SpotWeb: Running Latency-sensitive Distributed Web Services on Transient Cloud Servers

Ahmed Ali-Eldin, Jonathan Westin, Bin Wang, Prateek Sharma*, Prashant Shenoy
UMass Amherst, *Indiana University
{ahmeda,jwestin,binwang,shenoy}@cs.umass.edu,*prateeks@iu.edu

ABSTRACT

Many cloud providers offer servers with transient availability at a reduced cost. These servers can be unilaterally revoked by the provider, usually after a warning period to the user. Until recently, it has been thought that these servers are not suitable to run latency-sensitive workloads due to their transient availability. In this paper, we introduce SpotWeb, a framework for running latency-sensitive web workloads on transient computing platforms while maintaining the Quality-of-Service (QoS) of the running applications. SpotWeb is based on three novel concepts; using multi-period optimization—a novel approach developed in finance—for server selection; transiency-aware load-balancing; and using intelligent capacity over-provisioning. We implement SpotWeb and evaluate its performance in both simulations and testbed experiments. Our results show that SpotWeb reduces costs by up to 50% compared to state-of-the-art solutions while being scalable to hundreds of cloud server configurations.

ACM Reference Format:
Ahmed Ali-Eldin, Jonathan Westin, Bin Wang, Prateek Sharma*, Prashant Shenoy. 2019. SpotWeb: Running Latency-sensitive Distributed Web Services on Transient Cloud Servers. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3325397

1 INTRODUCTION

Cloud computing has become a popular paradigm for a wide range of applications ranging from online web services to scientific workloads. Application providers find cloud platforms attractive for hosting their applications since they can request and relinquish resources on demand and use a pay-as-you-go model to pay for their usage. Since customer demand for cloud servers can fluctuate over time, cloud data centers are provisioned for peak demand. As a result, the average utilization of the data center tends to be low (~20% [19]). To improve utilization and revenues, cloud providers have begun to offer idle servers at significant discounts under a revocation model—these servers, referred to as *transient servers*, often cost 70-90% less than traditional on-demand servers, but may be unilaterally revoked when needed [37]. Transient servers increase cloud provider revenues while retaining the ability to reclaim these resources and offer them to higher priority customers.

Since transient servers incur a fraction of the cost of non-revocable cloud servers, they have become popular for running disruption tolerant applications such as scientific batch jobs or data intensive workloads [32, 34, 39, 40] using techniques such as checkpointing and live-migration to mitigate the impact of revocations [18, 35]. A common use case is to acquire hundreds of transient servers for distributed processing of a very large dataset (e.g. using Hadoop, Spark or distributed machine learning).

Due to their revocable nature, transient servers have not been considered suitable for running interactive and latency-sensitive applications such as web services and interactive data anlytics. Such applications have service level objectives (SLOs) that must be met, but transient server revocations can result in downtimes and SLO violations. In this paper, we ask an interesting research question: *is it feasible to reliably run latency-sensitive distributed applications on transient cloud servers despite their revocable nature and yet provide SLO guarantees?* We use clustered web services as a canonical representative of latency-sensitive distributed applications for this paper; however our approach is general and is applicable to other latency-sensitive distributed applications for scientific computing.

Our work builds on two recent research efforts in making transient servers more useful to applications. The recent ExoSphere work [33] proposed the notion of server portfolios—groups of different transient server types—and judicious portfolio server selection to reduce the chances of correlated server revocations; each revocation event only preempts a fraction of the servers in the portfolio, rather than all of them, enabling the distributed application to continue execution but with fewer resources. However, ExoSphere does not incorporate the notion of SLOs into its portfolio selection and is unaware of an application's SLO needs. Tributary [15], on the other hand, addresses this issue by performing SLO-aware selection of a mix of transient servers. Neither ExoSphere nor Tributary are explicitly designed for latency sensitive clustered web servers and hence do not exploit application semantics. Further, both efforts use past historical knowledge for transient server selection and do not fully exploit future knowledge or predictions, if available, to guide their decision making.

Our approach, which we call SpotWeb, is based on multi-period portfolio theory from the domain of finance and can incorporate both past knowledge and future predictions into decision making. Knowledge of future predictions can often be used to improve the decision making— for instance, knowledge of future workload changes or price changes over a time horizon can yield better overall decisions when choosing transient servers. Multi-period portfolio theory incorporates predictions over a finite future time horizon into its decisions. In our case, predictions of future workload

HPDC '19, June 22–29, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3325397

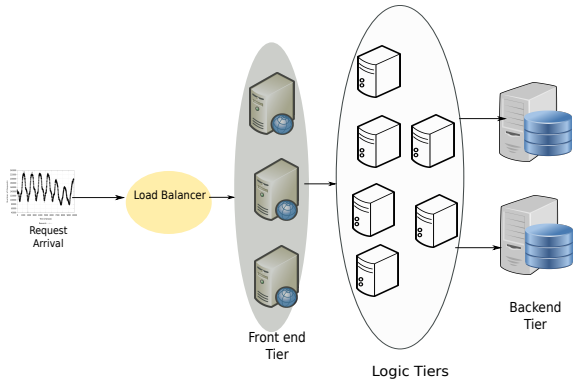


Figure 1: Multi-tier web architecture.

changes, price changes or future server demand can be incorporated into the server selection algorithm. Our approach combines SLO-aware provisioning of web servers with SLO-aware transient server selection to determine an appropriate selection of heterogeneous servers and the degree of over-provisioning. SpotWeb also includes a transiency-aware cluster load-balancer and prediction algorithms to mask the impact of server revocations on applications.

In designing and implementing SpotWeb, our paper makes the following contributions:

- We develop a new framework based on multi-period portfolio optimization, a novel approach from economics and finance, for running clustered web services on transient servers while enforcing SLOs, reducing provisioning costs, and decreasing latency.
- We present transiency-aware load-balancing and transiency-aware prediction algorithms and show how adding transiency-awareness to traditional load balancers and prediction algorithms is crucial for maintaining high QoS and low latency in clustered web services.
- We implement a full prototype of SpotWeb and deploy it on real cloud platforms. We open source our code and data to enable other researchers to build on our work.
- We evaluate SpotWeb extensively in a real testbed and in a simulated environment. We show how SpotWeb saves up to 50% of the costs compared to state-of-the-art techniques while reducing or completely eliminating SLO violations.

2 BACKGROUND

Transient Cloud Servers. Traditional cloud platforms have offered servers to customers on an on-demand basis—such servers can be requested by a customer at any time and are assigned to the customer until relinquished (i.e. they are non-revocable). More recently, cloud providers have begun to offer surplus server capacity (i.e. idling servers) in the form of transient servers. Transient servers are offered at deep discounts (70-90% cheaper than on-demand servers [14]) but are revocable by the cloud provider. Thus, when the demand for higher priced on-demand servers rises, the cloud provider can unilaterally preempt transient servers and offer them to higher paying customers. Typically the cloud providers offers an advance warning (e.g. 30s to 2min) to a transient server prior to terminating it; this advance warning allows for graceful saving of application state and application shutdown prior to server termination. Today, major cloud providers offer transient servers in the form of Amazon EC2 Spot Instances, Azure Low-priority Batch VMs, and Google Preemptible VMs.

Transient Server Characteristics. A typical cloud provider offers servers with many different hardware configurations (CPU cores, memory size, disk type and capacity, etc.). A cloud customer is expected to choose the appropriate configuration for their application. The number of different configurations continue to grow—Amazon EC2 offers over 160 different server types, while Google Cloud Platform offers more than 30 standard machine types, but also allows the user to custom-build a VM configuration. Most server configurations are offered in the form of on-demand and transient instances. The demand for different server configurations fluctuates over time based on variations in demand from customers for that server type. Thus, each cloud server configuration forms a *market* that sees varying demand. The revocation probability of a market is based on the demand dynamics of the market [2]. Previous work looked at the problem of bidding and capacity provisioning for clusters running on transient servers [8, 9, 23].

Multi-tier Web Servers and Elastic Scaling. Modern web applications employ a multi-tier architecture comprising of a front-end application tier containing the application logic, and a back-end data tier containing application data. The front-end application tier may also employ a microservices model where functionality is split into smaller microservices that collectively implement the font-end logic. Modern designs keep microservices stateless, to the extent possible, and persist application state (e.g. session state) as well as application data in the back-end tier, often using databases [25].

