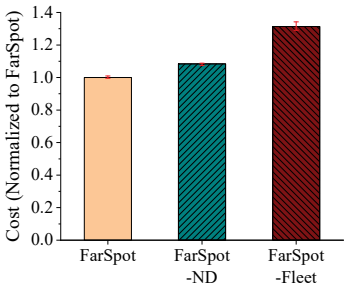


Fig. 8. Comparison of the normalized monetary cost when using FarSpot, FarSpot-ND, and FarSpot-Fleet under loose deadline. The error bars represent the maximum and minimum of each metric.

above. The error bars show the maximum and minimum error rates of each prediction. It can be seen that compared with LSTM, VWH presents more precise predictions.

For 24-hour predictions, the VWH model has obvious advantages in dealing with prices with large fluctuations: the average prediction error rate for I1 to I4 is 1%-2%, and the maximum error among them is just 18%. However, the average prediction error rate of LSTM for these types ranges from 10% to 12%, and the maximum error rate among them even reaches 85%. The reason for this occurrence is that the volatility of spot prices is relatively large, and LSTM cannot make correct predictions when encountered sudden changes in the price. But, when the volatility gets low, LSTM also carries out precise predictions. When it comes to I5 to I8, the average prediction error of LSTM for them is less than 2%, and the maximum error among them is only 5%. In contrast, the VWH model outshines LSTM obviously. The average error rate of VWH for I5 to I8 is much less than 1%, and the maximum error among them is as low as 2%.

As for 1-hour predictions, the error rate of LSTM is significantly reduced: the average prediction error rate for I1 to I4 is 4%-5%, and the maximum error rate among these types goes down to 26%. Also, the average error rate for I5 to I8 is less than 1%, and the maximum error rate among them is 2%. As before, our VWH model still outperforms LSTM greatly: the average prediction error rate for I1 to I4 is below 1%, and the maximum error rate among them is only 6%. In addition, the average prediction error rate for I5 to I8 is nearly to 0, and the maximum error rate among these types is just 1%.

On balance, LSTM has obvious disadvantages in forecasting volatile spot prices in a long-term manner, and its accuracy is unstable, therefore unreliable. However, VWH can efficiently extract features from the spot price with large fluctuation and then make accurate predictions with lower and more stable error rate.

7.4 Simulation results

Fig. 6(a) and 6(b) present the average monetary cost and execution time optimization results of the compared algorithms in our simulations. All results are normalized to Baseline. The number of migrations of the compared algorithms are shown in Fig.6(c). We further show the detailed execution results of the compared algorithms during one simulation in Fig. 7. We have the following observations.

First, compared to LSTMSpot, FarSpot reduces the monetary cost by 24%. This is mainly due to the inaccurate price prediction of LSTMSpot which leads to sub-optimal VM migrations. Although the makespan of FarSpot is just 2% less than that of LSTMSpot, FarSpot executes less migration, thereby ensuring better performance of applications. Second, FarSpot is able to obtain much better performance results compared to HotSpot. Specifically, FarSpot reduces the execution time and monetary cost by 50% and 32%, respectively, compared to HotSpot. This is mainly because HotSpot, unlike any prediction-base approach, can be easily hoodwinked by the current spot price. For example, HotSpot tends to hop to a new VM when the price of the VM is low enough. However, if the price of the VM increases greatly relative to other types of VMs in the next billing interval, HotSpot may continue to perform the migration without considering the long-term trade-off. Moreover, HotSpot may have difficulty in ensuring the performance constraint since it migrates too often and executes applications for a relatively long time.

Furthermore, we study the improvements brought by each individual technique of FarSpot, namely the deadline distribution policy and the migration policy. The evaluation results are shown in Fig. 8. FarSpot-ND is FarSpot without deadline assignment. FarSpot-Fleet is FarSpot with the migration policy replaced to the default allocation strategy of Amazon EC2 Spot Fleet [11]. Compared to FarSpot-ND

