Provisioning Spot Instances Without Employing Fault-Tolerance Mechanisms

Abdullah Alourani*†, Ajay D. Kshemkalyani*
*Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, USA
†Department of Computer Science and Information, Majmaah University, Al-Majmaah 11952, Saudi Arabia
aalour2@uic.edu, ajay@uic.edu

***Abstract*—Cloud computing offers a variable-cost payment scheme that allows cloud customers to specify the price they are willing to pay for renting spot instances to run their applications at much lower costs than fixed payment schemes, and depending on the varying demand from cloud customers, cloud platforms could revoke spot instances at any time. To alleviate the effect of spot instance revocations, applications often employ different fault-tolerance mechanisms to minimize or even eliminate the lost work for each spot instance revocation. However, these fault-tolerance mechanisms incur additional overhead related to application completion time and deployment cost. We propose a novel cloud market-based approach that leverages cloud spot market features to provision spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications. We evaluate our approach in simulations and use Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms.**

***Keywords*-cloud computing; spot instances; fault-tolerance mechanisms; cloud spot market features; cloud-based applications; payment schemes; spot instance revocations**

I. Introduction

Cloud computing offers a variable-cost payment scheme that allows cloud customers to specify the price they are willing to pay for renting spot instances to run their applications at much lower costs than fixed payment schemes, and depending on the varying demand from cloud customers, cloud platforms could revoke spot instances at any time. The price of a spot instance can go up if the demand increases and the number of available instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases. Conversely, the price of this spot instance can go down if the demand decreases and the number of available instances increases. Therefore, if the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to cloud customers' applications at the customer's price. However, when spot instances are already provisioned to cloud customer applications and the cloud provider's price goes above the customer's price, the cloud providers will terminate those spot instances within two minutes by sending termination notification signals [1]. As a result, even though cloud customers sometimes rent spot instances at 90% lower prices than on-demand prices [2], their applications that run on spot instances can be terminated based on price fluctuations that happen frequently; thus, those applications may incur additional overhead related to application completion time and deployment cost from re-executing lost work for each spot instance revocation.

Applications may benefit from different fault-tolerance mechanisms to alleviate the work lost for each spot instance revocation. However, these fault-tolerance mechanisms incur additional overhead related to application completion time and deployment cost. Fault-tolerance mechanisms are typically divided into three types: migration, checkpointing, and replication. First, migration mechanisms are often employed to reactively migrate the state of an application (i.e., memory and local disk state) to another instance prior to a spot instance revocation. The overhead of a migration mechanism is determined based on the migration time of an application and the number of spot instance revocations during the application execution. The migration time of an application mostly depends on the resource usage of the application, whereas the number of spot instance revocations depends on the volatility of cloud spot markets. A larger resource usage of an application often results in a higher overhead of a migration mechanism, and vice versa. A similar explanation is applicable for the volatility of cloud spot markets; thus, a higher overhead of a migration mechanism will lead to a higher overhead of an application's completion time and deployment cost. Second, checkpointing mechanisms are often employed to proactively checkpoint an application's state to remote storage (e.g., AWS S3). The overhead of a checkpointing mechanism is specified based on the time to checkpoint an application's state and the number of checkpoints, which represents how often an application's state is stored in remote storage during the application execution, along with the time to re-execute the lost work from the last checkpoint for each spot instance revocation. The checkpointing time of an application relies on the resource usage of the application and the number of checkpoints typically specified by engineers who maintain applications deployed on spot instances. If engineers specify a large number of checkpoints, the overhead time to re-execute the lost work from the last checkpoint for each spot instance revocation will likely decrease, whereas the overhead time to checkpoint the state of an application will likely increase. Conversely, if engineers specify a small number of checkpoints, the overhead time to checkpoint the state of an application will likely decrease, whereas the overhead time to re-execute the lost work from the

978-1-7281-8946-8/20/$31.00 ©2020 IEEE
DOI 10.1109/ISPDC51135.2020.00026

last checkpoint for each spot instance revocation will likely increase. Hence, checkpointing mechanisms require analyzing cloud spot markets and the resource usage of applications to optimize the tradeoff between the overhead of actual checkpoints and the overhead of re-executing lost work. Third, replication mechanisms are often employed to replicate the computations of an application among different instances. The overhead of a replication mechanism is based on the degree of replication (i.e., the number of replicated instances) and the number of revocations that depends on the volatility of cloud spot markets, and is independent of the resource usage of an application. As a result, a higher overhead of these fault-tolerance mechanisms leads to a higher overhead related to application completion time and deployment cost.

