SPA : Harnessing Availability in the AWS Spot Market

Wong, Walter

IEEE
2021

Wong , W , Zavodovski , A , Corneo , L , Mohan , N & Kangasharju , J 2021 , SPA : Harnessing Availability in the AWS Spot Market . in IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS) . IEEE Conference on Computer Communications Workshops , IEEE , IEEE Conference on Computer Communications Workshops (IEEE INFOCOM) , 09/05/2021 . https://doi.org/10.1109/INFOCOMWKSHPS

http://hdl.handle.net/10138/349714
https://doi.org/10.1109/INFOCOMWKSHPS51825.2021.9484646

unspecified
acceptedVersion

SPA: Harnessing Availability in the AWS Spot Market

Walter Wong*, Aleksandr Zavodovski*, Lorenzo Corneo†, Nitinder Mohan§, Jussi Kangasharju*
*Department of Computer Science, University of Helsinki, Finland
Email: {walter.wong, aleksandr.zavodovski, jussi.kangasharju}@helsinki.fi
†Department of Information Technology, Uppsala University, Sweden
Email: lorenzo.corneo@it.uu.se
§Department of Informatics, Technical University Munich, Germany
Email: nitinder.mohan@tum.de

***Abstract*—Amazon Web Services (AWS) offers transient virtual servers at a discounted price as a way to sell unused spare capacity in its data centers. Although transient servers are very appealing as some instances have up to 90% discount, they are not bound to regular availability guarantees as they are opportunistic resources sold on the spot market. In this paper, we present SPA, a framework that remarkably increases the spot instance reliability over time due to insights gained from the analysis of historical data, such as cross-region price variability and intervals between evictions. We implemented the SPA reliability strategy, evaluated them using over one year of historical pricing data from AWS, and found out that we can increase the transient instance lifetime by adding a pricing overhead of 3.5% in the spot price in the best scenario.**

I. Introduction

Cloud computing is the *de facto* way of providing large scale Internet services. Large cloud data centers leverage economies of scale and elastic demands, mostly via virtual machine offerings on which service providers deploy their own services. This model presents some benefits as enterprises do not need to make upfront investments in hardware purchases for their data centers. Cloud providers, such as Amazon, Google, and Microsoft, offer two different classes of virtual computing instances: on-demand, where users use them until they release the virtual instance; and *transient* instances, where cloud providers sell their spare capacity to users with some availability limitations. These less reliable transient instances attract users due to their cheaper pricing, which can be offered at up to 90% discount compared to the regular on-demand instances [1], [3], [4]. However, these instances do not have any availability guarantees, meaning that they can be shut down by the cloud operator with a very short notice period.

Although transient instances do no offer uptime guarantees, many studies in the past have explored the benefit of these instances. For example, SuperCloud [9] explores how to utilize transient resources via algorithms that deal with the expected failures by migrating the instance running state from one virtual machine to another. Other approaches include saving the service data in a shared partition [8], [10], [16], checkpointing the job processing at regular time intervals [23], increasing application availability by proactively migrating applications between spot and on-demand instances [17]–[19] and mix-and-match spot and on-demand instances to trade availability for cost reduction [2]. Users need to bid for a transient instance by setting the maximum price that they are willing to pay for that instance and, if the market price of the instance goes over the bid price, the instance is evicted from the data center. In order to optimize the bid price, authors in [20], [25] propose algorithms that optimize the bidding price to avoid eviction.

In this paper, we propose *SPot Availability* (SPA) – a framework that assists users in selecting the best AWS transient instance options based on service availability requirements. SPA operation can be summarized in three steps. First, it passively analyzes the historical pricing data of the transient instances. Second, it compares the offerings in different AWS regions and availability zones based on instant price, price volatility, and price change interval. Finally, it returns the user the best transient resource option that matches their availability requirements at a given time. The pricing data analysis shows *there is a minimum price update interval for all spot instances*, and SPA leverages this to place or migrate instances to regions where they can have a longer lifetime. The major contributions we make are as follows.

