Spotlytics: How to Use Cloud Market Places for Analytics?

Tim Kraska[1], Elkhan Dadashov[1], Carsten Binnig[1]

Abstract: In contrast to fixed-priced cloud computing services, Amazon's Spot market uses a demand-driven pricing model for renting out virtual machine instances. This allows for remarkable savings when used intelligently. However, a peculiarity of Amazon's Spot market is, that machines can suddenly be taken away from the user if the price on the market increases. This can be considered as a distinct form of a machine failure. In this paper, we first analyze Amazon's current spot market rules and based on the results develop a general market model. This model is valid for Amazon's current Spot service but also many potential variations of it, as well as other cloud computing markets. Using the developed market model, we then make recommendations on how to deploy analytical systems with the following three fault-tolerance/recovery strategies: re-execution as used by traditional database systems, checkpointing as, for example, used by Hadoop, and lineage-based recovery as, for example, used by Spark. The main insights are that for traditional database systems using significantly more instances/machines can be cheaper, whereas for systems with checkpoint recovery the opposite is true, while lineage-based recovery is not beneficial for cloud markets at all.

1 Introduction

Cloud Computing has become a true utility. In particular for analytics cloud computing offers many benefits such as instant access to virtually infinite resources. However, cloud computing is not for free. Service providers such as Amazon Web Services (AWS) charge a premium that can quickly add up, even for running small analytical tasks. Yet, cloud users are not the only ones struggling to keep costs down - providers are also doing so, for competitive reasons. For providers, it's critical to avoid the evil twins of under- and over-utilization. In the former case, machines sit idly, wasting energy and space, whereas in the latter, congested resources lead to violations of service-level agreements, incurring penalties and leave customers unsatisfied.

Market places are one of the most effective tools to control fluctuating demand and provide a win-win situation for consumers and service providers. In fact, almost all other major utilities, including electricity and water, are traded on market places. Therefore, it comes at no surprise that Amazon, among other providers, starts to offer its cloud computing resources on market places. Amazon calls its market for virtual machines "Amazon EC2 Spot instances". Instead of paying a fixed price, users bid the top price they're willing to pay for one hour of use of a virtual machine instance. As long as the current market price is lower than the bid, the user will be able to use the machine. For Amazon's Spot market place, we found that for the last 3 years, prices are on average 9× lower than Amazon's EC2 fixed prices (also referred to as on-demand market). However, the price differential can sometimes be as high as 30× more than the fixed prices.

Though the advantages of a market place for virtual machines are compelling, a fundamental difference exists between market prices for utilities such as electricity and those for virtual

[1] Brown University, Providence, RI, USA

machine instances such as EC2. If a utility price increases, the user can choose to use less electricity by turning off appliances. On Amazon's Spot Market, the machine automatically shuts down and becomes unavailable as soon as the price increases above the bid, resulting in a distinct form of machine failure. Whereas failures in the on-demand market (i.e., the fixed-price market) are unlikely, they are the norm with spot instances, and can be as extreme as having a failure every few hours or even minutes, depending on the bid price. Furthermore, machines within the same category (that is, the same type, OS, and region) and bid price usually shut down together because they observe the same price fluctuations, making failures not only highly correlated within a market, but also different from "normal" failures in the cloud.

In this paper we address the question, how analytical systems with different fault-tolerance strategies should be deployed on cloud market places. While recent studies looked at the "optimal" bid-price [An10] or how to adjust existing systems, like Hadoop or MySQL to spot instances [Ch10, Bi15], none of them systematically studied the implications of spot markets on analytics, specifically when using different fault-tolerant strategies. Instead, existing work is rather ad-hoc and heavily tailored towards Amazon's current market rules.

In this paper, we first systematically analyze and model Amazon's current spot market rules as well as discuss possible variations of it. Based on the results, we make theoretically sound recommendations for three common fault-tolerance strategies: (1) *fail-stop-redo* as used by traditional database systems, (2) *check-pointing* as implemented by Hadoop-like systems [Had, Bu10], and (3) *lineage-based* recovery as, for example, used by Spark [Za10]. Furthermore, we evaluate all strategies using real-world data sets obtained from Amazon's spot market place.

The remainder of this paper is organized as follows: In the next section, we summarize Amazon's current market rules and describe some of the unique characteristics we found. In Section 3 we abstract away from Amazon's rules and develop a more general market model, which we believe will also remain valid in the future. In Section 4 we use this general model to make recommendations on how to deploy systems with varies kinds of fault tolerance strategies. Within this section we also validate our results using real-world traces from Amazon Spot Market. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Amazon's Current Spot Market

Currently, Amazon offers the biggest cloud market place. In this section, we first describe Amazon's market model in more detail, before we generalize it in the next section.

2.1 Overview of the Market Rules

Amazon offers the same machine configurations on the spot market as on their on-demand/fixed-price platform: general purpose (m), compute intensive (c), memory intensive, etc., each targeting different use cases. Each category contains different specific machine configurations varying in the main memory and number of virtual CPUs etc (e.g., one configuration called *c3.8xlarge* has 32 virtual CPUs, 60GB of RAM and two 320GB SSDs).

Most configurations are offered in eight global regions and can be used with any of the four supported operating systems.Users can freely pick the machine, region, and OS they want. In this paper, we refer to a specific combination of the machine configuration, OS and location as an *instance type*. Every single instance type has its own independent market and, thus, price. That is, when an instance type running on the East Coast with Linux is expensive, the same machine with Linux on the West Coast can still be cheap. In 2015, Amazon had a total of 1640 different markets (i.e., prices).

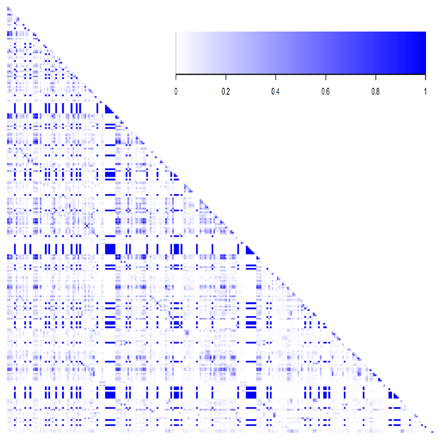


Fig. 1: US East1 Market (pearson) correlation heatmap with 1 min data interpolation of the latest 2 months. Every dot is the correlation between two machines on the spot-market. The darker the color, the stronger the correlation.

Users can buy machine time by bidding the maximum price they are willing to pay for an hour of use. Assume, that the user bids $0.75 per hour for an c3.8xlarge instance running Linux in the US-East region. If the user's bid price is higher than the selected spot instance's current price, then the instance is launched - (usually) in less than a minute. If the consumer's bid price is less or equal to the selected spot instance's current price, the request is pending. In this case, the user can either cancel the request or wait until the spot instance price drops below the bid. It is also possible to bid on N machines of the same type as a *bundle*, which ensures that the bid is either successful for all N machines or for none.

