K-Scope: Online Performance Tracking for Dynamic Cloud Applications

Li Zhang Xiaoqiao Meng Shicong Meng Jian Tan
IBM TJ Watson Research Center
{*zhangli, xmeng, smeng, tanji*}@*us.ibm.com*

1 Introduction

Cloud computing is an ongoing technology evolution that reshapes every aspects of computing. Cloud provides on-demand, flexible and easy-to-use resource provisioning. It is also an open platform where Cloud users can share software components, resources and services. These features give rise to several emerging Cloud application development and deployment paradigms, represented by continuous delivery and shared platform services.

Continuous Delivery [3], coined by Amazon, is a new way of releasing software wherein a Cloud application (e.g., Amazon web services) is delivered through frequent incremental updates. Cloud enables this paradigm by allowing developers to easily create a pipeline of automated application building, testing and deployment. For instance, application developers can quickly produce multiple application deployment for different development stages through virtual machine replication. Continuous delivery provides tremendous benefits in improving user experience and reduces the risk of each individual release substantially.

Shared Platform Services are commonly used in Cloud applications, and rapidly gaining popularity with increasing Platform-as-a-Service offers from Cloud service providers. Perhaps the most widely used platform service today is database or datastore services (e.g., SimpleDB from Amazon and Cloud SQL from Google) which are large-scale multi-tenant databases or datastores shared by multiple Cloud applications through a set of data access APIs. Enterprise users sometimes also deploy their own database/datastore servers shared by multiple applications in their virtual private Cloud (VPC). These shared data services reduce the management burden for application developers.

Despite the enormous convenience and great potential of these new paradigms, they also introduce new performance management challenges due to the volatility embedded in these techniques as well as the lack of well-defined performance requirements. For instance, updates in continuous deployment often change the behavior and the performance characteristics of an application, which may lead to performance degradation and service level agreement (SLA) violations. Similarly, due to the sharing nature of data services, one may experience fluctuation in data access performance when the overall workloads of the data service change. We refer to Cloud applications utilizing these features as *dynamic Cloud applications* to distinguish them from applications using traditional development life cycle and dedicated software components.

These challenges call for a fundamental piece missing from today's Cloud services, that is the ability to *continuously*, *efficiently* and *accurately* capture the most up-to-date performance characteristics of a dynamic Cloud application. Existing performance modeling approaches, however, do not readily provide this continuous modeling ability, primarily because they are designed with a traditional static deployment in mind where an application runs on dedicated machines and its implementation does not change during the modeling process. Some of them [9] must run offline with long model training time and high cost. Others [8, 12, 10] cannot explicitly model multiple request types or multiple functional layers which are common for Cloud applications. There are also techniques [1, 2] that can capture performance changes at different functional layers, but require instrumentation of the application.

In this paper, we introduce the first online, multi-request, multi-layer application performance modeling approach. It is non-intrusive in the sense that it infers critical performance model metrics such as request service time at different functional layers (e.g, web/application/database servers), which are usually unobservable, only from basic monitoring information such as end-to-end response time and CPU utilization, without instrumenting applications. Furthermore, it utilizes

Kalman filters [7] to continuously adjust model metrics to keep the model consistent with the dynamic Cloud application. As a result, an up-to-date performance model is always ready for users to query and perform tasks such as capacity planning and auto-scaling. For instance, it can quantitatively predict how much resources are needed at different functional layers to maintain a given performance, even when the application is constantly undergoing software updates.

2 Approach Overview

We consider a Cloud application consisting of multiple functional layers, e.g., web server layer, application server layer and database server layer. Such an application processes a number of different types of requests, each of which can be quite different in terms of execution time and resource consumption. Furthermore, the application has a targeted performance goal or service level agreement (SLA), e.g., the average response time should be smaller than 500ms. We assume the available monitoring data for these Cloud applications are basic system utilization metrics (e.g., CPU utilization), throughput and response time. These information are readily available on most Cloud platforms [6].

We choose to use non-intrusive modeling techniques that provide an easy-to-use performance model that can predict application resource utilization and performance, rather than using instrumentation based tracing techniques. Specifically, we use the queuing network model as the basic framework as it is general enough to model multi-layer multi-request applications. To cope with the changing performance characteristics in dynamic Cloud applications, in particular, the request service time which is the time a server spent to process a request, we need an agile, online model parameter estimation technique, rather than traditional constrained optimization based offline estimation techniques. Kalman filter, as a time-tested technique for estimating potentially changing future states, falls nicely into our design.

2.1 Queueing Network Model

Queueing network models are commonly used to capture the performance of complex computer systems [4]. They have been shown to provide accurate characterization of request level and system level performance metrics [5, 11]. Well calibrated queueing network models are the basis for performance sizing and capacity planning. Here we also use a general queueing network model for Cloud applications.

