Adaptive, Model-driven Autoscaling for Cloud Applications

Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, Li Zhang
IBM Research

Abstract

Applications with a dynamic workload demand need access to a flexible infrastructure to meet performance guarantees and minimize resource costs. While cloud computing provides the elasticity to scale the infrastructure on demand, cloud service providers lack control and visibility of user space applications, making it difficult to accurately scale the underlying infrastructure. Thus, the burden of scaling falls on the user.

In this paper, we propose a new cloud service, Dependable Compute Cloud (DC2), that automatically scales the infrastructure to meet the user-specified performance requirements. DC2 employs Kalman filtering to automatically learn the (possibly changing) system parameters for each application, allowing it to proactively scale the infrastructure to meet performance guarantees. DC2 is designed for the cloud - it is application-agnostic and does not require any offline application profiling or benchmarking. Our implementation results on OpenStack using a multi-tier application under a range of workload traces demonstrate the robustness and superiority of DC2 over existing rule-based approaches.

1 Introduction

With the advent of cloud computing, many application owners have started moving their deployments into the cloud. Cloud computing offers many benefits over traditional physical deployments including lower infrastructure costs and elastic resource allocation. These benefits are especially advantageous for applications with a dynamic workload demand. Such applications can be deployed in the cloud based on the current demand, and the deployment can be scaled dynamically in response to changing workload demand.

While cloud computing is a very promising option for application owners, it is not easy to take full advantage of the benefits of the cloud. Specifically, while cloud computing offers flexible resource allocation, it is up to the customer (application owner) to leverage the flexible infrastructure. That is, the user must decide when and how to scale the application deployment to meet the changing workload demand.

Dynamically sizing a deployment is challenging for many reasons (see, for example, the recent survey paper [15]). From the perspective of the user, who is also the application owner, some of the specific hurdles that complicate the dynamic sizing of the application are: (i) Requires expert knowledge about the dynamics of the application, including the service requirements of the application at each tier, and (ii) Requires sophisticated modeling expertise to determine when and how to resize the deployment. For small and medium businesses (SMB), which comprise the targeted customer base for many cloud service providers (CSPs) [8,24], these hurdles are non-trivial to overcome. SMB users would much rather contract a cloud service that manages their dynamic sizing than invest in employing a team of experts. The purpose of this research is to provide this exact service - an application-agnostic cloud offering that will automatically, and dynamically, resize user applications to meet performance requirements in a cost-effective manner.

Many CSPs today offer monitoring services to users (not necessarily for free) for tracking resource usage. While such monitoring services provide valuable information, the user still requires expert knowledge about the application and the performance modeling expertise to convert the monitored information into scaling actions.

Some CSPs also offer rule-based triggers to help users scale their applications. These rule-based triggers allow the users to specify some conditions on the monitored metrics which, when met, will trigger a pre-defined scaling action. Even with the help of rule-based triggers, however, the burden of determining the threshold conditions for the metrics still rests with the user. For example, in order to use a CPU utilization based trigger for scaling, the user must determine the CPU threshold at which to trigger scale-up and scale-down, and the number of instances to scale-up and scale-down.

Note that CSPs cannot gather all the necessary application-level statistics without intruding into the user-space application. Given the lack of control and visibility into the application, CSPs *cannot* leverage most of the existing work (see, for example, [7,25,26]) on dynamic scaling of applications since these works typically require access to the application for measurement and profiling purposes. Further, most of the existing work is *not* application-agnostic, which is a requirement for a practical cloud service. A detailed discussion of the related work can be found in Section 5

We propose a completely automated cloud service, Dependable Compute Cloud (DC2), that proactively and dynamically scales the application deployment based on user-specified performance requirements. DC2 leverages resource-level and application-level statistics to infer the

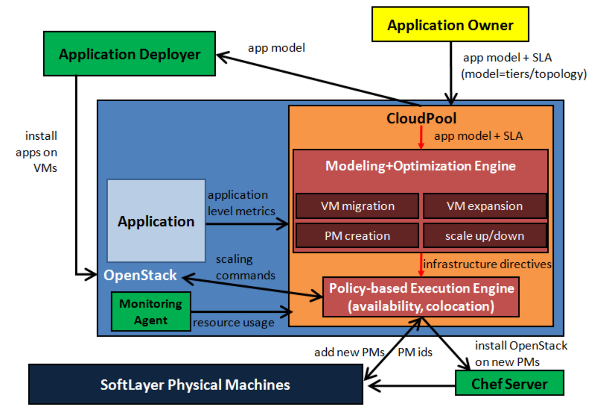


Figure 1: System architecture for DC2.

underlying system parameters of the application(s), and determines the required scaling actions to meet the performance goals in a cost-effective manner. A detailed discussion of our approach is presented in Section 2, along with details of our implementation.