Machines are billed based on the Spot price at the start of each instance-hour [Am]. For example, if a $1 bid is successful at 2pm at a price of $0.7, which shortly after drops to $0.5, the user still pays $0.7 for the hour of use. Even if the user has a successful bid and is able to reserve the machine, there is no guarantee that it is usable for the entire hour. If the price increases during the hour above the bid price, the machine is shut-down immediately resulting in a distinct form of machine failure.

An important peculiarity of the current Amazon Spot Market is that, if Amazon shuts a machine down, the user does not need to pay for the partially used hour. For example, if a machine shuts down after 30 minutes due to a price increase, the user pays nothing; if it shuts down after 90 minutes, the user pays for only one hour. Yet, if the user himself shuts down the machine before the end of the hour (and prior to any price increase that would have caused a shut-down), the user always has to pay for the full hour regardless of the price increase.

2.2 Amazon Failure Characteristics

When the price of a specific instance increases above a successful bid of a user, the Spot instance is shut down, resulting in a distinct form of failure. In the following, we describe our finding of studying spot failures over a period of almost 2 years:

Failures are correlated: In contrast to traditional hardware failures, machines with the same bid for a specific instance type on the Amazon Spot market usually fail together. If the price increases above the bid price, all reserved machines that have the same bid price fail, if the price increases exactly to the bid price some reserved machines might fail. However, even beyond a single instance type, failures are correlated. Figure 1 shows the price correlation between different markets (i.e., different machine types, OS and data centers) over 2 month. As shown, the demand for different instance types can be highly correlated (e.g., the prices between the same category in different data centers). Considering correlated failures is thus one of the main challenges.

Failures are unavoidable: In the spot market, the available number of instance is dependent on the on-demand market, making the spot market a two customer class auction. The primary class users of the on-demand market pay a fixed price (also referred to as a reserved price), whereas the second class of users, the spot instance customers, can only use the left-over machines. As a result, there is no way for the second class, the spot market, users to avoid failures as in the worst case the first class can consume all resources.

Other Peculiarities: We also noted that some markets have bursts of high demand, probably explainable by power users. However, we also observed that the price fluctuations in some markets suddenly stopped, or that some markets have a weird almost binary (high price, low price) behavior. Whereas we assume that in the former case, Amazon might have added additional machines creating oversupply, we assume for the latter case, that the market is currently simply too small and, thus, creates the binary market behavior. We also observed on several occasions, that less-powerful machines are more expensive than more powerful machines; i.e., the market is not arbitrage free.

2.3 Discussion

Given these observations from the previous sub-section, we found that advanced market-based bid strategies are not very effective right now when used for the spot market. Some recent work [Be11] even suggested, that Amazon generates the price at random from within a tight price interval via a dynamic hidden reserve price instead of using a real demand-driven auction that follows the $(N + 1)^{st}$ model.

As a consequence of the findings in [Be11] and our own observations, we decided not to develop bidding-strategies based on standard models. Instead, we model the price changes as a random process and only use high-level aggregates (e.g., the average mean-time-to-failure for a given bid price in the last x days as the main statistic). It should be noted though, that even many advanced stock trading models still use the same oversimplifying assumption that price changes are a random process. In the future, and with the maturity of the spot market, we might need to revisit this decision.

3 Cloud Computing Market Models

Amazon picked one configuration in the design space of possible market models but also keeps it open to change its rules in the future. Moreover, other providers might decide for a different setup of their cloud markets [Mu12]. In the following, we therefore discuss alternatives to Amazon's current Spot market rules, argue which rules are likely to persist or change, and based on those considerations develop a more general market model, which we then use throughout the remainder of the paper.

3.1 Instance Termination

While, at the moment Amazon can terminate an instance at any time, it is also easy to envision a market in which the machine is assigned to a user for a fixed amount of time, here referred to as a *reserved market*. In this case, a successful bid would allow the user to use the machine for at least X minutes (ignoring real hardware failures). On the one hand, this market would certainly provide benefits to the customer because it simplifies the assumptions. On the other hand, reserved markets reduce the revenue for the cloud provider. Moreover, as mentioned earlier, Amazon offers the over-capacity from the on-demand market (1st customer class) on the spot market (2nd customer class). It is reasonable to assume, that if more resources are needed for the on-demand market, they are taken away from the spot market by automatically shutting down machines. Thus, if the cloud provider would guarantee the resources for a fixed amount of time, he might not be able to sell the machine for a higher price on the on-demand market.

We therefore conclude, that Amazon's current product offerings are reasonable and likely to be used as blueprint by other providers. However, we can not exclude, that there might be other market models in the future. Studying them is beyond the scope of this paper.

3.2 Billing Policy

Amazon rents its virtual machines per hour. Obviously, an hour is just an arbitrary value. In the remainder, we assume that the customer is billed at the beginning of every ε-th minute of use:

Definition. *$\varepsilon \in \mathbb{N}$ minutes is the billing interval and $T_{startup} \in \mathbb{N}$ the time from the initial start of the machine until useful work can be done.*

Note, that the unit of ε is arbitrary. In this paper, we assume minutes to simplify the presentation. Moreover, we assume that the required startup time of a virtual machine $T_{startup}$ is (significantly) smaller than ε.

3.3 Auction and Pricing Principle

Amazon seems to use a $(N+1)^{st}$-based auction algorithm. However, it is likely, that they use a different price-finding scheme [Be11] or might even change the strategy in future. To generalize the findings of this paper, we do not rely on the knowledge of the strategy but only require to be able to observe the auction price over time. Furthermore, without loss of generality, we define that the price is redetermined at the beginning of every hour.

Definition. *$c_s^t \in \mathbb{R}$ is the Spot price at time t for instance type s. If the user successfully reserves an instance at time t_0 and uses it for m minutes, she has to pay*

$$C_s(m) = \sum_{i=0}^{\lfloor \frac{m}{\varepsilon} \rfloor} c_s^{t_0+i\varepsilon}$$

if the user bid b stays above the spot price for the duration of the reservation, i.e.,

$$\forall t \in [t_0, t_0 + m] : c_s^t \leq b$$

3.4 Notifications and Failures

We now address how to include the different failure cases into our model. Currently, Amazon shuts down the machine immediately when the price raises above the bid price. We already know from conversations with the Amazon Spot team, that they are considering to introduce a *warning* prior to the automatic shutdown to give the user time to save its intermediate result and allow for a clean shutdown.

Definition. *The time $\omega \in [0, \varepsilon]$ is a notification to the user before the automatic shutdown.*

While an interesting concept, the ω time also implies a minimum running time of the machine. In the extreme case, when $\omega = \varepsilon$ the auction changes to a reserved market auction as defined in Section 3.1 with all its benefits but also disadvantages for the cloud provider. In the remainder, we assume $\omega \ll \varepsilon$.

Furthermore, Amazon offers the functionality to start n machines together (i.e., either the bid for all of them is successful or none). We further observed, that reserved instances from the same user and with the same bid price either fail together (e.g., if the price increases above the bid), or they stay active together. We consider this as a valid assumption, which should also hold for other market models.