The web application is assumed to be clustered to scale application capacity—the front-end tier is replicated on multiple servers and a load balancer is used to distribute requests among front-end tier replicas [42] (see Figure 1). The load balancer is assumed to be capable of distributing the incoming requests across a heterogeneous front-end cluster via schemes like weighted round robin. The load balancer node is also assumed to run on a non-revocable cloud server. The back-end tier may be clustered as well (i.e. replicated or partitioned databases). For the purpose of this work, we assume that the back-end tier is well provisioned for peak and its capacity does not need to change dynamically. Since the back-end tier is responsible for storing data and state, we also assume that the back-end tier runs on non-revocable (i.e. on-demand) cloud servers. The front-end nodes on the other hand can run either on transient or on-demand servers and the servers can be heterogeneous.

Since web requests need to be serviced with low latencies to ensure good user-perceived performance, web applications specify these performance requirements in terms of Service Level Objectives (SLO). The SLO objectives may specify a target mean service time, an upper bound on the tail latency, a threshold on SLO violations, or other similar criteria. Web workloads are known to be dynamic in nature and exhibit fluctuations over multiple time scales—such as time of day effects and workload spikes [1, 5]. Since the performance SLO requirements need to be met in the presence of these fluctuations, there has been significant research on elastic scaling (also referred to as dynamic capacity provisioning) which involves varying the application (i.e. front-end tier) server capacity to respond to workload changes [12, 17, 28, 29]. Broadly, elastic scaling techniques can either be proactive, where predictive methods are

used to predict future peak workloads and capacity is provisioned proactively for these peaks, or reactive, where provisioning actions are taken in response to currently observed workload changes [30].

3 THE CASE FOR MULTI-PERIOD PORTFOLIO SELECTION

To reliably run a multi-tier web application on transient servers, our approach, SpotWeb, incorporates the following ideas:

Diversified Server Selection. Intelligent transient server selection is a key requirement. Intuitively, we should choose transient servers that belong to stable markets and have the least preemption probability (i.e. high MTTFs). At the same time, since cost is a consideration as well, the normalized cost (e.g. cost per request) should also be as low as possible. However, if we provision all front-end servers with a single transient server type, a revocation event can cause all front-end replicas to fail simultaneously, resulting in application downtime.

To avoid losing the entire cluster of front-end nodes. A second requirement is to employ diversification of the server pool by choosing transient servers of different types such that their demand is uncorrelated. In that case, revocations of one configuration will not impact other markets and the application will only lose a fraction of its nodes. The greater the diversification, the lower the risk of concurrent revocations. This idea was posed as a *portfolio selection* problem in ExoSphere [33], where a heterogeneous portfolio of servers was chosen to minimize risk-adjusted cost.

Judicious capacity over-provisioning. A clustered web application running on transient servers sees two types of dynamics. First, temporal changes in workload requires the resource allocation to change over time. Second, transient server revocations and price dynamics cause supply changes over time. While elastic scaling approaches have long studied the problem of handling workload dynamics, they do not address the issue of capacity changes caused by transiency dynamics. The two types of dynamics have similar effects—workload increases can cause the desired capacity to exceed provisioned capacity while revocations can cause the provisioned capacity to fall below the desired capacity—both resulting in SLO violations. Thus, intelligent capacity provisioning entails judiciously over-provisioning the chosen portfolio of servers so as to handle both workload spikes and revocation-driven capacity drops.

SLO-awareness. Since the overarching goal is to provide SLO guarantees, both server portfolio selection and provisioning server capacity for a chosen portfolio must be SLO-aware.

Exploiting Future Knowledge. Web workloads often exhibit temporal trends such as time of day effects or steady growth that are amenable to future predictions. Proactive autoscaling methods have employed future workload prediction techniques to forecast future workloads to auto provision capacity with the future prediction in mind. Portfolio selection methods for transient servers, on the other hand, have exploited past histories of prices and revocation rates to construct server portfolios that do not explicitly utilize future knowledge of workloads, or other factors, for decision making. *A key insight of this work is that future predictions of workload, price, or failure rates, when available, should be exploited for choosing SLO-aware portfolios and provisioning servers for the chosen portfolio mix.*

Application-awareness. Transiency dynamics can typically be handled either at system-level or application-level. Prior work on batch workloads has demonstrated the benefits of making applications such as Spark distributed data processing transiency-aware [32]. Building on this insight, we argue that latency-sensitive applications such as multi-tier web applications also benefit from awareness of transiency dynamics. In particular, the load balancer for the clustered front-end tier should be aware of revocation of servers within the cluster, and should mitigate the impact of such revocations on user requests/sessions.

3.1 Limitations of Prior Approaches

There are a number of prior research efforts that address various facets of the above requirements, but each has shortcomings necessitating a new approach. The significant body of research on predictive (aka proactive) autoscaling has exploited future workload knowledge for decision making [12, 17, 41], but autoscaling research only addresses workload dynamics and does not address transiency dynamics or the issue of cloud server selection [30].

The ExoSphere effort proposed the notion of server portfolios and mitigating revocation risk via diversification (and use of heterogeneous mixes to avoid concurrent revocations). ExoSphere was however designed for batch-style distributed applications such as Spark and is not SLO-aware—a key need for interactive web services. Whereas ExoSphere targets short-lived applications (running time of a few hours) and provides them with static portfolios, web applications are continuously running, and require *dynamic* portfolios. Moreover, ExoSphere is purely backward-looking, and does not attempt to forecast application load and server prices, which is essential in the context of long-running applications.

More recently, Tributary builds on ExoSphere portfolio notion and proposes a method for SLO-aware portfolio selection with scaling. While Tributary is the closest to our SpotWeb work, it is not designed specifically for multi-tier web applications and thus does not incorporate or exploit application-awareness for transiency dynamics. Furthermore, Tributary's server selection policy's time complexity is exponential, making it unsuitable for large public clouds that have a large number of markets. Tributary aims to predict future preemptions by predicting the market price of an instance, it does so with the aim of making use of markets that are about to fail to get "free" resources. This is based on AWS's obsolete cost model which was to refund the user if the spot market price of an instance exceeds the bid price during its first hour. The predictions are nevertheless not used to calculate the optimal diversification in terms of per-request costs, failure costs, or SLO violation costs. Both approaches use past histories of prices and revocations to make an intelligent decision for the next time step—they do not use a longer time horizon of the future for decision making.

Finally Qu's work uses [29] a user-specified threshold for the number of concurrent failures to determine the degree of over-provisioning of heterogeneous transient servers. Tributary as well as our approach only requires SLOs and determining the failure probabilities as part of the decision making. Table 1 provides a comparison of various approaches.

	ExoSphere	Tributary	Qu et al.	**SpotWeb**
Heterogeneous Servers	Yes	Yes	Yes	Yes
SLO-awareness	No	Yes	Indirect	Yes
Auto-scaling	No	Yes	Yes	Yes
Exploit Future Forecast	No	Partially	No	Yes
Latency-aware provisioning	No	No	Yes	Yes

Table 1: Comparison between different approaches

3.2 SpotWeb System Overview

The SpotWeb approach meets the above requirements and addresses the limitations of prior work by combining three key ideas: (i) multi-period optimization of portfolio selection over future horizons, (ii) a transiency-aware load balancing algorithm, and (iii) a set of predictors to predict the dynamics in the system.

Server portfolio selection is based on the problem of choosing a financial portfolio comprising of various asset classes such as stocks, bonds, etc. so as to maximize risk-adjusted returns. The key insight is that choosing an uncorrelated mix of assets can insulate the overall portfolio when one asset class (e.g. stocks) experiences large declines. ExoSphere [33] leverages a well known optimization approach from economics and finance called Modern Portfolio Theory [22], first proposed in 1952, for choosing an optimal portfolio of transient servers. However, such a "single period" portfolio selection approach uses past histories of prices, and price correlations across markets when choosing a mix. Economists have observed that rebalancing the portfolio periodically (as asset prices change) incurs transaction costs and can lead to suboptimal decision making. For instance, if a forecast calls for a decline in one asset class (e.g. stocks) since it is overvalued, such future knowledge should be exploited for better decision making.

The same insight applies to server portfolio selection as illustrated by the following example.

