Automated Anomaly Detection and Performance Modeling of Enterprise Applications

6

LUDMILA CHERKASOVA and KIVANC OZONAT
Hewlett-Packard Labs
NINGFANG MI
Northeastern University
JULIE SYMONS
Hewlett-Packard
and
EVGENIA SMIRNI
College of William and Mary

Automated tools for understanding application behavior and its changes during the application lifecycle are essential for many performance analysis and debugging tasks. Application performance issues have an immediate impact on customer experience and satisfaction. A sudden slowdown of enterprise-wide application can effect a large population of customers, lead to delayed projects, and ultimately can result in company financial loss. Significantly shortened time between new software releases further exacerbates the problem of thoroughly evaluating the performance of an updated application. Our thesis is that online performance modeling should be a part of routine application monitoring. Early, informative warnings on significant changes in application performance should help service providers to timely identify and prevent performance problems and their negative impact on the service. We propose a novel framework for automated anomaly detection and application change analysis. It is based on integration of two complementary techniques: (i) a regression-based transaction model that reflects a resource consumption model of the application, and (ii) an application performance signature that provides a compact model of runtime behavior of the application. The proposed integrated framework provides a simple and powerful solution for anomaly detection and analysis of essential performance changes in application behavior. An additional benefit of the proposed approach is its simplicity: It is not intrusive and is based on monitoring data that is typically available in enterprise production environments. The introduced

E. Smirni has been partially supported by NSF grants CSR-0720699 and CCF-0811417.
Author's addresses: L. Cherkasova, HP Labs, Palo Alto, CA; email: lucy.cherkasova@hp.com; K. Ozonat, HP Labs, Palo Alto, CA; email: kivanc.ozonat@hp.com; N. Mi, Northeastern University, Boston, MA; email: n.mi@neu.edu; J. Symons, HP, Cupertino, CA; email: julie.symons@hp.com; E. Smirni, College of William and Mary, Williamsburg, VA; email: esmirni@cs.wm.edu.

DOI 10.1145/1629087.1629089 http://doi.acm.org/10.1145/1629087.1629089

solution further enables the automation of capacity planning and resource provisioning tasks of multitier applications in rapidly evolving IT environments.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Special-Purpose and Application-Based Systems**]: Performance of Systems

General Terms: Experimentation, Management, Performance

Additional Key Words and Phrases: Anomaly detection, online algorithms, performance modeling, capacity planning, multitier applications

ACM Reference Format:

Cherkasova, L., Ozonat, K., Mi, N., Symons, J., and Smirni, E. 2007. Automated anomaly detection and performance modeling of enterprise applications. ACM Trans. Comput. Syst. 27, 3, Article 6 (November 2009), 32 pages.
DOI = 10.1145/1629087.1629089 http://doi.acm.org/10.1145/1629087.1629089

1. INTRODUCTION

Today's IT and services departments are faced with the difficult and challenging task of ensuring that enterprise business-critical applications are always available and provide adequate performance. As the complexity of IT systems increases, performance analysis becomes time consuming and labor intensive for support teams. For larger IT projects, it is not uncommon for the cost factors related to performance tuning, performance management, and capacity planning to result in the largest and least controlled expense.

We address the problem of efficiently diagnosing essential performance changes in application behavior in order to provide timely feedback to application designers and service providers. Typically, preliminary performance profiling of an application is done by using synthetic workloads or benchmarks which are created to reflect a "typical application behavior" for "typical client transactions." While such performance profiling can be useful at the initial stages of design and development of a future system, it may not be adequate for analysis of performance issues and observed application behavior in existing production systems. For one thing, an existing production system can experience a very different workload compared to the one that has been used in its testing environment. Secondly, frequent software releases and application updates make it difficult and challenging to perform a thorough and detailed performance evaluation of an updated application. When poorly performing code slips into production and an application responds slowly, the organization inevitably loses productivity and experiences increased operating costs.

Automated tools for understanding application behavior and its changes during the application lifecycle are essential for many performance analysis and debugging tasks. Yet, such tools are not readily available to application designers and service providers. The traditional *reactive* approach is to set thresholds for observed performance metrics and raise alarms when these thresholds are violated. This approach is not adequate for understanding the performance changes between application updates. Instead, a *proactive* approach that is based on *continuous* application performance evaluation may assist enterprises in reducing loss of productivity by time-consuming diagnosis of essential performance changes in application performance. With

complexity of systems increasing and customer requirements for QoS growing, the research challenge is to design an integrated framework of measurement and system modeling techniques to support performance analysis of complex enterprise systems. Our goal is to design a framework that enables automated detection of application performance changes and provides useful classification of the possible root causes. There are a few causes that we aim to detect and classify.

—*Performance Anomaly*. By performance anomaly we mean that the observed application behavior (e.g., current CPU utilization) cannot be explained by the observed application workload (e.g., the type and volume of transactions processed by the application suggests a different level of CPU utilization). Typically, it might point to either some unrelated resource-intensive process that consumes system resources or some unexpected application behavior caused by not fully debugged application code.

—*Application Transaction Performance Change*. By transaction performance change we mean an essential change (increase or decrease) in transaction processing time, for example, as a result of the latest application update. If the detected change indicates an increase of the transaction processing time then an alarm is raised to assess the amount of additional resources needed and provides the feedback to application designers on the detected change (e.g., is this change acceptable or expected?).

It is important to distinguish between *performance anomaly* and *workload change*. A performance anomaly is indicative of abnormal situation that needs to be investigated and resolved. On the contrary, a workload change (i.e., variations in transaction mix and load) is typical for Web-based applications. Therefore, it is highly desirable to avoid false alarms raised by the algorithm due to workload changes, though information on observed workload changes can be made available to the service provider.

Effective models of complex enterprise systems are central to capacity planning and resource provisioning. In Next Generation Data Centers (NGDC), where server virtualization provides the ability to slice larger, underutilized physical servers into smaller, virtual ones, fast and accurate performance models become instrumental for enabling applications to automatically request necessary resources and support design of utility services. Performance management and maintenance operations come with more risk and an increased potential for serious damage if they are done wrong, and/or decision making is based on "erroneous" data. The proposed approach aims to automatically detect the performance anomalies and application changes. It effectively serves as the foundation for an accurate, online performance modeling, automated capacity planning, and provisioning of multitier applications using simple *regression-based* analytic models [Zhang et al. 2007b].

The rest of the article is organized as follows. Section 2 introduces client versus server transactions. Section 3 provides two motivating examples. Section 4 introduces a regression-based modeling technique for performance anomaly detection. Section 5 presents a case study to validate the proposed technique

and discusses its limitations. Section 6 introduces a complementary technique to address observed limitations of the first method. Section 6.3 extends the earlier case study to demonstrate the additional power of the second method, and promotes the integrated solution based on both techniques. Section 7 applies the results of the proposed technique to automate and improve accuracy of capacity planning in production systems. Section 8 describes related work. Finally, a summary and conclusions are given in Section 9.

2. CLIENT VS. SERVER TRANSACTIONS

The term *transaction* is often used with different meanings. In our work, we distinguish between a client transaction and a server transaction.

A client communicates with a Web service (deployed as a multitier application) via a Web interface, where the unit of activity at the client side corresponds to a download of a Web page. In general, a Web page is composed of an HTML file and several embedded objects such as images. This composite Web page is called a *client transaction*.

Typically, the main HTML file is built via dynamic content generation (e.g., using Java servlets or JavaServer Pages) where the page content is generated by the application server to incorporate customized data retrieved via multiple queries from the back-end database. This main HTML file is called a *server transaction*. Typically, the server transaction is responsible for most latency and consumed resources [Cherkasova et al. 2003] (at the server side) during client transaction processing.

A client browser retrieves a Web page (client transaction) by issuing a series of HTTP requests for all the objects: First it retrieves the main HTML file (server transaction) and after parsing it, the browser retrieves all the embedded objects. Thus, at the server side, a Web page retrieval corresponds to processing multiple requests that can be retrieved either in sequence or via multiple concurrent connections. It is common that a Web server and application server reside on the same hardware, and shared resources are used by the application and Web servers to generate main HTML files as well as to retrieve page-embedded objects. Since the HTTP protocol does not provide any means to delimit the beginning or the end of a Web page, it is very difficult to accurately measure the aggregate resources consumed due to Web page processing at the server side. There is no practical way to effectively measure the service times for *all* page objects, although accurate CPU consumption estimates are required for building an effective application provisioning model. To address this problem, we define a client transaction as a combination of *all* the processing activities at the server side to deliver an entire Web page requested by a client, that is, generate the main HTML file as well as retrieve embedded objects and perform related database queries.

We use client transactions for constructing a "resource consumption" model of the application. The server transactions reflect the main functionality of the application. We use server transactions for analysis of the application performance changes (if any) during the application lifecycle.