3.5 Cost for partially-used Intervals

Amazon currently does not charge for a partially used hours, if the machine was shutdown by Amazon because of a price increase. Similarly to the billing interval, this is an arbitrary convention and should be relaxed. We therefore introduce the notion of a discount γ for partially used hour.

Definition. *Let $\gamma \in [0, 1]$ be the percentage the user has to pay for a partially used hour, t_0 be the starting time of the instance and m the minutes since the start until the price raises above the bid b, then the total cost is:*

$$C_s(m) = \sum_{i=0}^{\lfloor \frac{m}{\varepsilon} \rfloor - 1} c_s^{t_0 + i\varepsilon} + \gamma \frac{(m \bmod \varepsilon)}{\varepsilon} c_s^{(t_0 + \lfloor \frac{m}{\varepsilon} \rfloor \varepsilon)}$$

$$\gamma \in [0, 1] \wedge c_s^{(t_0+m)} > b \wedge \forall i \in [t_0, t_0 + m[: c_s^i < b$$

Note, that $\sum_k^l$ with $k > l$ defines the empty sum and that our model allows a maximum discount of 100% (i.e., $\gamma = 0$). Finally, in the equation above and for the remainder of the paper, we do not consider real hardware failures as they are relatively rare.

4 Deploying Analytical Systems on Spot Markets

In the previous section, we developed a generalized cloud market model for virtual machines, which not only covers Amazon's current market rules but will also remain valid for some potential changes (e.g., the discount). Given the model, we now discuss how analytical systems that use different fault-tolerant strategies are best deployed on a cloud market for virtual machines. In particular, we consider the following three fault-tolerance/recovery strategies that are most common in today's systems: (1) re-execution (typically used by traditional distributed database systems), (2) checkpointing (used by Hadoop and more recent frameworks), and (3) lineage-based recovery (e.g., Spark [Za10]).

4.1 Workload and Problem Statement

There are many different types of workloads and ways to model them. Here, we only consider the simplest form of an analytical task. We assume a single analytical task j, which requires a pre-determined amount of CPU cycles R_{CPU} per ε and memory/disk space R_{space}. A single task j could comprise an entire workflow or a query plan, composed of various smaller jobs. There exist a lot of work on determining the resource requirements of queries and analytical workflows[He11b, He11a]. Considering that sub-components of a task potentially have different resource requirements is beyond the scope of this paper.

If not mentioned otherwise, we assume that the analytical job is embarrassingly parallel and only dependent on the given CPU power (e.g., in the case of Amazon referred to as virtual CPUs).
Definition. *A task t takes $R = R_{CPU}/I_{CPU}$ minutes on an arbitrary instance that meets the requirements related to R_{space} and CPU "power" of I_{CPU} per billing interval ε.*

While this is an oversimplification as we assume no $T_{Startup}$ cost and perfect parallelism, we believe that this first simplified model helps already to gain interesting insights and is a first step in the right direction. It should be noted, that this approach is much in-line with any other development of a more theoretical model for any new problem. For example, even today most theoretical inventory models use oversimplified Poisson-based queuing theory or the famous Black–Scholes model to detect arbitrage never considered irrational traders and yet, is still heavily used worldwide. The reason for these simplifications is, that it creates mathematical more convenient models, which are human-understandable and function as the foundation for more complex but also more realistic models. This here presented model falls into the same category and we hope that the model will eventually be extended to include variants of Amdahl's law.

Given the assumptions before, our goal is to find the optimal deployment strategy:
Problem Statement 4.1. *Given the resource requirements R_{CPU} and I_{CPU} for a task j find the optimal instance type s and the optimal number of machines n to minimize the total expected cost $E(C)$ associated with executing the task.*

Thus, we consider the cost as most important factor with others, such as the real execution time, coming later. In the following, we will show how to achieve this goal for the different fault-tolerant strategies. Independent of the fault-tolerant strategy, a simple observation can already be made:
Proposition 1. *It is never beneficial to shut down an instance before the end of the billing interval ε.*

Even though it might sound obvious, it is an important observation (we omitted a formal proof). Terminating an instance before the end of the billing interval does never save money because the instance is billed at the beginning of the interval. On the contrary, it can even increase the cost if a discount for partially used units exists ($\gamma < 1$) because the longer the machine runs after the finishing, the higher the chance that the discount will be applied to the hour. As we will see later, this observation plays an important role for all deployment strategies.

4.2 Strategy 1: Re-Execution

As a first fault-tolerant model we consider the simple *re-execution* strategy. Traditional database systems do not employ intra-query recovery — that is, if one or all machines fail during query execution, the analytical job/query must started from scratch. In the case of cloud markets, that even means that in case of failure the whole cluster with all its software has to re-start and re-execute the task.

The first observation we can make is:

Proposition 2. *A failure notification time ω has no impact on the deployment strategy*

This is a simple fact of the re-execution fault-tolerant model. A model which is not capable of re-starting, does not benefit from a shutdown notification.

However, an important question remains: What is the optimal deployment strategy. To tackle this question, we need to model the failures rates. We assume that the market price follows a Lévy process (e.g., a Wiener process) without a drift and an average price of $\bar{c}_s^b$ given a bid b. Note, that most financial market models are based on this assumption except that they might also consider a drift [Bl73]. For the remainder of this section, we assume a fixed bid price and simply write c for $\bar{c}_s^b$.

Furthermore, we model the price changes as a Poisson process. The bid-price induced failure event follows a Poisson process with intensity λ^b for the interval ε, a given bid price b and some instance type. It should be noted, that a Poisson process is a frequently used model for the arrival rate of failures. Different from other failure models (e.g., for hardware) where the likelihood increases over time, for bid price-induced failures there is no wear and tear making it a stochastic process with *independent* and *stationary* increments. Thus, the Poisson process is actually a better model for this type of failure than for "normal" failures.

Finally, it is relatively easy to determine the maximum likelihood estimator based on the history for both, λ^b and $\bar{x}_s^b$, by simply counting the time between failures and averaging the prices less or equal to b.

Giving these assumptions and assuming that $E(C_s^1)$ is the expected cost of running m machines of type s for exactly 1 billing interval ε to finish the job (we use capital C here to refer to the entire cost), the following holds:[2]

Proposition 3. *If $\lambda > 0$, $\gamma < 1$ and $(R_{CPU} \bmod q \cdot m \cdot I_{CPU}) = 0$, then running a task in a single billing interval ε is cheaper than running the job with fewer resources over several intervals until completion, i.e., in this case it holds that*

[2] Note, that we assume that we can perfectly split the task among the machines ($(R_{CPU} \bmod q \cdot n \cdot I_{CPU}) = 0$). For reasonably large tasks and with Amazon's large spectrum of machine types, this is reasonable. However, for smaller tasks the bin-packing problem becomes an issue and remains future work.