At the heart of DC2 lies the modeling and execution engine that internalizes the monitored statistics and infers the necessary system parameters. While this engine can employ any grey-box or black-box modeling approach, in this paper we use Kalman filtering to infer the system parameters. Kalman filtering is a robust feedback control algorithm that combines monitoring information with a user-specified system model to create accurate estimations of the system state. In this paper we employ Kalman filtering by specifying a generic queueing-theoretic model (details of our modeling engine can be found in Section 3). Fortunately, since Kalman filtering leverages monitored statistics to come up with estimations, the underlying system model need not be accurate, as is often the case when using queueing theory (or any other mathematical modeling technique) to model complex systems.

We evaluate DC2 via implementation on OpenStack. We employ the three-tier bidding benchmark, RUBiS, as the user application and experiment with various workload traces. Our results demonstrate that DC2 successfully scales the application in response to changing workload demand without any user input and without any offline profiling. Importantly, we compare with existing rule-based triggers and show that DC2 is superior to such approaches. A detailed evaluation of our DC2 implementation is presented in Section 4.

2 Implementation

Figure 1 shows the proposed system architecture for the DC2 service environment. The Application Owner (customer) is responsible for providing the initial deployment model consisting of the multi-tier topology for the application (in the form of a graph or a configuration file) and the performance SLA requirements. The Application Deployer customizes the image for deployment and ties up the endpoints for the application during installation and configuration. We leverage Chef [18] to automate the installation of software on VMs during boot. We use OpenStack [17] as the underlying scalable cloud operating system. The VMs for the application are created on an OpenStack managed private cloud deployment on SoftLayer [23]. The CloudPool component in Figure 1 is a logical entity that models the application and issues the directives (such as VM scale up/down) required to maintain the performance SLA for the application. The Monitoring Agent is responsible for retrieving the resource-level metrics from the hypervisor and the application-level metrics from the application. The Modeling + Optimization Engine (described in detail in Section 3) takes as input the monitored metrics and outputs a list of directives indicating the addition or removal of VMs, migration of VMs, or a change in the resources allocated to VMs. These directives are passed on to the Policy-based Execution Engine that issues commands to OpenStack API, that in turn performs the scaling.

2.1 Application

We use the open source multi-tier application, RUBiS [2], for our experiments. RUBiS is an auction site prototype modeled after eBay.com supporting different classes of web requests such as bid, browse, buy, etc. Our RUBiS implementation employs Apache as the frontend web server, Tomcat as the Java servlets container, and MySQL as the backend database. In our experiments we focus on scaling the Tomcat application tier. We employ RUBiS's benchmarking tool to generate load by defining sessions consisting of a sequence of requests. The think time between requests is exponentially distributed with a mean of 1 second. We fix the number of clients for each experiment and vary the load by dynamically changing the composition of the workload mix.

2.2 Experimental setup

We employ multiple hypervisors with 8 CPU cores and 8 GB of memory each. The Apache and MySQL tiers are each hosted on a 4 CPU VM. The Tomcat application tier is hosted on multiple 2 CPU VMs. The provisioning time for a new Tomcat VM is about 30-40 seconds. Once the new VM is online, our automated scripts configure the JDBC with the IP address of the MySQL database and update the load balancer on Apache web server to include the new Tomcat VM.

2.3 Monitoring agent

We use virt-top (part of the libvirt [1] package) to collect VM CPU utilization statistics from each hypervisor periodically. For the application-level metrics, we periodically analyze the request URLs directed at the RU-

BiS application to compute the request rate and response time. Note that the user can choose to provide these metrics to us directly (for example, using a REST call). The monitoring interval is set to 10s. The collected statistics are then provided as input to the modeling engine.

2.4 Execution engine

The execution engine is primarily responsible for issuing commands for VM scaling based on the scaling directives received from the modeling engine. For robustness, the execution engine issues the VM scaling commands to OpenStack only after two successive scaling directives from the modeling engine. The execution engine is also responsible for placing the new VMs on specific hypervisors. We use host aggregates (which are essentially logical cloud partitions) to place the Apache and MySQL VMs on one hypervisor and Tomcat VMs on a different set of hypervisors.