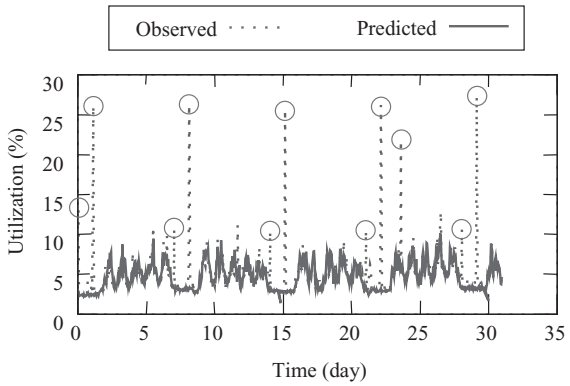

Fig. 1. CPU utilization of OVSD service.

3. TWO MOTIVATING EXAMPLES

Frequent software updates and shortened application development time dramatically increase the risk of introducing poorly performing or misconfigured applications to production environment. Consequently, the effective models for online, automated detection of whether application performance deviates from its normal behavior pattern become a high-priority item on the service provider's "wish list."

Example 1. *Resource Consumption Model Change*. In earlier papers [Zhang et al. 2007a, 2007b], a regression-based approach is introduced for resource provisioning of multitier applications. The main idea is to use a statistical linear regression for approximating the CPU demands of different transaction types (where a transaction is defined as a client transaction). However, the accuracy of the modeling results critically depends on the quality of monitoring data used in the regression analysis: If collected data contain periods of performance anomalies or periods when an updated application exhibits very different performance characteristics, then this can significantly impact the derived transaction cost and can lead to an inaccurate provisioning model. Figure 1 shows the CPU utilization (red line) of the HP Open View Service Desk (OVSD) over a duration of 1 month (each point reflects an 1-hour monitoring period). Most of the time, CPU utilization is under 10%. Note that for each weekend, there are some spikes of CPU utilization (marked with circles in Figure 1) which are related to administrator system management tasks and which are orthogonal to transaction processing activities of the application. Once provided with this information, we can use only weekdays' monitoring data for deriving CPU demands of different transactions of the OVSD service. As a result, the derived CPU cost accurately predicts CPU requirements of the application and can be considered as a normal resource consumption model of the application. Figure 1 shows predicted CPU utilization which is computed using the CPU cost of observed transactions. The predicted CPU utilization accurately models the observed CPU utilization with an exception of weekends' system management periods. However, if we were not aware of

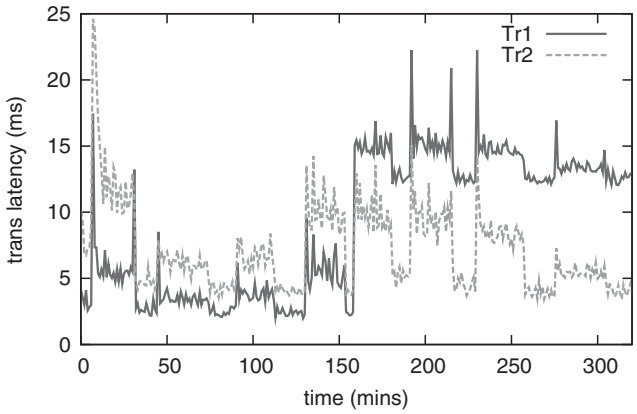

Fig. 2. The transaction latency measured by HP (Mercury) Diagnostics tool.

"performance anomalies" over weekends, and would use all the days (i.e., including weekends) of the 1-month dataset, the accuracy of regression would be much worse (the error would double) and this would significantly impact the modeling results.

Example 2. *Updated Application Performance Change*. Another typical situation that requires a special handling is the analysis of the application performance when it was updated or patched. Figure 2 shows the latency of two application transactions, $Tr1$ and $Tr2$, over time (here, a transaction is defined as a server transaction).

Typically, tools like HP (Mercury) Diagnostics [Mercury Diagnostics] are used in IT environments for observing latencies of the critical transactions and raising alarms when these latencies exceed the predefined thresholds. While it is useful to have insight into the current transaction latencies that implicitly reflect the application and system health, this approach provides limited information on the causes of the observed latencies and cannot be used directly to detect the performance changes of an updated or modified application. The latencies of both transactions vary over time and get visibly higher in the second half of the figure. This does not look immediately suspicious because the latency increase can be a simple reflection of a higher load in the system.

The real story behind this figure is that after timestamp 160, we began executing an updated version of the application code where the processing time of transaction $Tr1$ is increased by 10 ms. However, by looking at the measured transaction latency over time we can not detect this: The reported latency metric does not provide enough information to detect this change.

Problem Definition. Application servers are a core component of a multitier architecture. A client communicates with a service deployed as a multitier application via request-reply transactions. A typical server reply consists of the Web page dynamically generated by the application server. The application server may issue multiple database calls while preparing the reply. Understanding the cascading effects of the various tasks that are sprung by a single request-reply

transaction is a challenging task. Furthermore, significantly shortened time between new software releases further exacerbates the problem of thoroughly evaluating the performance of an updated application.

Typically, a performance analyst needs to perform additional data-cleansing procedures to identify and filter out time periods that correspond to abnormal application performance. These procedures are manual and time consuming, and are based on offline data analysis and ad hoc techniques [Arlitt and Farkas 2005]. Inability to extract "correct" data while eliminating erroneous measurements can lead to inappropriate decisions that have significant technical and business consequences. At the same time, such a cleansing technique cannot identify periods when the updated or patched application performs "normally" but its performance exhibits a significant change. While there could be a clear difference in system performance before and after such an application update, the traditional data-cleansing techniques fail to identify such a change. In this way, while collected measurements are correct and accurate, the overall dataset has measurements related to two, potentially very different, systems: before the application update and after the application update. For accurate and efficient capacity planning and modeling future system performance, one needs to identify a subset of measurements that reflect the updated/modified application.

The goal of this article is to design an online approach that automatically detects the performance anomalies and application changes. Such a method enables a set of useful performance services:

—early warnings on deviations in expected application performance,

—raise alarms on abnormal resource usage,

—create a consistent dataset for capacity planning and modeling future application resource requirements (by filtering out performance anomalies and pointing out the periods of changed application performance).

The next sections present our solution that is based on integration of two complementary techniques: (i) a regression-based transaction model that correlates processed transactions and consumed CPU time to create a resource consumption model of the application; and (ii) an application performance signature that provides a compact model of runtime behavior of the application.

4. REGRESSION-BASED APPROACH FOR DETECTING MODEL CHANGES AND PERFORMANCE ANOMALIES

We use statistical learning techniques to model the CPU demand of the application transactions (client transactions) on a given hardware configuration, to find the statistically significant transaction types, to discover the time segments where the resource consumption of a given application can be approximated by the same regression model, to discover time segments with performance anomalies, and to differentiate among application performance changes and workload-related changes as transactions are accumulated over time.

Prerequisite to applying regression is that a service provider collects the application server access log that reflects all processed client transactions

(i.e., client Web page accesses), and the CPU utilization of the application server(s) in the evaluated system.

4.1 Regression-Based Transaction Model

To capture the site behavior across time we observe a number of different client transactions over a monitoring window t of fixed length L. The transaction mix and system utilization are recorded at the end of each monitoring window. Assuming that there are totally n transaction types processed by the server, we use the following notation.

—T_m denotes the time segment for monitored site behavior and $|T_m|$ denotes the cardinality of the time segment T_m, that is, the number of monitoring windows in T_m;
—$N_{i,t}$ is the number of transactions of the ith type in the monitoring window t, where $1 \leq i \leq n$;
—$U_{CPU,t}$ is the average CPU utilization of application server during this monitoring window $t \in T_m$;
—D_i is the average CPU demand of transactions of the ith type at application server, where $1 \leq i \leq n$;
—D_0 is the average CPU overhead related to activities that "keep the system up". There are operating system processes or background jobs that consume CPU time even when there are no transactions in the system.

From the utilization law, one can easily obtain Eq. (1) for each monitoring window t.

$$D_0 + \sum_{i=1}^{n} N_{i,t} \cdot D_i = U_{CPU,t} \cdot L \tag{1}$$

Let $C_{i,m}$ denote the approximated CPU cost of D_i for $0 \leq i \leq n$ in the time segment T_m. Then, an approximated utilization $U'_{CPU,t}$ can be calculated as

$$U'_{CPU,t} = C_{0,m} + \frac{\sum_{i=1}^{n} N_{i,t} \cdot C_{i,m}}{L}, \tag{2}$$

To solve for $C_{i,m}$, one can choose a regression method from a variety of known methods in the literature. A typical objective for a regression method is to minimize either the absolute error or the squared error. In all experiments, we use the nonnegative Least Squares Regression (nonnegative LSQ) provided by MATLAB to obtain $C_{i,m}$. This nonnegative LSQ regression minimizes the error

$$\epsilon_m = \sqrt{\sum_{t \in T_m} (U'_{CPU,t} - U_{CPU,t})^2}.$$

such that $C_{i,m} \geq 0$.

When solving a large set of equations with collected monitoring data over a large period of time, a direct (naive) linear regression approach would attempt to set nonzero values for as many transactions as it can to minimize the error when the model is applied to the training set. However, this may lead to poor prediction accuracy when the model is later applied to other datasets, as the

model may have become too finely tuned to the training set alone. In statistical terms, the model may "overfit" the data if it sets values to some coefficients to minimize the random noise in the training data rather than to correlate with the actual CPU utilization. In order to create a model which utilizes only the statistically significant transactions, we use stepwise linear regression [Draper and Smith 1998] to determine which set of transactions are the best predictors for the observed CPU utilization.