Contributions: We address a challenging problem for applications deployed on cloud spot instances that results from the overhead of employing fault-tolerance mechanisms. We propose a novel cloud market-based approach that leverages features of cloud spot markets for *Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms (`P-SIWOFT`)* to reduce the deployment cost and completion time of applications. We evaluate `P-SIWOFT` in simulations and use Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms. `P-SIWOFT` code and our simulation results are publicly available [3].

II. Problem Statement

In this section, we discuss sources of overhead of fault-tolerance mechanisms and formulate the problem statement.

A. Sources of Overhead of Fault-Tolerance Mechanisms

There are three main sources of overhead of fault-tolerance mechanisms. First, various resource usage of an application imposes various overhead of fault-tolerance mechanisms depending on the settings of each fault-tolerance mechanism type. A larger resource usage of an application (i.e., memory footprint) often results in a higher overhead of a fault-tolerance mechanism, and vice versa. The time to migrate/checkpoint the state of an application depends on the sizes of the application's memory and local disk state. Additionally, the choice of the type of fault-tolerance mechanism depends on the resource usage of an application. For example, a live migration requires a limited size of an application's memory footprint and cannot be employed when the application's memory footprint is greater than 4 GB [4]. As a result, the resource usage of an application not only affects the overhead of a fault-tolerance mechanism but also affects the choice of the type of fault-tolerance mechanism.

Second, the volatility of cloud markets is represented by the number of spot instance revocations over the application runtime. A higher number of spot instance revocations often results in higher overhead of fault-tolerance mechanisms, and vice versa. Checkpointing mechanisms will re-execute the lost work from the last checkpoint for each spot instance revocation, whereas migration mechanisms will reactively migrate an application to another instance prior to each spot instance revocation. Unlike migration and checkpointing mechanisms, a replication mechanism might re-execute the lost work from the beginning of an application's runtime for each spot instance revocation when all replicated instances are being revoked. As a result, the volatility of cloud markets has an impact on the overhead of various types of fault-tolerance mechanisms.

Third, the overhead of fault-tolerance mechanisms relies on the settings of each type of fault-tolerance mechanism. A main parameter of replication settings is the degree of replication, which represents the number of replicated servers needed to execute the same application's job across these replicated servers. When the degree of replication is small, the overhead that results from re-executing the lost work from the beginning of an application's runtime for each spot instance revocation will likely increase. In contrast, when the degree of replication is large, the overhead that results from a high number of servers will likely increase. A main parameter of checkpointing settings is the number of checkpoints, which represents how often an application's state is stored in remote storage over the application runtime. When the number of checkpoints is small, the overhead that results from re-executing the lost work from the last checkpoint for each spot instance revocation will likely increase. In contrast, when the number of checkpoints is large, the overhead that results from the time to checkpoint an application's state will likely increase. A main parameter of migration settings is the number of migrations, which represents how often an application's state migrates to another server over the application runtime. When the number of migrations is small, the overhead that results from re-executing the lost work from the beginning of an application's runtime for each spot instance revocation will likely increase. In contrast, when the number of migrations is large, the overhead that results from the time to migrate an application's state will likely increase. As a result, the fundamental problem for cloud customers is determining how to find the optimal settings of various types of fault-tolerance mechanisms to reduce the overhead resulting from employing fault-tolerance mechanisms.

B. The Problem Statement

Cloud computing offers a variable-cost payment scheme that allows cloud customers to specify the price they are willing to pay for renting spot instances to run their applications at much lower costs than fixed payment schemes.

In exchange, applications deployed on spot instances are often exposed to revocations by cloud providers, and as a result, these applications often employ different fault-tolerance mechanisms to minimize or even eliminate the lost work for each spot instance revocation. However, these fault-tolerance mechanisms incur additional overhead related to application completion time and deployment cost. In this paper, we address a challenging problem for applications deployed on cloud spot instances that results from the overhead of employing fault-tolerance mechanisms—determining how to effectively deploy applications on spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications. The root of this problem is that applications often employ fault-tolerance mechanisms to minimize the lost work for each spot instance revocation without taking into consideration the overhead of fault-tolerance mechanisms, leading to significantly larger deployment costs and completion times of applications, and as a result, the advantages of cloud spot instances could be significantly minimized or even completely eliminated.

III. Our Approach

In this section, we state our key ideas for our approach for *Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms (P-SIWOFT)* and explain the P-SIWOFT algorithm.

A. Key Ideas

A goal of our approach is to automatically provision spot instances without employing fault-tolerance mechanisms to reduce the deployment cost and completion time of applications. Our approach leverages features of cloud spot markets such as the spot instance lifetime, revocation probability, and revocation correlation between cloud spot markets to provision spot instances for applications. The spot instance lifetime represents the average time until a spot instance's price rises above the corresponding on-demand instance price (i.e., mean time to revocation (MTTR)) because cloud customers are often not willing to pay more than the on-demand price to rent spot instances. The revocation probability of each spot instance represents the estimated lifetime of a spot instance during a job execution and is calculated by dividing the job's execution length by the MTTR of the provisioned spot instance. The revocation correlation between cloud spot instances represents how often these spot instances were revoked at the same time (i.e., the same hour representing a single billing cycle in cloud platforms [2]) over the past three months.