3 Modeling

The modeling engine lies at the heart of our DC2 approach. We use a queueing-network model to *approximate* our multi-tier cloud application. However, since we cannot access the user application to derive the parameters of our model, we use a Kalman filtering technique to infer these unobservable parameters. We now describe our queueing model and Kalman filtering technique, followed by an analysis of our modeling engine, and finally, an explanation of how our modeling engine determines the required scaling actions for SLA compliance.

3.1 Queueing-network model

Figure 2 shows a queueing-network model of a generic three-tier system with each tier representing a collection of homogeneous servers. We assume that the load at each tier is distributed uniformly across all the servers in that tier. The system is driven by a workload consisting of i distinct request classes, each class being characterized by its arrival rate, λ_i, and end-to-end response time, R_i. Let n_j be the number of servers at tier j. With homogeneous servers and perfect load-balancing, the arrival rate of requests at any server in tier j is $\lambda_{ij} := \lambda_i / n_j$. Since servers at a tier are identical, for ease of analysis, we model each tier as a single representative server. With some abuse of terminology, we refer to the representative server at tier j as tier j. Let $u_j \in [0,1)$ be the utilization of tier j. The background utilization of tier j is denoted by u_{0j}, and models the resource utilization due to other jobs (not related to our workload) running on that tier. The end-to-end network latency for a class i request is denoted by d_i. Let $S_{ij}(\geq 0)$ denote the average service time of a class i request at tier j. Assuming we have Poisson arrivals and a processor-sharing policy at each server, the stationary distribution of the queueing network is known to have a

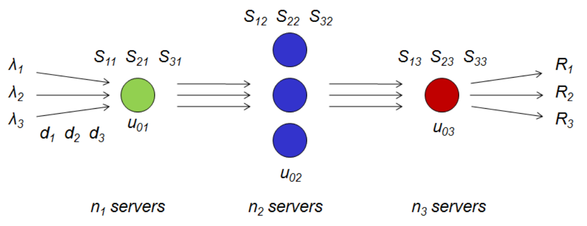


Figure 2: Queueing model for our system.

product-form [28], for any general distribution of service time at servers. Under the product-form assumption, we have the following analytical results from queueing theory:

$$u_j = u_{0j} + \sum_i \lambda_{ij} S_{ij}, \quad \forall j \tag{1}$$

$$R_i = d_i + \sum_j \frac{S_{ij}}{1-u_j}, \quad \forall i \tag{2}$$

While u_j, R_i and λ_i, $\forall i, j$, can be monitored easily and are thus observable, the parameters S_{ij}, u_{0j}, and d_i are non-trivial to measure and are thus unobservable. While existing work on auto-scaling (see, for example, [25,26]) typically obtains these values by directly accessing or modifying the application software (for example, by parsing the log files at each tier), our proposed application-agnostic cloud service *cannot* encroach the user's application space. Instead, we employ a parameter estimation technique, Kalman filtering (see Section 3.2 below), to derive estimates for the unobservable parameters. Further, since the system parameters can dynamically change during runtime, we employ the Kalman filter as an on-line parameter estimator to continually adapt our parameter estimates.
It is important to note that while the product-form is shown to be a reasonable assumption for tiered web services [5], we only use it as an approximation for our complex system. By employing the Kalman filter to leverage the actual monitored values, we minimize our dependence on the approximation.

3.2 Kalman filtering

For a three-class, three-tier system (i.e., $i = j = 3$), let $\mathbf{z} := (u_1, u_2, u_3, R_1, R_2, R_3)^T = \mathbf{h}(\mathbf{x})$ and $\mathbf{x} = (u_{01}, u_{02}, u_{03}, d_1, d_2, d_3, S_{11}, S_{21}, S_{31}, S_{12}, S_{22}, S_{32}, S_{13}, S_{23}, S_{33})^T$. Note that $\mathbf{z}$ is a 6-dimensional vector whereas $\mathbf{x}$ is a 15-dimensional vector. The problem is to determine the unobservable parameters $\mathbf{x}$ from measured values of $\mathbf{z}$ and $\lambda = (\lambda_1, \lambda_2, \lambda_3)$.
We use Kalman filtering to estimate the unobservable parameters (for a detailed explanation of Kalman filters, we refer the reader to [22]). The dynamic evolution of system parameters can be described through the following