1) We propose SPA, a software tool that calculates the best spot instances to provide the same reliability as the on-demand instances but at a fraction of the cost.
2) We analyze the AWS historical spot pricing data and highlight the main insights in the data, such as best prices and more stable instances across availability zones.
3) We propose and evaluate one strategy that reduces spot instance failure rates. The results show that our proposed Spot Instance MarketPLace Exploitation (SIMPLE) strategy can achieve the same reliability as on-demand instances with only 3.5% overhead in cost.

The rest of the paper is structured as follows. Section II presents related work. Section III presents the background on AWS. Section IV describes the data analysis and insights in the pricing data. Sections VI and VII present the implementation and evaluation of the SPA framework and the experimental results. Section VIII presents the discussion and some takeaways of the paper. Finally, Section IX concludes the work.

II. Related Work

Cloud economics has attracted the interest of academia, resulting in attempts to deconstruct AWS price formation model [6], infer the optimal bidding strategy using statistical and optimization tooling [25], predict adaptively price dynamics [13], and offer a better market model employing learning techniques [15]. Abhishek et al. [5] explore trade-offs between fixed and spot pricing, whereas [7] analyze the problem of fixed and spot market coexistence. Besides pure economics, there are works addressing other aspects of the spot market utilization, such as dynamic resource allocation and management [14], demand scheduling [12], cost-aware migration, and checkpointing [24].

Next, we examine practical systems and architectures aimed to exploit the advantages of the spot market. HotSpot [18] uses containers to move between spot instances by periodically computing the lowest spot pricing and proactively migrates to a new instance to avoid preemption. One benefit of HotSpot is it does not require any changes in the application. However, the migration mechanism only works within the same availability zone. SpotOn [19] is a batch computing service running on top of spot instances, enabling automatic selection of spot instances and implementing fault tolerance mechanisms to mitigate data loss without modifying the application. It uses containers to encapsulate jobs with their dependencies, and it may use reactive or proactive container state checkpointing on the disk. SpotCheck [17] is a derivative cloud market where SpotCheck purchases cloud resources from providers and resell them with customizations to customers. This is aimed at customers with demands not supported by the cloud providers and SPA can again reduce the overhead and costs of SpotCheck. SuperCloud [11] is a cloud architecture running over OpenStack integrating multiple cloud providers and allowing for live migration across those providers using Xen virtualization. Supercloud uses nested virtualization to enable complete VM migration from one server to another, allowing users to relocate virtual machines from one cloud provider to another without disrupting the running application. Bricklayer [21] optimizes resource composition using Spot instances. It uses a variety of attributes such as ECU, volatility, and eviction rate to compose spot instances aiming to optimize the availability. SPA is different from the previous approaches as it explores the minimum update interval to trigger the migration rather than using prediction techniques or redundant instances. The benefit is that SPA is more assertive in the migration policy and migrates only when required, reducing the overall cost due to the redundant number of instances.

III. AWS Spot Instances Background

AWS offers virtual instances with two types of availability Service Level Agreement (SLA): reserved and on-demand instances have an SLA of 99.95% of guaranteed uptime, and transient instances (known as *spot instances* [1] in AWS) that do not have any guarantees and are available while AWS has spare capacity in their data-centers. Both reserved and on-demand instances have a higher hourly price, and AWS cannot reclaim the instances back unless the user releases them, while the spot instances can be reclaimed at any time as they don't have any uptime SLA.

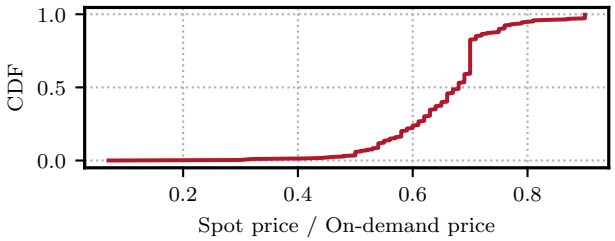

Fig. 1: Spot instance price distribution. More than 50% of spot instances have over 67% off discount over on-demand prices.

