Analysis of Application Performance and Its Change via Representative Application Signatures*

Ningfang Mi
College of William and Mary
Williamsburg, VA 23187, USA
ningfang@cs.wm.edu

Ludmila Cherkasova, Kivanc Ozonat, Julie Symons
Hewlett-Packard Labs
Palo Alto, CA 94304, USA
{lucy.cherkasova, kivanc.ozonat, julie.symons}@hp.com

Evgenia Smirni
College of William and Mary
Williamsburg, VA 23187, USA
esmini@cs.wm.edu

***Abstract*—Application servers are a core component of a multi-tier architecture that has become the industry standard for building scalable client-server applications. A client communicates with a service deployed as a multi-tier 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. 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.**

In this work, we propose a new approach based on an *application signature* that enables a quick performance comparison of the new application signature against the old one, while the application continues its execution in the production environment. The application signature is built based on new concepts that are introduced here, namely the *transaction latency profiles* and *transaction signatures*. These become instrumental for creating an application signature that accurately reflects important performance characteristics. We show that such an application signature is representative and stable under different workload characteristics. We also show that application signatures are robust as they effectively capture changes in transaction times that result from software updates. Application signatures provide a simple and powerful solution that can further be used for efficient capacity planning, anomaly detection, and provisioning of multi-tier applications in rapidly evolving IT environments.

I. Introduction

Fundamental to the design of reliable enterprise applications is an understanding of the performance characteristics of the service under different workload conditions and over time. In multi-tier systems, frequent calls to application servers and databases place a heavy load on resources and may cause throughput bottlenecks and high server-side processing latency. 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".

* This work was completed in summer 2007 during N. Mi's internship at HPLabs. E. Smirni is partially supported by NSF grants ITR-0428330 and CNS-0720699, and a gift from HPLabs.

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 the performance issues and the observed application behavior in existing production systems. First, an existing production system can experience a very different workload compared to the one that has been used in its testing environment. Second, 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 looses productivity and experiences increased operating costs.

Automated tools for understanding application behavior and its changes during the application development life-cycle 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 *pro-active* approach that is based on *continuous* application performance evaluation may assist enterprises in avoiding loss of productivity by the timely diagnosis of essential performance changes in application performance.

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 multi-tier systems. Some of them concentrate on measuring end-to-end latencies observed by the clients [9], [16], [5], [13], [14]. 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 [2], [12], [10], [7], [15] using different levels of transaction tracking that are useful for "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", i.e., 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 [19]. Variability across different customer behaviors and workload activity further exacerbates the problem of accurately measuring and understanding transaction demands.

In this work, we propose a new approach based on an *application performance signature* that provides a model of "normal" application behavior. We argue that online performance modeling should be a part of routine application monitoring and can be useful for performance debugging, capacity planning, and anomaly detection. The application signature approach enables a quick and efficient performance analysis of application transactions while the application is executing in the production environment. We introduce several new concepts such as *transaction latency profiles* and *transaction signatures* that are used to create a collective application signature to accurately reflect important application performance characteristics. We show that such an application signature is stable for different workload characteristics. Furthermore, continuous calculation of the application signature allows to capture events such as software updates that may significantly affect transaction execution time. Comparing the new application signature against the old one allows detection of specific application performance changes and enables further analysis to determine whether these are intended and acceptable.

An additional benefit of the proposed approach is its simplicity: it is not intrusive and based on monitoring data that is typically available in enterprise production environments. We illustrate the effectiveness of application signatures via a detailed set of experimentation using the TPC-W e-commerce suite [18].

II. Related Work

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 [9], [10], [12], [15] for monitoring Java applications by instrumenting the Java Virtual Machine (JVM) which provides a convenient locus for non-intrusive 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. While it is useful to have detailed information into the current transaction latencies, the above tools provide limited information on the causes of the observed latencies, and can not 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 [3] 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 [1] to identify sources of high latency in communication paths. Magpie [2] 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. [6] use a statistical approach to model performance problems of distributed applications using low-level system metrics. They designed 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 above works, the one most closely related to this paper is Magpie. 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.