Example 1: Consider a web application that has a choice of two server types—a small server that can service 10 req/s and a large server that can service 100 req/s. Let the costs of these servers be 2 c/hour and 15 c/hour respectively. Suppose that the web application is initially provisioned using two small servers and the incoming request rate for the next hour is 25 req/s. Single-period optimization will then provision a third small server yielding a capacity of 30 requests. However, if workload forecasts show that a prediction of 25 for the next hour rising to 110 requests for the following hour, a better choice would be to provision the third server as a large server since it yield a lower cost overall and also there are fewer "transactions" in terms of starting and stopping servers in the portfolio. Thus, any type of future knowledge over a finite time horizon can be useful for better decision making. This motivates the need for multi-period portfolio selection approach employed by SpotWeb.

Similarly application-awareness of transiency dynamics can help mitigate the impact of server revocations on SLO violations. In the case of multi-tier web applications, this can be done by making the load balancing algorithm transiency aware since transient server revocations include an advance warning (of 30-120s) from the cloud, passing this warning to the load balancing algorithm allow for a graceful "failover" of user sessions from the servers to other cluster servers with spare capacity. Since the system is over-provisioned for both peak workload and server revocations, transient revocations may often occur at times other than peak load duration, allowing for spare capacity on other servers to absorb load from revoked servers. Of course, revocation during peak load duration may lead to some SLO violations and can be mitigated by appropriate level of over-provisioning. Having provided the intuition for our approach we now present the design of SpotWeb.

Components Overview Our SpotWeb system has four main components that implement the above key ideas (see Figure 2). The four components are:

(1) **Transiency-aware Load Balancer.** Load balancers are a crucial component of any web system that directly affect the overall latency of the requests [42, 43]. Deploying web clusters on heterogeneous resources is challenging as most available load balancers do not necessarily handle resource heterogeneity well [13]. Running web-clusters on Spot instances adds even more complexity as resource heterogeneity will depend on the market mix and the workload dynamics.
(2) **Load Monitoring.** SpotWeb's optimization depends on accurate and up-to-date measurement data from the transient cloud. The monitoring system collects data on the failure probabilities, and the price changes of the different markets. In addition, the monitoring system polls data from the load balancer on the average and tail response times of the service requests, and on the request arrival rate.
(3) **Transiency-aware predictors.** SpotWeb uses three predictors to predict failure probability, per request price changes, and request arrival rates.
(4) **MPO Optimizer.** The central component of SpotWeb is the multi-period portfolio optimizer to calculate an optimal resource allocation that considers both past trends and future predictions to select a portfolio of servers. The optimizer uses the outputs from the predictors along with data from the monitoring system to choose the optimal portfolio allocation.

4 SPOTWEB DESIGN

To better understand how SpotWeb works, we discuss and explain how the optimizer, the load balancer, and the predictors are designed. The core of SpotWeb's functionality is the optimizer which is responsible for the server portfolio selection based on the system dynamics. In this Section, we start with an overview of modern portfolio theory, and then present SpotWeb's portfolio selection mechanism that exploits future knowledge over a time horizon. We then explain the load balancer and prediction algorithms, and how they integrate with the optimizer.

4.1 Overview of Portfolio Theory

Portfolio optimization theory was pioneered by Markowitz in his seminal paper published in 1952, where he formulated the choice of an investment portfolio as an optimization problem balancing off risks and returns [4, 22]. The theory has since evolved and found applications in many domains including computer science [16, 38] where it has been used to solve problems related to scheduling in

scientific workflows [6], transient computing [33], and industrial workflows [20].

Single Point Portfolio Optimization. The classical portfolio optimization problem – used by ExoSphere – seeks to select a portfolio of financial assets, such as stocks and bonds, in order to maximize risk-adjusted returns. The classical formulation of the problem seeks to optimize the selection over a single time period, and is thus referred to as Single Point portfolio Optimization (SPO). The SPO problem can be formulated as follows:

$$\text{Maximize } E\big[Return(t)\big] - \big(cost(t) + \alpha(Risk(t))\big), \qquad (1)$$

$E[Return]$ is the expected return from holding a portfolio for the trading period duration t, *cost* is the cost of owning and holding a portfolio plus any trading fees, *Risk* denotes the risk function which is an estimate of the variance of the return as a function of the return covariance of the different assets, and α denotes the *risk aversion parameter*. The risk aversion parameter quantifies the investor's attitude towards risk. There is a rich theory on how to choose and calculate α for financial portfolios [10]. Note that the optimization chooses an optimal portfolio for the next interval t based on current and past information, but does not use any predictions.

Since asset prices fluctuate over time, a given portfolio will become sub-optimal over a longer time horizon beyond the interval t, and will need to be adjusted. This can be achieved by re-running the optimization periodically; however doing so may yield a completely different selection of assets, resulting in significant churn and costs. Further, the approach does not incorporate any future knowledge even if available. Thus, if one asset class is overvalued and expected to decline in the future, such knowledge is not explicitly used when choosing the current portfolio.

Multi-period Portfolio Optimization (MPO) is an optimization technique that is designed to avoid the main drawbacks with SPO [24]. In multi-period portfolio optimization, future predictions of the market dynamics are used to select a portfolio. Instead of solving the portfolio selection problem for time-window t as done by SPO, MPO solves the portfolio selection problem over a *planning horizon*, H, using future prediction values of various time-dependant variables. In other words, we solve the portfolio selection problem for time intervals: $t, t+1, t+2, \ldots, t+H-1$. By doing so, MPO aims to reduce churn in the allocation while maximizing the best future selection strategy.

MPO generalizes the SPO approach and calculates a portfolio for each time interval over a horizon H:

$$\text{Maximize } \sum_{\tau=t}^{t+H-1} E\big[Return(\tau)\big] - \big(cost(\tau) + \alpha(Risk(\tau))\big). \qquad (2)$$

Clearly, MPO depends on the prediction accuracy of the time-dependent variable of concern (e.g., prices). Since no predictor is entirely accurate, and since prediction errors tend to increase with the length of the prediction horizon, these errors propagate to the portfolio selection. It is thus important to use a predictor that has self-correcting capabilities (as discussed later in Section 4.3), and to limit error propagation while selecting a portfolio. Therefore, while all trades over the horizon H are computed, only the first interval portfolio allocation is actually executed to limit error propagation and allow the predictor to adjust to prediction errors. This is repeated for every interval of time τ.

4.2 Intelligent Transient Server Selection

We now turn our attention to how SpotWeb uses MPO theory to select a portfolio of servers. Consider a multi-tier web application that needs to be provisioned in a cost-effective fashion on a cloud platform. We assume that the load balancing node and the back-end tier are provisioned on non-revocable on-demand servers and adequately provisioned for worst-case peaks. The application tier, on the other hand, can be provisioned using an arbitrary mix of on-demand and transient servers of various configurations. The application specifies a Service Level Objective (SLO) in terms of a threshold on the high percentile of the service time and a per-request penalty P for any SLO violations. We assume that both the incoming workload to the application and the price of computing resources fluctuate over time. Our goal is to adequately provision capacity so as to handle workload dynamics and server revocation dynamics while meeting SLO guarantees and reducing provisioning costs.

Let λ_t denote the predicted peak request rate seen by the application in time interval t. Since workload forecasting is well-studied we assume any suitable workload predictor is used to forecast the future workload in each time interval t over the future planning horizon H.

To understand how SpotWeb provisions a multi-tier web application on transient servers, consider a cloud platform that offers different server configurations. Each server configuration is offered as a non-revocable on-demand server and a revocable transient server, yielding a total of $N = 2S$ choices. Each server configuration s_i can serve up to r^i requests with no SLA violations.

Let $price_t^i$ denote the price of a server configuration, $i \in N$, over time interval t. On-demand servers have fixed prices and $price_t^i$ for an on-demand server is constant. Some cloud providers offer fixed discounts for transient servers and $price_t^i$ will be constant for such servers. Other providers (e.g. EC2 spot servers) see fluctuations in prices. If a price predictor is available, then $price_t^i$ will vary over the time horizon H. If price prediction is unavailable, a fixed $price_t^i$ may be used (in which case the algorithm will optimize for workload and/or failure dynamics but not price dynamics).

Let f_t^i denote the mean expected revocation probability over each interval t. On-demand servers are non-revocable $f_t^i = \emptyset$ for such servers. For transient servers, Amazon publishes expected revocation probabilities using Spot Instances Advisor [1]. Tributary [15] uses an LSTM neural network model to predict f_t^i for the next interval $t+1$. Such models can be extended to make predictions over a time window $t+H$. We assume a covariance matrix M which capture pairwise covariance in revocation events.