AWS spot instances are offered in the spot market and use a dynamic pricing model, where each spot instance has a flexible price at each availability zone and its hourly price varies based on the supply and demand of resources in the data-center. To get a spot instance, users need to bid for it in the market. In AWS, instance bidding can be understood as a pricing threshold. The bid price will remain valid as long as it is higher than the market price and, consequently, the user will get that spot instance. Despite higher price bids for the spot instance, the user only pays for the current market price of that instance, and the bid amount cannot be changed once it has been submitted to AWS. Whenever the market price for that spot instance goes over the bid price, the owner of the spot instance will receive a 2-minute eviction notice to save or migrate the spot data, and AWS will reclaim the resources.

Fig. 1 shows the discount percentage offered in spot instances compared to on-demand ones with the same specification. More than 50% of the spot instances have a discount of over 67% compared to the on-demand pricing, thus, users are able to run up to 3 spot instances (with 3x more resources) at the cost of 1 on-demand instance in AWS.

There are two main challenges of using spot instances regarding selection and availability. For spot instance selection, users have too many options to select, i.e., AWS has 21 regions, 39 availability zones, and 292 spot instances, resulting in a combination of $\approx$ 10k individual spot markets. AWS recommends looking at AWS spot advisor for 3-month of historical data and manually select which is the best instance at a given time. However, this is unfeasible in practice as users would need to analyze too many options and compare each of them. With regards to availability, AWS recommends creating a SpotFleet [2], which is a group of virtual instances where on-demand and spot instances can be mixed together to increase system availability. However, we find through our analysis that users end up paying for the more expensive instance in the group when using SpotFleet.

We explore the historical pricing data to find actionable insights on how to use the spot instances on AWS better. Our aim is to *achieve the same level of availability as on-demand*

but at a fraction of the price.

IV. AWS Spot Instance Observations

We collected and analyzed 15 months of AWS spot pricing data (March 2019 – June 2020) across all regions and availability zones – exceeding 2M price entries. We start analyzing three configuration parameters (region, availability zone, and instance type) and three insights found in the pricing data. We will discuss each of them below.

A. Spot Instance Attributes

Users select an instance using three configuration parameters: geographical region, availability zone, and instance type. Most of the time, customers select a region due to external requirements, i.e., closeness to the customer or availability of specific instance types such as GPUs. Also, the instance type is selected based on the customer's workload, i.e., the amount of CPU and RAM is dependent on the type of application being deployed. However, the *availability zone* (AZ) configuration parameter can provide some economical and availability gains if properly chosen. An AZ is a segmented data center inside the same AWS region and is set up to provide both hardware redundancy and scalability. Each AZ works separately and offers different spot instance types and prices. Thus, our interest is to check which AZ offers the best pricing and the lowest eviction rate for the same instance type. Fig. 2 shows region *us-east-1* has 6 availability zones, and they have different volatility. In Fig 3, it can be observed that a given instance type can have a different average number of changes across the available zones.

B. Spot Instance Insights

We performed the exploratory data analysis (EDA) on the historical spot pricing data and found out three insights that will assist with the spot instance selection.

Instance Price Volatility. The instance price volatility is the number of price changes a spot instance has over time. This metric is important because a high number of price changes can result in two main problems: (a) higher price volatility, resulting in users paying more than expected, and (b) higher eviction rate, as the spot bid price can go over the spot market price. Fig. 4 shows the price volatility of the top 10 instances in *us-east-1* and some prices can increase up to $\approx$ 8x, reaching almost the same as that of the on-demand variant. On the other hand, Fig. 5 shows instances that did not experience much change in their price over the 90-day observation period. Selecting these instances instead would ensure a certain level of guarantee in the price a user pays to acquire them and reduces the probability of eviction.