In general, cloud spot markets show a broad range of characteristics. These important characteristics are at the core of our approach. First, revocations rarely happen in some cloud spot markets, so the MTTR of these markets is very high (i.e., $>$ 600 h) [5]. Second, employing fault-tolerance mechanisms often results in additional overhead related to application completion time and deployment cost [4]. Third, cloud spot markets exhibit variations in price characteristics for a similar type of spot instance across various cloud spot markets. Thus, a spot instance in a cloud market is often independent of a spot instance in another cloud market, which suggests that a spot instance's revocation in a cloud market is often uncorrelated with a spot instance in another cloud market [5]. Based on these characteristics, our key idea is that we could eliminate the additional overhead resulting from employing fault-tolerance mechanisms by provisioning the spot instance with the highest MTTR as long as the spot instance's MTTR is at least twice the application's execution length. Another idea is that we could reduce consequent revocations when a spot instance is revoked by provisioning a new spot instance with the next highest MTTR and a low revocation correlation with the revoked spot instance. When we provision a spot instance that is uncorrelated with the revoked spot instance, it is more unlikely that the new spot instance will be revoked again than another spot instance that is highly correlated with the revoked spot instance. As a result, these key ideas enable cloud customers to avoid unnecessary overhead resulting from employing fault-tolerance mechanisms; hence, cloud customers can execute jobs with a completion time near that of on-demand instances but at a cost of only spot instances.

Algorithm 1 P-SIWOFT's algorithm for provisioning spot instances without employing fault-tolerance mechanisms.

```
1: Inputs: Jobs $J$, Cloud Markets $M$, Resources $R$
2: $U \leftarrow$ FindSuitableServers($J$, $R$)
3: $L \leftarrow$ ComputeLifeTime($M$, $U$)
4: for each j in $J$ do
5:     $S_j \leftarrow$ ServerBasedLifeTime($j$, $M$, $L$)
6:     while $j \neg$ Completed do
7:         $s_j \leftarrow$ Highest($S_j$)
8:         if $length(s_j) >> length(j)$ then
9:             $v_{s_j} \leftarrow$ RevocationProbability($j$, $s_j$)
10:            ProvisionHighestLifeTime($j$, $s_j$)
11:            if $s_j$ encounters $v_{s_j}$ then
12:                $C_j, T_j \leftarrow C_j \cup \{c_{s_j}\}, T_j \cup \{t_{s_j}\}$
13:                $W_{s_j} \leftarrow$ FindLowCorrelation($j$, $s_j$))
14:                $S_j \leftarrow (S_j \setminus \{s_j\}) \cap W_{s_j}$
15:            end if
16:        end if
17:    end while
18:    $C_j, T_j \leftarrow C_j \cup \{c_{s_j}\}, T_j \cup \{t_{s_j}\}$
19:    $C, T \leftarrow$ ComputeCostExeTime($C_j$, $T_j$)
20: end for
21: return $C$, $T$
```