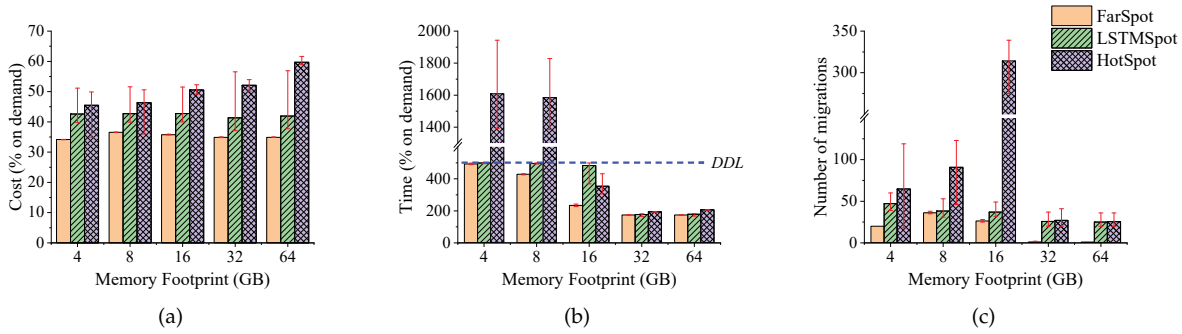


Fig. 9. Comparison of the average (a) monetary cost, (b) execution time and (c) number of migrations when using HotSpot, LSTMSpot, and FarSpot under the loose deadline as the size of transmitted data varies.

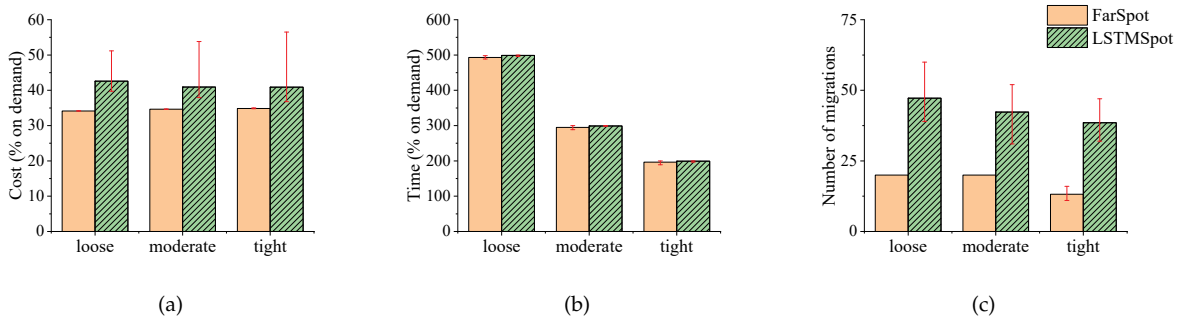


Fig. 10. Comparison of the average (a) monetary cost, (b) execution time and (c) number of migrations when using LSTMSpot and FarSpot with 4GB memory footprint as the deadline requirement varies. The error bars represent the maximum and minimum of each metric. The result of HotSpot is omitted since it is not aware of the job's deadline.

and FarSpot-Fleet, FarSpot reduces the average cost by 8% and 31%, respectively, which shows the effectiveness of each individual technique. It also shows the need for combining the deadline assignment and sophisticated migration policy together to further reduce job execution cost in spot market.

We also study the generality of FarSpot using recently collected spot price traces within the ap-southeast-2 and eu-west-2 region for over a month, starting from January 3^{rd}, 2021 to February 25^{th}, 2021. FarSpot still manages to reduce the monetary cost over LSTMSpot by 16% and HotSpot by 24%. While ensuring performance constrictions, FarSpot also outperforms LSTMSpot and HotSpot by 10% and 47%, respectively, in terms of the execution time.

7.5 Sensitivity studies

We also conduct sensitivity studies to exercise our algorithm and isolate key factors, such as application's memory footprint and deadline requirement, that affect FarSpot's relative cost and performance. To enable control of the key variables, we set the default runtime of the workflow as 115 hours to execute on a c5.9xlarge VM. In each study, we vary one parameter at a time and keep others as default.

7.5.1 Varying memory footprint

We compare FarSpot with HotSpot and LSTMSpot under the loose deadline requirement while varying the memory footprint (4GB-64GB) to illustrate the effectiveness of FarSpot, further figuring out its correlation with an application's memory footprint. Fig. 9(a) and 9(b) denote the average monetary cost and execution time optimization results of the compared algorithms on our simulation. All the results are normalized to Baseline. The number of migration of the compared algorithms is also plotted as Fig.9(c).

First, each of these three approaches have a markedly lower monetary cost than merely utilizing an on-demand instance. Specifically, FarSpot is capable of further lowering the average cost compared to both LSTMSpot and HotSpot, outdoing them by 15%-42% when increasing the data size. Although the average cost of FarSpot and that of LSTMSpot stay relatively stable, the range of that of LSTMSpot grows larger and larger as the memory footprint increases, even reaching 20% for a 64GB memory footprint.