The SpotWeb algorithm aims to choose a subset of server configuration from N choices (the portfolio) as well as the number of servers of each chosen configuration for each time interval t over the horizon H so as to:

- Handle the expected peak workload λ_t in each interval while minimizing penalty P from SLO violations.
- Minimizes the cloud costs incurred based on server prices P_t^i based on the risk the application is willing to tolerate.

[1] https://aws.amazon.com/ec2/spot/instance-advisor/

- Reduce the frequency of revocation as well as the probability of concurrent revocation across different configurations.

The algorithm employs multi-period optimization that is SLO-aware and optimizes risk-adjusted cost over a time horizon H. The objective is to optimize the risk-return trade-offs by deploying a number of servers n_t^i of server type s_i, such that, the total number of servers deployed is sufficient to serve all the request λ_t at any given time, with minimal or no SLA violations. The number of requests served by a server of type s_i is r_i.

The cost of provisioning is the cost associated with acquiring a certain server type to serve the workload. While we can use the server prices, $price_t^i$, we note that a server can be cheaper in price, but serve a limited number of requests per second due to its configuration. The price should therefore be adjusted to the servers' ability to serve requests. Let C_t^i represent the adjusted cost of service per request on a given server configuration s_i. The per request average cost of service can be calculated as $C_t^i = price_t^i \div r^i$.

As we use a weighted round robin algorithm for load-balancing the requests across the cluster, the fraction of the total number of requests allocated to all servers of type s_i is $A_t^i = n_t^i r_i / \lambda_t$. For an over-provisioned system, $\sum_i A_t^i > 1$, while for a system that is under-provisioned $\sum_i A_t^i 1 < 1$.

The cost of provisioning for the next time unit is thus,

$$\text{Cost of Provisioning}_t^i = A_t^i \lambda_{t+1}^p C_t^i = n_t^i \frac{r_i}{\lambda_t} \lambda_{t+1}^p C_t^i, \tag{3}$$

where λ_{t+1}^p is the predicted workload at time $t+1$. If the workload is static, $\lambda_t = \lambda_{t+1}^p$ and Equation 4.2 is the cost of renting the servers using the portfolio allocation.

In addition to the provisioning costs, we need to account for SLA violation costs. These costs occur due to; a) capacity shortages when the number of deployed servers is less than the actual demand due to a misprediction by the predictor, b) the percentage of requests L dropped when an instance is revoked because they were not migrated to another instance before the warning period expires. Capacity shortages due to mispredictions can be calculated a posteriori as $\lambda_{t+1} - \lambda_{t+1}^p$. As the cost is calculated a priori, we need to account for this value by keeping track of the mean-absolute-error over a window of some recent predictions.

The number of requests dropped due to the failure dynamics of transient resources depends on the percentage of long running requests that can not be migrated within the warning period when a transient failure occurs L, the number of requests running on the failing server $A_{t+1}^i \lambda_{t+1}$, and the probability that a server fails f_{t+1}^i. Therefore, the SLA violation costs are represented by ,

$$SLACost_t^i = \begin{cases} PA_{t+1}^i (f_{t+1}^i \lambda_{t+1} L + \lambda_{t+1} - \lambda_{t+1}^p), & \text{if } \lambda_{t+1} - \lambda_{t+1}^p > 0 \\ PA_t^i f_t^i \lambda_{t+1} L, & \text{otherwise} \end{cases}. \tag{4}$$

In this formulation, we assume that there is penalty associated with not serving a request due to lack of capacity needed at $t+1$, but no extra penalty (besides the provisioning costs) for having some extra capacity. While the model supports adding such a penalty, for most latency-critical workloads, under-provisioning is more costly compared to over-provisioning.

Finally, the risk associated with a certain portfolio can be represented in different ways (see [3] for example risk representations). A common way for representing a given portfolio's risk is the traditional quadratic risk measure, which we model as,

$$\text{Risk}_t^i = \alpha (A_t^i)^\top M A_t^i, \tag{5}$$

where α is the risk aversion parameter as discussed earlier, and M is the covariance matrix of pairwise market revocation events which can be inferred from the changes in the failure probability over time.

The optimization can then be formulated to,

$$\text{Maximize} \sum_{\tau=t}^{t+H-1} \text{E[Return]} - (\text{Cost of Provisioning}_\tau^i + \text{SLA Violations costs}_\tau^i + \text{Risk}_\tau^i) \tag{6}$$

such that,

$$A_t^i \geq 0, \tag{7}$$

$$\sum_i A_t^i \geq A_{Min}. \tag{8}$$

$$\sum_i A_t^i \leq A_{Max} \tag{9}$$

$$A_t^i \leq a_{Max} \tag{10}$$

A_{Min} is the minimum percentage of requests that need to be served without violating and SLA, A_{Max} is the maximum percentage of requests that can be served by an allocation when a spike occurs, and a_{max} is the maximum percentage of requests served by any server type. In other words, A_{Min} provides the application owner with the ability to allow for some under-provisioning, A_{Max} sets the maximum over-provisioning allowed in the system, and a_{Max} allows the application owner to influence the diversification by setting the maximum requests directed to any server configuration. If a_{Max} is set to 1, the system relies on the optimizer only to choose the diversification level of the portfolio. If some applications need to control the maximum allocation per server type, a_{Max} is set to the maximum fractional allocation.

We set $E[Return]$ to zero. As previously stated, $E[Return]$ is the expected return from holding a certain portfolio for the duration of a "trading period t". Setting $E[Return]$ to zero turns the main objective of SpotWeb's optimization to a cost minimization problem.

4.3 Intelligent over-provisioning

Since revocations and failures are sure to occur, SpotWeb needs to mitigate for these failures by over-provisioning the resources allocated in order to handle failures when they occur. Nevertheless, SpotWeb should ensure that the cost of running the cluster does not increase significantly. Similar to traditional scaling mechanisms that correct prediction error by adding extra capacity or "padding" [36], it is crucial for SpotWeb to over-provision the system intelligently depending on both the predicted workload, prediction errors, and server revocations.

Web workload prediction and auto-scaling has been well studied in the literature [1, 12, 36]. Recent work has shown that no single workload prediction algorithm is accurate for all workloads [17, 28]. SpotWeb's predictor is based on the workload prediction algorithm introduced in [1], where the authors used a combination of cubic spline regression, and outlier detection to predict web workloads.

The authors' algorithm nevertheless does not support multi-horizon predictions. Cubic splines are a popular form of non-linear regression [31]. A spline is a piece-wise polynomial where a number of base functions are connected between "knots" or joints, with the coefficients of each function fixed between the knots, and with the derivatives of the right and left piece-wise polynomials being equal at the connecting knots.

In SpotWeb, we base our predictor on [1] and extend it to support transiency, multi-period predictions, and handling larger workload spikes. We train a cubic spline function using a moving window of two weeks, predicting the workload for the next time unit. Spline predictors with a moving window are adequate for modeling and predicting repeating trends in the workload, e.g., predicting the diurnal patterns, but they provide a poor prediction for non-repetetive patterns such as spikes. For spike prediction, [1] uses an Auto-Regressive Model (AR) model with lag structure one. We found that their model is sufficient for predicting small spikes but underestimates the amount of over-provisioning required to handle server revocations or larger spikes.

To calculate the over-provisioning required, SpotWeb calculates the 99th% confidence interval around each prediction. The 99th% confidence interval is a range where the probability of a predicted value lying within that range is 0.99. It has two bounds, an upper bound, and a lower bound. The upper bound is then used to provide the over-provisioning required by SpotWeb, and to set the predicted required capacity in the MPO algorithm. We note that SpotWeb can integrate any other predictors out-of-the-box.[2]

4.4 Transiency-Aware Load-Balancer

Running web-clusters on Spot instances requires more complex load-balancers that are transiency-aware. Besides server utilization, server revocations and changes in the portfolio should dictate how the load-balancing is performed. Server revocations require the load-balancer to adapt quickly to failures, ceasing to send requests to the revoked servers, and possibly migrating workloads running on the revoked servers to other servers. SpotWeb's load balancer implements a novel *adaptive* weighted round robin (WRR) algorithm to handle transiency and dynamic heterogeneity.

The total server capacity is provisioned using the workload predictor and the multi-period portfolio optimizer to handle peak workloads and expected revocations in each interval. Whenever a revocation occurs, the cloud VM receives an advance revocation warning signal before termination. SpotWeb monitors such signals and relays them to the load balancing algorithm. The transiency-aware load-balancer exploits the warning period W for load redistribution and capacity reprovisioning.