We will use a 3-class, 2-tier system to illustrate our performance modeling and tracking methodology. It can easily be extended to a general n class k tier system. We first define a set of variables for the model:

$$\begin{aligned}
\lambda_i &= \text{Arrival rate of class } i \text{ jobs.} \\
S_{ij} &= \text{Average service time of class } i \text{ jobs at tier } j. \\
d_i &= \text{Additional delay for class } i \text{ jobs in system.} \\
u_{0j} &= \text{Background utilization for tier } j. \\
u_j &= \text{Average utilization for tier } j. \\
R_i &= \text{Average response time for class } i \text{ jobs in system.}
\end{aligned}$$

Under appropriate assumptions, the system performance and resource utilization can be approximated by the queueing analytic relations below.

$$u_j = u_{0j} + \lambda_1 S_{1j} + \lambda_2 S_{2j} + \lambda_3 S_{3j}, j \in \{1,2\} \quad (1)$$

$$R_i = d_i + \frac{S_{i1}}{1-u_1} + \frac{S_{i2}}{1-u_2}, i \in \{1,2,3\} \quad (2)$$

In vector form: $\mathbf{z} := (u_1, u_2, R_1, R_2, R_3)^T = \mathbf{h}(\mathbf{x})$. The assumptions for the above formulate to hold are quite general. For example, under Poisson arrivals and processor sharing policy at each server, the formulate are exact. Processor sharing policy can reasonably approximate the scheduling behaviors in modern operating systems. Numerous studies have demonstrated that the queueing model above provides a good approximation to the real system.

It is relatively easy to measure the aggregate system utilization u_1, u_2, the request throughput $\lambda_1, \lambda_2, \lambda_3$, and the end-to-end response times R_1, R_2, R_3. The delay and service time parameters, however, are very difficult to measure directly. These parameters are the key quantitative information of the system model. In our 3-class 2-server example, the system parameters are

$$\mathbf{x} = (u_{01}, u_{02}, d_1, d_2, d_3, S_{11}, S_{21}, S_{31}, S_{12}, S_{22}, S_{32})^T \quad (3)$$

an 11-dimension vector. The important problem we need to solve now is to estimate the system parameters $\mathbf{x}$ based on the measurement data $\mathbf{z} = (u_1, u_2, R_1, R_2, R_3)^T$. The off-line parameter estimation problem has been addressed in [11] by formulating the problem as an optimization problem.

Below we address this on-line parameter estimation problem with noisy measurement data. The challenge is how to efficiently and accurately estimate $\mathbf{x}$ on line from a continuous stream of measurements $\mathbf{z}$. Kalman filter theory is a perfect tool to tackle this problem.

2.2 Kalman Filter

Kalman filter is developed by Rudolf E. Kalman around 1960. It is commonly used to estimate the values of hidden state variables of a dynamic system that is excited by stochastic disturbances and stochastic measurement noise. In real systems, all the variables are functions of

time. Measurements will change over time. Parameter values will have estimates that are updated over time. The dynamics of the system following the Kalman filter framework is

$$\mathbf{x}(t) = \mathbf{F}(t)\mathbf{x}(t-1)+\mathbf{w}(t) = \mathbf{x}(t-1)+\mathbf{w}(t), \quad (4)$$

$$\mathbf{z}(t) = \mathbf{H}(t)\mathbf{x}(t-1)+\mathbf{v}(t). \quad (5)$$

Here $\mathbf{x}$ is the state variable that is not observed. $F(t)$ is the state transition model that describes the evolution of the state over time. $\mathbf{w}(t)$ is the process noise which is assumed to be a zero mean, multi-variate Normal distribution with certain covariance matrix $\mathscr{Q}(t)$, i.e. $\mathbf{w}(t) \sim \mathscr{N}(0, \mathscr{Q}(t))$. $\mathbf{z}(t)$ is the measurement vector. $\mathbf{H}(t)$ is the observation model which maps the true state space into the observed space. $v(t)$ is the observation noise which is assumed to be a zero mean, multi-variate Normal distribution with certain covariance matrix $\mathscr{R}(t)$, i.e. $\mathbf{v}(t) \sim \mathscr{N}(0, \mathscr{R}(t))$. The covariance matrices $\mathscr{Q}$ and $\mathscr{R}$ are not directly measurable. They will be tuned based on best practice heuristics.