Kalman filtering equations [22]:

$$\begin{aligned}\text{System State} \quad \mathbf{x}(t) &= \mathbf{F}(t)\mathbf{x}(t-1)+\mathbf{w}(t),\\ \text{Measurement Model} \quad \mathbf{z}(t) &= \mathbf{H}(t)\mathbf{x}(t)+\mathbf{v}(t),\end{aligned}$$

where $\mathbf{F}(t)$ is the state transition model and $\mathbf{H}(t)$ is the observation model mapping the true state space into the observed state space. In our case, $\mathbf{F}(t), \forall t$, is the identity matrix. The variables $\mathbf{w}(t) \sim \mathscr{N}(0, \mathscr{Q}(t))$ and $\mathbf{v}(t) \sim \mathscr{N}(0, \mathscr{R}(t))$ are process noise and measurement noise which are assumed to be zero-mean, multi-variate Normal distributions with covariance matrices $\mathscr{Q}(t)$ and $\mathscr{R}(t)$ respectively. The matrices $\mathscr{Q}(t)$ and $\mathscr{R}(t)$ are not directly measurable but can be tuned via best practices [14].

Since the measurement model $\mathbf{z}$ is a non-linear function of the system state $\mathbf{x}$ (see Eqns. (1) and (2)), we use the Extended Kalman filer [22] with $\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right]_{\mathbf{x}(t)}$, which for our model is a 6×15 matrix with $\mathbf{H}(t)_{ij} = \left[\frac{\partial \mathbf{h_i}}{\partial \mathbf{x_j}}\right]_{\mathbf{x}(t)}$.

Since $\mathbf{x}(t)$ is not known at time t, we estimate it by $\hat{\mathbf{x}}(t|t-1)$, which is the *a priori* estimate of $x(t)$ given all the history up to time $t-1$. The state of the filter is described by two variables $\hat{\mathbf{x}}(t|t)$ and $\mathbf{P}(t|t)$, where $\hat{\mathbf{x}}(t|t)$ is the *a posteriori* estimate of state at time t and $\mathbf{P}(t|t)$ is the *a posteriori* error covariance matrix which is a measure of the estimated accuracy of the system state.

The Kalman filter has two phases: Predict and Update. In the predict phase, *a priori* estimates of state and error matrix are calculated. In the update phase, these estimates are refined using the current observation to get *a posteriori* estimates of state and error matrix. The filter model for the predict and update phases for our 3-class, 3-tier model is given by:

Predict:

$$\begin{aligned}\hat{\mathbf{x}}(t|t-1) &= \mathbf{F}(t)\hat{\mathbf{x}}(t-1|t-1)\\ \mathbf{P}(t|t-1) &= \mathbf{F}(t)\mathbf{P}(t-1|t-1)\mathbf{F}^T(t)+\mathscr{Q}(t)\end{aligned}$$

Update:

$$\begin{aligned}\mathbf{y}(t) &= \mathbf{z}(t)-\mathbf{h}(\hat{\mathbf{x}}(t|t-1))\\ \mathbf{H}(t) &= \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right]_{\hat{\mathbf{x}}(t|t-1)}\\ \mathbf{S}(t) &= \mathbf{H}(t)\mathbf{P}(t|t-1)\mathbf{H}^T(t)+\mathscr{R}(t)\\ \mathbf{K}(t) &= \mathbf{P}(t|t-1)\mathbf{H}^T(t)\mathbf{S}^{-1}(t)\\ \hat{\mathbf{x}}(t|t) &= \hat{\mathbf{x}}(t|t-1)+\mathbf{K}(t)\mathbf{y}(t)\\ \mathbf{P}(t|t) &= (\mathbf{I}-\mathbf{K}(t)\mathbf{H}(t))\mathbf{P}(t|t-1)\end{aligned}$$

We employ the above filter model by seeding our initial estimate of $\hat{\mathbf{x}}(t|t-1)$ and $\mathbf{P}(t|t-1)$ with random values, then applying the Update equations by monitoring $\mathbf{z}(t)$ to get $\hat{\mathbf{x}}(t|t)$ and $\mathbf{P}(t|t)$, and finally using the Predict values to arrive at the estimated $\hat{\mathbf{x}}(t|t-1)$ and $\mathbf{P}(t|t-1)$. We continue this process iteratively at each 10 second monitoring interval to derive new system state estimates.

