Comparison of Performance Analysis Approaches for Bottleneck Detection in Multi-Tier Enterprise Applications

Jason Parekh, Gueyoung Jung, Galen Swint, *Member, IEEE*, Calton Pu, *Senior Member, IEEE*, and Akhil Sahai, *Member, IEEE*

***Abstract*—The complexity of today's large-scale enterprise applications demands system administrators to monitor enormous amounts of metrics, and reconfigure their hardware and software at run-time without thorough understanding of monitoring results. Our on-going Elba project is designed to achieve an automated iterative staging to mitigate the risk of violating Service Level Objectives (SLOs) by providing efficient, accurate bottleneck detection using a fusion of machine learning and performance analysis. Therefore, the Elba project guides system administrators to effectively perform management activities focused on actual performance-limiting bottlenecks rather than a vast set of collected metrics. In this paper, we introduce our concrete bottleneck detection approach used in Elba, and then characterize the qualities of three classifiers with respect to our bottleneck detection process in enterprise applications. We utilize a well known benchmark application, RUBiS (Rice University Bidding System), to evaluate the classifiers.**

***Index Terms*—Bottleneck detection, machine learning, multi-tier enterprise systems, performance analysis**

I. Introduction

QUALITY of service (QoS) during operation is one of the key areas of systems research for large scale mission-critical applications. However, production is not the only phase during an application's life cycle during which QoS should apply; it must also be met during pre-production testing, or staging. Due to the complexity of today's enterprise class applications, system administrators monitor and analyze a massive number of application-specific metrics such as the number of active threads and the number of EJB entity bean instances, along with system metrics like CPU usage and disk I/O rate. This same complexity also demands automation of staging, and furthermore as the complexity of the applications increase, the importance of efficient analysis of testing results also increases. If staging can successfully identify metrics associated with performance limitation and subsequently correlate the metrics with performance goals, then the results can also be used in the production phase as valuable guidelines for system administrators.

The Elba project [6] addresses the automation of enterprise and tiered application staging. Automated staging in the Elba project is an iterative process whereby an application is refined and reconfigured at each iteration. Automating the process involves the creation of deployment plans, instrumentation, analysis tools, and recommendation engines from the policy level documents that govern both the staging and production policies of the application. Staging inherently demands an iterative approach to test an application adequately before placing it in a production environment. During each staging iteration, the application is subject to multiple trials of variable workload. These trials provide data used to identify bottlenecks in the hardware/software configuration. After identifying bottlenecks, the application can be reconfigured and tested, again going through a series of trials, in the next iteration.

Machine learning classifiers constitute an important part of the metrics analysis in Elba. We have chosen machine learning techniques because they allow us to analyze many more metrics simultaneously than manual or ad-hoc approaches. As a result, the classifiers allow us to sort through these metrics to identify particular "bottleneck metrics" that indicate application mis-configuration correspondent with failed QoS. Our ongoing work demonstrated that a machine learning approach aids application tuning but avoided the questions related to the suitability of various types of classifiers to tuning multi-tier applications.

The goal of this paper is to characterize the qualities of various classifiers with respect to bottleneck detection in enterprise applications. A superior classifier for bottleneck detection should have three advantages. First, it must detect bottlenecks more quickly. In other words, it must require fewer trials per iteration before detecting bottleneck metrics. This allows tests to be terminated earlier. Consequently, the application can be re-designed and re-tested more quickly. Second, better classifiers potentially detect multiple bottleneck metrics. Finally, its analysis results should be accurate—i.e. correct in spite of variety in system parameterizations and workloads. We use a well known distributed application, RUBiS, which is a benchmark designed to mimic auction-based e-commerce websites, to evaluate the classifiers. Even though the application is a benchmark, we do not use it for performance comparisons. Instead, it is an exemplar

Manuscript received February 28, 2006. This work was supported by Hewlett Packard.

J. Parekh, G. Jung, G. Swint, and C. Pu are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA. (e-mail: jason.parekh@cc.gatech.edu, gueyoung.jung@cc.gatech.edu, galen.swint@cc.gatech.edu, calton@cc.gatech.edu).