Takeaway: Not all instances are equal in price volatility, and an effective selection strategy should focus on those with lower prices and volatility (or higher durability).

Average Price Change Interval. Monitoring the interval between the price changes allows us to learn the minimum spot instance lifetime for a given instance type before a price increase that may evict the user of that instance. Fig. 6 shows the price update interval for *m4.16xlarge* instance plotted hourly. We can observe that there is a pattern of update times over days – ranging between 3 to 5 updates per day. On average, the price of the instance updates after almost 6 hours. Therefore, intuitively, we conjecture that there is a recurrent policy at AWS that enforces price updates of spot instances at regular intervals based on the data usage.

Fig. 7 shows price update intervals over one month (specifically September 2019). One can observe from the figure that most of the price change intervals are clustered around 6 hours (fitted with a red dashed line), further strengthening our findings above. Another interesting observation from the figure is that there is a lower boundary of price updates, indicating a minimum lifetime for a spot instance.

Takeaway: Our findings indicate scope for a strategy that picks instances with longer price change intervals, thus guaranteeing more instance running time before the price update that leads to its eviction.

Minimum and Percentile of 1% Price Change Interval. We now investigate both minimum and percentile of 1% (or quantile *q1*) of the price update intervals. By focusing on *(*q1), we can observe the top 99% of the price update intervals and remove any outliers that lie in the lower part of the distribution. Fig. 8 shows both the minimum and *q1* price update intervals for a given spot instance across different availability zones. Our analysis reveals that the *q1* change interval is always at least 4h35min12sec in *all* availability zones, while the minimum update time is the same as the *q1* for 35 out of 39 zones (only 4 zones have lower minimum times).

This finding, along with price change interval, reveals that AWS has rather a roughly fixed price update interval per day (3 to 5 times per day) with a minimum update interval – indicating space for a strategy that exploits this behavior.

Takeaway: AWS's price update policy has a minimum update interval for most availability zones, which can be exploited to maximize service availability.

V. SIMPLE Algorithm to keep the availability

Driven by our findings from historical spot instance price data analysis, we develop the *Spot Instance MarketPLace Exploitation* (SIMPLE) algorithm. SIMPLE's aim is to maximize service availability over spot instances by retroactively migrating it to new instances before potential price updates. SIMPLE calculates the minimum update interval of all instances and updates its historical data whenever there is a change in the pricing update interval from AWS. A rudimentary workflow of SIMPLE is as follows.

First, SIMPLE's engine fetches the historical pricing data filtered by the user's selected region, AZ, and instance type. Based on this information, the engine calculates the instance's price volatility, average price update interval, and the minimum price update time. Then, SIMPLE calculates the remaining lifetime of all candidate instances by subtracting the last price update time from 4h32m (minimum update interval we observed from our analysis). For example, if the last price