B. P-SIWOFT Algorithm

P-SIWOFT is illustrated in Algorithm 1 that takes in the batch job set J; the resource requirement set R; and the entire set of cloud markets M, containing on-demand instance types, prices of on-demand instances, spot instance types, their availability zones, their regions, and spot instance prices over the past three months. Starting from Step 2, the algorithm finds a suitable set of spot instances U that meet the resource requirements. In P-SIWOFT, we use the memory size to determine suitable types of spot instances that are supported by EC2 markets [2]. In Step 3, for each suitable spot instance, the spot instance's lifetime (i.e., the spot instance's MTTR) is computed based on the corresponding on-demand instance price, as discussed in Section III-A. L is the set of such lifetimes. In Steps 4-20, for each job, the algorithm is executed until the jobs in the job set are completed. In Step 5, the cloud spot markets are first filtered to include only a set of suitable spot instances S_j for the job j according to their lifetimes L, as discussed in Section III-A, and then these spot instances are sorted in descending order based on their lifetimes. In Steps 6–17, the job j is executed until the job's execution is completed. In Step 7, the algorithm selects a spot instance s_j with the highest lifetime. In Step 8, we ensure that the highest lifetime for the spot instance s_j is at least twice the job $j's$ execution length to reduce the revocation probability of the provisioned instance during the job execution. In Step 9, the algorithm computes the revocation probability of the provisioned instance v_{s_j} by dividing the job $j's$ execution length by the lifetime of the provisioned instance s_j. In Step 10, the spot instance s_j with the highest lifetime is provisioned to (re)start executing the job j. In Steps 11–15, the algorithm checks if the provisioned spot instance s_j is revoked based on its revocation probability v_{s_j} during the job execution j. When a spot instance s_j is revoked, the deployment time t_{s_j} and cost c_{s_j} are added to the total deployment time set T_j and cost set C_j, respectively, in Step 12. In P-SIWOFT, the deployment time represents the job's execution time until the spot instance is revoked, the deployment cost of a spot instance represents the price of the provisioned spot instance at a certain execution point, and the cost is computed at a per hour rate [2]. In Step 13, the low revocation correlation set W_{s_j} with the revoked spot instance is computed using the revocation correlation between cloud spot instances, as discussed in Section III-A. In Step 14, the revoked spot instance is removed from the set of suitable spot instances S_j, and the set of suitable spot instances S_j is filtered based on a low revocation correlation set W_{s_j}. The cycle of Steps 6–17 repeats until the job $j's$ execution is completed. When the job $j's$ execution is completed, the deployment time t_{s_j} and cost c_{s_j} are added to the total deployment time set T_j and cost set C_j, respectively, in Step 18. In Step 19, the total deployment time set T_j and cost set C_j are computed and then added to the overall deployment time T and cost C, respectively. The cycle of Steps 4–20 repeats until the jobs in the job set are completed. Finally, the total deployment time T and cost C are returned in Step 21 as the algorithm ends.

IV. Evaluation

In this section, we describe the design of the study to evaluate P-SIWOFT and state threats to its validity. We pose the following Research Questions (RQs):

RQ$_1$: How efficient is P-SIWOFT compared to a fault-tolerance approach in executing applications?

RQ$_2$: How effective is P-SIWOFT compared to a fault-tolerance approach in reducing the deployment cost of applications?

RQ$_3$: Do different settings of a fault-tolerance approach contribute to different types of overhead?

A. Subject applications

We evaluate P-SIWOFT in simulations and use Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. P-SIWOFT packages jobs in Docker containers to simplify restoring and checkpointing. We use Amazon spot instances since their MTTR often exceed hundreds of hours, unlike MTTR for Google preemptible instances that are less than 24 hours [6]. Also, we use Docker containers since they support checkpointing and restoring container images. We use a load generator called Lookbusy [7] to create synthetic jobs with different amounts of resource usage. In addition, P-SIWOFT uses EC2's REST API to collect realistic price traces for all spot instances across all markets (i.e., availability zones and regions) for the past three months.

B. Methodology

Some objectives of the experiments are to demonstrate that P-SIWOFT can efficiently execute applications and can effectively decrease the deployment cost of applications compared to a fault-tolerance approach. For these objectives, we use different combinations of job execution length and job memory footprint to show the impact on the completion time and the deployment cost when a spot instance is provisioned for the job using P-SIWOFT and the fault-tolerance approach. We define two revocation rules with different ranges for P-SIWOFT and the fault-tolerance approach to show the impact on the completion time and the deployment cost for different numbers of revocations during a job's execution. When a spot instance is provisioned for a job using the fault-tolerance approach, we randomly send a fixed number of revocations per day of the job's execution length, as suggested by prior work [4]. Conversely, when a spot instance is provisioned for a job using P-SIWOFT, we use the revocation probability of a spot instance that relies on realistic price traces from the Amazon cloud to revoke the provisioned spot instance. Since another goal is to understand how different settings of jobs and different settings of the fault-tolerance approach contribute to different types of overhead (e.g., checkpoint overhead), we investigate how different job execution lengths, job memory footprints, numbers of revocations, and numbers of checkpoints contribute to different overhead types that are related to a job's completion time and deployment cost.

P-SIWOFT is implemented using a load generator API (Lookbusy), EC2's REST API, Docker containers, AWS S3, and EC2 spot instances. The experiments for the subject applications were carried out using spot instances from Amazon EC2 called m5ad.12xlarge with a 48 GHz CPU and 192 GB of memory. We package jobs in Docker containers that run on Ubuntu 18.04 LTS with a limited CPU and memory capacity for the provisioned spot instances to assess the effectiveness of P-SIWOFT for different job memory footprints and job execution lengths.

C. Threads to validity

One potential threat to our empirical evaluation is that our experiments were conducted only on batch job applications, which may make it difficult to generalize the results of the experiments to other types of applications (e.g., interactive job applications) that may have various workflows and behaviors. However, cloud spot instances are often used to run batch job applications. As a result, we expect the results of the experiments to be generalizable.