A. Sahai is with Hewlett Packard Laboratories, Palo Alto, CA 94304 USA. (e-mail: akhil.sahai@hp.com).

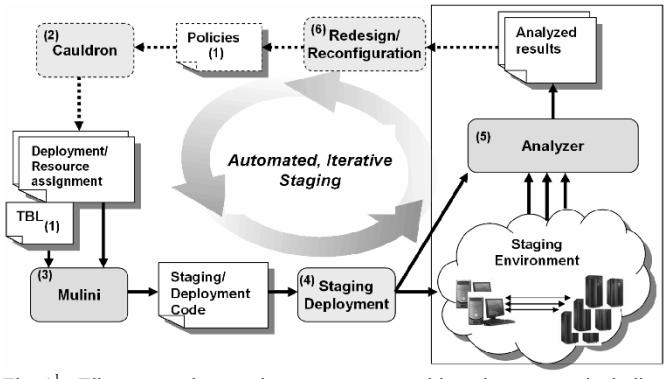

Fig. 1[1]. Elba approaches staging as an automated iterative process including analysis, the focus of this paper. One iteration of the staging process involves 6 steps: (1) Administrators and customers devise policy-level guidelines; (2) Cauldron converts the policy documents into resource assignments; (3) Mulini re-maps these assignments, applies further policies, and generates three types of code: instrumented application code, deployment code, and monitoring code; (4) a deployment tool installs and configures the application and then executes it; (5) monitoring data is fed into automated analysis tools; and (6) the analysis triggers recommendations for policy changes.

application.

The remainder of this paper is organized as follows. Section 2 outlines Elba project, classifiers used in our evaluation, and RUBiS. Section 3 describes our approach to bottleneck detection. Section 4 presents the evaluation environment. Section 5 presents our evaluation results utilizing RUBiS. Section 6 discusses related work, and Section 7 presents our conclusions.

II. Background

A. The Elba Project

The goal of Elba project is to automate the staging process, determine shortfalls in application performance and reconfigure deployed applications. Elba, which encompasses both existing and new tools that we are constructing, views staging as an iterative process in which staging results and recommendations can be passed back to system designers, as illustrated in Fig. 1.

The Elba project brings together several technologies. Policy formalizations and formal methods are used to create mappings guaranteed to satisfy policy-level constraints. Mulini and a domain specific test-bed language (TBL) capture the staging process and map policy documents into a staging environment. Distributed deployment tools and scripts are used to deploy, configure, and execute the application in the staging environment. In addition, we have extended the earlier version of Elba to include additional code generation and instrumentation, and employed machine learning methods to automatically analyze and detect bottlenecks from multiple staging executions. In this paper, we focus on the analysis and bottleneck detection (the rectangle box in Fig. 1). Elba and Mulini are described in [2]. Interested readers refer to [2] for more detail on our approach for automated staging and code generation.

[1] We have improved the figure of Elba used in [2].

B. Machine Learning Classifiers

1) Tree-augmented naïve (TAN) Bayesian network

A Bayesian network is a directed acyclic graph whose nodes embody domain attributes and arcs between nodes embody the probability dependency for the connected nodes. Given the conditional probability distribution for each attribute, a Bayesian network allows for the prediction of an unknown attribute, named the prediction attribute, in the network by computing its posterior probability distribution. In a Naïve Bayesian network, the graph is a tree with height one where the root node is the prediction attribute, disallowing dependencies between non-prediction attributes. A tree-augmented Naïve Bayesian network extends this by allowing each node to have a single incoming correlation edge, allowing non-prediction attributes to have probabilistic dependencies on one another. For more details about Bayesian Networks, refer to [9].

2) C4.5 decision tree (J48 implementation)

A decision tree consists of directed edges that embody decisions based on attribute values at the nodes, and whose leaves are the prediction nominal groups. Each node in the tree corresponds to an attribute (many-to-one relationship) and each outward edge from that node contains a range of values for that attribute, one of which is followed when predicting an outcome using the decision tree. Each attribute and its ranges selected best differentiate the child (direct or indirect) prediction attributes from other prediction attributes. J48 is the WEKA toolkit [3] implementation of the C4.5 algorithm, which generates decision trees based on information gain. For more details about decision trees, refer to [8].

3) LogitBoost

Boosting classifiers utilize simple classifiers but dramatically improve prediction performance by repeated training of re-weighted input data, and taking a majority vote of the resulting simple classifiers. Particularly, LogitBoost can be viewed as an approximation to additive modeling on the logistic scale. For our experiments, LogitBoost boosts the Decision Stump classifier, which generates a decision tree with only one split. For more details about LogitBoost and boosting techniques, refer to [10].