III. Motivating Example and Intuitive Conjecture

Typically, tools like HP (Mercury) Diagnostics [12] are used in IT environments for observing latencies of the critical transactions via an interactive GUI as well as for 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 can not be used directly to detect the performance changes of an updated or modified application.

Fig. 1 shows the latency of two application transactions, $Tr1$ and $Tr2$, over time. 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 min, 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

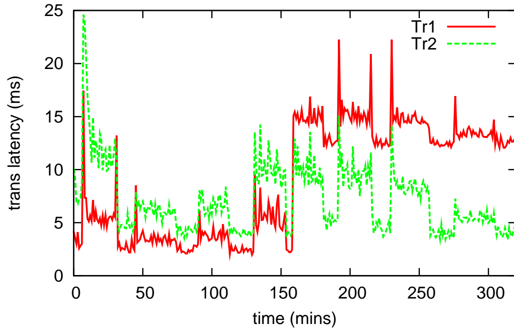


Fig. 1. The transaction latency measured by HP (Mercury) Diagnostics tool.

reported latency metric does not provide enough information to detect this change.

The problem addressed in this paper is whether by using measured transaction latency and its breakdown information, we can process and present it in a special way to quickly and efficiently diagnose essential performance changes in the application performance and to provide fast feedback to application designers and service providers.

Our main idea is, first, to partition transaction latency into complementary portions that represent time spent at different tiers. In particular, we will distinguish latency portions at the application server versus the database server. After that, we augment the transaction latency at the application server with CPU utilization of application server measured during the same monitoring window. The hypothesis is that if we plot transaction latency at the application server against its CPU utilization we will get a representative transaction latency profile. Our intuitive conjecture is that this transaction profile is very similar under different transaction mixes, i.e., it is uniquely defined by the transaction type and CPU utilization of the server and is practically independent on the transaction mix.

IV. Experimental Environment

A. TPC-W Testbed and Non-Stationary TPC-W Workloads

In our experiments, we use a testbed of a multi-tier e-commerce site that simulates the operation of an on-line bookstore, according to the classic TPC-W benchmark [18]. This allows to conduct experiments under different settings in a controlled environment in order to evaluate the proposed profiling and modeling approach. Specifics of the software/hardware used are given in Table I. We use terms "front server" and "application server' interchangeably in this paper.

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

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, i.e., the one with 10,000 items and 1,440,000 customers.

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

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)

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

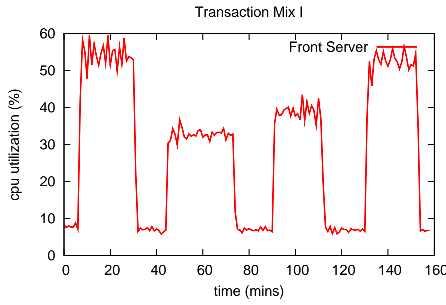
- the *browsing mix* with 95% browsing and 5% ordering;
- the *shopping mix* with 80% browsing and 20% ordering;
- the *ordering mix* with 50% browsing and 50% ordering.

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. 2. The pseudocode for the random process.

One drawback of directly using the transaction mixes described above in our experiments is that they are *stationary*, i.e., the transaction mix and load do not change over time. Since real enterprise and e-commerce applications are typically characterized by *non-stationary* transaction mixes with variable load [8], [4], [17], we designed an approach that enables us to generate non-stationary workloads using the TPC-W setup. To generate a non-stationary transaction mix with variable 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 4-th, 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



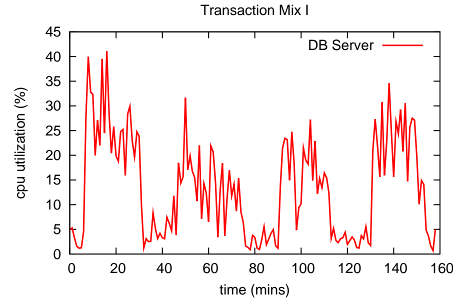


Fig. 3. CPU utilization of the Front server and the DB server across time under Transaction Mix I.

navigate and control this random process we use specified ranges for the "random" parameters in this workload. The pseudo-code of this random process is shown in Fig. 2.