The algorithm initializes with an "empty" model which includes none of the transactions. At each following iteration, a new transaction is considered for inclusion in the model. The best transaction is chosen by adding the transaction which results in the lowest mean squared error when it is included.

For each N_i $(0 \leq i \leq n)$ we try to solve the set of equations in the form

$$D_0 + N_{i,t} \cdot D_i = U_{CPU,t} \cdot L, \tag{3}$$

while minimizing the error

$$\epsilon_i = \sqrt{\sum_{t \in T_m} (U'_{CPU,t} - U_{CPU,t})^2}.$$

Once we performed this procedure for all the transactions, we select the transaction N_k $(0 \leq k \leq n)$ which results in the lowest mean squared error, that is, such that

$$\epsilon_k = \min_{0 \leq i \leq N} \epsilon_i.$$

Then, transaction N_k is added to the empty set. After that, the next iteration is repeated to choose the next transaction from the remaining subset to add to the set in a similar way.

Before the new transaction is included in the model, it must pass an F-test which determines if including the extra transaction results in a statistically significant improvement in the model's accuracy. If the F-test fails, then the algorithm terminates since including any further transactions cannot provide a significant benefit. The coefficients for the selected transactions are calculated using the linear regression technique described before. The coefficient for the transactions not included in the model is set to zero.

Typically, for an application with n transactions, one needs at least $n + 1$ samples to do regression using all n transactions. However, since we do transaction selection using a stepwise linear regression and an F-test, we can do regression by including only a subset of n transactions in the regression model. This allows us to apply regression without having to wait all $n + 1$ samples.

4.2 Algorithm Outline

Using statistical regression, we can build a model that approximates the overall resource cost (CPU demand) of application transactions on a given hardware configuration. However, the accuracy of the modeling results critically depends on the quality of monitoring data used in the regression analysis: If the collected data contain periods of performance anomalies or periods when an updated application exhibits very different performance characteristics, then this

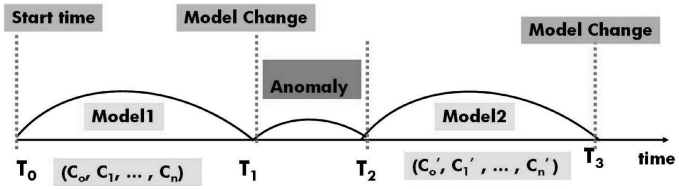

Fig. 3. Finding optimal segmentation and detecting anomalies.

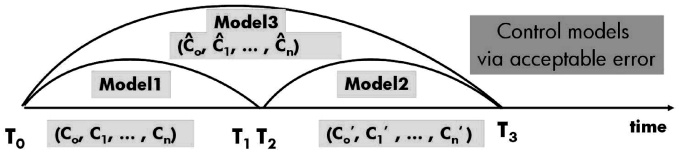

Fig. 4. Model reconciliation.

can significantly impact the derived transaction cost and can lead to an inaccurate approximation model. The challenge is to design an online method that alarms service providers of model changes related to performance anomalies and application updates. Our method has the following three phases.

—*Finding the Optimal Segmentation*. This stage of the algorithm identifies the time points when the transaction cost model exhibits a change. For example, as shown in Figure 3, the CPU costs of the transactions $(Tr_1, Tr_2, \ldots, Tr_n)$ during the time interval (T_0, T_1) are defined by a model $(C_0, C_1, C_2, \ldots, C_n)$. After that, for a time interval (T_1, T_2) there was no a single regression model that provides the transaction costs within a specified error bound. This time period is signaled as having anomalous behavior. As for time interval (T_2, T_3), the transaction cost function is defined by a new model $(C'_0, C'_1, C'_2, \ldots, C'_n)$.

—*Filtering Out the Anomalous Segments*. Our goal is to continuously maintain the model that reflects a normal application resource consumption behavior. At this stage, we filter out anomalous measurements identified in the collected dataset, for example, the time period (T_1, T_2) that corresponds to anomalous time fragment as shown in Figure 3.

—*Model Reconciliation*. After anomalies have been filtered out, one would like to unify the time segments with no application change/update/modification by using a single regression model: We attempt to "reconcile" two different segments (models) by using a new common model as shown in Figure 4.

We try to find a new solution (new model) for combined transaction data in (T_0, T_1) and (T_2, T_3) with a given (predefined) error. If two models can be reconciled then an observed model change is indicative of the workload change and not of the application change. We use the reconciled model to represent application behavior across different workload mixes.

If the model reconciliation does not work, then it means these models indeed describe different consumption models of application over time, and it is indicative of an actual application performance change.

4.3 Online Algorithm Description

This section describes the three phases of the online model change and anomaly detection algorithm in more detail.

(1) *Finding the optimal segmentation*. This stage of the algorithm identifies the time points where the transaction cost model exhibits a change. In other words, we aim to divide a given time interval T into time segments T_m ($T = \bigcup T_m$) such that within each time segment T_m the application resource consumption model and the transaction costs are similar. We use a cost-based statistical learning algorithm to divide the time into segments with a similar regression model. The algorithm is composed of two steps:

—construction of weights for each time segment T_m;
—dynamic programming to find the optimum segmentation (that covers a given period T) with respect to the weights.

The algorithm constructs an edge with a weight, w_m, for each possible time segment $T_m \subseteq T$. This weight represents the cost of forming the segment T_m. Intuitively, we would like the weight w_m to be small if the resource cost of transactions in T_m can be accurately approximated with the same regression model, and to be large if the regression model has a poor fit for approximating the resource cost of all the transactions in T_m.

The weight function, w_m is selected as a Lagrangian sum of two cost functions: $w_{1,m}$ and $w_{2,m}$, where

—the function $w_{1,m}$ is the total regression error over T_m:

$$w_{1,m} = \sqrt{\sum_{t \in T_m} (U'_{CPU,t} - U_{CPU,t})^2},$$

—the function $w_{2,m}$ is a length penalty function. A length penalty function penalizes shorter time intervals over longer time intervals and discourages the dynamic programming from breaking the time into segments of very short length (since the regression error can be significantly smaller for a shorter time segments). It is a function that decreases as the length of the interval T_m increases. We set it to a function of the entropy of segment length as

$$w_{2,m} = -(|T_m|) \cdot \log(|T_m|/|T|).$$

Our goal is to divide a given time interval T into time segments T_m ($T = \bigcup T_m$) that minimize the Lagrangian sum of $w_{1,m}$ and $w_{2,m}$ over the considered segments, that is, the segmentation that minimiz

$$W_1(T) + \lambda W_2(T), \tag{4}$$

where the parameter λ is the Lagrangian constant that is used to control the *average* regression error ϵ_{allow} (averaged over T) allowed in the model, and

$$W_1(T) = \sum_m w_{1,m} \quad \text{and} \quad W_2(T) = \sum_m w_{2,m}.$$

We denote the set of all possible segmentations of T into segments by S. For instance, for T with three monitoring windows, t_1,t_2,t_3, S would have 4 elements: (t_1,t_2,t_3), ($t_1 \bigcup t_2$,t_3), (t_1, $t_2 \bigcup t_3$), and ($t_1 \bigcup t_2 \bigcup t_3$). Each element of the

set S consists of segments, that is, T_m's. For instance, $(t_1, t_2 \bigcup t_3)$ consists of the segments t_1 and $t_2 \bigcup t_3$.

The minimization problem defined by formula (4) can be expressed as

$$s^* = \text{argmin}_{s \in S} W(s) = W_1(s) + \lambda W_2(s), \tag{5}$$

where λ is the fixed Lagrangian constant, s denotes an element of S, and

$$W_1(s) = \sum_{T_m \in s} w_{1,m}$$

and

$$W_2(s) = \sum_{T_m \in s} w_{2,m}.$$

We first provide the segmentation algorithm for a fixed value of λ (we have an explanation after the algorithm, how to find the "right" value of λ to be used in the algorithm for a given, allowable regression error ϵ).

In the algorithm, l represents the index of the monitoring window, where the monitoring windows are denoted by t_l. We represent segments using the following form: $[t_j, t_k]$. It indicates the segment that extends from monitoring window t_j to monitoring window t_k. W^{t_l} denotes the minimum Lagrangian weight sum for segmentation of the first l time points, that is, t_1,t_2, ... t_l.

(1) Set $W^{t_1} = 0$, and set $l = 2$.
(2) For $1 \leq j < l$, set $w_{1,[t_j,t_k]}$ to the total regression error when a regression model is fit over the time samples in the segment $[t_j, t_k]$, then set
$$w_{2,[t_j,t_k]} = -|t_k - t_j| \log(|t_k - t_j|/|T|)$$
and, finally, set
$$w_{[t_j,t_k]} = w_{1,[t_j,t_k]} + \lambda w_{2,[t_j,t_k]}.$$
(3) Set $W^{t_l} = min_{1 \leq j \leq l-1}(W^{t_j} + w_{[t_j,t_l]})$.
(4) Set $j^* = \text{argmin}_{1 \leq j \leq l-1}(W^{t_j} + w_{[t_j,t_l]})$.
(5) Then the optimum segmentation is the segmentation result for j^* (already obtained in previous iterations) augmented by the single segment from j^* to l.
(6) Set $l = l + 1$, and return to step 2.