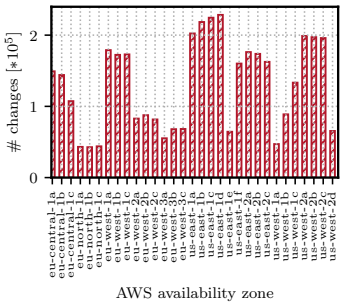

Fig. 2: Number of changes vs. Availability Zones

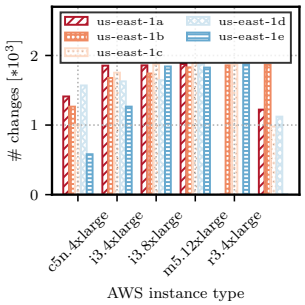

Fig. 3: Price change variation vs. availability zone

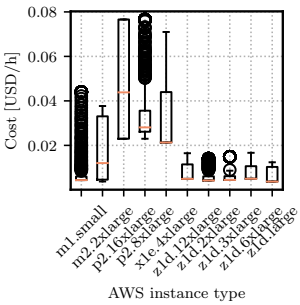

Fig. 4: Top 10 Volatility

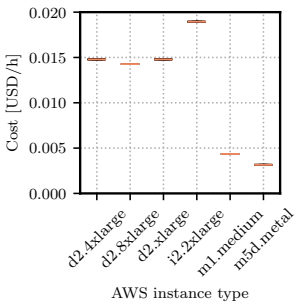

Fig. 5: Bottom 6 Volatility

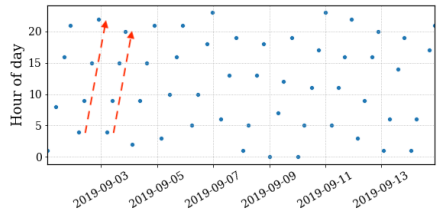

Fig. 6: Spot price update time vs. hour of day (red line shows the trend over the day) for instance *m4.16xlarge*.

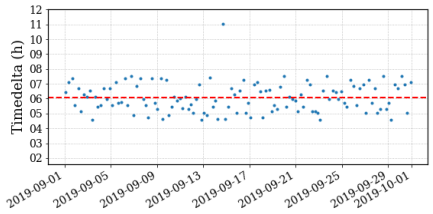

Fig. 7: Spot price update interval (the red dotted line shows the average of 6 hours) for instance *m4.16xlarge*. Both measurements show a sample of a 2-week period of time.

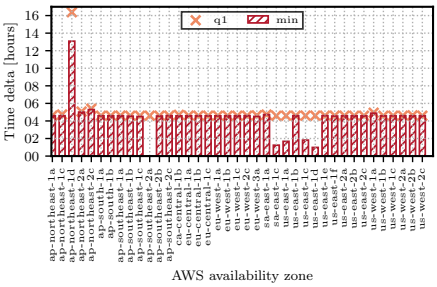

Fig. 8: Comparison between the instance price change in percentile 1% and the minimum interval.

Algorithm 1: SIMPLE's Instance Migration Algorithm

```
Input: region, availability_zone, instance_type, and
       buffer_time
retrieve_historical_data(instance_type)
calculate_price_volatility(instance_type)
calculate_average_update_time(instance_type)
calculate_minimum_update_interval(instance_type)
retrieve_instance_provisioning_time(instance_type)
calculate_migration_time(instance_type)
calculate_time_to_migrate()
while active do
    while current_time ≤ time_to_migrate do
        use_spot_instance()
    provision_new_spot_instance()
    migrate_state_to_new_spot_instance()
    update_last_price_change()
```

change time for an instance was 2 hours ago, then the instance still has 2h32 of guaranteed uptime. Next, the algorithm calculates the time to bootstrap service on a new spot instance (in a different zone or at a different price), deploys a replica of the application, transfers the service's state to the new instance, and kills the older version of the service.

Our experimental data shows that instance provisioning and application deployment roughly takes 10 min to complete. Users can also add buffer time to allow any state transfer after the spot instance is deployed. Since our analysis reveals a guaranteed uptime ranging around 4h32m, we conservatively select the 4-hour mark as a trigger to migrate the service to a new spot instance, practically guaranteeing a continuously running service. One limitation of SIMPLE is that it requires a transition time for transferring data and application state between instances. Therefore, for stateful applications, this can result in unwanted service interruption. In order to prevent any downtime, we use additional redundant spot instances during the transition period.

VI. Implementation

SPA is implemented using Python for the core components and Numpy, Scipy, Pandas for data analytics and feature engineering, and the components run in Docker containers configured in a Docker Swarm setup [22]. Fig. 9 shows the architecture with its components and correlates to each step of the algorithm presented in 1.