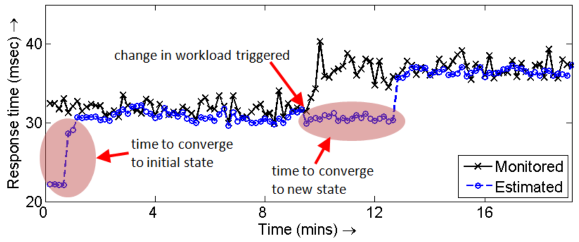


Figure 3: Accuracy and convergence of our Kalman filtering technique when employed in our experiments.

3.3 Performance analysis

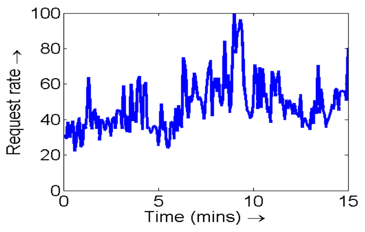
The Kalman filtering technique described above gives us estimates of the unobservable system parameters S_{ij}, u_{0j}, and d_i. We then use these estimates, along with Eqns. (1) and (2), to predict the future values of u_j and R_i. Figure 3 demonstrates our Kalman filtering technique in action. The solid line with crosses shows the monitored values of response time for a specific class of requests in our three-tier application (see Section 2 for details of our application setup). Here, the monitoring interval is 10 seconds. The dashed line with circles shows our estimated values for the predicted response time based on our Kalman filtering technique. It initially takes about a minute for our estimates to converge. After convergence, our estimated values are in very good agreement with the monitored values, thus validating our technique and highlighting its accuracy. Since we leverage the *current* monitored values of $\mathbf{z}$ and λ, our estimated system parameters can adapt to changes in the application. In order to demonstrate this ability, we trigger a change in our workload at about the 10-minute mark (shown in Figure 3) which causes the response time to increase. The change in the workload causes a change in the service time of the requests. Our Kalman filter detects this change based on the monitored values, and quickly adapts (in about 2 minutes) its estimates to converge to the new system state.

3.4 Scaling directives

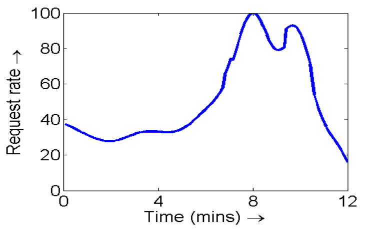
The estimated values of the system state are used to compute the required scaling actions for DC2. Specifically, given the response time SLA, we use Eqns. (1) and (2) to determine the minimum n_j required to ensure SLA compliance. Note that $\lambda_{ij} = \lambda_i / n_j$ in Eqn. (1). We demonstrate the auto-scaling abilities of the Kalman filtering-based DC2 approach in Section 4.

4 Evaluation

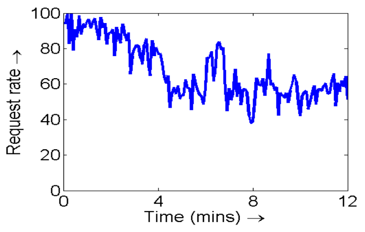
We now evaluate our DC2 scaling policy in various settings using the RUBiS application. We use traces from the WITS traffic archive [29] and the WorldCup98 dataset from the Internet Traffic Archive (ITA) [11] to drive our load generator. The WITS archive contains a large collection of recent internet traces from ISPs and University networks. The WorldCup98 dataset contains



(a) Bursty trace (source:WITS [29])



(b) Rampdown trace (source:ITA [11])



(c) Hill trace (source:WITS [29])

Figure 4: Traces (normalized) used for our experiments.

3 months worth of requests made to the 1998 World Cup Web site. We scaled the traces to fit our deployment. The normalized traces are shown in Figure 4. The workload we use for evaluation is a mix of different RUBiS request classes that together stress the application tier more than the other tiers.

In our experiments, we focus on the response time of browse requests since customers often base their web experience based on how long it takes to browse through online catalogues. We want the response time for the browse requests to be less than 40ms, on average, for *every* 10s monitoring interval. Note that this goal is much more challenging than requiring the response time be less than 40ms over the entire length of the experiment. We set the response time SLA for all other classes to be 100ms. The secondary goal is to minimize the number of application tier VMs employed during the experiment. We consider the following two metrics: **V**, the percentage of time that the response time SLA was violated, and **K**, the average number of application tier VMs used during the experiment. For each experiment, we compare DC2 with the following class of policies:

THRES(x,y) is a family of rule-based provisioning policies that adds one application VM when the average application tier utilization exceeds y% for successive intervals and removes one application VM when the average utilization falls below x% for successive intervals. In practice, it suffices to consider two successive intervals for the scaling decisions, just as in the case of DC2.