In the preceding algorithm, step 2 sets the values for the w_1, w_2, and w terms, and step 3 and step 4 apply dynamic programming to find the minimum cost W^l of segmentation up to the l^{th} time sample.

The algorithm shows the best segmentation for a fixed value of λ. Here is the additional sequence of steps to find the appropriate value of λ for the use in the algorithm. In our implementation, we first decide on an allowable regression error ϵ (typically, provided by the service provider), and then seek the λ that gives the best segmentation for that allowable error ϵ by iterating over different values of $\lambda = \lambda_0, \lambda_1, \ldots, \lambda_k$. In particular, if the algorithm for $\lambda_0 = 1$ results in the optimal segmentation with regression error *greater* than ϵ, then we choose $\lambda_1 = 2 \cdot \lambda_0$, and repeat the segmentation algorithm. Once we find a value λ_k

that results in the optimal segmentation with regression error *smaller* than ϵ, then we use a binary search between values λ_{k-1} and λ_k to find the best value of λ for a use in the algorithm with a given allowable regression error ϵ.

Let us consider an example to explain the intuition for how the Eq. (5) works. Let us first consider the time interval T with no application updates or changes. Let time interval T be divided into two consecutive time segments T_1 and T_2.

First of all, $W_1(T_1) + W_1(T_2) \leq W(T)$, hence there are two possibilities.

—One possibility is that a regression model constructed over T is also a good fit over time segments T_1 and T_2, and the combined regression error of this model over time segments T_1 and T_2 is approximately equal to the total regression error over the original time interval T.

—The other possibility is that there could be different regression models that are constructed over shorter time segments T_1 and T_2 with the sum of regression errors smaller than a regression error obtained when a single regression model is constructed over T.

For the second possibility, the question is whether the difference is due to a noise or small outliers in T, or do segments T_1 and T_2 indeed represent different application behaviors, that is, "before" and "after" the application modification and update.

This is where the W_2 function in Eq. (4) comes into play. The term $\log(|T_m|/|T|)$ is a convex function of $|T_m|$. Therefore, each time a segment is split into multiple segments, W_2 increases. This way, the original segment T results in the smallest W_2 compared to any subset of its segments, and λ can be viewed as a parameter that controls the amount of regression error allowed in a segment.

By increasing the value of λ, we allow a larger W_1, regression error, in the segment. This helps in reconciling T_1 and T_2 into a single segment representation T. In such a way, by increasing the value of λ one can avoid the incorrect segmentations due to noise or small outliers in the data T. When an average regression error over a single segment T is within the allowable error ϵ_{allow} (ϵ_{allow} is set by a service provider), the overall function (4) results in the smallest value for the single time segment T compared to the values computed to any of its subsegments, for instance, T_1 and T_2. Therefore, our approach groups all time segments defined by the same CPU transaction cost (or the same regression model) into a single segment.

By decreasing the value of λ, one can prefer the regression models with a smaller total regression error on the data, while possibly increasing the number of segments over the data.

There is a trade-off between the allowable regression error (it is a given parameter for our algorithm) and the algorithm outcome. If the allowable regression error is set too low then the algorithm may result in a high number of segments over data, with many segments being neither anomalies or application changes (these are the false alarms, typically caused by significant workload changes). From the other side, by setting the allowable regression error too high, one can miss a number of performance anomalies and application changes that happened in these data and masked by the high allowable error.

(2) *Filtering out the anomalous segments*. An anomalous time segment is one where observed CPU utilization cannot be explained by an application workload, that is, measured CPU utilization cannot be accounted for by the transaction CPU cost function. This may happen if an unknown background process(es) is using the CPU resource either at a constant rate (e.g., using 40% of the CPU at every monitoring window during some time interval) or randomly (e.g., the CPU is consumed by the background process at different rates at every monitoring window). It is important to be able to detect and filter out the segments with anomalous behavior, as otherwise the anomalous monitoring windows will corrupt the regression estimations of the time segments with normal behavior. Furthermore, detecting anomalous time segments provides an insight into the service problems and a possibility to correct the problems before they cause major service failure.

We consider a time segment T_m as anomalous if one of the following conditions take place.

—*The constant coefficient, $C_{0,m}$, is large*. Typically, $C_{0,m}$ is used in the regression model to represent the average CPU overhead related to "idle system" activities. There are operating system processes or system background jobs that consume CPU time even when there is no transaction in the system. The estimate for the "idle system" CPU overhead over a monitoring window is set by the service provider. When $C_{0,m}$ exceeds this threshold a time segment T_m is considered as anomalous.

—*The segment length of T_m is short*, indicating that a model does not have a good fit that ensures the allowed error threshold. Intuitively, the same regression model should persist over the whole time segment between the application updates/modifications unless something else, anomalous, happens to the application consumption model and it manifests itself via the model changes.

(3) *Model reconciliation*. After anomalies have been filtered out, one would like to unify the time segments with no application change/update/modification by using a single regression model. This way, it is possible to differentiate between the segments with application changes from the segments which are the parts of the same application behavior and were segmented out by the anomalies in between. In such cases, the consecutive segments can be reconciled into a single segment after the anomalies in the data are removed. If there is an application change, on the other hand, the segments will not be reconciled, since the regression model that fits to the individual segments will not fit to the overall single segment without exceeding the allowable error (unless the application performance change is so small that it still fits within the allowable regression error).

4.4 Algorithm Complexity

The complexity of the algorithm is $O(M^2)$, where M is the number of time samples collected so far. This is problematic since the complexity is quadratic in a term that increases as more time samples are collected. In our case study, we have not experienced a problem as we used only 30 hours of data with 1-minute

Table I. Testbed Components

	Processor	RAM
Clients (Emulated-Browsers)	Pentium D / 6.4 GHz	4 GB
Front Server - Apache/Tomcat 5.5	Pentium D / 3.2 GHz	4 GB
Database Server - MySQL5.0	Pentium D / 6.4 GHz	4 GB

intervals, namely $M = 1800$. However, in measuring real applications over long periods of time, complexity is likely to become challenging. To avoid this, one solution might retain only the last X samples, where X should be a few orders larger than the number of transaction types n and/or cover a few weeks/months of historic data. This way, one would have a sufficiently large X to get accurate regression results, yet the complexity will not be too large.

5. CASE STUDY

In this section, we validate the proposed regression-based transaction model as an online solution for anomaly detection and performance changes in application behavior and discuss limitations of this approach. The next subsection describes the experimental environment used in the case study as well as a specially designed workload for evaluating the proposed approach.

5.1 Experimental Environment

In our experiments, we use a testbed of a multitier e-commerce site that simulates the operation of an online bookstore, according to the classic TPC-W benchmark [TPC-W Benchmark]. This allows to conduct experiments under different settings in a controlled environment in order to evaluate the proposed anomaly detection approach. We use the terms "front server" and "application server" interchangeably in this article. Specifics of the software/hardware used are given in Table I.

Typically, client access to a Web service occurs in the form of a *session* consisting of a sequence of consecutive individual requests. According to the TPC-W specification, the number of concurrent sessions (i.e., customers) or emulated browsers (EBs) is kept constant throughout the experiment. For each EB, the TPC-W benchmark statistically defines the user session length, the user think time, and the queries that are generated by the session. The database size is determined by the number of items and the number of customers. In our experiments, we use the default database setting, that is, the one with 10,000 items and 1,440,000 customers.

TPC-W defines 14 different transactions which are classified as either of browsing or ordering types as shown in Table II. We assign a number to each transaction (shown in parentheses) according to their alphabetic order. Later, we use these transaction *id*s for presentation convenience in the figures.

According to the weight of each type of activity in a given traffic mix, TPC-W defines 3 types of traffic mixes as follows:

—the *browsing mix* with 95% browsing and 5% ordering;
—the *shopping mix* with 80% browsing and 20% ordering;
—the *ordering mix* with 50% browsing and 50% ordering.

Table II. 14 Basic Transactions and Their Types in TPC-W

Browsing Type	Ordering Type
Home (8)	Shopping Cart (14)
New Products (9)	Customer Registration (6)
Best Sellers (3)	Buy Request (5)
Product detail (12)	Buy Confirm (4)
Search Request (13)	Order Inquiry (11)
Execute Search (7)	Order Display (10)
	Admin Request (1)
	Admin Confirm (2)

1. initialize a variable $dur \leftarrow$ 3hours
2. while ($dur > 0$) do
  a. set the execution time $exe\_dur \leftarrow$ random(20min, 30min)
  b. set the number of EBs $curr\_EBs \leftarrow$ random(150, 700)
  c. execute shopping mix with $curr\_EBs$ for exe_dur time
  d. set the sleep time $sleep\_dur \leftarrow$ random(10min, 20min)
  e. sleep for $sleep\_dur$ time
  f. adjust $dur \leftarrow dur$ - (exe_dur+$sleep\_dur$)

Fig. 5. The pseudocode for the random process.