Due to the 4-th random process the workload is non-stationary and the transaction mix and load vary significantly over time.

In this paper, we use fixed ranges of (20min,30min) for process "execution" time and (10min,20min) for process "sleep" time. Thus in order to describe the random process in different workloads used in our case studies we only specify ranges for choosing a random number of EBs. In particular, we consider the following three non-stationary transaction mixes with different number of EBs for the base processes and the additional random process as shown in Table III (each workload is executed for 3 hours):

TABLE III
THE PARAMETERS OF THE THREE NONSTATIONARY WORKLOADS

	Number of EBs			
	Browsing	Shopping	Ordering	Random
Trans. Mix I	20	30	50	random(150,700)
Trans. Mix II	50	50	50	random(150,700)
Trans. Mix III	50	30	100	random(150,700)

Fig. 3 shows the CPU utilization over time at 1-minute granularity for the front and database servers, respectively, for *Transaction Mix I*. CPU utilizations at both the application and database servers vary dramatically over time. Due to space limitations we omit figures representing CPU utilization for the other two workloads. Table IV summarizes CPU utilization ranges for all the three workloads.

TABLE IV
CPU UTILIZATION RANGES FOR THREE NONSTATIONARY WORKLOADS

	Ranges of CPU Utilization	
	Application Server	Database Server
Trans. Mix I	6% – 60%	1% – 42%
Trans. Mix II	10% – 61%	1% – 54%
Trans. Mix III	10% – 64%	1% – 39%

B. Transaction Latency Monitoring

TPC-W implementation is based on 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 [12] tool which offers a monitoring solution for J2EE applications. The Diagnostics tool consists of two components: the Diagnostics Probe and the Diagnostics Server as shown in Fig. 4.

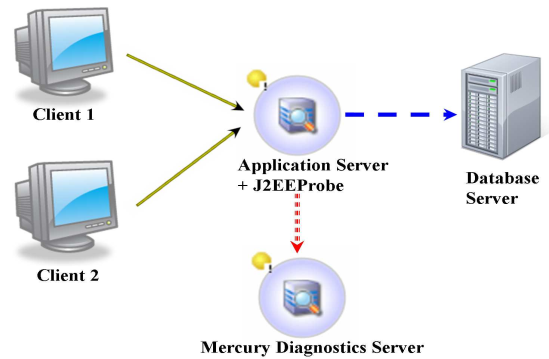


Fig. 4. TPC-W experimental configuration with the Diagnostics tool.

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 Fig. 4 is responsible for capturing events from the application, aggregating the performance metrics, and sending these captured performance metrics to the Diagnostics Server. In a monitoring window, Diagnostics provides the following information for each transaction type:

- a transaction count;
- an average overall transaction latency for observed transactions. This overall latency includes transaction processing time at the application server as well as all related query processing at the database server, i.e., 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 Fig. 5;
- 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 request is issued

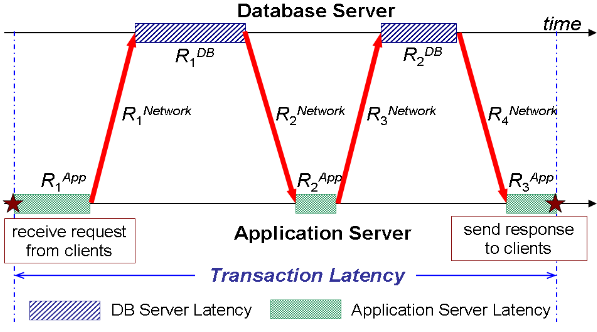


Fig. 5. The transaction latency measured by the Diagnostics tool.