Further, each approach increases the execution time relative to Baseline, but only FarSpot and LSTMSpot can satisfy the deadline requirement under any circumstance. HotSpot may not be able to ensure user goals when the memory footprint is lower than 16GB. In specific, the execution time of LSTMSpot decreases from 4.9x to 1.7x of that of Baseline while HotSpot reduces it from 16.1x to 2.1x of that of Baseline; FarSpot lessens the runtime compared to Baseline greatly by 65% when the memory footprint changes from 4GB to 64GB. Also, each method presents a slightly downward trend as the memory footprint grows.

Since HotSpot always computes the cost-efficiency based on the historical spot price without taking future price into account, HotSpot is likely to migrate so many times (up to 339) in order to recoup its transaction cost when the memory footprint is small (less than 32GB). It will not only incur additional overheads, but also affect the coherence of the application seriously. As for LSTMSpot, although it adopts FarSpot's migration policy, its prediction accuracy is pretty low compared with our VWH model. Thus, it may assign unfavorable subdeadlines for tasks and make wrong migration decisions frequently, thereby bringing about additional cost as well. However, note that all the three methods migrate less frequently and finish the execution greatly ahead of deadline when the memory footprint gets bigger (more than 16GB). That is determined by their migration policies. Their choices for new server are narrowed down

since they all don't migrate to a VM with smaller memory than an application's memory footprint. Generally, the big-memory instances also possess more vCPUs and are more expansive than the smaller ones, that is, execute jobs faster and lead to more cost.

7.5.2 Changing deadline requirement

We compare FarSpot with HotSpot and LSTMSpot with the fix size of memory footprint (4GB) under divers deadline requirements to illustrates its correlation with the deadline requirement. Fig. 10(a) and 10(b) denote the average monetary cost and execution time optimization results of the compared algorithms on our simulation. All the results are normalized to Baseline. The number of migration of the compared algorithms is also plotted as Fig.10(c). Since HotSpot is not aware of the job's deadline, we will not plot the results of it in these figures.

To begin with, the monetary cost of FarSpot stays relatively stable, outperforming Baseline by about 35%. However, LSTMSpot presents downward trends while the deadline is tightened, outstripping Baseline by 43% under the loose deadline requirement and around 41% under the tight deadline requirement, respectively. Nonetheless, the maximum cost of LSTMSpot becomes larger and larger. It can even climb up to nearly 60% when the deadline is strict. HotSpot can only outdo Baseline by 46% under any requirement.

Further, the execution time of FarSpot, LSTMSpot, and HotSpot are 4.9x, 5.0x, 16.1x of that of Baseline, respectively, under the loose deadline requirement. In addition, the runtime of FarSpot and LSTMSpot both show declining trends with tightening the deadline, going down to as low as 1.9x of that of Baseline under the strict deadline requirement. That is, the execution time of these two approaches are declining while narrowing the deadline. Importantly, FarSpot and LSTMSpot can meet the deadline requirements while HotSpot cannot make it.

Fig. 10(c) gives us information to account for these simulation results. The downtime incurred by migration may lead to applications' timeout. Thus, all of the three approaches adaptively cut down migration in order to execute applications on time. Notwithstanding, HotSpot and LSTMSpot still perform more migrations than FarSpot. As for LSTMSpot, numerous inappropriate migrations can explain its increasing range of cost. This phenomenon can be attributed to its low accuracy on price prediction.

8 Conclusion and Future Work

To reduce the monetary cost with little performance degradation in executing long-running HPC applications, this paper presents FarSpot, a framework that can provide precise spot price forecast and then automatically select and migrate to new spot instances when the benefit outweighs cost. We first analyze the benefits and necessity of VM migrations in EC2's spot market. Then we construct our VWH model by combining RF model and LightGBM to predict the spot prices. We next come up with a cost-aware deadline distribution algorithm based on the price prediction from the VWH model. Furthermore, we propose an effective migration policy for making migration decision. After that, we carry out experiments to evaluate the prediction accuracy of our VWH model. Finally, we conduct simulations on FarSpot using NAS Parallel Benchmarks. The experimental results show that FarSpot is able to save user's budget as it reduces the monetary cost by 32% on average (up to 37%) compared to the state-of-the-art algorithms [35] , and it can ensure the performance of applications and user goals.