One drawback of directly using the transaction mixes described previously in our experiments is that they are *stationary*, that is, the transaction mix and load do not change over time. Since real enterprise and e-commerce applications are typically characterized by *nonstationary* transaction mixes (i.e., with changing transaction probabilities in the transaction mix over time) under variable load [Douglis et al. 1997; Cherkasova and Karlsson 2001; Stewart et al. 2007] we have designed an approach that enables us to generate nonstationary workloads using the TPC-W setup. To generate a nonstationary transaction mix with variable transaction mix and load we run 4 processes as follows:

—the three concurrent processes each executing one of the standard transaction mixes (i.e., browsing, shopping and ordering respectively) with the arbitrary fixed number of EBs (e.g, 20, 30, and 50 EBs, respectively). We call them *base* processes;
—the fourth, so-called *random* process executes one of the standard transaction mixes (in our experiments, it is the shopping mix) with a random execution period while using a random number of EBs for each period. To navigate and control this random process we use specified ranges for the "random" parameters in this workload. The pseudocode of this random process is shown in Figure 5 (the code also shows parameters we use for the nonstationary mix in this article).

Due to the fourth random process the workload is nonstationary and the transaction mix and load vary significantly over time.

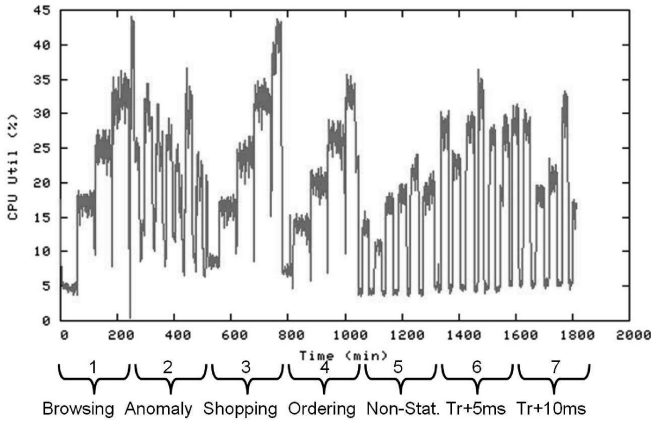

Fig. 6. 30-hour TPC-W workload used in the case study.

In order to validate the online anomaly detection and application change algorithm, we designed a special 30-hour experiment with TPC-W that has 7 different workload segments shown in Figure 6, which are defined as follows.

(1) The browsing mix with the number of EBs equal to 200, 400, 600, 800, and 1000 respectively.
(2) In order to validate whether our algorithm correctly detects performance anomalies, we generated a special workload with nonstationary transaction mix as described earlier and an additional CPU process (that consumes random amount of CPU) on a background.
(3) The shopping mix with the number of EBs equal to 200, 400, 600, 800, and 1000 respectively.
(4) The ordering mix with the number of EBs equal to 200, 400, 600, 800, and 1000 respectively.
(5) The nonstationary TPC-W transaction mix described earlier in this section.
(6) In order to validate whether we can automatically recognize the application change, we modified the source code of the "Home" transaction (the 8th transaction) in TPC-W by inserting a controlled CPU loop into the code of this transaction and increasing its service time by 5 ms. Using this modified code, we performed experiment with the nonstationary TPC-W transaction mix described before.
(7) Another experiment with the modified TPC-W benchmark, where the service time of the "Home" transaction is increased by 10 ms and the nonstationary TPC-W transaction mix described previously.

5.2 Validation of Online Regression-Based Algorithm

We applied our online regression-based algorithm to the special 30-hour workload shown in Figure 6. The expectations are that the algorithm should exhibit a model change and issue an alarm for anomaly workload described by segment 2 in Figure 6. Then a single model should represent workload segments

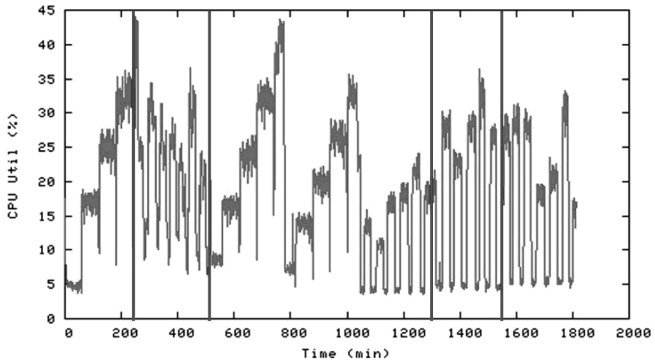

Fig. 7. Model changes in the studied workload.

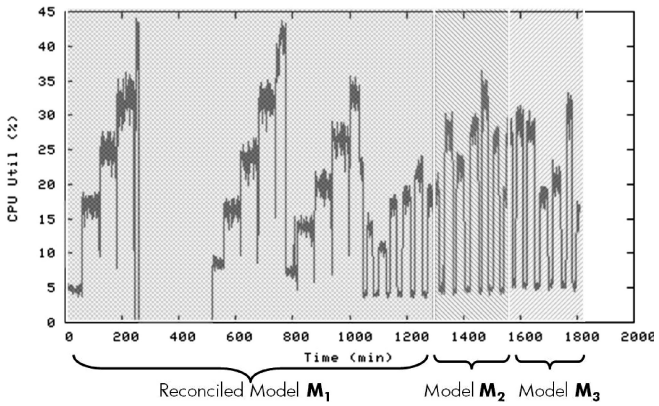

Fig. 8. Model reconciliation in the studied workload.

3, 4, and 5 as shown in Figure 6. Moreover, the algorithm should reconcile segments 1, 3, 4, and 5 with a single model (since they are all generated with the same application operating under different workloads), while issuing the model changes for application modifications described by segments 6 and 7.

We experimented with two values of allowable error in our experiments: $\epsilon^1_{allow} = 3\%$ and $\epsilon^2_{allow} = 1\%$ to demonstrate the impact of error setting and stress the importance of tuning this value.

When we used $\epsilon^1_{allow} = 3\%$, the algorithm had correctly identified 4 major model changes as shown in Figure 7. In fact, for the second segment there were 42 model changes (not shown in this figure to simplify the presentation) with maximum segment being 5 epochs.

The algorithm accurately detected that the whole segment 2 is anomalous. Then the tool correctly performed the model reconciliation for the consecutive segments around the anomalous segment as shown in Figure 8.

Finally, the algorithm correctly raised alarms on the application change when the regression model has changed and could not be reconciled (last two segments in Figure 8).

When we used $\epsilon^1_{allow} = 1\%$, the algorithm had identified 6 major model changes: In addition to 4 model changes shown in Figure 7 the algorithm reported 2 extra segments at timestamps 790 and 1030 (separating apart segments 3, 4, and 5 shown in Figure 6) that correspond to workload changes and that are false alarms. It is important to use the appropriate error setting to minimize the number of false alarms. When the allowable error is set too small the proposed algorithm often overreacts and identifies model changes that correspond to significant workload changes under the same, unmodified application. In the next section, we will introduce a complementary technique that helps to get an insight in whether the model change is indeed an application change or whether it rather corresponds to a workload change. This technique can also be used for error tuning in the earlier described online algorithm.

The power of the introduced regression-based approach is that it is sensitive to accurately detect a difference in the CPU consumption model of application transactions. However, it cannot identify the transactions that are responsible for this resource consumption difference. To complement the regression-based approach and to identify the transactions that cause the model change we use a different method described in the next section and which is based on building a representative application performance signature.

6. DETECTING TRANSACTION PERFORMANCE CHANGE

Nowdays there is a new generation of monitoring tools, both commercial and research prototypes, that provide useful insights into transaction activity tracking and latency breakdown across different components in multitier systems. However, typically such monitoring tools just report the measured transaction latency and provide an additional information on application server versus database server latency breakdown. Using this level of information it is often impossible to decide whether an increased transaction latency is a result of a higher load in the system or whether it can be an outcome of the recent application modification, and is directly related to the increased processing time for this transaction type.

In this section, we describe an approach based on an *application performance signature* that provides a compact model of runtime behavior of the application. The application signature is built based on new concepts: the *transaction latency profiles* and *transaction signatures*. These become instrumental for creating an application signature that accurately reflects important performance characteristics. Comparing new application signature against the old application signature allows detecting transaction performance changes.

6.1 Server Transaction Monitoring

Many enterprise applications are implemented using the J2EE standard, a Java platform which is used for Web application development and designed to meet the computing needs of large enterprises. For transaction monitoring we use the HP (Mercury) Diagnostics [Mercury Diagnostics] tool which offers a monitoring solution for J2EE applications. The Diagnostics tool consists of

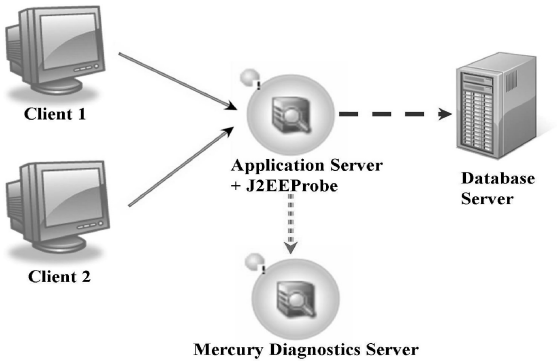

Fig. 9. Multitier application configuration with the Diagnostics tool.

two components: the Diagnostics Probe and the Diagnostics Server as shown in Figure 9.