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

V. Transaction Latency Profile

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. While in this work, we use only a subset of the extracted fields, we believe that the proposed application log format enables many value-added services such as capacity planning and anomaly detection as proposed in [20], [21], where the authors rely on the existence of application logs similar to the one described above.

Assuming that there are totally M transaction types processed by the server, we use the following notation:

- T =1 min is the length of the monitoring window;
- N_i is the number of transactions Tr_i, i.e., i-th type, where $1 \leq i \leq M$;
- R_i is the average latency of transaction Tr_i;
- P_i is the total number of *types* of outbound DB calls for transaction Tr_i;
- $N_{i,j}^{DB}$ is the number of DB calls for each type j of outbound DB call for transaction Tr_i, where $1 \leq j \leq P_i$;
- $R_{i,j}^{DB}$ is the average latency [1] for each type j of outbound DB call, where $1 \leq j \leq P_i$;
- U_{CPU} is the average CPU utilization of the application server tier during this monitoring window.

Recall that 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 i-th transaction type at the front and database servers, respectively. We then have the transaction latency breakdown calculated as follows:

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

Using this equation we can easily compute R_i^{front}. After that, for each transaction Tr_i we generate 100 CPU utilization buckets $\{U_1^i = 1, U_2^i = 2, ..., U_k^i = k, ..., U_{100}^i = 100\}$.

From the extracted application log, for each 1-minute monitoring window, we classify observed transactions into the corresponding CPU utilization buckets. For example, if during the current monitoring window there are N_i transactions of type i with average latency R_i^{front} under observed CPU utilization of 10% at the application server, then a pair (N_i,R_i^{front}) goes in the CPU utilization bucket U_{10}^i. Finally, for each CPU bucket U_k, we compute the average latency $R_{i,k}^{front}$ and overall transaction count $N_{i,k}$. In such a way, for each transaction Tr_i we create a *transaction latency profile* in the following format $[U_k^i, N_{i,k}, R_{i,k}^{front}]$, where $1 \leq i \leq M$ and $1 \leq k \leq 100$. In each CPU bucket, we keep information on overall transaction count $N_{i,k}$ because we will use it in assessing whether the bucket is representative.

Fig. 6 and Fig. 7 illustrate examples of latency profiles for "Home" and "Shopping cart" transactions, respectively. In each figure, the three curves correspond to the three workloads introduced in Section IV-A (see Table III).

Overall, the transaction latency profiles do look similar under different workloads. However, the existence of "outliers" in these curves makes formal comparison difficult. Typically, the "outliers" correspond to some "under-represented" CPU utilization buckets with few transaction occurrences, and as a result an average transaction latency being not representative for the corresponding CPU utilization bucket.

In the next section, we describe a derivation of transaction service time (transaction CPU demand) that uniquely defines the transaction latency curve and hence can be efficiently used for formal comparison of the transaction latency profiles.

VI. Approximating Transaction Service Time

In this section, we outline some classic queueing theory formulas that help to relate transaction latency, transaction service time, and observed system utilization aiming at defining transaction signatures that compactly characterize application transactions under different workload characteristics.

Consider a simple queue system. Let S be the mean service time for a job in the system, and let A be the queue length at the instant a new job arrives. The residence time (denoted as R) at the queueing center is the sum of the total time spent in service and the total time spent waiting for other jobs to complete service, which are already queued at that center when a job arrives. Thus, the average residence time R in such a system is given by:

$$R = S + S * A \quad (2)$$

As in a closed model, the queue length A seen upon arrival when there are N customers in the network is equal to the

[1] In reality, the measured latency of outbound call includes the addtional communication latency.