On load redistribution, the load balancer migrates all user sessions on the revoked server to the remaining servers while directing new request to the non-revoked servers. Since capacity is overprovisioned, the non-revoked servers will usually have idle capacity to absorb the load from the revoked servers. In such cases, no SLO violation will occur, since the transiency aware load balancer seamlessly switch over all sessions within the revocation warning period W prior to the termination. Of course, this only occurs under the assumption that application servers or microservices running on each frontend node are stateless, which allows requests and sessions to move to a different node.

Note that such a seamless switch-over with no SLO violation may not always be feasible. For example, If the application sees peak workload, server utilization in the cluster will be high and the remaining servers may not have adequate headroom to absorb the load from revoked servers. Further, despite attempting to minimize concurrent revocation across server types (markets) such revocation may still occur (since predicted correlation may not always be accurate). Concurrent revocation further reduces cluster capacity and can cause SLO violation. When a revocation occurs, the load-balancer communicates the failure to the workload predictor. If there is not enough resources to handle these failures via redistribution, the predictor decides to reprovisions capacity, starting new VMs to handle the load on the revoked VMs. These VMs need to be started as quickly as possible in order to migrate the load of the revoked machines to these new machines. In some cases, the start-up time for the new servers can be longer than the revocation warning period. In such cases, the load-balancer acts as an admission controller, dropping or delaying requests that can not be served without overloading the running servers to protect the remaining servers from becoming overwhelmed.

When the MPO algorithm changes the portfolio, adding or removing new instance types to the mix, the load balancer needs to adapt to these changes. The newly added instance type(s) might have different capacities from the old instance types of the previous portfolio. On each new portfolio selection, the MPO algorithm updates the load balancer with the new portfolio, adapting the weights of the round robin algorithm to reflect the new portfolio allocation.

5 SPOTWEB IMPLEMENTATION

We now turn our attention to a more detailed description of SpotWeb's implementation. As already mentioned, SpotWeb's code and data are open-source [3].

5.1 Architecture Overview

SpotWeb's system architecture is shown in Figure 2 with its four main components (the yellow boxes). The main user-facing component is the load-balancer. Similar to any web cluster, user requests are directed to the load balancer to decide which server in the cluster should serve that request. From a user-perspective, SpotWeb's load-balancer behaves similar to any traditional load-balancer, concealing the transiency behavior from the user.

The load-balancer also collects application level monitoring data, monitoring the response time distribution, the request arrival rate, the system throughput, the queue lengths of the servers, and the dropped request rate. This data is exposed via a REST interface to the workload predictor, and is used to predict the future workload. Similarly, SpotWeb's System Monitoring component monitors vital system metrics, namely, the market prices, and the revocation probability of a server, and feeds this data to the price and failure predictors. We have found that for almost all markets, there is no, to very little dynamics, in the revocation probability. The failure

[2]For example, our implementation of other predictors: https://github.com/ahmedaley/Autoscalers

[3]Code repo: https://github.com/ahmedaley/SpotWeb

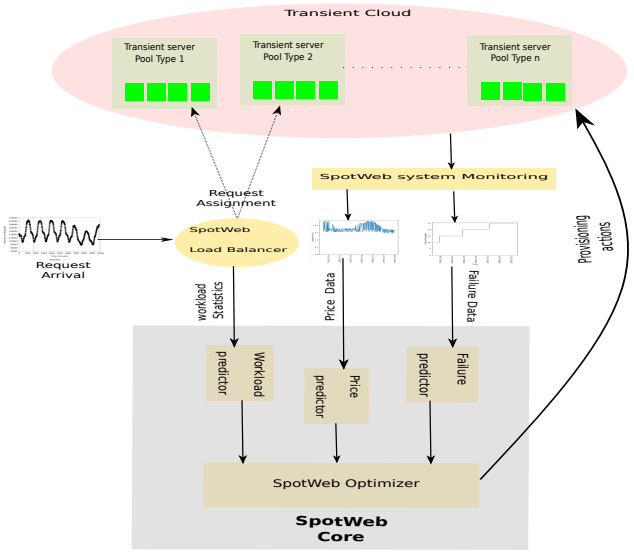


Figure 2: SpotWeb System Architecture.

predictions in our experiments are thus done reactively, i.e., we assume that for the next time unit, the failure probability will be equal to the measured probability now.

SpotWeb's optimizer runs periodically to compute an optimal portfolio of servers. When the optimizer runs, it polls the predictors, to get new predictions for the future request arrival rates, failure rates, and the future per request price. Once a new portfolio is calculated, the optimizer starts the new machines and updates the load-balancer with the new portfolio. As we show later (see Figure 7(b)), SpotWeb's optimizer is scalable, requiring subseconds to 5 seconds to calculate a portfolio, thus allowing us to run the optimizer frequently. That being said, there are reasons to run the optimizer on longer time-scales. First, starting a transient server and initializing the application can take seconds to minutes depending on the time taken to get the spot VM, but also to load the application and reconfigure it to join the cluster. Second, while most cloud providers have moved to a per-second billing model, some cloud providers, such as Microsoft Azure, support only hourly billing.

5.2 Implementation Considerations

SpotWeb is implemented in Python and R. The design is modular allowing the user to use either all the components of SpotWeb, or some of them with, e.g., different predictors or load-balancers.

SpotWeb Optimizer is the main component that implements the MPO logic described in Section 4.2. It is implemented in Python 2.7 with more than 1500 LOCs, and is based on CVX-Portfolio [3]. The optimizer is designed to enable users to easily integrate new costs models, constraints, and risks functions. For solving the optimization problem, we use CVXPY [7] with the SCS solver [26, 27]. The optimizer runs periodically with a configurable parameter for the period length. On initialization, the user specifies the look-ahead period length, and the constraints to be used,

SpotWeb's Load-Balancer To implement SpotWeb's load balancer, we have modified HAProxy, a widely used open-source load-balancer, to enable transiency awareness. We chose HAProxy as it works as both a level-4 (TCP) and a level-7 (HTTP) load-balancer. In addition, HAProxy implements weighted round robin as one of four main load balancing algortihms. It does not nevertheless provide an interface to change the weights for the different server types online. We have thus implemented a wrapper around HAProxy's Weighted Round Robin Algorithm changing the weights online. The wrapper provides a REST interface that gets called by the optimizer to reset the weights after each new portfolio is calculated. The weights are set to be equal to the relative weight of a market within the portfolio. HAProxy provides halog, a tool for reporting service statistics. We use this tool to get updates monitoring data by exposing the tool via a REST interface that can be called by the workload predictor to get up to date performance statistics, in addition to the request arrival rate. The load balancer changes are implemented in Python and Kotlin in a total of around 300 LOCs, and are wrapped in a Docker container.

SpotWeb's system monitoring. The system monitoring component keeps track of all price changes, failure probability changes, and the warning periods for any revoked machines. On a revocation warning, the monitoring system forwards it to the Load balancer. In addition to data collection, the system monitoring component also does some data cleaning and conversions. For example, as the AWS EC2 posted prices are posted per machine type, the system monitoring component calculates the per-request price. The time-series for failure probability and prices are polled by the predictors periodically. We implemented the system monitoring component for EC2 in Python in around 300 LOCs.

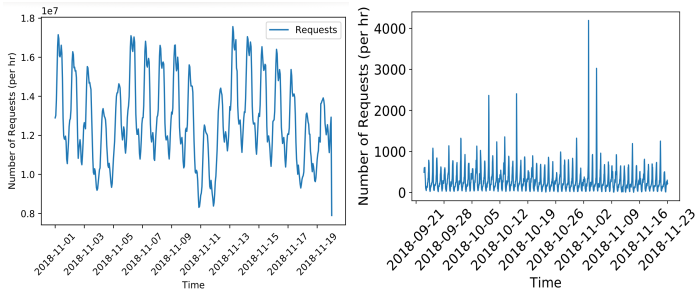
SpotWeb's predictors The predictors are implemented in both R and Python 2.7, using RPY2 for interfacing. The predictors are mainly used by the MPO algorithm to predict future request arrival rates, future price dynamics, and future revocation probability.

For cost and arrival rate predictions, we mainly use the predictor described in Section 4.3. Our implementation is around 200 LOCs for the predictor. In addition, we provide implementations of multiple state-of-the-art open sourced prediction algorithms that can be used instead of our predictor.