4.1 Comparison of different policies

Figure 5(a) shows our experimental results for DC2 under the Bursty trace. The figure shows the monitored (black solid line) and estimated (green line with dots) response time under DC2, along with the response time SLA (dashed line). We only show the response time for the browse requests. We see that the monitored response time under DC2 is below the SLA throughout the experiment. The up and down triangles represent the points in time when a scale-up and scale-down action was triggered, respectively. As mentioned in Section 2, a scaling is triggered based on two successive recommendations from the Kalman filter. Observe that the estimated response time is typically in agreement with the monitored response time. This indicates the accuracy of our Kalman filtering technique. However, there is a difference between the estimated and monitored response time for the first few intervals. This is because it takes some time for the Kalman filter to calibrate its model based on the monitored data, as discussed in Section 3.

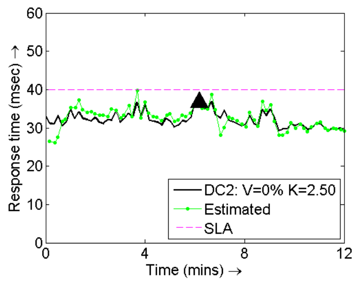
Using the THRES(x,y) policy in practice is tricky since it requires finding the right values for x and y. To find the optimal THRES policy, we start with $x = 20\%$ and $y = 70\%$, and then iterate via trial-and-error till we find the optimal values. Our results indicate that $y = 60\%$ results in the lowest **K** with $\mathbf{V} = 0$. We then experiment with different x values with $y = 60\%$. Based on our results, we conclude that THRES(30,60) is the optimal THRES policy for the Bursty trace.

Table 1 shows the performance of different policies for the Bursty trace. While both DC2 and THRES(30,60) result in zero SLA violations and low resource consumption, THRES requires a lot of experimentation and calibration to achieve the desired performance.

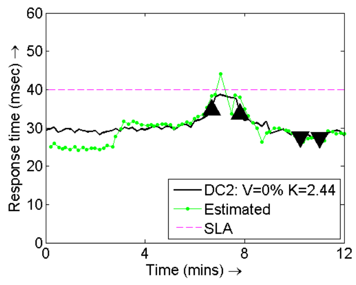
4.2 Comparison under different traces

We now consider the Hill trace and the Rampdown trace. Figures 5(b) and 5(c) show our experimental results for DC2 under these traces. We again see that the (monitored) response time under DC2 is below the SLA throughout the experiment for both traces. It is important to note that we do not change our DC2 algorithm between experiments. DC2 automatically adapts (based on the Kalman filtering technique discussed in Section 3) to the different traces and takes corrective actions to ensure that the SLA is not violated.

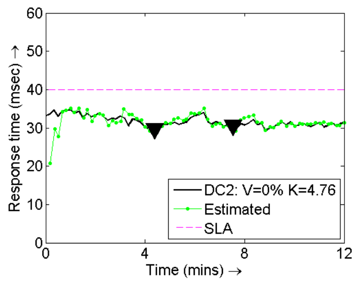
Unfortunately, the THRES(30,60) policy is no longer optimal for the Hill or Rampdown traces. For the Hill trace, we find that THRES(30,50) is optimal. This is because the Hill trace exhibits a steep rise in load, requiring more aggressive scaleup. For the Rampdown trace, we find that THRES(40,60) is optimal. This is because the Rampdown traces exhibits a gradually lowering request rate, allowing for more aggressive scaledown. Not using the right THRES policy for each trace can result in expensive SLA violations or increased resource consumption (see Table 1). We thus conclude that *DC2 is more robust to changes in arrival patterns than THRES*.



(a) DC2 under Bursty trace



(b) DC2 under Hill trace



(c) DC2 under Rampdown trace

Figure 5: Performance of DC2 for all traces.