$$qE(C^1) \leq E(C^q)$$

The intuition behind this result is as follows: As long as a failure rate and a discount exist, the cost of a successful task is the same, but every failed task costs more as the user has to potentially pay the full price for machines even though he did not receive any usable result. More formally, let us assume that $q \cdot m$ is the number of machines to run the job in exactly one billing interval and m the number to run the job in exactly q intervals. In both cases, the total cost for a successful run is $C_{suc} = qmc_s$ and, thus, it is not important if the job is done in q or in 1 interval. However, the average cost for an unsuccessful run is:

$$\begin{aligned} &\sum_{k=1}^{\frac{\varepsilon R}{n}} \left(\underbrace{P(X^{t=k} > 0 \cap X^{t<k} = 0)}_{i} \left(\underbrace{cn \left\lfloor \frac{k}{\varepsilon} \right\rfloor}_{ii} + \underbrace{cn\gamma \left(\frac{k}{\varepsilon} - \left\lfloor \frac{k}{\varepsilon} \right\rfloor \right)}_{iii} \right) \right) \\ &= \sum_{k=1}^{\frac{\varepsilon R}{n}} \left(\underbrace{\left(1 - e^{-\frac{\lambda}{\varepsilon}}\right) e^{-\frac{\lambda(k-1)}{\varepsilon}}}_{i} \left(\underbrace{cn \left\lfloor \frac{k}{\varepsilon} \right\rfloor}_{ii} + \underbrace{cn\rho \left(\frac{k}{\varepsilon} - \left\lfloor \frac{k}{\varepsilon} \right\rfloor \right)}_{iii} \right) \right) \end{aligned} \tag{1}$$

Here $X^{t=k}$ is a discrete random variable describing the number of failures in minute k. To determine the average cost, we sum up the likelihoods that the instances fail in a particular minute during the execution (i) times the cost for a failure during that particular minute (1-(ii) and 1-(iii)). The last minute before the job finishes can be calculated as $\varepsilon R/n$ as we normed the CPU requirements to one billing interval. The failure likelihood $1 - (i)$ is simply the probability mass function for at least one failure in a minute, $(1 - P(X = 0))$, times the likelihood that we had no failure in the previous $(k - 1)$ minutes, $P(X = 0)^{k-1}$. Note, that we make use of the fact, that the Poisson distribution is *infinitely divisible*. That is, we can divide λ by the billing interval length ε to calculate the likelihood of a failure per minute. The cost of the failure consist of two parts: $1 - (ii)$ defines the cost for every completed billing interval, whereas $1 - (iii)$ describes the cost for the partially used hours. Note, that $(k \bmod \varepsilon) = k/\varepsilon - floor(k/\varepsilon)$. A transformation of equation 1 leads to:

$$cn\left(e^{\frac{\lambda}{\varepsilon}} - 1\right) \sum_{k=1}^{\frac{\varepsilon R}{n}} \left(\underbrace{e^{-\frac{\lambda(k-1)}{\varepsilon}} \left\lfloor \frac{k}{\varepsilon} \right\rfloor (1-\gamma)}_{i} + \underbrace{e^{-\frac{\lambda(k-1)}{\varepsilon}} \frac{k}{\varepsilon}}_{ii} \right) \tag{2}$$

Thus, if no discount is given ($\gamma = 1$) (i.e., the user has to pay 100% for partially used hours), the first part of the equation $2 - (ii)$ equals zero and it does not matter if the task runs for more than one billing interval. However, if $\gamma < 1$ than allocating more machines to finish in one hour (or even less) is always beneficial as it guarantees that $\frac{\varepsilon R}{n} < \varepsilon$ and therefore, $\forall k \in [1, \varepsilon] : \lfloor \frac{k}{\varepsilon} \rfloor = 0$, forcing again to zero-out $2 - (i)$.

Given that, we can make another even more interesting observation:
Proposition 4. *Using more machines to finish before the end of the billing interval ε can be cheaper than using fewer machines.*

At a first glance, this is counter-intuitive: Finishing earlier and wasting resources can be cheaper than using the resources optimally until the end of the billing interval. The reason lies yet again in the discount in the case of a failure. In order to prove the proposition, we first need to derive the total expected cost for executing a task j.

From proposition 3 we already know, that executing a task in a single billing interval is always beneficial. Thus, we can restrict ourselves to one billing interval only. Based on equation 2 we can derive the expected cost for a failure between minute a and b as:

$$\begin{aligned} EIC(a,b) &= nc\sum_{k=a}^{b}\frac{k}{\varepsilon}\gamma P(X^{t=k}>0\cap X^{t<k}=0) \\ &= nc\sum_{k=a}^{b}\frac{k}{\varepsilon}\gamma\left(1-e^{-\frac{\lambda}{\varepsilon}}\right)e^{-\frac{\lambda(k-1)}{\varepsilon}} \end{aligned} \tag{3}$$

As before $\left(1-e^{-\frac{\lambda}{\varepsilon}}\right)e^{-\frac{\lambda(k-1)}{\varepsilon}}$ is the likelihood of a failure in minute k. Thus, every minute multiplied with the cost of n machines with cost c times relative discount $\gamma\frac{k}{\varepsilon}$ for minute k leads to the cost of the particular minute, and accordingly the sum over all the minutes in the time interval to the total cost of the interval. Given the expected cost for a failure in a specific time interval, the overall expected cost can be calculated as:

$$\begin{aligned} C = & \underbrace{\sum_{i=1}^{\infty}P\left(X^{t\le\frac{R}{n}}>0\right)^{i}EIC\left(1,\frac{R}{n}\varepsilon\right)}_{(i)}+ \\ & \underbrace{EIC\left(\frac{R}{n}\varepsilon+1,\varepsilon\right)}_{(ii)}+\underbrace{ncP\left(X^{\frac{R}{n}<t\le\varepsilon}=0\right)}_{(iii)} \\ = & \underbrace{\sum_{i=1}^{\infty}\left(\left(1-e^{-\frac{\lambda R}{n}}\right)^{i}\right)EIC\left(1,\frac{R}{n}\varepsilon\right)}_{(i)}+ \\ & \underbrace{EIC\left(\frac{R}{n}\varepsilon+1,\varepsilon\right)}_{(ii)}+\underbrace{nce^{-\lambda\left(1-\frac{R}{n}\right)}}_{(iii)} \end{aligned} \tag{4}$$

Here (i) is the expected cost of all failed attempts, (ii) is the expected cost if the task completes but a failure occurs before the end of the billing interval, and finally (iii) is the full cost for the billing interval times the likelihood that no failure occurs before the end of the billing interval.

Maybe surprisingly, it is possible to derive a closed-form solution and the first derivative of equation 4, but it is lengthy and thus, we omit to show it here. Instead, we discuss a special case of the parameter space, partially used hours after failures are free (i.e, $\gamma = 0$), to demonstrate that using more machines than necessary to finish in one billing interval can be cheaper. Note, that this special case, is also Amazon's current model. After our discussion, we again generalize the market model and explain why using more machines is also beneficial with any other discount (as long as a discount exists).