As future work, we are planning to apply FarSpot on more prevalent cloud computing platforms with spot instances, such as Windows Azure, Alibaba Cloud, and Tencent Cloud, to evaluate the generality of our techniques.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62172282, 61802260, 62072311, 61972259, 62122056 and U2001212), Guangdong Basic and Applied Basic Research Foundation (No. 2020B1515120028, 2019B151502055), Guangdong NSF 2019A1515012053, the Shenzhen Science and Technology Foundation (No. JCYJ20210324094402008, JCYJ20210324093212034) and Tencent "Rhinoceros Birds" - Scientific Research Foundation for Young Teachers of Shenzhen University.

References

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing amazon ec2 spot instance pricing. *TEAC*, pages 1–20, 2013.

[2] H. Al-Theiabat, M. Al-Ayyoub, M. Alsmirat, and M. Aldwair. A deep learning approach for amazon ec2 spot price prediction. In *AICCSA*, pages 1–5, 2018.

[3] S. Alkharif, K. Lee, and H. Kim. Time-series analysis for price prediction of opportunistic cloud computing resources. In *EDB*, pages 221–229, 2018.

[4] S. Angra and S. Ahuja. Machine learning and its applications: A review. In *ICBDAC*, pages 57–60, 2017.

[5] M. Baughman, C. Haas, R. Wolski, I. Foster, and K. Chard. Predicting amazon spot prices with lstm networks. In *ScienceCloud*, pages 1–7, 2018.

[6] L. Breiman. Random forests. *Machine learning*, pages 5–32, 2001.

[7] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. 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*, pages 23–50, 2011.

[8] M. Chhetri, M. Lumpe, Q. Vo, and R. Kowalczyk. On forecasting amazon ec2 spot prices using time-series decomposition with hybrid look-backs. In *EDGE*, pages 158–165, 2017.

[9] V. Chittora and C.P. Gupta. Dynamic spot price forecasting using stacked lstm networks. In *ICISS*, pages 1080–1085, 2020.

[10] A. Criminisi, J. Shotton, and E. Konukoglu. *Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning*, pages 81–227. 2012.

[11] Amazon EC2. Spot Fleet. https://go.aws/3GCE5IW, 2021.

[12] D. Fernando, J. Terner, K. Gopalan, and P. Yang. Live migration ate my vm: Recovering a virtual machine after failure of post-copy live migration. In *IEEE INFOCOM 2019*, pages 343–351, 2019.

[13] X. Gao, C. Shan, C. Hu, Z. Niu, and Z. Liu. An adaptive ensemble machine learning model for intrusion detection. *IEEE Access*, pages 82512–82521, 2019.

[14] Y. Gong, B. He, and J. Zhong. Network performance aware mpi collective communication operations in the cloud. *TPDS*, pages 3079–3089, 2013.

[15] Y. Gong, B. He, and D. Li. Finding constant from change: Revisiting network performance aware optimizations on iaas clouds. In *SC*, pages 982–993, 2014.

[16] Y. Gong, B. He, and AC. Zhou. Monetary cost optimizations for mpi-based hpc applications on amazon clouds: checkpoints and replicated execution. In *SC*, pages 1–12, 2015.

[17] A. Gupta, P. Faraboschi, F. Gioachin, L. Kale, R. Kaufmann, B. Lee, V. March, D. Milojicic, and C. Suen. Evaluating and improving the performance and scheduling of hpc applications in cloud. *TCC*, pages 307–321, 2016.

[18] AJ. Hasan and M. Hammad. Spot hopping: Increasing reliability and reducing cost. *IJCDS*, pages 1237–1250, 2020.

[19] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, page 1735–1780, 1997.

[20] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS*, page 3149–3157, 2017.

[21] V. Khandelwal, A.K. Chaturvedi, and C.P. Gupta. Amazon ec2 spot price prediction using regression random forests. *TCC*, pages 59–72, 2020.

[22] H. Liu, H. Jin, X. Liao, C. Yu, and C. Xu. Live virtual machine migration via asynchronous replication and state synchronization. *TPDS*, pages 1986–1999, 2011.

[23] H. Liu, R. Yang, and Z. Duan. Wind speed forecasting using a new multi-factor fusion and multi-resolution ensemble model with real-time decomposition and adaptive error correction. *Energy Convers. Manag.*, page 112995, 2020.