In step 1, SPA receives the user parameters such as region, availability zone, and instance type. Then, the historical data retriever periodically fetches spot pricing from AWS and updates the local database (step 2). The main purpose of this scrapper is to keep the last price update for each spot instance, to allow the analytics engine to calculate the price change interval. The data engineering component parses the raw data

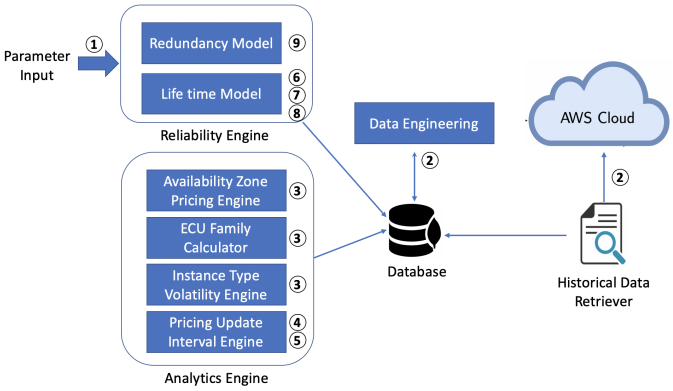

Fig. 9: SPA architectural components. The numbers correlate to the steps in Algorithm 1.

retrieved from the spot pricing, reshapes it, and performs feature engineering and data processing for consumption by the other components (step 2). The analytics engine calculates each metric that defines a spot instance in our model, e.g., the pricing of the instance in each availability zone, the instance price volatility, eviction rates, and the pricing update interval (step 3 through 5). The reliability component calculates the minimum update interval for a given instance and it is responsible for the correct migration to a new spot instance (steps 6 through 8). It also runs the SIMPLE algorithm and, once the component triggers the migration event, it provisions a new instance on AWS and migrates to it (step 9).

VII. Evaluation

We evaluate the SIMPLE algorithm in two different scenarios: first, we compare the cost to run the SIMPLE algorithm vs. a naive deployment of redundant spot instances; second, we evaluate SIMPLE's efficiency in providing availability on the historical pricing data.

For the first evaluation, we calculate the additional cost incurred by the SIMPLE algorithm, shown in 1. Recall that the SIMPLE algorithm states that we migrate to a new spot instance every 4h30min. Assuming the spot instance provisioning and state migration takes $\approx$ 10 minutes, we will have an additional instance running in parallel for 10 minutes, thus, $\approx$ 17% of a 1-hour running time for a redundant spot instance.

$$C_s(t) = (p_{spot} * t) + (0.17 * p_{spot}) * m_{spot}(t) \quad (1)$$

where t is the time, p_{spot} the spot instance hourly price, and m_{spot} the number of migrations at time t (per instance type). If we consider a 24-hour period, the total number of migrations is going to be 5. Therefore, the total cost $C_s(t = 24)$ is 24.85 hours (24 hours of running instance plus 50 min for the overlapping running machine), representing an overhead of 3.5% for SIMPLE.

Tab. I compares the cost of different approaches for a sample of spot instances. The costs are given as percentages of the on-demand instance price, i.e., the on-demand price is 100% for reference. The table shows the strategy selection

TABLE I: Cost comparison between different strategies. On-demand instance represents 100% of the price.

Instance Type	Spot Price	SIMPLE	$r = 1$	$r = 2$
m2.xlarge	10.00%	10.35%	20.00%	30.00%
r3.xlarge	19.49%	20.17%	38.98%	58.47%
m5dn.24xlarge	25.00%	25.88%	50.01%	75.01%
c3.4xlarge	32.36%	33.49%	64.71%	97.07%
a1.large	44.31%	45.86%	88.63%	132.94%
m5a.2xlarge	52.50%	54.34%	105.00%	157.50%
r5dn.xlarge	69.10%	71.52%	138.20%	207.31%