Theorem 5. *With a discount of 100% ($\gamma = 0$), the optimal number of machines for the re-execution strategy is:*

$$n_{\gamma=0} = max(\lambda R, R) \cdot m$$

In this formula, m are the number of machines require to finish R in one billing interval. Thus, m is a simple scale factor. To prove this theorem, we again consider equation 4 and for simplicity assume that R can be executed in a single time interval (e.g., $m = 1$). With $\gamma = 0$ the EIC from equation 3 is always zero and thus, the whole equation reduces to the cost that we have to pay for a successful task times the likelihood that we have to pay (i.e., the likelihood of no failure):

$$C_{\gamma=0} = nce^{-\lambda\left(1-\frac{R}{n}\right)} \tag{5}$$

Deriving the first derivative of $C_{\gamma=0}$ yields:

$$C'_{\gamma=0} = ce^{-\lambda\left(1-\frac{R}{n}\right)} - \frac{c\lambda Re^{-\lambda\left(1-\frac{R}{n}\right)}}{n} \tag{6}$$

Given that, the minimum is $n = \lambda R$. It should be noted, that we require the *maximum* from Theorem 5 as the equation does not model the effect of tasks running longer than one billing interval. This is also not necessary as Proposition 3 already showed, that it is never beneficial to run over more than one billing interval.

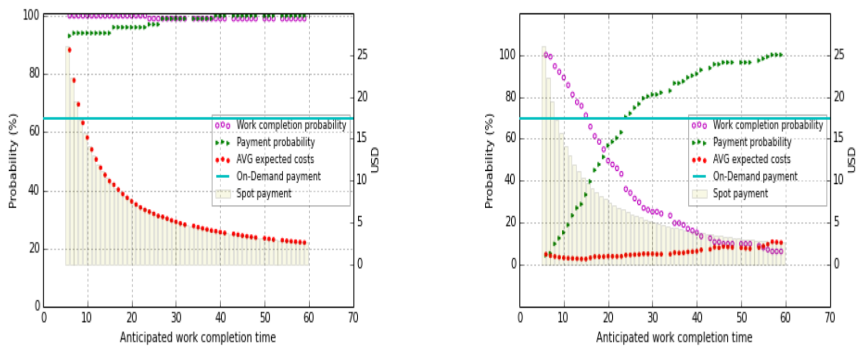
In the more general case of an arbitrary discount, the function in equation 4 remains convex and, depending on the failure rate and the discount, using more machines than minimally necessary to finish in a single billing interval can still be beneficial. In fact, the impact of parts (i) and (ii) of equation 4 on the optimum is very limited. That is due to the fact, that these cases are rather unlikely and thus, do not play an important role for the optimum.

4.2.1 The Optimization Goal

The last step of determining the optimum deployment for the re-execution strategy now depends on finding the optimal machine type. Again, we first consider the case with a discount of 100% ($\gamma = 0$). Using the optimum number of machines from Theorem 5 in cost equation 5 yields to the following optimization goal:

$$\underset{s}{\text{minimize}} \quad c \cdot min\left(\lambda Re^{1-\lambda}, R\right) \tag{7}$$

Looking closely at this minimization goal, and assuming a linear relation between the price of a machine and its failure rate (i.e., that the market values machines with a lower failure



(a) (b)

Fig. 2: Failure rate (λ) influence on work completion and payment probability, average expected cost, what we have to pay when we have to pay, on-demand cost, spot cost

rate), we can conclude that in an arbitrage-free market (i.e., a market where a more powerful machine is not cheaper), instance types with a higher failure rate will always reduce the cost and will potentially get the work done faster as we reserve more machines because of $n = \lambda R$. A surprising artifact created by the discounts for partially used hours. However, it should be noted, that our model does not consider the cost of starting the cluster and the software components (e.g., the services of the analytical system).

Similarly to the minimization goal of equation 7 , it is possible to define the optimization goal for market places with a discount of $0 < \gamma < 1$. However, due to space constraints we omit further details. Furthermore, we omit to consider $T_{startup}$ as part of the optimization because of space constraints (note, that $T_{startup}$ can be considered as a penalty function and easily be integrated).

4.2.2 Experimental Validation

While the goal of this paper are mainly the theoretical results, we also validated our findings using price traces from Amazon Spot Instance market. First, we analyzed the spot markets if there are actually really instance types which have on average more than one failure per hour (i.e., $\lambda > 1$). We calculated the statistics using a 5 day sliding window over 4 years of traces for every machine instance type (e.g., every data center, with every OS, etc) and different bid-prices. Not surprisingly, we found all types of failure rates (see also Section 2.2)

Here we show the results for the *us-east-1c–m1.large–Linux* instance type with on-demand price of \$0.175 and a bid price of \$0.0263 (15% of on-demand price) during two different time intervals $2012-09-12$ until $2012-09-17$ (Figure 2a), which had a lambda of $\lambda = 0.075$, and $2013-09-02$ until $2013-09-07$ (Figure 2b), which had a lambda of $\lambda = 1.8$. The former means, we had only 9 failures in total during the interval, whereas the latter refers to one failure every 30 minutes. We assumed a rather large task, which requires 101 instances to finish in 59 minutes (while 1057 machines are necessary to finish in 6 minutes).

We simulated the job execution using real Amazon spot traces from [Ja11] in every hour during the above mentioned interval. Figure 2a shows the that for the case with the low *out-of-bid* induced failure rate, using fewer machines is beneficial. That is, the work completion probability (the likelihood that the job finishes in the billing interval) stays almost constant independent on number of instances used. However, the probability that we have to pay (recall that Amazon has a 100% discount for failures), is also almost constant and only slightly drops with a lot of machines (e.g., to 95% with a 6 minuted anticipated completion time on 1057 machines). That is in almost all cases, the user would have to pay for the usage of the machine. As a result, using fewer machines to finish in exactly one billing interval (i.e., 60 minutes), has the lowest expected cost.

In contrast, when the failure rate is high the probability that the job completes quickly drops as fewer machines are used (see Figure 2b), while the likelihood that the user has to pay for a successful job increases. Looking at the expected cost reveals that there is a sweet spot at roughly 15 minutes execution time. However, according to our analyses the optimum should be at ≈ 30 minutes. The difference can be explained by the fact, that these are real world traces and that the failure likelihood heavily varies during the days. Furthermore, the difference in savings is only marginal between these two points. Finally, the figure shows that with such a high failure rate, it is actually a bad idea to use fewer machines: while the cost quickly decreases for the cases in which the user has to pay if no failures occurs (yellow bars in the background), the expected cost increases quickly as longer the computation takes.

4.3 Strategy 2: Checkpointing

The key, arguably counter-intuitive, insight the previous section made is that using more machines and thus, finishing a task earlier, can save money for analytical systems, that use re-execution to“overcome” failures, if the failure rate is high enough (more than 1 failure per billing interval). In this section, we study the optimal deployment strategy for systems that use check-pointing to reduce the failure recovery time, such as Hadoop. In Hadoop every output of a map-reduce job in a more complex in the query plan (i.e., data flow) is immediately check-pointed to HDFS. If a failure occurs, the checkpointed result is read from HDFS to continue. Our goal is not to model Hadoop’s recovery strategy in detail, in contrary we want to abstract from it to keep the presentation simple and make our model also applicable to other systems. The surprising result of our analysis is, that with check-pointing the optimal deployment strategy completely changes: instead of exploring parallelism at all, it is the best possible strategy to use a single machine as long as possible.