[24] X. Liu, J. Jin, W. Wu, and F. Herz. A novel support vector machine ensemble model for estimation of free lime content in cement clinkers. *ISA Trans.*, pages 479–487, 2020.

[25] Z. Liu, X. Wang, Q. Zhang, and C. Huang. Empirical mode decomposition based hybrid ensemble model for electrical energy consumption forecasting of the cement grinding process. *Measurement*, page 314–324, 2019.

[26] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *FGCS*, pages 1–18, 2015.

[27] A. Marathe, R. Harris, D. Lowenthal, B. De Supinski, B. Rountree, and M. Schulz. Exploiting redundancy for cost-effective, time-constrained execution of hpc applications on amazon ec2. In *HPDC*, pages 279–290, 2014.

[28] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *HPCC*, pages 296–303, 2011.

[29] A. K. Mishra, A. Kesarwani, and D. K. Yadav. Short term price prediction for preemptible vm instances in cloud computing. In *I2CT*, pages 1–9, 2019.

[30] M. Naghshnejad and M. Singhal. Adaptive online runtime prediction to improve hpc applications latency in cloud. In *CLOUD*, pages 762–769, 2018.

[31] NASA. NPB. https://go.nasa.gov/2TRqn18, 2021.

[32] Roshni Pary. New amazon ec2 spot pricing model: Simplified purchasing without bidding and fewer interruptions. https://amzn.to/3hS3FxR, 2018.

[33] L. Rokach. Ensemble-based classifiers. *Artificial intelligence review*, pages 1–39, 2010.

[34] SAS. Machine learning: What it is and why it matters. https://bit.ly/3hVoc4H, 2021.

[35] S. Shastri and D. Irwin. Hotspot: Automated server hopping in cloud spot markets. In *SoCC*, page 493–505, 2017.

[36] S. Shastri and D. Irwin. Cloud index tracking: Enabling predictable costs in cloud spot markets. In *SoCC*, page 451–463, 2018.

[37] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. Spoton: A batch computing service for the spot market. In *SoCC*, page 329–341, 2015.

[38] S. Subramanya, A. Rizk, and D. Irwin. Cloud spot markets are not sustainable: The case for transient guarantees. In *HotCloud*, page 13–18, 2016.

[39] M. Taifi, J. Shi, and A. Khreishah. Spotmpi: A framework for auction-based hpc computing using amazon spot instances. In *ICA3PP*, pages 109–120, 2011.

[40] F. Xu, H. Zheng, H. Jiang, W. Shao, H. Liu, and Z. Zhou. Cost-effective cloud server provisioning for predictable performance of big data analytics. *TPDS*, pages 1036–1051, 2019.

[41] H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang. Optimal resource rental planning for elastic applications in cloud market. In *IPDPS*, pages 808–819, 2012.

[42] A. C. Zhou and B. He. Transformation-based monetary costoptimizations for workflows in the cloud. *TCC*, pages 85–98, 2014.

Amelie Chi Zhou is currently an Assistant Professor in Shenzhen University, China. Before joining Shenzhen University, she was a Postdoc Fellow in Inria-Bretagne research center, France. She received her PhD degree in 2016 from School of Computer Engineering, Nanyang Technological University, Singapore. Her research interests lie in cloud computing, high performance computing, big data processing and resource management.

Jianming Lao is currently working toward the B.S. degree in the College of Computer Science and Software Engineering, Shenzhen University, China. His research interests focus on high performance computing.

Zhoubin Ke is currently working toward the B.S. degree in the College of Computer Science and Software Engineering, Shenzhen University, China. His research interests focus on high performance computing.

Yi Wang is currently a professor in the College of Computer Science and Software Engineering, Shenzhen University, China. He received the BE and ME degrees in electrical engineering from the Harbin Institute of Technology, China, in 2005 and 2008, respectively, and the PhD degree in computer science from the Department of Computing, the Hong Kong Polytechnic University, in 2011. His research interests include embedded systems, non-volatile memory, and real-time scheduling for multi-core systems.

Rui Mao is currently a professor and a vice dean in the College of Computer Science and Software Engineering, Shenzhen University, China. He received his BS and MS degrees in computer science from the University of Science and Technology of China, China, in 1997 and 2000, respectively, and the PhD degree in computer science from the University of Texas at Austin, USA, in 2007. His research interests include universal data management and analysis in metric space, and high performance computing.