We experimented with a certain price ratio between spot instances and on-demand instances that is based on realistic price traces from EC2 markets, whereas other ratios between spot instances and on-demand instances could result in different effects on the deployment cost and completion time of jobs when spot instances are provisioned using P-SIWOFT and the fault-tolerance approach. However, understanding the effect of various price ratios between spot instances and on-demand instances is beyond the scope of this empirical study and shall be considered in future studies.

V. Results

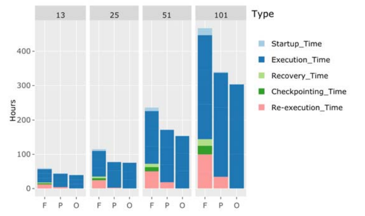
In this section, we describe and analyze the results of the experiments to answer the RQs listed in Section IV.

A. Completion Time

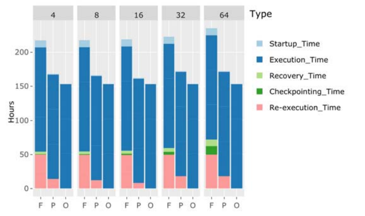
The experimental results that summarize the completion time for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different job execution lengths are shown in the stacked bar plots in Fig. 1a. We observe that the completion time using P-SIWOFT is consistently shorter than the completion time using the fault-tolerance approach, and the completion time using P-SIWOFT is consistently near that of on-demand instances, which do not incur any additional overhead [2]. This result shows that a higher job length leads to a steadily higher overhead of completion time resulting from the job's checkpointing, recovery, and re-execution times, as well as the startup time of a spot instance when using the fault-tolerance approach. However, a higher job length leads to a slightly higher overhead of the completion time, as a result of the job's re-execution time and the startup time of a spot instance when using P-SIWOFT. Our explanation is that P-SIWOFT does not incur frequent job re-execution time and the startup time of a spot instance since the startup time of a spot instance using P-SIWOFT does not increase with the increase in job execution length. This is expected based on the way P-SIWOFT provisions a spot instance with the highest MTTR.

The experimental results that summarize the completion time for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different job memory footprints are shown in the stacked bar plots in Fig. 1b. We observe that the completion time for P-SIWOFT is consistently shorter than the completion time for the fault-tolerance approach, and the completion time for P-SIWOFT is consistently near that of on-demand instances, which do not incur any additional overhead [2]. This result shows that a higher job memory footprint leads to a higher overhead of the completion time resulting from the job's checkpointing time and recovery time when using the fault-tolerance approach. In contrast, the overhead of the completion time resulting from the job's re-execution time and the startup time of a spot instance when using the fault-tolerance approach stays approximately the same across various job memory footprints, which suggests that the overhead resulting from the job's re-execution time and the startup time of a spot instance for the fault-tolerance approach is independent of the job resource usage. Also, the overhead of an application's completion time resulting from the job's re-execution time and the startup time of a spot instance when using P-SIWOFT stays approximately the same across various job memory footprints, which suggests that the completion time for the subject applications when using P-SIWOFT is also independent of the resource usage.

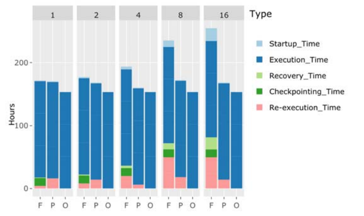
The experimental results that summarize the completion time for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different numbers of revocations are shown in the stacked bar plots in Fig. 1c. We observe that the completion time for P-SIWOFT—except for when the number of revocations equals one—is consistently shorter than the completion time for the fault-tolerance approach, and the completion time for P-SIWOFT is consistently near that of on-demand instances, which do not incur any additional overhead [2]. When the number of revocations equals one, the job's checkpointing time for the fault-tolerance approach balances the job's re-execution for P-SIWOFT. This result suggests that the fault-tolerance approach incurs additional overhead due not only to the number of revocations, but also the number of checkpoints. It also suggests that the effectiveness of P-SIWOFT may decrease when the number of revocations decreases, and it is very difficult to guarantee that the number of revocations is small [8]. The job's recovery time, the job's re-execution time, and the startup time of a spot instance—except for the job's checkpointing time—all increase steadily when



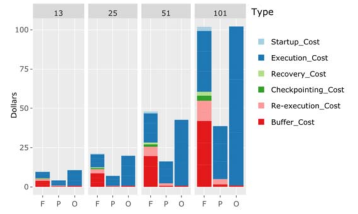
(a) Job Length (Time)



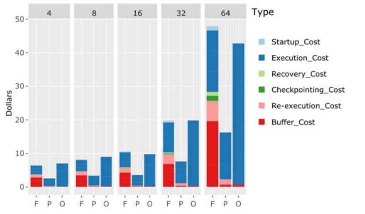
(b) Memory Footprint (Time)