First, it should be noted, that systems like Hadoop normally consider isolated failures of single machines. Though failures induced by markets normally cause multiple machines to fail (see Section 3) simultaneously.

In the following, we assume that a check-pointing system is able to recover all machines in t_R minutes from a checkpoint. Furthermore, we assume, that the whole system can checkpoint an output in t_C minutes and that t_C is smaller than the shut-down warning time: $t_C < \omega$. Finally, we assume that the number of machines can be re-adjusted after every (system-wide) failure.

Effectively that means, that except for the recovery time, it does not matter how often a machine fails. If we assume for the moment, that we can not effect the failure rate of a job to run on a specific machine, the question becomes again how many machines we should use. Again, for simplicity lets also assume, that bin-packing does not pose a challenge (i.e., the job can be divided into equal pieces). In this case, we can derive the following result:

Proposition 6. *The expected cost of using n or* $2 \cdot n$ *machines for a job is the same with check-pointing:* $\frac{R}{n} \geq \varepsilon$

The reason is simple: in contrast to re-execution, after a failure the machine can start where the previous instances left it of. This is not completely accurate since results are often cached and not directly checkpointed to disk. However, the additional time for re-execution is often minimal and we further on ignore it in our model. Therefore, it is no more the question of how much a failed attempt cost, instead the question becomes how often do we need restart the process to finish a job. If we reserve an instance on the sport market, on average we have to pay the following price per single instance:

$$\begin{aligned} E(C) &= \underbrace{\sum_{k=1}^{\varepsilon} P(X^{t=k} > 0 \cap X^{t<k} = 0)\frac{k}{\varepsilon}\gamma c}_{(i)} + \underbrace{P(X^{t<\varepsilon} = 0)c}_{(ii)} \\ &= \underbrace{\sum_{k=1}^{\varepsilon} \left(1 - e^{-\lambda}\right) e^{-\lambda(k-1)} \frac{k}{\varepsilon}\gamma c}_{(i)} + \underbrace{e^{-\lambda}c}_{(ii)} \end{aligned} \tag{8}$$

That is, we calculate the likelihood of a failure in minute k times the cost for a failure during that minute for the entire billing interval (i). In addition, we add the cost in the case there is no failure (ii). A small reformulation of equation 8 is able to remove the sum and yields to:

$$= \frac{c\delta e^{\lambda(-\varepsilon)}\left(-e^{\lambda} - e^{\lambda}\varepsilon + e^{\lambda+\lambda\varepsilon} + \varepsilon\right)}{(e^{\lambda} - 1)\varepsilon} + ce^{-\lambda} \tag{9}$$

Similarly, the expected number of minutes a single instance runs is given as (assuming it is not shut-down manually):

$$\begin{aligned} E(T)^{\infty} &= \sum_{k=1}^{\infty} kP(X^{t=k} > 0 \cap X^{t<k} = 0) \\ &= \sum_{k=1}^{\infty} k\left(1 - e^{-\lambda}\right) e^{-\lambda(k-1)} \\ &= \frac{e^{\lambda}}{e^{\lambda} - 1} \end{aligned} \tag{10}$$

In contrast, if the machine will be shutdown manually at the end of the billing interval ε, we get:

$$
\begin{aligned}
E(T)^{\varepsilon} &= \sum_{k=1}^{\varepsilon} k\left(1-e^{-\lambda}\right)e^{-\lambda(k-1)} + e^{-\lambda}\varepsilon \\
&= \frac{\left(e^{\lambda}-1\right)e^{-\lambda}\varepsilon + e^{\lambda(-\varepsilon)}\left(-e^{\lambda}-e^{\lambda}\varepsilon+e^{\lambda+\lambda\varepsilon}+\varepsilon\right)}{e^{\lambda}-1}
\end{aligned} \tag{11}
$$

If the user plans to use n machines, we have to restart the machines (due to a failure) on average:

$$
\frac{\frac{R}{n}}{E(T)^{\infty}} = \frac{E(T)^{\infty}R}{n} \tag{12}
$$

and has thus to pay an average of:

$$
\left(\frac{\frac{R}{n}}{E(T)^{\varepsilon}}\right) nE(C) = E(T)^{\varepsilon}E(C)R \tag{13}
$$

That is we multiply the expected number of times we have to restart the machine, with the expected cost per machine times the number of concurrent machines n. As the average price in equation 13 does no longer depend on n our proposition holds. Note, that we use $E(T)^{\varepsilon}$ to account for the end of every billing interval. Also note, this multiplication is correct because of the special characteristics of the Poisson process: stationary and infinite divisible. Moreover, this equation only holds as long as $\frac{R}{n} \geq \varepsilon$.

Given these results we now know, that the expected cost of using one spot instance vs. using more instances to finish in a multiple of billing intervals is the same.However, there is still a significant difference between finishing in one or multiple billing-intervals: the risk the user has.

Theorem 7. *Using a single instance to finish a job in a single check-pointing interval is the cheapest and most risk-averse option.*

For example: lets assume the user decides to use 100 spot instances with a failure rate of $\lambda = 2$ and a cost of .1 to finish a job in exactly one billing interval (i.e., the machine might need to be restarted several times) and the discount is 100%. That is, usually — with his bid-price — he expects 2 failures per billing interval. According to our cost equation, this leads to an expected cost of $nE(C) = \$.1 \cdot e^{-2} = \$1.35335$. However, this is only the expected cost. Lets assume he got unlucky and the machines do not fail before the end of the billing interval. Now, he has suddenly to pay $n \cdot c = \$10$. A huge difference. In contrary, if he would have used a single instance for longer, the variance would have been much lower. This is do to the (strong) law of large numbers: If $\bar{C}_n = \frac{1}{n}(C_1 + C_2 + \ldots + C_n)$ and $E(C_1) = E(C_2) = \ldots = E(C_n)$ then for any positive number δ:

$$
\lim_{n\to\infty} P\left(\mid \bar{C}_n - E(C) \mid\right) > \delta\Big) = 0 \tag{14}
$$

That is the likelihood to diverge from the expected mean goes to zero. In other words, as more intervals *n* we use, the more we have to pay for individual intervals, and the more stable is our expected cost.

Finally, similarly to the previous section, we need to determine the optimal machine. Here the optimization goal is not much different than before, except for the new cost function. Again on Amazon the machine with the best ratio of high failure rate and powerful hardware will lead to the maximum savings.

4.3.1 Experimental Validation

Again, we evaluated our findings using real traces from Amazon spot markets from the last 4 years. We simulated two types of deployment for every single instance type on Amazon Spot markets: (1) one deployment in which we allocated 100 instances to finish a job in 1 hour, and (2) another deployment in which we allocated 1 instance for 100 hours. We assumed that we would always use the median of the prices from the 4 years as the bid-price (note, that the results hold for any other bid-price). We then calculated the cost that the job would have caused the user based on the traces.

In Table 1 we show the results for three machine types, *m2.2xlarge*, *m2.4xlarge*, and *m2.xlarge* all from the *us-east-1a* data center. We omitted other instance types because they showed similar behavior, unless their failure rate was very low in which case there was no difference.