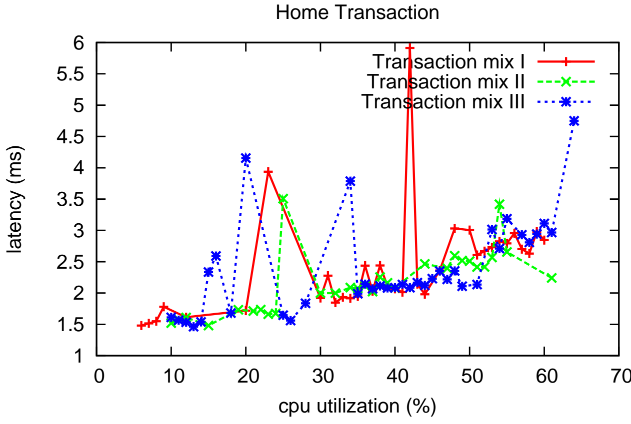


Fig. 6. The "Home" transaction latency profiles under the three workloads.

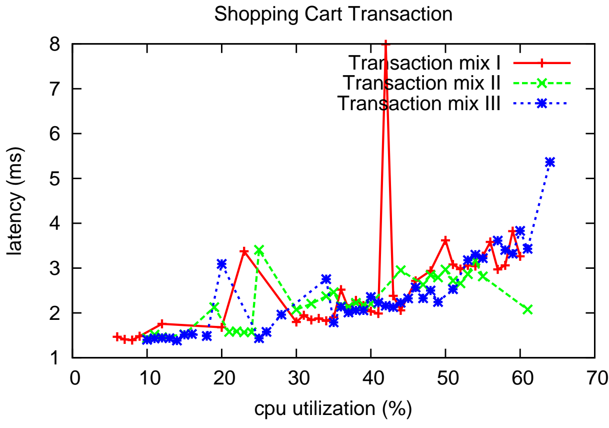


Fig. 7. The "Shopping cart" latency profiles under the three workloads.

time averaged queue length when the number of customers is $N-1$. If we let Q be the time averaged queue length, then we have an approximation for A, i.e., $A \approx \frac{N-1}{N} * Q$ [11]. As N increases, the approximation parameter $\frac{N-1}{N}$ approaches 1. Consequently, the queue length seen upon arrival approaches the time averaged queue length Q. As a result, we get the following formula:

$$R = S + S * Q \tag{3}$$

By applying Little's law $Q = X * R$, we have

$$R = S + S * (X * R), \tag{4}$$

where X is the average system throughput.

Since, by the utilization law the server utilization (denoted as U) is equal to throughput multiplied by service time, i.e., $U = X * S$, we can simplify Eq. 4 as follows:

$$R = S + S * ((U/S) * R) = S + U * R \tag{5}$$

Finally, after solving for R, we have the following form for the residence time:

$$R = S/(1-U) \tag{6}$$

Let us multiply both sides of Eq. 6 by the throughput X.

$$X * R = X * S/(1-U) \tag{7}$$

By Little's law, we can replace Q by XR:

$$Q = \frac{U}{1-U} \tag{8}$$

The formula (8) illustrates the relationship between the average number of jobs in the system queue and the server utilization.

Now, consider a transaction-based workload executed by a computer system. Let $Tr_1, \ldots, Tr_i$ be different transaction types in the overall system, where $1 < i \leq M$. Let S_i denote the mean service time of the transaction Tr_i (i.e., service time for processing Tr_i when it is the only job in the system). Assume we have on average N jobs (concurrent transactions or customers) in the system.