6 EXPERIMENTAL EVALUATION

For evaluating SpotWeb, we run both testbed experiments and extensive simulations with multiple workloads and SpotWeb configurations. The testbed experiments run on Amazon's EC2 spot markets using up to 36 spot markets. For the simulations, we develop a discrete-event simulator in Python which enables us to test SpotWeb more extensively. We run testbed experiments to evaluate latency related performance of SpotWeb, while we use simulations to evaluate long-term cost savings for large-scale clusters with many markets, and large numbers of instances.

Workloads. Figure 3 shows the workload traces we use in our experiments. The first workload is a 3 weeks trace of the request arrival rate on the English Wikipedia for the first 3 weeks of June 2008 (Figure 3(a)). The second trace is a 3 weeks long trace from TV4, a major Swedish Video-on-Demand (VoD) service provider, detailing the requests issued by the premium service subscribers to TV4's VoD service in January, 2013. The first workload has very few spikes, while the second one has multiple, hard to predict spikes. As



(a) Three weeks trace from Wikipedia (Time in hours)

(b) Traces from a major European Video-on-Demand provider (Time in hours).

Figure 3: Workload traces used in our experiments.

the revocation probability and the price per request data are recent, we assume the same time period for all three, i.e., November 2018.

Testbed Application. For our testbed experiments, we assume that the web cluster is a wiki cluster running the Mediawiki software and hosting the whole German Wikipedia which we replicate. MediaWiki is a custom-made, free and open-source wiki software platform written in PHP and JavaScript. Mediawiki is a traditional LAMP stack software, running on Linux, Apache web-server, deploying a MySQL database to store the data, and written in PHP. LAMP stack based software are prevalent in today's web [21]. Our setup also uses Memcached [11] for caching database objects in memory. Since Wikipedia in general receives far more read requests to articles than edit or write requests, making the workload read heavy, we replicate both the database and the Mediawiki software in each virtual machine, and experiment only with read requests.

Baseline comparisons. In our experiments we mainly take ExoSphere (using SPO) as our baseline for comparison. As noted by the authors of Tributary, Tributary's extra savings over ExoSphere are mostly due to making use of "free-hours", a pricing model that no longer exists in any major cloud provider, and since Tributary's does not scale beyond a very small number of markets, we decided against comparing SpotWeb to Tributary.

SpotWeb's configuration. There are three main configurable parameters used in SpotWeb, namely, the penalty for delayed request P, the percentage of long running requests L, and the risk aversion parameter α. While we ran experiments with a large set of parameters, unless otherwise stated, the evaluation results here are for experiments where $P = 0.02$, $L = 0$, and $\alpha = 5$. P is set such that dropping a request carries a penalty of double the maximum cost to serve a request (which is 0.01 on the x1e.16xlarge market). If the penalty is set lower that the cost of serving a request, the optimizer might favor dropping all requests over serving them as this will be "cheaper". L is set to zero as our testbed application, Wikipedia, has an average response time of less than 0.5 seconds, much lower than the warning period. Finally, M is chosen based on correlation between the failure probabilities matrix.

6.1 Transiency-aware load balancing Efficacy

When no revocations occur, SpotWeb's load-balancer is identical to vanilla HAProxy. However, when revocations occur, the transiency-aware aspects of SpotWeb's load-balancer are activated. SpotWeb's load-balancer actions to revocations are based on the overall system utilization, and the expected load on the system. On revocation and receiving a warning, SpotWeb's load-balancer checks if;

(1) The overall system utilization is low or medium, and the load can be migrated to the other running instances with no degradation in the SLOs.
(2) The system utilization is high, and new instances *can* be started within the warning period, with the load migrated to the newly started instances.
(3) The system utilization is high, and new instances *can not* be started within the warning period. Load will be migrated to the other running instances, or dropped until the new instances are available.

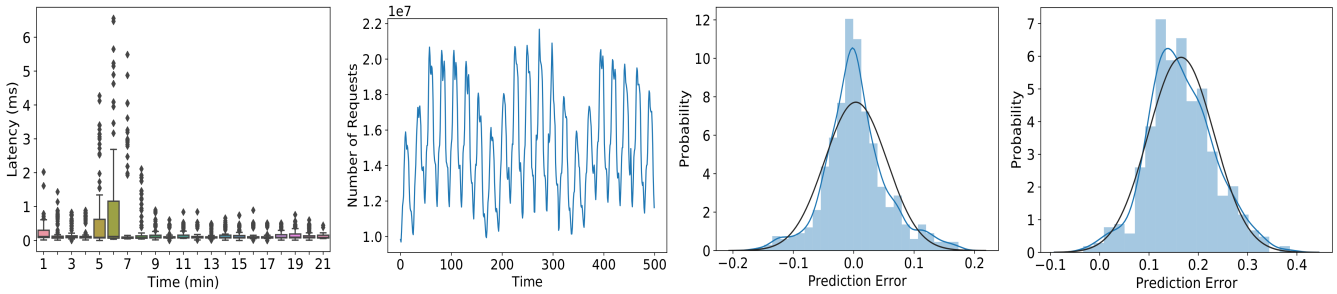
We have evaluated all of the different scenarios. As the first one is trivial and the last one is dependant on the time taken for the instance to start, we show results for the second scenario only.

In our first experiment, we run 6 machines on Amazon's US-east-1 datacenter, with two m4.xlarge, two m4.2xlarge, and two m2.4xlarge machines. The average utilization of the machines is kept between 70 and 95% with an average load of 600 requests/second. We induce correlated failures on the m4.2xlarge, and the m4.4xlarge machines 3 minutes into the experiment triggering SpotWeb's load-balancer to reactively start 4 new machines instead of the ones that we revoked. The new machines are started within the warning period.

Figure 4(a) shows a boxplot of the latencies reported at the load-balancer. The average response time is wells below 200 milliseconds during normal operation. Right after the revocation warning arrives at the 3rd minute, and four new machines are started. We measured the machine start-up time to be less 1 minute. Once the new machines are started, SpotWeb's load balancer migrates the load from the failing machines to the new ones. The new machines start with cold caches as Memcached is initialized with an empty cache. We have measured the cache warm-up period for one of the servers to be between 30 to 90 seconds. SpotWeb managed to bring down the 90%ile response time to less than 700 ms with no requests dropped. In contrast, unmodified HAProxy drops 85% of the requests within a few seconds after revocations, and the resulting average response time jumps to 2 seconds for the served requests.

6.2 Intelligent over-provisioning

In order to quantify how effective SpotWeb's padding is, we test the prediction accuracy using the approach used in [1] versus SpotWeb's predictions. Figure 4(b) shows a 3 weeks workload trace which we use to test the two approaches. We compare the two algorithms calculating the relative prediction error using the trace and plotting the error distribution histograms. Figure 4(d) shows the error distribution histogram using SpotWeb's predictor, while Figure 4(c) shows the same distribution for the algorithm in [1]. The positive error in the figure represents cases where the prediction algorithm over-provisions the resources, while the negative error is when it under-estimates the load (under provisions). We also plot a fitted normal distribution to the prediction errors. Clearly, SpotWeb tends to over-provision resources, reducing the under-provisioning error and adding enough padding for revocations. We found that SpotWeb on average over-provisions the resources by 15% compared to the actual needed capacity, with a maximum



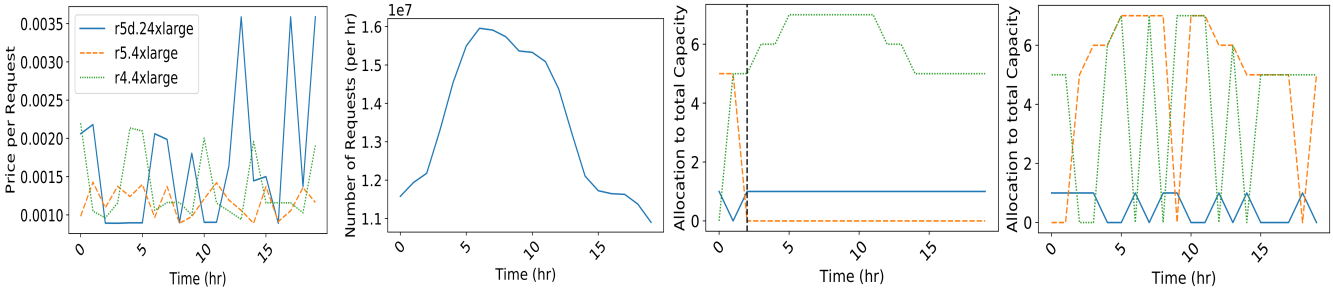
(a) Transiency aware load balancing rapidly reacts to revocations, which occur at around the 3 minute mark.