	1 instance for 100 hours		100 instance for 1 hours	
	$\bar{\mu}$ ($)	$\bar{\sigma}$	$\bar{\mu}$	$\bar{\sigma}$
m2.2xlarge	9	12	17	15
m2.4xlarge	15	18	32	30
m2.xlarge	5	5	11	7

Tab. 1: Simulated execution time and variance for two different deployments on two machine types

The result show what we expected. The cost standard deviation of using 1 instance is lower than of using 100 instances. However, we also noticed that the average cost is lower when running 1 instance for 100 hours than 100 instances for 1 hour. The reason is, that the failure rate does not follow a perfect Poisson process. Instead failures sometimes come in bursts at Amazon, while during other periods almost always the full price has to be paid. However, if we use 1 instance for 100 hours it better reflects the expected cost as we cover more billing intervals with a single job.

4.4 Strategy 3: Lineage

As a last fault-tolerance strategy we analyse *lineage*-based recovery, as for example used by Spark [Za10]. Lineage-based recovery keeps track of the lineage of every (partial) intermediate state. If a machine fails, the system recovers the missing intermediate state by re-executing only the work of the lost state. Lineage-based recovery has the huge benefit, that it has much less overhead than check-pointing and does not require to re-execute the compete query (but only parts of it). Unfortunately, lineage-based recovery has a severe draw-back if used on spot instances:

Proposition 8. *Lineage-based recovery does not protect from failures if used on n instances with a single spot instance type and bid-price.*

The reason is simple: as explained in Sections 2 and 3 a price increase of the spot price in this case either causes all machines to fail simultaneously or none. Thus, lineage-based recovery does not provide any benefit over re-execution for failures induced by the market.

Still, there exists one possibility to profit from lineage-based recovery on spot markets: We can combine different instance types, potentially with different bid-prices, into one portfolio, that is used to deploy the system and this way ensure, that not all machines fail at the same time. The challenge here is, that failures on today's spot markets are often correlated (see the discussion in Section 2 and also Figure 1).

Thus, this scenario requires a completely new optimization goal: We want to find the optimal portfolio, so that the expected cost is minimized

$$\underset{w_i}{\text{minimize}} \quad E(C) = \sum_i \omega_i E \tag{15}$$

with the cost variance of:

$$\sigma_p^2 = \sum_i \sum_j \omega_i \omega_j \sigma_i \sigma_j \rho_{ij} \tag{16}$$

where ρ_{ij} is the correlation coefficient between the cost on instance types i and j and $\rho_{ij} = 1$ for $i = j$.

We explored this option on Amazon's current market and our initial results are not very promising, that this is indeed a feasible option. Thus, we omit further details. The main reason is, that lineage-based recovery models often come with a significant overhead to even recover a single machine. For instance, Spark showed a 30% overhead in a cluster of 75 nodes. That is, to recover 60 seconds of work on a single machine, the **entire** cluster with all remaining nodes spend roughly 20s.

4.4.1 Experimental Validation

To better understand how machines are correlated in a portfolio, we selected 14 instances within a single region and with the same Unix OS. We then built all possible portfolios of sizes 2 to 14 using the 14 selected instance types. For each of the portfolios we again used the 4 years of Amazon traces to determine per portfolio size the portfolio with the minimum number of failures. Afterwards, for the "best" portfolio per portfolio size we calculated the number of times 1 instance type was failing alone, 2 instance types were failing together, 3 instance types were failing together, etc. using again the 4 years of Amazon traces. Figure 3 shows the result of this simulation using the real-world traces. As can be seen, building portfolios can help with reducing the correlated failures, but it is not a way to entirely overcome them. Furthermore, even though the correlation figure in Section 2.2 shows that for some machines no correlation exists, this is typically only true for instance types in different data centers or with different operating systems. Yet, it is questionable how useful it is to combine different instance types from different data centers or with different operating systems into one portfolio. As a consequence we believe that at least on the current Amazon Spot market building portfolios to take advantage of lineage-based recovery is not feasible.

	Number of failed instances at the same time (%)										
Size	1	2	3	4	5	6	7	8	9	10	11
1	100.0										
2	65.1	34.9									
3	48.5	40.8	10.7								
4	48.7	33.3	16.3	1.7							
5	39.1	36.0	19.0	5.8	0.1						
6	29.6	34.5	20.9	12.0	3.0						
7	25.0	34.9	22.3	11.8	4.9	1.1					
8	21.8	36.1	23.6	12.0	5.4	1.1					
9	23.8	33.5	20.3	11.3	6.1	3.9	1.1				
10	17.6	24.3	23.1	16.3	9.2	4.9	3.7	0.9			
11	15.7	21.6	21.2	18.5	10.6	5.2	4.6	2.0	0.6		
12	11.9	20.7	20.6	17.5	12.2	6.7	4.9	3.4	1.7	0.4	
14	7.6	17.7	19.9	17.6	14.3	8.8	5.6	4.1	2.8	1.3	0.3

Fig. 3: Portfolio: The percentage of one isolated instance type failures, 2 correlated failures etc, for the best portfolio of size 2 to 14 over the last 4 years. For example, for a portfolio size of 3, the portfolio with the lowest failure rate had 48.5% of single instance failures, in 33.3% of the cases two instance types failed together, and in 10.7% of the cases all three.

5 Related Work

Spot prices and bidding strategies: Various attempts have been made to analyse Amazon's spot prices and to create predictive models [Ja11, Be11, Ma11, Ta12a, Vo12]. For instance, [Be11] analyzed the cloud market and came to the conclusion, it might not be a real market, *yet*. However, this is pure speculation and was neither confirmed nor disputed by Amazon. In [Ja11] the authors extensively analyze the Amazon spot market. Furthermore, the authors model the price changes using a mixture Gaussian model to predict the cost of a given task. Different approaches for bidding strategies have been proposed and evaluated [Ta12b, Ta14]. All this work is orthogonal to our results. We developed neither a new bidding strategy nor a (yet another) predictive model. Instead, we tried to make general recommendations on how to use spot instances for different fault-tolerance models.

Checkpointing, Hadoop and spot instances: Similarly, there has been work on tuning the check-pointing interval for Spot instances [Yi12, Yi10, Kh13, Yi10, An10, Vo12, Bi15]. In this paper, we only considered the simplest form of check-pointing for which every single intermediate result is immediately written to disk. Note, that this is also the model Hadoop uses. Looking at other check-pointing strategies (e.g., check-pointing the intermediate state every 30 seconds) as done in [Yi12, Yi10, Kh13] is future work. There has also been work on deploying MapReduce, specifically Apache Hadoop, on Amazon spot instances [Li11]. However, their focus was on the required modification of the Hadoop architecture (e.g., to enable full cluster recovery) not on optimizing the deployment itself. Chohan et al. [Ch10] used a Markov chain model to predict the expected lifetime of the spot instance and have used spot instances only as potential task accelerators to improve the runtime of MapReduce jobs. Conductor [Al12] formulates cloud services as well as the performance of many data-parallel distributed computations as linear dependencies and uses dynamic linear programming for determining the optimal plan for deploying MapReduce jobs. Qin Zheng [Zh10] proposes provisioning new redundant copies for MapReduce tasks to improve MapReduce fault tolerance in the cloud (using Amazon EBS) while reducing latency. SpotAdapt [Ka15] automatically adapts to spot price changes and tries to find a better redeployment. Again, our goal was much more general since we theoretically analyzed the most cost-effective deployment strategy for different classes of systems.