C. RUBiS

RUBiS, the Rice University Bidding System, is a multi-tiered e-commerce application consisting of a web server, web container, EJB container, and database server. RUBiS is based on the scenario of an online auction site with 26 interaction types, such as browsing categories; browsing items within a category; bidding, buying, or selling items; registering users; and writing or reading comments. RUBiS provides two workload transition matrices describing two different user behaviors: a browsing transition consisting of read-only interactions and a bidding transition, including 15% write interactions. We utilize the bidding transition in our evaluation since this transition is better representative of an auction site workload [1]. Our system reuses and extends a recent version

of RUBiS from ObjectWeb [4]. Generally, experiments show that RUBiS is application-server tier intensive. In other words, it is characteristically constrained by performance in the EJB container tier as introduced in [1].

III. Bottleneck Detection Process

An effective staging phase assures system administrators that a hardware/software configuration is capable of handling workloads to be seen during production. Starting at an initial configuration, this phase augments resources allowing the configuration to better satisfy the SLOs in cases of, for example, unhandled requests or poor response time. Possibilities of inadequacy stem from incorrectly configured software parameters crucial to performance, or lack of hardware needed to support the system under the stressed conditions outlined by the SLOs. Locating an exact point of bottleneck is a difficult problem, but, by applying machine learning we introduce a process that identifies metrics that potentially hinder system performance. The bottleneck detection process divides into three steps: staging the system with varying workload, training a machine learning classifier with metric data collected during staging, and finally querying the trained machine learning classifier to identify potential bottlenecks.

A. Staging with Varying Workload

The first step of the bottleneck detection process is to stage the enterprise system with varying workload in order to collect metric data which is analyzed and delivered to the latter steps of the process. The workload variation allows the process to formulate the correlations between load increase and system performance, which are then used to discover the limitations causing an SLO to become violated. The SLO policy describes behavior that causes violation, such as average response time for an interaction exceeding a certain threshold, which are then translated to form a starting point for staging: an initial workload and response time levels (per interaction type) that satisfy the SLO.

During the staging phase, each machine in the enterprise system is instrumented with monitors that collect metric data from both system-level and application-specific metrics. For our RUBiS experiment, we collected 220 metrics. We summarize sample metrics in Table I which have correlation to SLO violations in our experiment. Most of these metrics are used during the second step of the bottleneck detection process, but the response time is used immediately after the staging trial to calculate the degree of SLO satisfaction. Based on this satisfaction level, the next staging trial's workload is determined by a set of simple rules. The first step is to find an upper workload that violates the SLO below a certain level (in our experiments, this level is set at 40%) by exponentially increasing the workload. Then, based on the number of trials desired, the bottleneck detection process uniformly fills the gap starting from a small workload (20 concurrent users in our experiments) to the upper limit. While it is recommended to fill the entire spectrum of workload trials, the number of trials desired is an important variable as it should balance the speed of the bottleneck detection process with the accuracy of the identified bottlenecks. Upon completion of all staging trials, the collected data is analyzed to form calculations useful for the second step of the bottleneck detection process, such as the average value of each metric.

Metric	Description
CPU usage	Measured spent time in user and kernel. Applied all tiers.
Mem usage	Measured overall memory usage including operating system caching. This makes high memory utilization observed. Applied all tiers.
Paging rate	Unlike system memory utilization, this provides a more accurate picture of the pressure on memory. Applied all tiers.
Disk I/O	Measured total amount of data read/written to the drive in blocks per second. Applied all tiers.
NW bandwidth	Measured sent/received bytes per second between tiers.
EJB cache size	Measured cache size allocated for entity beans in EJB container tier.
Num of EJB instances	Measured the number of active entity instances in EJB container tier.
JVM memory usage	Measured actively used bytes of application server over JVM.
Num of connections to DB	Measured the number of actively opened connections to database tier in EJB container tier.
Num of transactions	Measured the number of committed transactions in EJB container tier.
Num of threads	Measured the number of active threads in application server tier.
Num of DB connections	Measured the number of opened connections in database tier.

Table I. Sample system-level and application-specific metrics in RUBiS experiment.

B. Classifier Training

The second step of the bottleneck detection process is to train a machine learning classifier with previously accumulated metric data allowing the classifier to form correlations between the performance of the system and the resulting SLO satisfaction. A machine learning classifier is trained with multiple training instances where each instance has a set of attributes along with a prediction attribute. Each attribute is a variable that may have some correlation to the prediction attribute, the variable that should be predicted by a trained classifier given a set of attribute values. Upon training, a classifier finds correlations between the attributes and the prediction attribute and saves them to a model.