Trace	Bursty		Hill		Rampdown	
Policy \ Metric	**V**	**K**	**V**	**K**	**V**	**K**
THRES(30,60)	**0%**	**2.50**	6.66%	2.56	0%	6.00
THRES(30,50)	0%	2.79	**1.21%**	**2.72**	0%	6.00
THRES(40,60)	2.02%	2.19	15.87%	2.13	**0%**	**4.62**
DC2	**0%**	**2.50**	**0%**	**2.44**	**0%**	**4.76**

Table 1: Comparison of policies for all traces. For each trace, the optimal policies' values are displayed in bold.

5 Related Work

Auto-scaling approaches: Prediction models [6,20,21] use historical data to predict future demand and proactively allocate resources. In most cases, however, some information about the application is required to convert predicted demand into resource requirements. Control-theoretic techniques [6,7,9,13] react to the current system state and adjust the resource allocation accordingly. However, these approaches typically rely on system profiling to convert the system state into scaling actions. Queueing-based models [19,25,26] also require information about the application to make informed scaling decisions. Black-box models do not require information about the application, and instead leverage statistical techniques [16] or machine learning [4] to infer system parameters. DC2 uses a grey-box approach by modeling the system as a queueing network, and then leverages Kalman filtering to infer the parameters of the queueing model. Grey-box approaches typically require less time to converge and infer the system state as opposed to black-box models.

Kalman filtering approaches: An Extended Kalman filter (EKF) based approach was proposed in [31,33] where the system was modeled using a queueing network. The authors in [14] study a three-class, single-tier system and conduct an extensive experimental evaluation of EKF. Recently, [32] applied the EKF approach to track resource usage for a three-class, two-tier system. In this work, we generalize EKF to a three-class, three-tier system, and specifically use EKF for auto-scaling, as opposed to the above works that focus on modeling and offline analysis. The authors in [12] use Kalman filtering for allocating CPU resources to VMs by modeling the application performance as a function of CPU utilization. Our work leverages application-level metrics in addition to resource usage metrics, and employs queueing-theoretic models to capture the interaction between the resources, application load, and performance.

Rule-based approaches: Auto-scaling features are now offered by almost every major CSP including Amazon (AWS) [3], VMware [27], Windows Azure [30], and Google [10]. However, to the best of our knowledge, existing CSP-offered auto-scaling solutions are rule-based and typically require the user to specify the threshold values on the resource usage (e.g., CPU, memory, storage). Further, such rule-based approaches have to be tuned to the specific demand pattern for best results, as demonstrated by the THRES policy in Section 4. By contrast, DC2 does *not* require the user to specify scaling rules.

6 Conclusion

In this paper we present the design and implementation of a new cloud service, Dependable Compute Cloud (DC2), that automatically scales user applications in a cost-effective manner to provide performance guarantees. Since cloud service providers (CSPs) do not have complete control and visibility of a user's cloud deployment, we designed DC2 to be application-agnostic. In particular, unlike most of the existing auto-scaling research, DC2 does *not* require any offline profiling or application benchmarking. Instead, DC2 employs a Kalman filtering technique in combination with a queueing theoretic model to proactively determine the right scaling actions for an application deployed in the cloud. An overarching goal behind the conception of DC2 is to make a case for a CSP-offered auto-scaling service that is superior to existing rule-based offerings. Since the cloud is marketed as a platform designed for all levels of tenants, we believe that users who do not have expert knowledge in performance modeling and system optimization should be able to easily scale their applications. Existing auto-scaling research has ignored this segment of users. We hope that DC2 motivates further research in the area of easy-to-use, application-agnostic auto-scaling.

References

[1] libvirt virtualization API. `http://libvirt.org`.

[2] RUBiS: Rice University Bidding System. `http://rubis.ow2.org`.

[3] AMAZON INC. Amazon Auto Scaling. `http://aws.amazon.com/autoscaling`.

[4] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, UT, USA, 2014), pp. 127–144.

[5] DUBE, P., YU, H., ZHANG, L., AND MOREIRA, J. Performance evaluation of a commercial application, trade, in scale-out environments. In *Proceedings of the 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2007), pp. 252–259.

[6] GANDHI, A., CHEN, Y., GMACH, D., ARLITT, M., AND MARWAH, M. Minimizing Data Center SLA Violations and Power Consumption via Hybrid Resource Provisioning. In *Proceedings of the 2011 International Green Computing Conference* (Orlando, FL, USA, 2011), pp. 49–56.

[7] GANDHI, A., HARCHOL-BALTER, M., RAGHUNATHAN, R., AND KOZUCH, M. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *Transactions on Computer Systems 30* (2012).