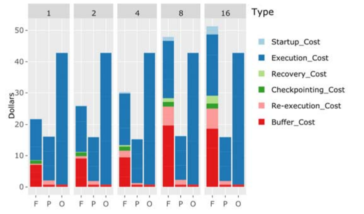
(c) Revocation Number (Time)



(d) Job Length (Cost)



(e) Memory Footprint (Cost)



(f) Revocation Number (Cost)

Figure 1: Comparing the completion time (top row) and the deployment costs (bottom row) for the subject applications using P-SIWOFT (P), the fault-tolerance approach (F), and on-demand instances (O) for different job execution lengths (a and d), memory footprints (b and e), and revocation numbers (c and f), while keeping other job features constant.

using the fault-tolerance approach, whereas in P-SIWOFT, the job's re-execution time and the startup time of a spot instance stay approximately the same. This observation suggests that the job's checkpointing time for the fault-tolerance approach as well as the job's re-execution time and the startup time of a spot instance for P-SIWOFT, are independent of the number of revocations. In summary, these experimental results allow us to conclude that P-SIWOFT is more efficient in executing applications for different job execution lengths, job memory footprints, and numbers of revocations than the fault-tolerance approach, thus **positively addressing *RQ*$_1$**.

B. Deployment Costs

The experimental results that summarize the deployment costs for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different job execution lengths are shown in the stacked bar plots in Fig. 1d. We observe that the deployment costs using P-SIWOFT are consistently lower than the deployment costs using the fault-tolerance approach or those of on-demand instances. This result identifies the steady rise in overhead related to deployment costs that result from the job's checkpointing costs, its recovery costs, its re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles when using the fault-tolerance approach with the increased job length. However, this result also identifies a slight rise in the overhead of deployment costs that result from the job's re-execution cost, the startup costs of spot instances, and the buffer costs of billing cycles when using P-SIWOFT with the increased length. Our explanation is that P-SIWOFT does not frequently incur the job's re-execution costs and the startup costs of spot instances since the startup costs of spot instances using P-SIWOFT do not increase with the increase of the job execution length, which is expected based on the way that P-SIWOFT provisions a spot instance with the highest MTTR. Interestingly, we observe that unlike P-SIWOFT, the buffer costs of billing cycles significantly increase compared to the other types of overhead costs when using the fault-tolerance approach with the increase of the job length, which suggests that the fault-tolerance approach incurs not only overhead related to the settings of the fault-tolerance approach (e.g., the job's checkpointing cost) but also additional overhead related to the cloud billing policies (i.e., the buffer costs of billing cycles). Also, we observe that the deployment costs of the fault-tolerance approach across all job lengths are equal to or higher than the deployment costs of on-demand instances [2], which suggests using on-demand for larger job lengths may reduce deployment costs and the completion time when compared to the fault-tolerance approach.

The experimental results that summarize the deployment costs for the subject applications using P-SIWOFT, the fault-tolerance approach, and on-demand instances for different job memory footprints are shown in the stacked bar plots in Fig. 1e. We observe that the deployment costs using P-SIWOFT are consistently lower than the deployment costs using the fault-tolerance approach and on-demand instances. This result demonstrates the steady rise of the overhead related to deployment costs resulting from the job's checkpointing, recovery, re-execution, and startup costs of spot instances, as well as the buffer costs of billing cycles when using the fault-tolerance approach with the increase of job memory footprint. However, this result demonstrates a slight rise of the overhead of deployment costs resulting from the job's re-execution and startup costs of spot instances, and the buffer costs of billing cycles when using `P-SIWOFT` with the increase of job memory footprint. Our explanation is that `P-SIWOFT` does not incur the job's re-execution and startup costs of spot instances, since the startup costs of spot instances using `P-SIWOFT` do not increase with the increase of the job memory footprint, which is expected based on the way that `P-SIWOFT` provisions a spot instance with the highest MTTR. We observe that, unlike the buffer costs of billing cycles for `P-SIWOFT`, the buffer costs of billing cycles for the fault-tolerance approach significantly increase with the higher job memory footprints (i.e., 32 and 64 GB), suggesting that the buffer costs increase when there is a significant change in deployment time between consecutive job memory footprints (i.e., exceeds the period for a billing cycle). Additionally, we observe that the deployment costs of the fault-tolerance approach across all job memory footprints are equal or higher than the deployment costs of on-demand instances [2], which suggests provisioning on-demand for large job memory footprints may result in lower deployment costs and completion time than the fault-tolerance approach.