Fault-tolerance Schemes: There is no shortage on systems and different types of fault-tolerance schemes. Traditional database systems, like MySQL or Postgres, typically do not use any inter-query fault tolerance and require to re-execute the whole query. Fine-granular fault-tolerance schemes are typically found in modern analytical systems, such as Hadoop [Had], Dryad [Is07]) as well as in many stream processing engines [Hw05, Ta06]. While stream processing engines check-point the internal state of each operator for recovering continuous queries, MapReduce-based systems [De08] such as Hadoop [Had] typically materialize the output of each operator to handle mid-query failures. For being able to recover, they rely on the fact that the intermediate results are persistent even when a node in the cluster fails, which requires expensive replication and prevents support for latency-sensitive queries. Other systems, like Impala [Cl] and Shark [Xi13] store their intermediates in main-memory in order to better support short running latency-sensitive queries. Spark uses the idea of resilient distributed datasets [Za12], which store their lineage in order to enable re-computation instead of replicating intermediates. We believe, that we were able to address a wide-range of these models in our analyzes and that the results can (easily) be adopted for any variation used by specific implementations.

6 Conclusion

A major aspect of future cloud-based computing platforms will be service and pricing models based on economic market principles. The Amazon Spot Market uses a bidding-based pricing model and is a pioneering step in this direction. If used wisely, spot instances can make analytics in the cloud significantly cheaper. In this paper, theoretically analyzed how analytical systems with different fault-tolerance strategies are best deployed on a cloud market place, like Amazon Spot, to reduce the overall cost. The key findings are: (1) that for systems, that do not implement any fault tolerance and have to re-execute the query, using more machines can be cheaper, whereas (2) for systems, that use an intermediate check-pointing strategy, such as Hadoop, a single machine is the best option. In contrast, (3) lineage-based recovery, as for example used by Spark, does not help on cloud market places. In addition to the theoretical results, we also backed-up the results using real data traces from the Amazon Spot Instance market.

References

[Al12] Alexander, W. et al.: Orchestrating the Deployment of Computations in the Cloud with Conductor. In: Networked systems design and implementaion(NSDI). 2012.

[Am] Amazon: , Amazon Spot Instances. `http://aws.amazon.com/ec2/purchasing-options/spot-instances/`.

[An10] Andrzejak, A. et al.: Decision Model for Cloud Computing under SLA Constraints. In: Proceedings of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). S. 257 – 266, 8 2010.

[Be11] Ben-Yehuda, O.A. et al.: Deconstructing amazon EC2 spot instance pricing. In: Proceedings 3rd IEEE Int'l Conf. on Cloud Computing Technology and Science. 2011.

[Bi15] Binnig, Carsten et al.: Spotgres - parallel data analytics on Spot Instances. In: ICDE Workshops. S. 14–21, 2015.

[Bl73] Black, Fischer et al.: The Pricing of Options and Corporate Liabilities. Journal of Political Economy, 81(3):637–54, May-June 1973.

[Bu10] Bu, Y. et al.: HaLoop: efficient iterative data processing on large clusters. In: Proceedings of the VLDB Endowment. 9 2010.

[Ch10] Chohan, N. et al.: See Spot run: Using spot instances for mapreduce workflows. In: USENIX HotCloud. 2010.

[Cl] Impala, http://www.cloudera.com.

[De08] Dean, Jeffrey et al.: MapReduce: simplified data processing on large clusters. Commun. ACM, 51(1):107–113, 2008.

[Had] Apache Hadoop. `http://hadoop.apache.org/`.

[He11a] Herodotou, H. et al.: , No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics, 2011.

[He11b] Herodotou, H. et al.: Starfish: A Self-tuning System for Big Data Analytics. In: In CIDR. S. 261 – 272, 2011.

[Hw05] Hwang, Jeong-Hyon et al.: High-Availability Algorithms for Distributed Stream Processing. In: ICDE. S. 779–790, 2005.

[Is07] Isard, Michael et al.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys. S. 59–72, 2007.

[Ja11] Javadi, B. et al.: Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In: Utility and Cloud Computing (UCC) IEEE. S. 219 – 228, 12 2011.

[Ka15] Kaulakiene, Dalia et al.: SpotADAPT: Spot-Aware (re-)Deployment of Analytical Processing Tasks on Amazon EC2. In: DOLAP. S. 59–68, 2015.

[Kh13] Khatua, Sunirmal et al.: Application-Centric Resource Provisioning for Amazon EC2 Spot Instances. In: Euro-Par. S. 267–278, 2013.

[Li11] Liu, H.: Cutting MapReduce cost with spot market. In: USENIX Workshop on Hot Topics in Cloud Computing (HotCloud). 2011.

[Ma11] Mazzucco, M. et al.: Achieving Performance and Availability Guarantees with Spot Instances. In: HPCC. S. 296 – 303, 9 2011.

[Mu12] Murthy, M. K. M. et al.: Pricing models and pricing schemes of IaaS providers: a comparison study. In: Proceedings of the International Conference on Advances in Computing, Communications and Informatics. S. 143 – 147, 2012.

[Ta06] Tatbul, Nesime et al.: Load Management and High Availability in the Borealis Distributed Stream Processing Engine. In: GSN. S. 66–85, 2006.

[Ta12a] Tang, Sh. et al.: Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In: High Performance Computing and Communications (HPCC). S. 91 – 98, 6 2012.

[Ta12b] Tang, ShaoJie et al.: Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In: IEEE CLOUD. S. 91–98, 2012.

[Ta14] Tang, ShaoJie et al.: A Framework for Amazon EC2 Bidding Strategy under SLA Constraints. IEEE Trans. Parallel Distrib. Syst., 25(1):2–11, 2014.

[Vo12] Voorsluys, W. et al.: Reliable Provisioning of Spot Instances for Compute-intensive Applications. In: Advanced Information Networking and Applications (AINA). S. 542 – 549, 3 2012.

[Xi13] Xin, Reynold S. et al.: Shark: SQL and rich analytics at scale. In: SIGMOD Conference. S. 13–24, 2013.

[Yi10] Yi, Sangho et al.: Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In: IEEE CLOUD. S. 236–243, 2010.

[Yi12] Yi, Sangho et al.: Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. IEEE T. Services Computing, 5(4):512–524, 2012.

[Za10] Zaharia, M. et al.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 6 2010.

[Za12] Zaharia, Matei et al.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: NSDI. S. 15–28, 2012.

[Zh10] Zheng, Qin: Improving MapReduce fault tolerance in the cloud. In: Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). S. 1 – 6, 4 2010.