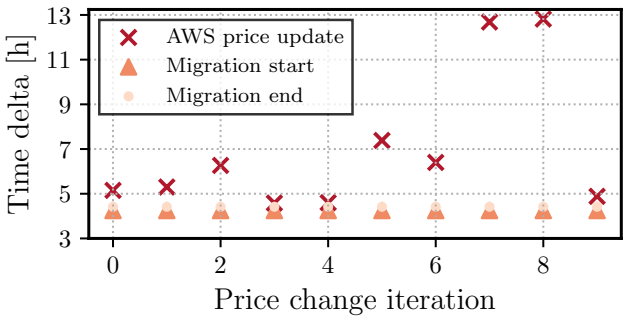

Fig. 10: Comparison between SIMPLE migration time - start and end of migration - and the AWS price update time. SIMPLE anticipates the price update time in all cases.

that can be used when choosing which strategy to select, e.g., spot instances that have a discount of over 67% can have up to 2 redundant elements, while other instances with less discount can only use the SIMPLE strategy. Note that in the case of redundant instances, we don't need to have an extra overlapping spot instance to migrate given we already have redundancy. Thus, the cost is proportional to the number of redundant elements provisioned in the spot instance set.

The second evaluation aims to check if SIMPLE works over the historical data. Fig. 10 compares the spot migration events (start and end) and the actual AWS spot price change. The figure shows that SIMPLE always starts before the price update event and, in some cases, it triggers a migration in 4h30min, despite the migration only going to happen over 12 hours later. Although this scenario can happen, SIMPLE guarantees that by migrating every 4h30min, it is able to avoid any eviction. We also evaluated it on all historical data and the experimental results show that SIMPLE can avoid spot instance preemption in all cases.

VIII. Discussion

Many research works have tried to figure out the best strategy to use spot instances, as we discussed in Section II. In this paper, we started analyzing the historical pricing data to find out insights into the AWS price update algorithm. One insight that we found is the minimum update interval, which is 4 hours and 32 minutes for roughly all availability zones. If we take into account the 1% percentile, then this

minimum update interval applies to all availability zones. We believe there is a system that periodically polls the number of resources available at each availability zone and updates the price accordingly and that polling interval should be around that percentile 1% update time. Based on that, we crafted the SIMPLE algorithm that provisions a new spot instance when the lifetime of the original instance is going to expire.

Despite begin a simple algorithm, it only increases the overall instance price by 3.5% due to the overlapping running time of a parallel instance for the migration. Compared to the other approaches discussed in the literature (see Section II), SIMPLE is much simpler and more predictable. It only looks at the last price update time and plans for a spot instance migration when the lifetime is about the expire. This makes SIMPLE very easy to implement. One limitation of SIMPLE is it requires a transition period in which data is transferred from one instance to another, and this may lead to downtime. To tackle this issue, we designed a reliability model over spot instances, allowing for the optimal composition of redundant instances to guarantee availability at the cost of an increased hourly price. The redundant instance can work as a hot standby or run in parallel, depending on the user's requirements. Compared to the on-demand model, which has an SLA of 99.95% of reliability, the pricing is worth it when the spot instance is at least 70% off. Thus, the user can have 3 spot instances running in parallel, providing the same reliability but with 3 times more processing power.

SPA also can adapt to variations in the AWS pricing model. If AWS moves away from the minimum pricing model, SPA can detect this change as it recurrently calculates the minimum pricing of each instance and updates the SIMPLE algorithm to take into account the new minimum time. Another important aspect is if too many users start to use SPA, the demand for spot instances will increase and, consequently, the spot prices. However, even if they get more expensive, SPA still will be able to get savings compared to on-demand instances, as it will optimize the use of existing spot instances through the SIMPLE algorithm.

IX. Conclusion

In this paper, we presented SPA, a framework that assists users to select the best spot instance across different availability zones. SPA provides insights based on the historical pricing data and provides a model in which more stable instances (with less price variation) and with larger price update intervals are selected. SPA leverages the observation that the minimum update interval time is 4h32min and proposes the SIMPLE algorithm, which exploits the minimum price update interval and triggers the migration to another spot instance to prevent any instance evictions, resulting in the same availability as on-demand instances. The experimental evaluation shows the SIMPLE algorithm only has an overhead of 3.5% over the regular spot price and, comparing with the 15-month of historical pricing data, SPA is able to anticipate all AWS price updates and migrate to another spot instance before any eviction occurs.