The experimental results that summarize the deployment costs for the subject applications using `P-SIWOFT`, the fault-tolerance approach, on-demand instances for different numbers of revocations are shown in the stacked bar plots in Fig. 1f. We observe that the deployment costs using `P-SIWOFT` and that of on-demand instances are consistently lower than the deployment costs using the fault-tolerance approach. The job's recovery and re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles, except for the job's checkpointing costs, increase steadily when using the fault-tolerance approach whereas, for `P-SIWOFT`, the job's re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles stay approximately the same. This observation suggests that the job's recovery time and re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles depend on the number of revocations when using the fault-tolerance approach. However, the job's checkpointing costs for the fault-tolerance approach and the job's re-execution costs, the startup costs of spot instances, and the buffer costs of billing cycles for `P-SIWOFT`, are independent of the number of revocations, respectively. Our explanation is that `P-SIWOFT` does not incur the job's re-execution costs and the startup costs of spot instances. We observe that unlike the buffer costs of billing cycles for `P-SIWOFT`, the buffer costs of billing cycles for the fault-tolerance approach significantly increase with the higher numbers of revocations (i.e., 8 and 16), which suggests that the buffer costs increase when there is a significant change in deployment time between consecutive numbers of revocations (i.e., exceeds the period for a billing cycle). Interestingly, we observe that the deployment costs for the fault-tolerance approach when the number of revocations is high (i.e., 8 and 16) is significantly higher than the deployment costs for on-demand instances [2], which confirms that provisioning on-demand for a large number of revocations may result in lower deployment costs and completion time than the fault-tolerance approach. In summary, these experimental results allow us to conclude that `P-SIWOFT` is more effective in reducing the deployment costs of applications for different job execution lengths, job memory footprints, and numbers of revocations than the fault-tolerance approach, thus **positively addressing RQ_2**.

C. Impact on Different Types of Overhead

An interesting question is how different job execution lengths, job memory footprints, and numbers of revocations, contribute to different overhead types that are related to a job's completion time and deployment cost when using the fault-tolerance approach. Consider the stacked bar plots that are shown in Fig. 1a, Fig. 1b, and Fig. 1c — the visual inspection identifies the highest overhead related to the completion time results from the job's re-execution time, then the job's checkpointing time and the job's recovery time, followed by the startup time of a spot instance, with the increase of the job execution length. Also, with the rise of the job memory footprint, the highest overhead related to the completion time when using the fault-tolerance approach results from the job's checkpointing time and the job's recovery time. With the increase of the number of revocations, the highest overhead related to the completion time when using the fault-tolerance approach results from the job's re-execution time, then the job's recovery time, followed by the startup time of a spot instance.

Similarly, it is shown in the stacked bar plots in Fig. 1d, Fig. 1e, and Fig. 1f that the highest overhead related to the deployment costs when using the fault-tolerance approach results from the buffer costs of billing cycles, the job's re-execution costs, then the job's checkpointing cost, the job's recovery cost, followed by the startup costs of spot instances, with the increase of the job execution length. With the rise of the job memory footprint, the highest overhead related to the deployment costs when using the fault-tolerance approach results from the buffer costs of billing cycles, the job's re-execution costs, then the job's checkpointing and recovery costs, followed by the startup costs of spot instances. With the increase of the number of revocations, the highest overhead related to the deployment costs when using the fault-tolerance approach results from the buffer costs of billing cycles, the job's re-execution costs, then its recovery costs, followed by the startup costs of spot instances. The results confirm that different job execution lengths, job memory footprints, and numbers of revocations contribute to different overhead types related to a job's completion time and deployment cost when using the fault-tolerance approach, thus **positively addressing RQ_3**.

VI. Related Work

While many of the prior works focused on reducing the effect of spot instance revocations by modeling spot markets [9]–[12] and using fault-tolerance methods [4], [5], [8], [13], [14], these works are subject to altering pricing algorithms and are exposed to incurring overhead related to application completion time and deployment cost, respectively. In contrast, `P-SIWOFT` leverages features of cloud spot markets to mitigate the effect of spot instance revocations. Also, other works [1], [15] focused on testing the effect of spot instance revocations on cloud-based applications. However, `P-SIWOFT` focused on reducing the deployment cost and completion time of applications by provisioning spot instances using features of cloud markets. Other researchers worked on transient resource reclamations [6], [16], resource elasticity [17], [18], containerised clouds [19], [20], workflow scheduling in clouds [21], [22] to decrease deployment costs for users while improving the performance of cloud-based applications.

VII. Conclusion

We addressed a challenging problem for applications deployed on cloud spot instances that results from the overhead of employing fault-tolerance mechanisms. We proposed a novel cloud market-based approach that leverages features of cloud spot markets for *Provisioning Spot Instances WithOut employing Fault-Tolerance mechanisms (`P-SIWOFT`)* to reduce the deployment cost and completion time of applications. We evaluated `P-SIWOFT` in simulations and used Amazon spot instances that contain jobs in Docker containers and realistic price traces from EC2 markets. Our simulation results show that our approach reduces the deployment cost and completion time compared to approaches based on fault-tolerance mechanisms.