Typical computer systems use a time-sharing discipline to service multiple outstanding jobs. With simple time-sharing each job in the system receives a fixed quantum of service time. For example, the Linux operating system used in our experiments operates with 1 ms time slices. If the job completes within the given quantum, it then leaves the system. Otherwise, this job returns to the end of the queue to wait for the next allotted quantum. When the fixed quantum approaches zero in the limit, the time-sharing policy becomes the same as the processor sharing (PS) policy, where the processor shares its (fixed) capacity equally among all jobs in the system. That is, if there are N jobs in the system, they receive $1/N$ of their mean service time. Consequently, in such a processor sharing system, the residence time R_i of the transaction Tr_i is given:

$$R_i = S_i * N \tag{9}$$

Since $N = Q + 1$, we can replace Q using formula (8) to compute a residence time for a concrete transaction type as follows:

$$\begin{aligned} R_i &= S_i * (Q+1) = S_i * (U/(1-U)+1) \\ &= S_i/(1-U) \end{aligned} \tag{10}$$

In such a way, for a concrete transaction type Tr_i, we have a relationship (i.e., Eq. 11) based on transaction service time S_i, transaction residence time R_i and utilization U of the system:

$$S_i = R_i * (1-U) \tag{11}$$

In the next section, we show how to best approximate the transaction service time S_i using Eq. 11 and how it can be used to fit the latency curves shown in Fig. 6 and Fig. 7. This enables us to formally compare the transaction latency profiles under different workload conditions.

VII. Application Performance Signature

In this section, we describe how to create representative application signature that compactly reflects important performance characteristics of application. As shown in Section VI, we can compute the transaction service times from transaction latency profiles. In reality, when we collect measured latencies for each transaction type i over time, we have multiple

equations that reflect transaction latencies at different CPU utilization points as shown below [2]:

$$\begin{aligned} S_i &= R_{i,1}^{front} * (1 - U_1/100) \qquad (12) \\ S_i &= R_{i,2}^{front} * (1 - U_2/100) \\ ... & \quad .... \end{aligned}$$

Our goal is to find the solution that is the best fit for the overall equation set (12).

A linear regression-based (LSR) method can be chosen to solve for S_i. However, there are two reasons that forced us to choose a different method. First, as shown in Fig. 6 and Fig. 7 a number of outliers exists which could significantly affect the accuracy of the final solution as LSR aims to minimize the absolute error across all points. The outliers may significantly impact and skew the solution while these outliers are non-representative points in the first place. Second, even if we decide to use the most representative CPU utilization buckets (e.g., the top 10 or 20 most populated CPU buckets) then again since LSR aims to minimize the absolute error it treats all the CPU buckets equally. There may be a significant difference in the number of transactions contributed to different CPU buckets, but these "additional weights" are not taken into consideration when using LSR.

Therefore, we propose another method to compute the service time S_i for the i-th transaction type. By solving $S_i = R_{i,k}^{front} * (1 - U_k/100)$ in Eq. 12, a set of solutions S_i^k is obtained for different utilization points U_k in transaction latency profile. We generate a Cumulative Distribution Function (CDF) for the S_i. For example, Fig. 8 shows the CDF of the service time S_8 for the Home transaction.

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 Fig. 8 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 50-th percentile value as the solution for S_i as most representative.[3] The 50-th percentile heuristics works well for all transactions in our study.

Finally, an *application performance signature* is created:

$$\begin{aligned} Tr_1 &\longrightarrow S_1 \\ Tr_2 &\longrightarrow S_2 \\ ... & \qquad ... \\ Tr_n &\longrightarrow S_n \end{aligned}$$

We believe that such an application signature uniquely reflects the application transactions and their CPU requirements and is invariant for different workload types as we will show in the next section. The application signature compactly represents a model of "normal" application behavior. Comparing the new application signature against the old one allows detection of specific application performance changes and analysis of their impacts. We illustrate the effectiveness of this approach in the next section.

[2]Since we collect CPU utilization expressed in percents, we need to divide it by 100 to use correctly in equation (11).

[3]Selecting the mean of S_i allows the outliers (thus 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 50-th percentile is a good heuristics.