References

[1] Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/, 2020. Accessed: 2020-06-30.

[2] How Spot Fleet Works - Amazon Elastic Compute Cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html, 2020. Accessed: 2021-01-10.

[3] Low-priority VMs in Batch. https://azure.microsoft.com/en-us/pricing/details/batch/, 2020. Accessed: 2021-01-10.

[4] Preemptible VM Instances. https://cloud.google.com/compute/docs/instances/preemptible, 2020. Accessed: 2021-01-10.

[5] V. Abhishek, I. Kash, and P. Key. Fixed and market pricing for cloud services. In *2012 Proceedings IEEE INFOCOM Workshops*. IEEE.

[6] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing amazon ec2 spot instance pricing. *ACM Transactions on Economics and Computation (TEAC)*, 1(3):1–20, 2013.

[7] Ludwig Dierks and Sven Seuken. Cloud pricing: The spot market strikes back. *Available at SSRN 3383420*, 2019.

[8] A. Harlap, A. Chung, A. Tumanov, G. Ganger, and P. Gibbons. Tributary: spot-dancing for elastic services with latency slos. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, July 2018.

[9] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. Smart spot instances for the supercloud. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud '16. ACM, 2016.

[10] P. Joaquim, M. Bravo, L. Rodrigues, and M. Matos. Hourglass: Leveraging transient resources for time-constrained graph processing in the cloud. In *Proceedings of the 14th EuroSys 2019*, 2019.

[11] I. Kash, Q. Jia, Z. Shen, W. Song, R. Renesse, and H. Weatherspoon. Economics of a supercloud. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud '16, 2016.

[12] Robert Keller, Lukas Häfner, Thomas Sachs, and Gilbert Fridgen. Scheduling flexible demand in cloud computing spot markets. *Business & Information Systems Engineering*, 62(1):25–39, 2020.

[13] M. Khodak, L. Zheng, A. Lan, C. Joe-Wong, and M. Chiang. Learning cloud dynamics to optimize spot instance bidding strategies. In *IEEE Conference on Computer Communications (INFOCOM 2018)*, 2018.

[14] M. Nagpure, P. Dahiwale, and P. Marbate. An efficient dynamic resource allocation strategy for vm environment in cloud. In *IEEE International Conference on Pervasive Computing (ICPC 2015)*, 2015.

[15] A. Prasad, M. Arumaithurai, D. Koll, Y. Jiang, and X. Fu. Ofm: An online fisher market for cloud computing. In *IEEE Conference on Computer Communications (INFOCOM 2019)*, 2019.

[16] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *the 11th European Conference on Computer Systems*, EuroSys '16, 2016.

[17] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *the 10th European Conference on Computer Systems (EuroSys '15)*, 2015.

[18] S. Shastri and D. Irwin. Hotspot: Automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[19] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. Spot-on: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, 2015.

[20] S. Tang, J. Yuan, and X. Li. Towards optimal bidding strategy for amazon ec2 cloud spot instance. In *the 2012 IEEE 5th International Conference on Cloud Computing*, CLOUD '12, 2012.

[21] W. Wong, L. Corneo, A. Zavodovski, P. Zhou, N. Mohan, and J. Kangasharju. Bricklayer: Resource composition on the spot market. In *IEEE International Conference on Communications (ICC 2020)*.

[22] W. Wong, L. Corneo, A. Zavodovski, P. Zhou, N. Mohan, and J. Kangasharju. Container deployment strategy for edge networking. MECC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, SoCC '16, 2016.

[24] Sangho Yi, Artur Andrzejak, and Derrick Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2011.

[25] L. Zheng, C. Joe-Wong, C. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.