The Diagnostics tool collects performance and diagnostic data from applications without the need for application source-code modification or recompilation. It uses bytecode instrumentation and industry standards for collecting system and JMX metrics. Instrumentation refers to bytecode that the Probe inserts into the class files of the application as they are loaded by the class loader of the virtual machine. Instrumentation enables a Probe to measure execution time, count invocations, retrieve arguments, catch exceptions, and correlate method calls and threads.

The J2EE Probe shown in Figure 9 is responsible for capturing events from the application, aggregating the performance metrics, and sending these captured performance metrics to the Diagnostics Server. We have implemented a Java-based processing utility for extracting performance data from the Diagnostics Server in real time and creating a so-called application log that provides a complete information on all transactions processed during the monitoring window, such as their overall latencies, outbound calls, and the latencies of the outbound calls. In a monitoring window, Diagnostics collects the following information for each transaction type:

—a transaction count;

—an average overall transaction latency for observed transactions.[1] This overall latency includes transaction processing time at the application server as well as all related query processing at the database server, that is, latency is measured from the moment of the request arrival at the application server to the time when a prepared reply is sent back by the application server; see Figure 10;

—a count of outbound (database) calls of different types;

—an average latency of observed outbound calls (of different types). The average latency of an outbound call is measured from the moment the database

[1]Note that here a latency is measured for the server transaction (see the difference between client and server transactions described in Section 2).

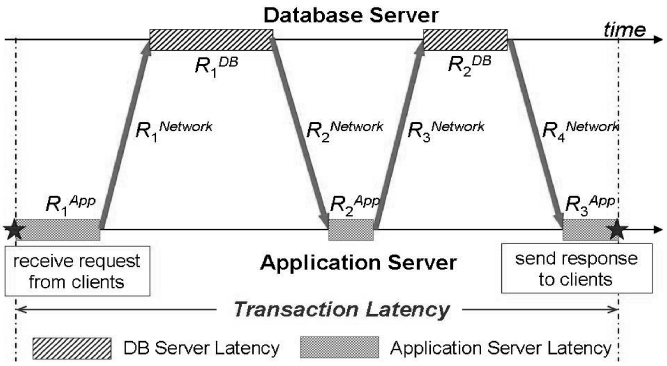

Fig. 10. The transaction latency measured by the Diagnostics tool.

request is issued by the application server to the time when a prepared reply is returned back to the application server, that is, the average latency of the outbound call includes database processing and communication latency.

The transaction latency consists of the waiting and service times across the different tiers (e.g., front and database servers) that a transaction flows through. Let R_i^{front} and R_i^{DB} be the average latency for the ith transaction type at the front and database servers, respectively. We then have the transaction latency breakdown calculated as follows.

$$\begin{aligned} R_i &= R_i^{front} + R_i^{DB} \\ &= R_i^{front} + \frac{\sum_{j=1}^{P_i} N_{i,j}^{DB} * R_{i,j}^{DB}}{N_i} \end{aligned} \tag{6}$$

Using this equation we can easily compute R_i^{front}.

6.2 Application Performance Signature

In this section, we describe how to create a representative application signature that compactly reflects important performance characteristics of application. As shown in Mi et al. [2008], we can compute the transaction service times using measured transaction latencies and corresponding system utilization. For a concrete transaction type Tr_i, we have a relationship based on transaction service time S_i, transaction residence time R_i (measured at the application server, i.e., R_i corresponds to R_i^{front} in formula 6), and utilization U of the system (the application server).

$$R_i = S_i/(1-U) \tag{7}$$

Therefore, it is equivalent to

$$S_i = R_i * (1-U). \tag{8}$$

Since in real production system we collect measured latencies for each transaction type i over different monitoring windows, we have multiple equations that reflect transaction latencies at different CPU utilization points as shown next

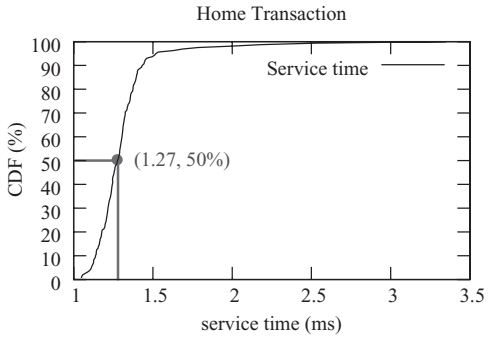

Fig. 11. Service time CDF of a typical server transaction.

(since we collect CPU utilization expressed in percents, we need to divide it by 100 to use correctly in Eq. (8)).

$$\begin{aligned} S_i &= R_{i,1} * (1 - U_1/100) \\ S_i &= R_{i,2} * (1 - U_2/100) \\ &\ldots \ \ldots. \end{aligned} \tag{9}$$

Our goal is to find the solution that is the best fit for the overall equation set (9). A linear regression-based (LSR) method can be chosen to solve for S_i. However, there are two reasons why we chose a different method. First, a number of outliers that often present in production data could significantly affect the accuracy of the final solution, as LSR aims to minimize the absolute error across all points. Second, there may be a significant difference in the number of transactions contributing to different CPU utilization points. LSR aims to minimize the absolute error across the equations, and it treats all these equations equally.

Therefore, we propose another method to compute the service time S_i for the ith transaction type. By solving $S_i = R_{i,k} * (1 - U_k/100)$ in Eq. (9), a set of solutions S_i^k is obtained for different utilization points U_k in the transaction latency profile. We generate a Cumulative Distribution Function (CDF) for S_i. Intuitively, since we conjecture that each transaction type is uniquely characterized by its service time, then we should see a curve similar to shown in Figure 11 with a large number of similar points in the middle and some outliers in the beginning and the tail of the curve. We then select the 50th percentile value as the solution for S_i as most representative.[2] The 50th percentile heuristics works well for all transactions in our study.

Finally, an *application performance signature* is created.

$$\begin{aligned} Tr_1 &\longrightarrow S_1 \\ Tr_2 &\longrightarrow S_2 \\ \ldots &\quad\ \ldots \\ Tr_n &\longrightarrow S_n \end{aligned}$$

[2]Selecting the mean of S_i allows the outliers (i.e., the tail of the distribution) to influence our service time extrapolation, which is not desirable. Because of the shape of the CDF curve, the selection of the 50th percentile is a good heuristics.

As shown in Mi et al. [2008], such an application signature uniquely reflects the application transactions and their CPU requirements and is invariant for different workload types. The application signature compactly represents a model of application runtime behavior.

Continuous calculation of the application signature allows us to detect events such as software updates that may significantly affect transaction execution time. By comparing the new application signature against the old, one can detect transaction performance changes and analyze their impacts.

The application signature technique is complementary to the regression-based resource consumption model described in Section 4. For example, it is not capable of detecting abnormal resource consumption caused by processes unrelated to the application and its transaction processing.

6.3 Case Study Continued: Application Signature Analysis

The power of regression-based approach is that it is sensitive and accurate to detect a difference in the CPU consumption model of application transactions. The proposed online algorithm correctly raises alarms on the application change when the regression model has changed and could not be reconciled (last two segments in Figure 8). However, it cannot identify those transactions that are responsible for this resource consumption difference. To complement the regression-based approach and to identify the transactions that cause the model change we use the complementary method based on the application performance signature. Comparison of the new application signature against the old one allows efficient detection of transactions with performance changes.

The application signature stays unchanged for the first 5 segments of the studied 30-hour workload. It is plotted in Figure 12 as the baseline. The new application signatures for the 6th and 7th segments reflect the change in service time of the "Home" (8th) transaction, while for the other transactions their service times stay unchanged. Thus, indeed, 6th and 7th segments correspond to the application change.

When we used $\epsilon^1_{allow} = 1\%$, the regression-based algorithm had identified 6 major model changes: In addition to four model changes shown in Figure 7 the algorithm reported two extra segments at timestamps 790 and 1030 that correspond to workload changes and that are false alarms as follows from the application signature method. One can use the application signature technique while performing the allowable error tuning for regression-based approach: It enables a quick and efficient performance analysis of transactions' service times while the application is executing in the production environment.

The preceding experiments show that the proposed integrated framework of regression-based transaction model and application signature provides a simple and powerful online solution for anomaly detection and analysis of essential performance changes in application behavior. The proposed algorithms provide necessary data-analysis and data-cleansing support for automated capacity planning described in the next section.