(b) Three weeks of the total workload on the English Wikipedia website (Time in hours).

(c) Prediction error relative to the current number of servers when predicting for 1 time unit ahead using [1].

(d) Prediction error relative to the current number of servers when predicting for 1 time unit ahead and using the upper limit of the 99% confidence interval as the prediction.

Figure 4: SpotWeb's transiency-aware load balancing, and intelligent over-provisioning together help eliminate SLO violations.



(a) The per-request price for each market.

(b) Zoomed-in Wikipedia workload.

(c) Constant portfolio with autoscaling.

(d) Portfolio chosen with MPO.

Figure 5: Having a constant portfolio with autoscaling fails to make use of price changes that occur in different markets

over-provisioning of 40% extra resource. On the other hand, the maximum under-provisioning error is less than 3.2% and it occurs for very brief periods, i.e., we never have capacity shortage of more than 3.2% while we have an SLO allowing for up to 5% of the requests to be served with some delay. We also note that our portfolio approach seldom chooses a portfolio with a probability of failure greater than 0.1. The prediction error using the algorithm in [1] is much worse, with the maximum under-provisioning error reaching 16.1% while the average and maximum over-provisioning being 0.03% and 17.3%, respectively. In addition to proactive padding, SpotWeb implements a reactive algorithm to handle any observed SLO violations that go beyond the predicted padding. Reactive provisioning involves requesting on-demand servers of one or more types within the chosen portfolio configuration to add additional capacity to the cluster for the remainder of the interval t.

6.3 Benefits of Price-awareness

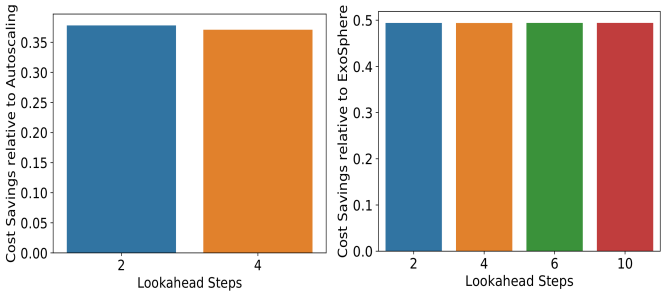
The ability to vary the portfolio mix over times enables Spotweb to exploit pricing changes over time which a fixed portfolio with autoscaling is unable to do. To show the benefit of price awareness, we compare Spotweb with ExoSphere and an auto-scaling algorithm that chooses a fixed portfolio and uses an auto-scaler to adjust the number of servers of each type. To show why it is important to recompute a new portfolio periodically, we run an experiment where we use three server markets on Amazon's US-east-1 datacenter, namely, r5d.24xlarge, r5.4xlarge, and r4.4xlarge markets, capable of serving 1920, 320, and 320 request per second respectively. We use pricing data for the three markets between 25 and 28, September, 2018 and assume that the all the servers have equal failure revocation probabilities of lower than 5%. Figure 5(a) shows the per request price dynamics for the three markets during the first 20 hours in this period. We can clearly see that the cheapest market changes with time.

We run an experiment where a fixed portfolio is chosen with an auto-scaler adjusting the number of servers, and compare it to SpotWeb. Figure 5(c) shows server allocation when a constant portfolio is set based on the market prices after 2 hours of running (marked with the vertical line) and with the allocation adjusted based on an oracle auto-scaler. Since at t=2 the r5d.24xlarge machine has the cheapest per request price, along with the r4.4xlarge, the autoscaler increases and decreases the allocation within these two markets only. The allocation by SpotWeb is shown in Figure 5(d), where we see that SpotWeb adjusts the portfolio according to pricing data, switching the portfolio allocation to cheaper markets.

Figure 6(a) shows the cost savings when using SpotWeb relative to using a constant portfolio with an auto-scaler with a short prediction horizon of 2, or a longer prediction horizon of 4. SpotWeb's cost is 37% lower than using a constant portfolio. We note that in this experiment we assumed an oracle predictor, thus this cost does not include any SLO costs, or any costs for over-provisioning.

6.4 Exploiting Future Workload Knowledge

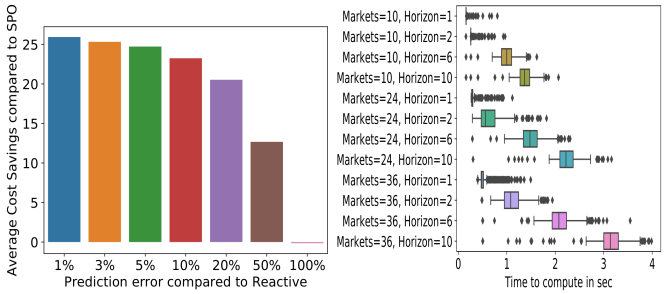
Now that we have established that simply using a constant portfolio with auto-scaling is not enough to provide optimal cost savings, we show why simply using ExoSphere in a loop, re-evaluating the portfolio in every time step based on the current load, and the price



(a) SpotWeb versus constant portfolio with an auto-scaler.

(b) SpotWeb versus running ExoSphere in a loop.

Figure 6: SpotWeb provides up to 50% cost savings compared to state-of-the-art techniques to handle transiency.



(a) SpotWeb's cost savings depend on the predictor accuracy.

(b) SpotWeb is highly scalable requiring less than 4 seconds to reach a new allocation.

Figure 7: Left: SpotWeb's sensitivity to predictor accuracy. Right: SpotWeb's scalability.

and failure hisotry. In this experiment, we choose the portfolio from 36 markets offered by Amazon's US-east-1 datacenter. We use pricing and revocation probability data that covers a period of 2 months. We note that these 36 markets cover most of the markets that mainly offer conventional x86 CPUs, i.e., no GPUs. For this experiment, we use the Wikipedia workload.

Figure 6(b) shows the cost saving of using SpotWeb with a look-ahead period of 2, 4, 6, and 10, versus using ExoSphere to calculate a new portfolio periodically. SpotWeb saves up to 50% of the costs compared to running ExoSphere in a loop. The Figure also shows other interesting aspects of SpotWeb. First, increasing the number of markets truly increases the expected savings from using SpotWeb compared to a limited number of markets. This result is repeated consistently across our experiments. As more choices are available with different market dynamics, future knowledge truly helps to decide the best portfolio. In our data, we saw that this is especially true when multiple markets appear to have the same price per request at one time point while having the same revocation probability, SpotWeb's predictions of future prices enables it to have a better "tie-breaking" mechanism than ExoSphere.

Second, as one can see in this Figure, in most of our experiments, we consistently saw that using longer look-ahead windows does not necessarily lead to better decision making. In most cases, the improvements where not significant or did not exist at all. As longer term predictions are usually less accurate than short term predictions, this result actually means that SpotWeb's savings are possible, even with simple, but accurate, future predictions.

When using the TV4 workload, the cost savings compared to ExoSphere where around 25% with similar behaviour, longer look-ahead horizons do not have a significant improvement over shorter ones. We omit the results due to lack of space.

Comparing ExoSphere's latency versus SpotWeb's latency in the testbed experiments, we saw that ExoSphere has on average 10 to 15% requests that are severely delayed (taking over 5 seconds on average) even when ExoSphere is integrated with our transiency aware load-balancer. On the occurance of large spikes, ExoSphere can have up to 25% of the requests dropped. SpotWeb on the other-hand maintain's a 99% response time below 1 second end-to-end. We omit the Figure due to lack of space.

6.5 Impact of Prediction Accuracy

The cost savings using SpotWeb depends on multiple factors, but one major factor is the prediction accuracy of the system dynamics. To quantify the effect of predictor accuracy, we run an experiment where we change the prediction error of the predictors relative to using a reactive predictor, i.e., relative to predicting that the workload, failure, and price for the next time step will be equal to the current values. Figure 7(a) shows the savings as a function of the prediction accuracy. As the prediction accuracy decreases, the savings decrease. Nevertheless, even when the error is large, there are still some significant savings using SpotWeb. To give some perspective, SpotWeb's predictor has a 3-5% prediction error.

6.6 Scalability of our Approach

One of the main drawbacks of Tributary is the computation tractability of the Acquire Manager component. On the other hand, SpotWeb's algorithm is highly scalable. SpotWeb's tractability is a function of the number of markets considered and the look-ahead horizon of MPO. To evaluate the scalability of SpotWeb, we ran multiple experiments with SpotWeb, varying the number of markets considered, and the length of look-ahead horizon, measuring the time required by SpotWeb to calculate the new optimized allocation. These experiments use the Wikipedia workload shown in Figure 3(a).