The bottleneck detection process assigns the metrics as attributes and the SLO satisfaction level as the prediction attribute. Each variation in workload (a separate trial) has its own training instance where each attribute is a function of the averaged metric data for that workload, and the prediction attribute is the SLO satisfaction level for that workload; thus, training the classifier allows it to analyze any correlations that may exist between each metric and the SLO satisfaction level.

The initial set of collected metrics consists of 220 application-specific and system-level metrics. In order to reduce the amount of extraneous non-correlated metrics, we apply a correlation coefficient function introduced in [7], comparing metrics to the overall response time. From this function, we apply a threshold to reduce our working set of metrics for training.

To create a training set, the bottleneck detection process does a difference in each metric value from the previous workload trial—that is, each training instance attribute equals the change in metric value from the previous workload divided by the change in workload. Using this delta metric value as a training attribute allows the machine learning classifier to correlate trends of metric values to SLO satisfaction rather than the metric values themselves. The trends are particularly important since as a metric becomes saturated, it becomes the bottleneck in which case it cannot grow any further (in terms of metrics that have a limit, for example CPU utilization), and

will thus have a trend (first order derivative) approaching zero. Metrics that are not limited, such as database query latency, will have a trend that approaches infinity (that is, when the metric is bottlenecked and the workload keeps increasing, the metric value will grow exponentially larger with each workload increase). Both of these types of metrics can be correlated to the SLO satisfaction since their derivative (the delta metric) will not be constant. Those metrics that are not bottlenecks and whose values grow linearly with increase in workload will have constant trends, which disregards them from being correlated to the SLO satisfaction (for example, if network throughput grows by 100 KB/s through all SLO satisfaction levels, the delta would be constant at 100 therefore a correlation cannot be formed). Thus, our bottleneck detection approach focuses on metrics showing throttled behavior, rather than linear metrics which show room for expansion.

In order to remove jittery data collected through the multiple staging trials, we use an average window applied to the delta metric values. To compute the average window, each delta metric value is formed by averaging itself with its surrounding delta metric values (in the space of contiguous workload trials). In our experiments, we found an average window size of 25 smoothes oscillating delta values dramatically and strengthens the results of our bottleneck detection process.

The training set used by the bottleneck detection process consists of training instances each with n attributes, where n is the number of metrics (each metric corresponds to one attribute), and a prediction attribute that corresponds to the SLO satisfaction. There are $k-1$ training instances, where k is the number of workload trials with all but the first workload corresponding to a training instance. Since the process uses the change in metric values between workload trials, the first workload serves as the base and is not represented by a training instance. The k^{th} training instance consists of attributes where each attribute is the difference in its corresponding metric's value between the $k+1^{th}$ workload and the k^{th} workload.

The machine learning classifiers employed by our bottleneck detection process all require nominal prediction attributes. However, the SLO satisfaction level is not nominal (it is a continuous variable, ranging from 0% to 100%). To allow proper training of the classifier, the SLO satisfaction level is converted to nominal values by using a partitioning scheme that generates uniformity in the number of training instances that fall into each SLO satisfaction nominal group. This uniformity is important as having a skewed number of training instances per group affects the classifier model. To generate these nominal groups, the bottleneck detection process sorts the SLO satisfaction levels (from each training instance) and divides the sorted list into 10 uniform groups with each group labeled by the highest SLO satisfaction level in the group.

Finally, the training set is serialized into the ARFF-file format (used for machine learning training data in the WEKA toolkit) and each of the classifiers (TAN, J48, and LogitBoost) is trained while recording the generated models to a file.

C. *Identifying Bottleneck Metrics*

The final step of the bottleneck detection process is to query a trained classifier using an approach that discovers candidate bottlenecks. The process queries the classifier by creating a test set consisting of test instances that mimic training instances, with the only difference being that the prediction attribute is not included in the test instance since this is the information the classifier predicts using its model.

In order to discover potential bottlenecks, the bottleneck detection process must realize which metric trends the classifier correlated to the SLO satisfaction. To accomplish this, the bottleneck detection process carefully constructs a set of instances that is used to query the previously generated classifier model. The test set consists of n test instances where n is the number of metrics, with each instance i testing whether the i^{th} metric is a candidate bottleneck – that is whether the metric trend was highly correlated to SLO satisfaction. First, all n test instances are cloned from a single base instance (described later). Second, each test instance i is modified so that only the i^{th} attribute (which corresponds to the i^{th} metric) differs from the other test instances. Finally, the classifier uses its previously trained model to predict the SLO satisfaction nominal group for each instance in the test set.