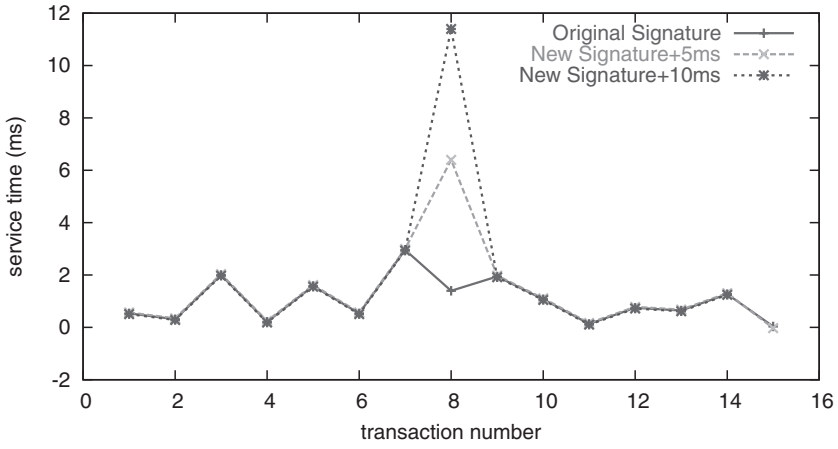

Fig. 12. Original application signature vs. the application signatures of the modified application.

7. AUTOMATED CAPACITY PLANNING

There is a continuing trend in enterprise computing to automate different infrastructure management and decision making processes. It is motivated by the fact that these management tasks account for half of enterprise IT budget [Chou 2004].

Performance management and maintenance operations come with more risk and an increased potential for serious damage if they are wrong. Imagine introducing a change or making an adjustment that results in dramatically slow application response times or/and creates significantly higher resource consumption in the shared infrastructure. Today's IT and services departments simply cannot afford to operate without some help in the form of a tool providing automated, comprehensive, and highly informative performance management and capacity planning functions.

Many enterprise services are built using the three-tier architecture paradigm. It is important to design effective and accurate performance models that predict behavior of multitier applications when they are placed in enterprise production environment and operate under a real workload mix. Understanding application behavior and its changes during the application lifecycle is a part of this problem: One needs to separate performance issues that are caused by a high load of incoming workload from the performance problems caused by possible errors, anomalies, or inefficiencies that often occur during the application updates and new software releases.

To automate capacity planning for existing production system with real workload mix we propose a refined capacity planning framework that is based on the following three components shown in Figure 13.

—*Online Workload Profiler*. Our workload profiler is built on top of HP (Mercury) Diagnostics tool. It extracts information on number of processed transactions and their outbound DB calls.

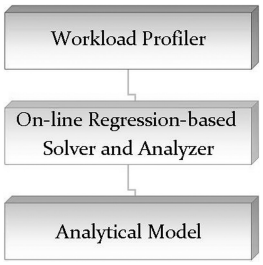

Fig. 13. The overall capacity planning framework.

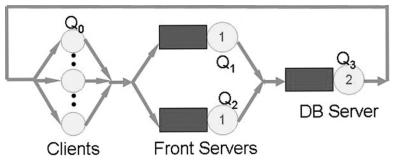

Fig. 14. The queuing model of the enterprise three-tier application.

If the solution has multiple application servers in the configuration then there are multiple Diagnostics Probes installed at each server and they collect data independently at these application servers supported by (possibly) heterogeneous machines with different CPU speeds. Data processing is done for each probe separately.

—*Online Regression-Based Solver and Analyzer.* This module uses the information provided by the workload profiler and performs an automated anomaly detection and application change analysis.

Our online regression algorithm discovers the time segments where the resource consumption of a given application cannot be approximated by the same regression model and filters out time segments with performance anomalies. It differentiates among application performance changes and workload-related changes as transactions are accumulated over time. The application signature approach refines the time intervals where the application might have been updated in order to create a consistent dataset for capacity planning and modeling stage. This dataset is used by the solver to approximate the up-to-date resource cost (CPU demand) of the application transactions on a given hardware as well as estimates the CPU cost of each outbound DB call.

Such an approach aims to eliminate error-prone manual processes for data cleansing in order to support a fully automated solution.

—*Analytical Model*. For capacity planning of multitier applications with session-based workloads, an analytical model based on network of queues (shown in Figure 14) is developed, where the queues represent different tiers of the application.

Additionally, this model uses a set of refined measurements for the DB tier that are provided by the workload profiler and HP (Mercury) Diagnostics tool. The results of the regression method in the previous module are used to parameterize an analytic model of queues for capacity planning and resource

Inputs:
N = number of clients
Z = think time
L = number of servers
S_i = service time per visit to the i-th queue
V_i = number of visits to the i-th queue

Outputs:
X = system throughput
Q_i = average number of jobs at the i-th queue
R_i = average response time of the i-th queue
R = system response time (excluding think time)
U_i = utilization of the i-th queue

1. Initialization: for $i = 1$ to L do $Q_i \leftarrow 0$
2. Iterations:
for $n = 1$ to N do
a. for $i = 1$ to L do
$$R_i = S_i(1 + Q_i)$$
b. $$R = \sum_{i=1}^{L} R_i V_i$$
c. $$X = \frac{n}{Z + R}$$
d. for $i = 1$ to L do $Q_i = XV_iR_i$
3. for $n = 1$ to L do
a. $X_i = XV_i$
b. $U_i = XS_iV_i$

Fig. 15. The MVA algorithm.

provisioning tasks. For most production multitier services CPU is a typical system bottleneck. However, in practice, when one needs to make a projection of the maximum achievable system throughput, additional "back of the envelope" computations for estimating memory and network requirements under the maximum number of concurrent clients are required to justify this maximum throughput projection. These computations can augment the results of the analytic model module.

Due to the session-based client behavior, a multitier system is usually modeled as a closed system with a network of queues (see Figure 14). The number of clients in the system is fixed. When a client receives the response from the server, it issues another request after certain think time. This think time is modeled as an infinite server Q_0 in Figure 14. Once the service time in each queue is obtained, this closed system can be solved efficiently using Mean-Value Analysis (MVA) [Menasce et al. 1994]. MVA is based on the key assumption that when a new request enters a queue, this request sees the same average system statistics in the system as without this new request. Figure 15 presents a description of the detailed MVA algorithm.

The visit ratio V_i (definition in Figure 15) is controlled by the load balancing policy. For example, if the load balancing policy used is equally partitioning the

transactions across all servers, then the number of visits V_s to server s in tier l is equal to $1/m_l$, where m_l is the number of servers in tier l.

Note that the original MVA (as in Figure 15) takes the number of clients N as input, and computes the average performance metrics for a system with N clients. In capacity planning, the number of clients is unknown. In the contrary, the model needs to be solved for exactly this unknown variable. Here, we assume that the Service Level Agreement (SLA) specifies a threshold Γ_R (i.e., upper bound) of the average transaction response time. Then the condition in step 2 of MVA is changed to the following condition: "while $R \leq \Gamma_R$ do".

Workload characterization of real traces [Zhang et al. 2007a] shows that the workload mix changes over time, and hence the service time could not be modeled as a fixed distribution for the entire lifetime of the system, but one can treat the workload as fixed during shorter time intervals (e.g., 1 hour), perform the capacity planning procedure for each monitoring time window (of 1 hour), and then combine the results across these time points to get the overall solution.

Next we show how to answer a typical service provider question: How many clients can be supported by the existing system, while

—providing the desirable performance guarantees, for example, response time under Γ_R, and
—assuming that the system processes a given (varying, nonstationary) type of workload?

The detailed sequence of steps performed by our tool to answer this question is summarized in Figure 16.

The first two steps that use the *workload profiler* and the *regression-based solver and analyzer* have been presented in details in the previous sections. We use the same workload as input to the third step of the analytic model, and this completes and automates the overall capacity planning process.

8. RELATED WORK

Nowadays, a new generation of monitoring tools, both commercial and research prototypes, provides useful insights into transaction activity tracking and latency breakdown across different components in multitier systems. Some of them concentrate on measuring end-to-end latencies observed by the clients [IBM Corporation; Rajamony and Elnozahy 2001; Cherkasova et al. 2003; Mercury Real User Monitor; NetQoS Inc]. Typically, they provide a latency breakdown into network- and server-related portions. While these tools are useful for understanding the client network-related latencies and improving overall client experience by introducing a geographically distributed solution at the network level, this approach does not offer sufficient insights in the server-side latency as it does not provide a latency breakdown into application- and database-related portions.

Another group of tools focuses on measuring server-side latencies [Barham et al. 2004; Mercury Diagnostics; Nimsoft Co.; CA Willy Introscope; Quest Software Inc.] using different levels of transaction tracking that are useful for

1. Workload profiler with Diagnostics tool:
For each monitoring window w, and each application server s:
- **a.** collect the number of transactions $N_{i,s,w}$ of each type i;
- **b.** collect the number of outbound DB calls of different j types $N^{DB}_{i,j,w}$ per each transaction type i;
- **c.** collect the transaction mix in the system, i.e., the percentage p_i of the transactions for type i and the percentage P_j of the outbound DB calls of type j) per each transaction type i;
- **d.** collect the average utilizations: $U_{s,w}$ of each application server s and U^{DB}_w for the DB server;
- **e.** collect the average think time Z_w.

2. Regression-based solver and analyzer:
Perform anomaly detection and application signature analysis.
For each application (or DB) server s:
Compute the latest cost function $C_{i,s}$ for each transaction type i (or $C^{DB}_{i,j}$ for outbound DB call of type j).