[8] GARTNER, INC. Gartner's Advice for CSPs Becoming Cloud Service Providers. `https://www.gartner.com/doc/2155315`, 2012.

[9] GHANBARI, H., SIMMONS, B., LITOIU, M., BARNA, C., AND ISZLAI, G. Optimal Autoscaling in a IaaS Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing* (San Jose, CA, USA, 2012), pp. 173–178.

[10] GOOGLE CLOUD PLATFORM. Auto Scaling on the Google Cloud Platform. `http://cloud.google.com/resources/articles/auto-scaling-on-the-google-cloud-platform`.

[11] ITA. The Internet Traffic Archives: WorldCup98. `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`.

[12] KALYVIANAKI, E., CHARALAMBOUS, T., AND HAND, S. Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters. In *Proceedings of the 6th International Conference on Autonomic Computing* (Barcelona, Spain, 2009), pp. 117–126.

[13] KRIOUKOV, A., MOHAN, P., ALSPAUGH, S., KEYS, L., CULLER, D., AND KATZ, R. NapSAC: Design and implementation of a power-proportional web cluster. In *Proceedings of the 1st ACM SIGCOMM Workshop on Green Networking* (New Delhi, India, 2010), pp. 15–22.

[14] KUMAR, D., TANTAWI, A., AND ZHANG, L. Estimating model parameters of adaptive software systems in real-time. In *Run-time Models for Self-managing Systems and Applications*, D. Ardagna and L. Zhang, Eds., Autonomic Systems. Springer Basel, 2010, pp. 45–71.

[15] LORIDO-BOTRÁN, T., MIGUEL-ALONSO, J., AND LOZANO, J. A. Auto-scaling Techniques for Elastic Applications in Cloud Environments. Tech. Rep. EHU-KAT-IK-09-12, University of the Basque Country, 2012.

[16] NGUYEN, H., SHEN, Z., GU, X., SUBBIAH, S., AND WILKES, J. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of the 10th International Conference on Autonomic Computing* (San Jose, CA, USA, 2013), pp. 69–82.

[17] OPENSTACK.ORG. OpenStack Open Source Cloud Computing Software. `http://www.openstack.org`.

[18] OPSCODE INC. Chef. `http://www.opscode.com/chef`.

[19] PACIFICI, G., SPREITZER, M., TANTAWI, A., AND YOUSSEF, A. Performance management for cluster-based web services. *Selected Areas in Communications, IEEE Journal on 23*, 12 (2005), 2333–2343.

[20] ROY, N., DUBEY, A., AND GOKHALE, A. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *IEEE International Conference on Cloud Computing* (2011), pp. 500–507.

[21] SHEN, Z., SUBBIAH, S., GU, X., AND WILKES, J. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal, 2011), pp. 1–14.

[22] SIMON, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. John Wiley & Sons, 2006.

[23] SOFTLAYER TECHNOLOGIES, INC. `http://www.softlayer.com`.

[24] SP HOME RUN INC. Cloud Service Provider (CSP) and Inbound Marketing. `http://www.sphomerun.com/cloud-service-provider-csp`, 2013.

[25] URGAONKAR, B., AND CHANDRA, A. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing* (Seattle, WA, USA, 2005), pp. 217–228.

[26] URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Banff, Alberta, Canada, 2005), pp. 291–302.

[27] VMWARE, INC. VMware vFabric AppInsight. `http://pubs.vmware.com/appinsight-5/index.jsp`.

[28] WALRAND, J. *An Introduction to Queueing Networks*. Prentice Hall, 1988.

[29] WAND NETWORK RESEARCH GROUP. WITS: Waikato Internet Traffic Storage. `http://www.wand.net.nz/wits/index.php`.

[30] WINDOWSAZURE. How to Scale an Application. `http://www.windowsazure.com/en-us/manage/services/cloud-services/how-to-scale-a-cloud-service`.

[31] WOODSIDE, M., ZHENG, T., AND LITOIU, M. Service system resource management based on a tracked layered performance model. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 175–184.

[32] ZHANG, L., MENG, X., MENG, S., AND TAN, J. K-scope: Online performance tracking for dynamic cloud applications. In *Proceedings of the 10th International Conference on Autonomic Computing* (2013), pp. 29–32.

[33] ZHENG, T., WOODSIDE, M., AND LITOIU, M. Performance model estimation and tracking using optimal filters. *Software Engineering, IEEE Transactions on 34*, 3 (2008), 391–406.