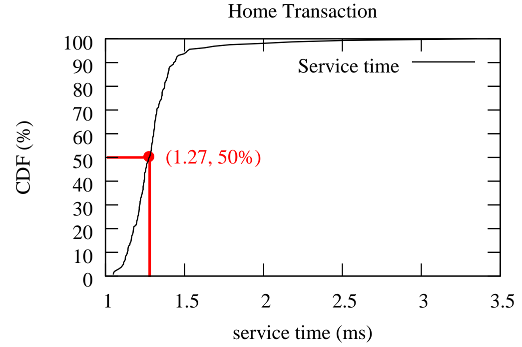


Fig. 8. CDF of the service time for the "Home" transaction.

VIII. Case Study

We ran the TPC-W benchmark under the three nonstationary workloads described in Section IV. Each experiment is performed for 3 hours, and we collected transaction data and performance metrics using the Diagnostics tool.

Fig. 9 plots the three TPC-W application signatures under different workloads. X-axes represent the transaction number, while Y-axes represent the estimated service time of transactions. Indeed, the application signatures are practically identical under different workload mixes and can be used as a compact performance model of the application behavior.

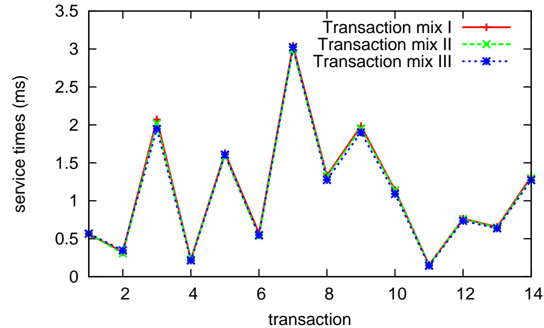


Fig. 9. The application signatures under the three workloads.

In order to see whether an application signature can reflect the application change, we modified the source code of the "Home" transaction in TPC-W: we've increased the transaction execution time by inserting a controlled CPU-hungry loop into the code of this transaction.

After that we performed three additional experiments with differently modified versions of the TPC-W benchmark running under Transaction Mix I, where the service time of the "Home" transaction (the 8th transaction) is increased by *i)* 2 milliseconds, *ii)* 5 milliseconds, and *iii)* 10 milliseconds, respectively. The application signatures of the modified applications are shown in Fig. 10, where the original application signature is plotted as the base line.

Indeed, comparing a new application signature against the original one allows detection of the application performance

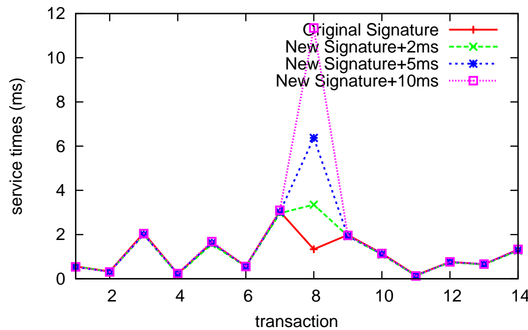


Fig. 10. Original application signature vs the application signatures of the modified application.

changes related to the "Home" transaction. The application signature approach enables a quick check of the possible performance changes in the application behavior between updates while the application continues its execution in the production environment.

IX. Conclusion and Future work

In this work, we propose a new approach based on the application performance signature that aims to provide a compact model of application behavior. We believe that the application signature is a valuable addition to the service provider's arsenal for automatically detecting performance changes of multi-tier applications in rapidly evolving IT environments. Comparing a new application signature against the old one allows detection of specific application performance changes and enables further analysis of whether these are intended and acceptable performance changes.