References

[1] A. Alourani, A. D. Kshemkalyani, and M. Grechanik, "Testing for bugs of cloud-based applications resulting from spot instance revocations," in *12th IEEE International Conference on Cloud Computing, CLOUD*, 2019, pp. 243–250.

[2] I. Amazon Web Services, "Amazon ec2 instances," https://aws.amazon.com/ec2/, 2020.

[3] "Our source code and experimental data," https://www.dropbox.com/s/10b0ut1d6qx3a6s/P-SIWOFT.zip?dl=0, 2020.

[4] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy, "Spoton: a batch computing service for the spot market," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 329–341.

[5] P. Sharma, D. E. Irwin, and P. J. Shenoy, "Portfolio-driven resource management for transient cloud servers," in *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Urbana-Champaign, IL, USA, June 05 - 09, 2017*, 2017, p. 59.

[6] P. Sharma, A. Ali-Eldin, and P. J. Shenoy, "Resource deflation: A new approach for transient resource reclamation," in *Proceedings of the Fourteenth EuroSys Conference*. ACM, 2019, pp. 33:1–33:17.

[7] D. Carraway, "lookbusy – a synthetic load generator," https://devin.com/lookbusy/, 2020.

[8] S. Shastri and D. Irwin, "Hotspot: automated server hopping in cloud spot markets," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 493–505.

[9] M. Khodak, L. Zheng, A. S. Lan, C. Joe-Wong, and M. Chiang, "Learning cloud dynamics to optimize spot instance bidding strategies," in *2018 IEEE Conference on Computer Communications, INFOCOM*. IEEE, 2018, pp. 2762–2770.

[10] A. K. Mishra, A. Kesarwani, and D. K. Yadav, "Short term price prediction for preemptible vm instances in cloud computing," in *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*. IEEE, 2019, pp. 1–9.

[11] V. Khandelwal, A. Chaturvedi, and C. P. Gupta, "Amazon ec2 spot price prediction using regression random forests," *IEEE Transactions on Cloud Computing*, 2017.

[12] C. Wang, Q. Liang, and B. Urgaonkar, "An empirical analysis of amazon ec2 spot instance features affecting cost-effective resource procurement," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, no. 2, p. 6, 2018.

[13] M. Khaldi, M. Rebbah, B. Meftah, and O. Smail, "Fault tolerance for a scientific workflow system in a cloud computing environment," *International Journal of Computers and Applications*, pp. 1–10, 2019.

[14] J. Zhou, Y. Zhang, and W.-F. Wong, "Fault tolerant stencil computation on cloud-based gpu spot instances," *IEEE Transactions on Cloud Computing*, 2017.

[15] A. Alourani, A. D. Kshemkalyani, and M. Grechanik, "T-BASIR: finding shutdown bugs for cloud-based applications in cloud spot markets," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1912–1924, 2020.

[16] M. Shahrad, C. Klein, L. Zheng, M. Chiang, E. Elmroth, and D. Wentzlaff, "Incentivizing self-capping to increase cloud utilization," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 52–65.

[17] F. B. Elloumi, H. Brabra, M. Sellami, W. Gaaloul, and S. Bhiri, "A transactional approach for reliable elastic cloud resources," in *2019 IEEE International Conference on Services Computing (SCC)*. IEEE, 2019, pp. 154–161.

[18] A. Alourani, M. A. N. Bikas, and M. Grechanik, "Search-based stress testing the elastic resource provisioning for cloud-based applications," in *International Symposium on Search Based Software Engineering*, 2018, pp. 149–165.

[19] J. Gao, W. Li, Z. Zhao, and Y. Han, "Provisioning big data applications as services on containerised cloud: a microservices-based approach," *IJSTM*, vol. 26, no. 2/3, pp. 167–181, 2020.

[20] H. Li, L. Zhang, W. Li, and J. Gao, "A service performance based dynamic provisioning approach in containerized cloud environments," in *Proceedings of the 2nd International Conference on Big Data Technologies*, 2019, pp. 177–181.

[21] S. Cao, K. Deng, K. Ren, X. Li, T. Nie, and J. Song, "An optimizing algorithm for deadline constrained scheduling of scientific workflows in iaas clouds using spot instances," in *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking*. IEEE, 2019, pp. 1421–1428.

[22] T.-P. Pham and T. Fahringer, "Evolutionary multi-objective workflow scheduling for volatile resources in the cloud," *IEEE Transactions on Cloud Computing*, 2020.