Since the measurement model is a non-linear function of the system state parameters (due to the utilization u in the denominator), we must use the 'Extended' version of the Kalman filter. $\mathbf{H}(t)$ is computed as, $\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right](\mathbf{x}(t))$ Since we don't really know $\mathbf{x}$ at time t, we will estimate it based on all the information we have before time t. $\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right](\hat{\mathbf{x}}(t|t-1))$ Here $\hat{\mathbf{x}}(t|t-1)$ is the estimate of $\mathbf{x}(t)$ given all the information up to time $t-1$.

The state of the filter is represented by two variables:

- $\hat{\mathbf{x}}(t|t)$ is the estimate of state at time t given observations up to and including time t.
- $\mathbf{P}(t|t)$ is the error covariance matrix (a quantitative measure of estimated accuracy of the state estimate).

Here are the two sets of equations for the Kalman filter algorithm:

Predict:

$$\hat{\mathbf{x}}(t|t-1) = \mathbf{F}(t)\hat{\mathbf{x}}(t-1|t-1) \quad (6)$$

$$\mathbf{P}(t|t-1) = \mathbf{F}(t)\mathbf{P}(t-1|t-1)\mathbf{F}^T(t)+\mathscr{Q}(t) \quad (7)$$

Update:

$$\mathbf{H}(t) = \left[\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right](\hat{\mathbf{x}}(t|t-1)) \quad (8)$$

$$\mathbf{S}(t) = \mathbf{H}(t)\mathbf{P}(t|t-1)\mathbf{H}^T(t)+\mathscr{R}(t) \quad (9)$$

$$\mathbf{K}(t) = \mathbf{P}(t|t-1)\mathbf{H}^T(t)\mathbf{S}^{-1}(t) \quad (10)$$

$$\hat{\mathbf{x}}(t|t) = \hat{\mathbf{x}}(t|t-1)+\mathbf{K}(t)(\mathbf{z}(t)-\mathbf{h}(\hat{\mathbf{x}}(t|t-1))) \quad (11)$$

$$\mathbf{P}(t|t) = (\mathbf{I}-\mathbf{K}(t)\mathbf{H}(t))\mathbf{P}(t|t-1) \quad (12)$$

In our 3-class 2-server queueing network example, the Jacobian is given by, $\frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \begin{bmatrix} J_{11} & J_{12} & J_{13} & J_{14} \\ J_{21} & J_{22} & J_{23} & J_{24} \end{bmatrix}$ The algorithm iterates between the predict and update steps as new measurement data arrives.

2.3 Applications

K-Scope has a wide range of applications, including performance diagnosis, answering what-if queries, capacity planning and performance-driven dynamic provisioning.

Performance Diagnosis. Performance diagnosis for multi-layer applications is painful as generic monitoring provides only end-to-end performance statistics which offer little insight on the performance of individual functional layers. K-Scope explicitly estimates request service time at different layers, and provides a clear breakdown of the response time.

Answering What-If Queries. A simple approach is that we first apply the model to track the system in a stable period; with all the model parameters estimated, the question can be generally solved by varying certain parameters and re-calculate the other parameters.

Capacity Planning. K-Scope also simplifies capacity planning as application developers can leverage the performance model produced by K-Scope to virtually explore a large number of deployment options and predict the corresponding performance.

Dynamic Provisioning. As K-Scope provides a breakdown of request execution time at different layers, it can guide dynamic provisioning to the bottlenecked layer. In addition, dynamic provisioning can query K-Scope to find out how many additional virtual instances are needed to maintain the targeted performance, and quickly adds the required number of instances in a single batch to minimize the window of performance violation.

3 Evaluation

We apply K-Scope to a real-world multi-layer application. In addition, we describe a simple usage scenario in which the model is used for capacity planning.

The tested workload is SOABench, an IBM internal benchmark widely used to measure the performance of Web servers. Our testbed consists of a client and a server machine. Each machine is equipped with an Intel 1.6GHz 8-core Xeon processor. The client machine runs the SOABench workload generator, a Java program that could spawn multiple threads to simulate concurrent Web service users. The server machine runs IBM WAS(WebSphere Application Server). Each Java thread in the workload generator sends a service request to the WAS server. Upon receiving the response, the thread continues to send another request. Three types of service requests are sent by the generator: for Type 1, both the request and the response have 3K Byte payload. Type 2 and 3 have 10K and 1M Byte payload respectively. This SOABench testbed follows the three-class two-tier model in the previous sections.