Figure 7(b) shows a box-plot of the time taken by SpotWeb to calculate an optimal new portfolio allocation with different number of markets and look-ahead horizons. Our results indicate that SpotWeb scales sub-linearly with increasing the number of markets, or increasing the look-ahead horizon. Doubling the number of markets does not lead to doubling the time needed to compute the optimal portfolio. This is one of the main features of SpotWeb, its ability to consider large numbers of markets compared to other current approaches.

7 DISCUSSION

When to use longer look-ahead. There are some cases where we found that the longer prediction horizon improved the performance of SpotWeb. The case where we saw the most savings is when the time it takes to start the new instance is longer than the period between two predictions. This situation can arise in transient servers when the cloud provider can not fulfill resource requests immediately due to resource pressure. This situation also occurs when the application running requires a longer start up time before it can

serve requests or when there is a cache warm-up period before the machine can meet the required SLOs. We omit these results due to space limitations.

Other Cloud providers. While our experiments are based on measurements from different AWS data centers, our results hold true for other cloud providers as our solution is not dependant on anything particular to AWS. For example, in the Google Cloud, while prices are constant, both the workload variations, and the probability of preemption – which varies between 0.05 and 0.15 – will lead to cost savings. In addition, since all instances are terminated after running for 24 hours on the Google Cloud, SpotWeb can utilize its transiency-aware load-balancer to relinquish the resources.

8 CONCLUSION

In this paper, we looked at how to run low-latency web-based applications on low-cost cloud transient servers. Our system, SpotWeb, is able to provide low-cost, SLO-aware server provisioning and autoscaling for web clusters. SpotWeb builds on and extends portfolio-based resource allocation, and uses multi-period portfolio optimization that can effectively use of traffic and price predictors. Deploying web applications on SpotWeb results in a cost savings of up to 90% compared to conventional on-demand cloud servers, and up to 50% when compared to state of the art transiency-specific systems.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. This research was supported by the Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196, NSF grants 1763834, 1802523, 1836752, and 1405826, and Amazon AWS cloud credits.

REFERENCES

[1] Ahmed Ali-Eldin et al. 2014. How will your workload look like in 6 years? analyzing wikimedia's workload. In *IEEE IC2E*. 349–354.
[2] Amazon. 2018. Spot Instance Interruptions. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html.
[3] Stephen Boyd et al. 2017. Multi-period trading via convex optimization. *Foundations and Trends® in Optimization* 3, 1 (2017), 1–76.
[4] Stephen Boyd, Mark T Mueller, Brendan O'Donoghue, Yang Wang, et al. 2014. Performance bounds and suboptimal policies for multi–period investment. *Foundations and Trends® in Optimization* 1, 1 (2014), 1–72.
[5] Giuliano Casale, Ningfang Mi, Ludmila Cherkasova, and Evgenia Smirni. 2008. How to parameterize models with bursty workloads. *ACM SIGMETRICS* 36, 2 (2008), 38–44.
[6] Kefeng Deng, Junqiang Song, Kaijun Ren, and Alexandru Iosup. 2013. Exploring portfolio scheduling for long-term execution of scientific workloads in IaaS clouds. In *SC*. IEEE, 1–12.
[7] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *JMLR* 17, 1 (2016), 2909–2913.
[8] Daniel J Dubois and Giuliano Casale. 2015. Autonomic provisioning and application mapping on spot cloud resources. In *ICCAC*. IEEE, 57–68.
[9] Daniel J Dubois and Giuliano Casale. 2016. OptiSpot: minimizing application deployment cost using spot cloud resources. *Cluster Computing* (2016), 1–17.
[10] Louis Eeckhoudt, Christian Gollier, and Harris Schlesinger. 2005. *Economic and financial decisions under risk*. Princeton University Press.
[11] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 124 (2004), 5.
[12] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. 2012. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM TOCS* 30, 4 (2012), 14.
[13] Anshul Gandhi, Xi Zhang, and Naman Mittal. 2015. HALO: Heterogeneity-Aware Load Balancing. In *IEEE MASCOTS*. 242–251.
[14] Weichao Guo, Kang Chen, Yongwei Wu, and Weimin Zheng. 2015. Bidding for highly available services with low price in spot instance market. In *ACM HPDC*. 191–202.
[15] Aaron Harlap et al. 2018. Tributary: Spot-dancing for elastic services with latency SLOs. In *USENIX ATC*.
[16] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. 1997. An economics approach to hard computational problems. *Science* 275, 5296 (1997), 51–54.
[17] Alexey Ilyushkin et al. 2018. An Experimental Performance Evaluation of Autoscalers for Complex Workflows. *ACM TOMPECS* 3, 2 (2018), 8.
[18] Qin Jia et al. 2016. Smart spot instances for the supercloud. In *Workshop on CrossCloud Infrastructures & Platforms*. ACM Press.
[19] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 158–169.
[20] Shenjun Ma, Alexey Ilyushkin, Alexander Stegehuis, and Alexandru Iosup. 2017. ANANKE: a Q-Learning-Based Portfolio Scheduler for Complex Industrial Workflows. In *ICAC*. IEEE, 227–232.
[21] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. 2010. Turning Down the LAMP: Software Specialisation for the Cloud. *HotCloud* (2010).
[22] Harry Markowitz. 1952. Portfolio selection. *The journal of finance* 7, 1 (1952), 77–91.
[23] Michele Mazzucco and Marlon Dumas. 2011. Achieving performance and availability guarantees with spot instances. In *HPCC*. IEEE.
[24] Jan Mossin. 1968. Optimal multiperiod portfolio policies. *The Journal of Business* 41, 2 (1968), 215–229.
[25] Sam Newman. 2015. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".
[26] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. 2016. Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding. *Journal of Optimization Theory and Applications* 169, 3 (June 2016), 1042–1068.
[27] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. 2017. SCS: Splitting Conic Solver, version 2.0.2. https://github.com/cvxgrp/scs.
[28] Alessandro Papadopoulos, Ahmed Ali-Eldin, Karl-Erik Årzén, Johan Tordsson, and Erik Elmroth. 2016. PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications. *ACM TOMPECS* 1, 4 (2016), 15.
[29] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. 2016. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *JNCA* 65 (2016), 167–180.
[30] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. 2018. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 73.
[31] Christian H Reinsch. 1967. Smoothing by spline functions. *Numerische mathematik* 10, 3 (1967), 177–183.
[32] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. 2016. Flint: batch-interactive data-intensive processing on transient servers. In *EuroSys*. ACM, 6.
[33] Prateek Sharma, David Irwin, and Prashant Shenoy. 2017. Portfolio-driven resource management for transient cloud servers. *ACM SIGMTERICS* (2017).
[34] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. 2015. Spotcheck: Designing a derivative iaas cloud on the spot market. In *EuroSys*. ACM, 16.
[35] Supreeth Shastri and David Irwin. 2017. HotSpot: automated server hopping in cloud spot markets. In *Symposium on Cloud Computing*. ACM Press, 493–505.
[36] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *ACM SoCC*.
[37] R. Singh, P. Sharma, D. Irwin, P. Shenoy, and K.K. Ramakrishnan. 2014. Here Today, Gone Tomorrow: Exploiting Transient Servers in Data Centers. *IEEE Internet Computing* 18, 4 (July/August 2014).
[38] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research* 32 (2008), 565–606.
[39] Fan Yang and Andrew A Chien. 2016. ZCCloud: Exploring wasted green power for high-performance computing. In *IPDPS*. IEEE, 1051–1060.
[40] Fan Yang and Andrew A Chien. 2018. Large-Scale and Extreme-Scale Computing with Stranded Green Power: Opportunities and Costs. *IEEE TPDS* 29, 5 (2018), 1103–1116.
[41] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. 2007. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *IEEE ICAC*.
[42] Qi Zhang, Alma Riska, Wei Sun, Evgenia Smirni, and Gianfranco Ciardo. 2005. Workload-aware load balancing for clustered web servers. *IEEE TPDS* 16, 3 (2005), 219–233.
[43] Tao Zhu et al. 2017. Limitations of load balancing mechanisms for n-tier systems in the presence of millibottlenecks. In *ICDCS*. IEEE, 1367–1377.