The application signature provides a concise and representative performance model of the application but one needs to apply it with care. The TPC-W benchmark under study is a well designed application with CPU being a primary bottleneck. In this case, the theoretical foundation designed in Section VI for deriving service time from transaction latency and system utilization provides correct results. However, we had received false alarms under high utilization rates for "Execute search" transaction (transaction 7). Due to lack of space, the results are not presented in the paper. The derived service time for this transaction showed an increase while there were no modification of the application code for these experiments. There are a few explanations for this:

- Under high load there could be additional bottlenecks in the system that contribute to transaction latency. Note that in this case, the latency is not a direct outcome of CPU contention only. As a result, when we apply formula (11) we will see an "increased" service time. To avoid such false alarms, one needs to limit the CPU utilization ranges that are used for constructing the application signature. It helps to make the proposed method more robust.
- There could be some inefficiencies and lock contention within the transaction code itself. Depending how bad such inefficiencies are the application signature might reflect an "increased" service time for involved transactions at different levels of system utilization.

We are working on refining our approach and extending the transaction signature with a few control points for service time derivation. We expect that it might provide additional useful information for application designers: the transactions with "tight signatures", i.e., similar service times across different CPU utilization levels, represent transactions with a well written, scalable code, while the transactions with "spread signatures", i.e., with visible differences in the derived service time across different CPU utilization levels, indicate transactions with opportunities for code improvement. In addition, we plan to extend our approach to more complex multi-tiered systems which may consist of more distributed servers or more transaction types. These are directions for our future work.

References

[1] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In Proc. 19th ACM Symposium on Operating Systems Principles, 2003.

[2] P. Barham, A. Donnelly, R. Isaacs, R. Mortier. Using Magpie for request extraction and workload modelling December 2004 6th Symposium on Operating Systems Design and Implementation (OSDI'04), 2004.

[3] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI'04), 2004.

[4] L. Cherkasova, M. Karlsson. Dynamics and Evolution of Web Sites: Analysis, Metrics and Design Issues. In Proc. of the 6-th International Symposium on Computers and Communications (ISCC'01), 2001.

[5] L. Cherkasova, Y. Fu, W. Tang, A. Vahdat: Measuring and Characterizing End-to-End Internet Service Performance. Journal ACM/IEEE Transactions on Internet Technology, (TOIT), November, 2003.

[6] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In Proc. 20th ACM Symposium on Operating Systems Principles, 2005.

[7] Computer Associates Co. Enterprise IT Management: Wily SOA Manager. www.ca.com/us/eitm/

[8] F. Douglis and A. Feldmann. Rate of change and other metrics: a live study of the world wide web. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[9] IBM Corporation. Tivoli Web Management Solutions, http://www.tivoli.com/products/demos/twsm.html.

[10] Indicative Co. http://www.indicative.com/products/End-to-End.pdf

[11] E.D. Lazowska, J. Zahorjan, G.S. Graham and K.C. Sevcik.Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1984.

[12] Mercury Diagnostics. www.mercury.com/us/products/diagnostics/

[13] Mercury Real User Monitor. http://www.mercury.com/us/products/business-availability-center/end-user-management/real-user-monitor/

[14] NetQoS Inc. http://www.netqos.com.

[15] Quest Software Inc. Performasure. http://java.quest.com/performasure.

[16] R. Rajamony, M. Elnozahy. Measuring Client-Perceived Response Times on the WWW. Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS), March 2001, San Francisco.

[17] C. Stewart, T. Kelly, A. Zhang. Exploiting nonstationarity for performance prediction. Proc. of the 2007 conference on EuroSys, 2007

[18] TPC-W Benchmark. URL http://www.tpc.org

[19] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-tier Internet Services and its Applications. In Proc. of the ACM SIGMETRICS'2005. Banff, Canada, June 2005.

[20] Q. Zhang, L. Cherkasova, and E. Smirni: A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. Proc. of the 4th IEEE International Conference on Autonomic Computing (ICAC'2007), 2007.

[21] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, and E. Smirni: R-Capriccio: A Capacity Planning and Anomaly Detection Tool for Enterprise Services with Live Workloads. Proc. of the ACM/IFIP/USENIX 8th International Middleware Conference (Middleware'2007), 2007.