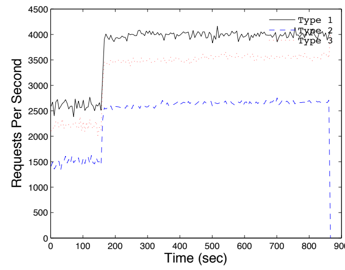
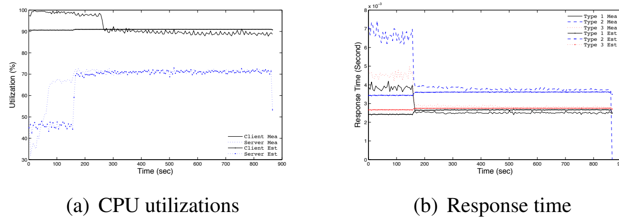


Figure 1: Workload charasterics in SOABench testbed



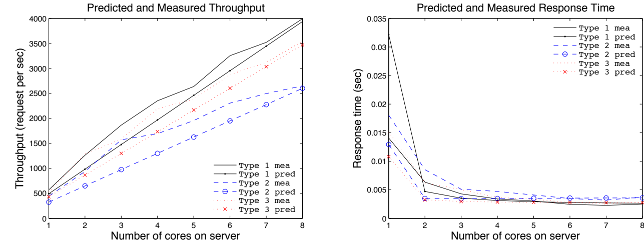
(a) CPU utilizations

(b) Response time

Figure 2: Measured and Predicted Results

In our first experiment, we want to use the model to track the performance of SOABench when the system resources are close to full utilization. To this end, we increase the number of threads on the client until either the client or the server has a saturated CPU usage. At this saturation point, the workload generator spawns 24 threads: 8 threads for each request type. Figure 1 shows the throughput for each request type. We run the experiment for about 15 minutes. The first 3 minutes are a warm-up period. After the warm-up, all the performance metrics become stable. We then collect data for the observable performance metrics, feed the data to the proposed model, and measure the model accuracy by comparing the predicted CPU utilization and response time to their actual values. Figure 2(a) compares the measured and estimated CPU utilization. Figure 2(b) compares the measured and the estimated response time for each request type. All these comparisons clearly show that the model can precisely track the performance.

Now we describe a case in which the model is used to address a simple capacity planning issue. The WAS server has eight cores and all these cores are dedicated to the WAS application. If the server allocates fewer cores to the WAS, how will this impact the throughput and response time? Such a typical *what-if* question can be easily answered by applying the model. In principle, if fewer CPU cores are allocated to a task, the task processing time should increase. We approximately assume that if the allocated core number on the server is reduced to $\frac{1}{x}$ of the original core number, the service time for each request, namely, S_{12}, S_{22} and S_{32}, should be multiplied by x respectively. Now if the server keeps the same utiliza-



(a) Predicted throughput when reducing CPU cores on server

(b) Predicting response time when reducing CPU cores on server

Figure 3: Application in Capacity Planning

tion ratio, according to Equation (1), λ_1, λ_2 and λ_3 are reduced to $\frac{1}{x}$ of their original values respectively. After computing the adjusted u_1 from (1), we can further compute the new response time from (2). To evaluate the accuracy of this simple computation, we vary the allocated core number on the server from 1 to 8, and for each setting, we restart the WAS server. Figure 3(a) compares the computed throughput and the actual measurements. Figure 3(b) compares the response time. On both aspects, the estimation follows the ground truth.

References

[1] Barham, P., Isaacs, R., Mortier, R., and Narayanan, D. Magpie: Online modelling and performance-aware systems. In *HotOS* (2003), pp. 85–90.

[2] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *NSDI* (2007).

[3] Humble, J., and Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[4] Kleinrock, L. *Queueing Systems*. Wiley, 1976.

[5] Kumar, D., Olshefski, D. P., and Zhang, L. Connection and performance model driven optimization of pageview response time. In *MASCOTS* (2009), IEEE, pp. 1–10.

[6] Meng, S., Wang, T., and Liu, L. Monitoring continuous state violation in datacenters: Exploring the time dimension. In *ICDE* (2010), pp. 968–979.

[7] Simon, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley, 2006.

[8] Solomon, B., Ionescu, D., Litoiu, M., and Mihaescu, M. A real-time adaptive control of autonomic computing environments. In *CASCON* (2007), pp. 124–136.

[9] Urgaonkar, B., Pacifici, G., Shenoy, P. J., Spreitzer, M., and Tantawi, A. N. Analytic modeling of multitier internet applications. *TWEB 1*, 1 (2007).

[10] Woodside, C. M., Zheng, T., and Litoiu, M. Service system resource management based on a tracked layered performance model. In *ICAC* (2006), pp. 175–184.

[11] Zhang, L., Xia, C. H., Squillante, M. S., and III, W. N. M. Workload service requirements analysis: A queueing network optimization approach. In *MASCOTS* (2002), IEEE.

[12] Zheng, T., Woodside, C. M., and Litoiu, M. Performance model estimation and tracking using optimal filters. *IEEE Trans. Software Eng. 34*, 3 (2008), 391–406.