3. Analytical model:
For each monitoring window w:
- **a.** approximate the service time S^{front}_s for each 1st-tier (i.e., front tier or application server) machine s as $S^{front}_s = \sum_i p_i \cdot C^{front}_{i,s}$;
- **b.** approximate the service time S^{DB} for the DB server as $S^{DB} = \sum_i p_i \cdot \sum_j P_j \cdot C^{DB}_{i,j}$;
- **c.** compute the maximum number of clients MAX_w can be handled with average response time less than Γ_R using MVA algorithm.

Build the profile with entry as (w, MAX_w).
Find the minimal MAX_w value X.
X is the number of concurrent customers the system can support with the average transaction response time less than Γ_R

Fig. 16. The tool framework.

"drill-down" performance analysis and modeling. Unfortunately, such monitoring tools typically report the measured transaction latency and provide additional information on application server versus database server latency breakdown. Using this level of information it is often difficult to decide whether an increased transaction latency is a result of a higher load in the system or whether is an outcome of the recent application modification that is directly related to the increased processing time for this transaction type. Measurements in real systems cannot provide accurate transaction "demands", that is, execution times without *any* delays due to queuing/scheduling in each tier/server. Approximate transaction demands are extrapolated using measurements at very low utilization levels or with nearly 100% utilization [Urgaonkar et al.

2005]. Variability across different customer behaviors and workload activity further exacerbates the problem of accurately measuring and understanding transaction demands.

Applications built using Web services can span multiple computers, operating systems, languages, and enterprises. Measuring application availability and performance in such environments is exceptionally challenging. However, the tightly defined structures and protocols that have been standardized by the Web services community have opened the door for new solutions. There is a set of commercial tools [IBM Corporation; Nimsoft Co.; Mercury Diagnostics; Quest Software Inc.] for monitoring Java applications by instrumenting the Java Virtual Machine (JVM) which provides a convenient locus for nonintrusive instrumentation (some systems focus on .Net instead of Java). These tools analyze transaction performance by reconstructing the execution paths via tagging end-to-end user transactions as they flow through a J2EE-based system and looking for performance problems using one or more of the following techniques.

—Fixed or statistical baseline-guided threshold setting in HP BTO product suite [Mercury Diagnostics], IBM Tivoli Web Management Solutions [IBM Corporation], CA Application Performance Management [CA Willy Introscope], and Symantec I^3 Application Performance Management [Symantec I^3]. This approach can be labor intensive and error prone.

—Adaptive threshold setting, where the baselines and statistical workload is evaluated periodically, for instance, every 24 hours, and thresholds are adjusted. Examples include BMC ProactiveNet [BMC] and Netuitive [Netuitive Inc.]. This approach can result in a lot of false alarms while adjusting to change.

—Change detection combined with statistical baselining and thresholding, such as, CA Application Performance Management [CA Willy Introscope].

While it is useful to have detailed information into the current transaction latencies, the aforesaid tools provide limited information on the causes of the observed latencies, and cannot be used directly to detect the performance changes of an updated or modified application.

In addition to commercial tools, several research projects have addressed the problem of performance monitoring and debugging in distributed systems. Pinpoint [Chen et al. 2004] collects end-to-end traces of client requests in a J2EE environment using tagging and identifies components that are highly correlated with failed requests using statistics. Statistical techniques are also used by Aguilera et al. [2003] to identify sources of high latency in communication paths. Magpie [Barham et al. 2004] provides the ability to capture the resource demands of application requests as they are serviced across components and machines in a distributed system. Magpie records the communication path of each request and also its resource consumption, which allows for better understanding and modeling of system performance. Cohen et al. [2005] use a statistical approach to model performance problems of distributed applications

using low-level system metrics. They design a set of signatures to capture the essential system state that contributes to service-level objective violations. These signatures are used to find symptoms of application performance problems and can be compared to signatures of other application performance problems to facilitate their diagnosis.

From the preceding works, the two most closely related to our approach are Barham et al. [2004] and Cohen et al. [2005]. Magpie uses a more sophisticated tracing infrastructure than in our approach and concentrates on detecting relatively rare anomalies. The goal of our work is to detect performance changes in application behavior caused by application modifications and software updates that are complementary and independent on workload conditions in production environments.

9. CONCLUSION AND FUTURE WORK

Today, the three-tier architecture paradigm has become an industry standard for building enterprise client-server applications. The application server is a core component in this architecture and defines the main service functionality. Typically, when a new application update is introduced and/or unexpected performance problems are observed, it is important to separate performance issues that are caused by a high load of incoming workload from the performance issues caused by possible errors or inefficiencies in the upgraded software.

In this work, we propose a new integrated framework of measurement and system modeling techniques for anomaly detection and analysis of essential performance changes in application behavior. Our solution is based on integration of two complementary techniques: (i) a regression-based transaction model that characterizes the resource consumption pattern of the application; and (ii) an application performance signature that provides a compact model of runtime behavior of the application. The proposed online regression-based algorithm accurately detects a change in the CPU consumption pattern of the application and alarms about either observed performance anomaly or possible application change. However, it cannot distinguish which of the transactions is responsible for a changed CPU consumption of the application. To complement the regression-based approach and to identify the transactions that cause the model change, we use the application performance signature.

Finally, we demonstrate the automated capacity planning framework for enterprise services that integrates the designed modules (workload profiler and regression-based solver) with analytic model to answer additional service provider questions on required future capacity and resource provisioning while providing desirable QoS guarantees.

This article concentrates on performance anomalies and model changes in the CPU consumption of enterprise applications. Another representative group of performance problems in multitier applications is related to memory usage anomalies (e.g., memory leaks). The question is whether a similar approach can be applied for evaluating memory usage. We plan to exploit this avenue in our future work.

ACKNOWLEDGMENTS

Both the tool and the study would not have been possible without generous help of our HP colleagues (HP Diagnostics team): B. Enck, M. Glenna, and D. Gershon. We would like to thank the anonymous referees for insightful questions and suggestions which helped to improve the content and presentation of the article.

REFERENCES

AGUILERA, M., MOGUL, J., WIENER, J., REYNOLDS, P., AND MUTHITACHAROEN, A. 2003. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*.

ARLITT, M. AND FARKAS, K. 2005. The case for data assurance. HP laboratories rep. No. HPL-2005-38. http://www.hpl.hp.com/techreports/2005/HPL-2005-38.html.

BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. 2004. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*.

BMC. ProactiveNet. http://www.bmc.com/.

CA WILLY INTROSCOPE. http://www.ca.com/us/application-management-solution.aspx.

CHEN, M., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. 2004. Path-based failure and evolution management. In *Proceedings of the 1st International Conference on Networked Systems Design and Implementation (NSDI'04)*.

CHERKASOVA, L., FU, Y., TANG, W., AND VAHDAT, A. 2003. Measuring and characterizing end-to-end Internet service performance. *ACM/IEEE Trans. Internet Technol*.

CHERKASOVA, L. AND KARLSSON, M. 2001. Dynamics and evolution of Web sites: Analysis, metrics and design issues. In *Proceedings of the 6th International Symposium on Computers and Communications (ISCC'01)*.

CHOU, T. 2004. *The End of Software: Transforming Your Business for the On Demand Future*. Sams.

COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. 2005. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of of the 20th ACM Symposium on Operating Systems principles (SOSP'05)*.

DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. 1997. Rate of change and other metrics: A live study of the World Wide Web. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*.

DRAPER, N. R. AND SMITH, H. 1998. *Applied Regression Analysis*. John Wiley & Sons.

IBM CORPORATION. Tivoli Web management solutions. http://www.tivoli.com/products/demos/twsm.html.

MENASCE, D., ALMEIDA, V., AND DOWDY, L. 1994. *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*. Prentice Hall.

MERCURY DIAGNOSTICS. http://www.mercury.com/us/products/diagnostics/.

MERCURY REAL USER MONITOR. http://www.mercury.com/us/products/business-availability-center/end-use%r-management/real-user-monitor/.

MI, N., CHERKASOVA, L., OZONAT, K., SYMONS, J., AND SMIRNI, E. 2008. Analysis of application performance and its change via representative application signatures. In *Proceedings of the Network Operations and Management Symposium (NOMS'08)*.

NETQOS INC. http://www.netqos.com.

NETUITIVE INC. http://www.netuitive.com/.

NIMSOFT CO. http://www.nimsoft.com/solutions/.

QUEST SOFTWARE INC. Performasure. http://http://www.quest.com/performasure.

RAJAMONY, R. AND ELNOZAHY, M. 2001. Measuring client-perceived response times on the WWW. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*.

STEWART, C., KELLY, T., AND ZHANG, A. 2007. Exploiting nonstationarity for performance prediction. In *Proceedings of the Conference EuroSys Conference*.

SYMANTEC I^3. Application performance management. http://www.symantec.com/business/products/.

TPC-W BENCHMARK. http://www.tpc.org.

URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. 2005. An analytical model for multi-tier Internet services and its applications. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*.

ZHANG, Q., CHERKASOVA, L., MATHEWS, G., GREENE, W., AND SMIRNI, E. 2007a. R-Capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads. In *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware'07)*.

ZHANG, Q., CHERKASOVA, L., AND SMIRNI, E. 2007b. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing (ICAC'07)*.

Received November 2008; revised August 2009; accepted May 2009