Determining which base instance to use and how to modify each instance's unique attribute (which corresponds to a metric) is a difficult problem since the result of testing (the predicted SLO satisfaction) must be translated to identify whether the metric is a candidate bottleneck, and if so, how much confidence the classifier has in its judgment. First, the bottleneck detection process assigns the base instance to be an instance from the training set since this guarantees its attributes to be within the valid ranges for each corresponding metric delta. Next, the test set generated by cloning this base instance n times is modified so that each instance varies from the base instance in one attribute.

We set the base instance to the instance in the training set that has the least SLO satisfaction. Then, we modify the i^{th} cloned test instance on i^{th} attribute using the i^{th} attribute from the training instance that has the highest SLO satisfaction. That is, each delta metric in a test instance will be its value from the workload that has the least SLO satisfaction, except for one metric that will be its value from the workload that has the highest SLO satisfaction. After testing this instance, we get the difference between the predicted SLO satisfaction and the least SLO satisfaction and use this as a judgment for the degree of correlation for this delta metric to SLO satisfaction. If the metric corresponding to this instance has high correlation, it affects the decision of the classifier greatly. Since the base instance has the least SLO satisfaction, if modifying one delta metric to a value that is completely opposite (taken from the opposite SLO satisfaction level: the highest SLO satisfaction) tricks the classifier into predicting a different SLO satisfaction level from the base instance, then that metric has high correlation. This process happens for each test instance (one per metric) in the test set, so more than one

metric can be identified as a potential bottleneck. Based on the difference between the predicted SLO and least SLO satisfaction levels of each instance, we judge how to weigh the corresponding metric in terms of potential bottleneck. We normalize each metric by dividing its SLO difference by the sum of all differences in SLO for the test set. Finally, we generate a list of potential bottlenecks and the confidence the classifier has in each.

IV. Evaluation Setup

A. Software

To execute the staging phase with RUBiS, we employ Apache 2.0.54 as an HTTP server, MySQL max-3.23.58 as a database server with type 4 Connector/J 3.0.11 as a JDBC driver, and JOnAS4.4.6-Tomcat5.5.12 package as an EJB-Web container. Apache HTTP server is equipped with mod_jk so that it can be used as a front-end server to one or several Tomcat engines, and it can forward servlet requests to multiple Tomcat instances simultaneously via AJP 1.2 protocols. We increase the number of the maximum processes of Apache to avoid connection refusals from the server when numerous clients simultaneously request services. We also set the automated increment option on every primary key of RUBiS databases to prevent duplication errors when clients simultaneously attempt to insert data into a table with the same key. Finally, we adjust JOnAS to have adequate heap memory size for preventing out-of-memory exceptions during staging.

For gathering system-level metrics, we wrote a shell script to execute Linux/UNIX utilities, sar and ps, with monitoring parameters such as staging duration, frequency, and the location of monitored hosts. We also use JimysProbing 0.1.0 for metrics generated from JOnAS-Tomcat server, apachetop 0.12.5 for Apache HTTP server, and mysqladmin for MySQL database server. We slightly modified apachetop to generate XML encoded monitoring results. The client workload generator is designed to simulate remote Web browsers that continuously send HTTP requests, receiving corresponding HTML files, and recording response time as a performance metric during staging.

B. Hardware

We used Intel Xeon machines with 2.8 GHz processors and 1 GB of RAM in a cluster to run the servers. Two different types of hardware were tested, a single CPU and dual-CPU hyper-threading enabled machines, to identify whether CPU causes the performance limitation of RUBiS (Note that these machines are different at only the CPU configuration. Others such as memory, disk, and network are identical.). We have established a set of topology configurations to show whether application server tier causes the performance limitation as increasing the number of machines for application server tier. For instance, 1/2/1 means the configuration consists of one machine for the HTTP server tier, two machines for the application server tier, and one machine for database tier. Each configuration will be tested in a series of trials beginning with a minimum workload. On each trial, the workload will be increased by a workload increment.

V. Evaluation Results

The evaluation consists of three parts. First, we show the bottleneck patterns of RUBiS. Second, we select a list of candidate bottleneck metrics from 220 metrics using a correlation coefficient function and metrics' thresholds specified in given policy document. Finally, we evaluate three classifiers with chosen metrics to identify characteristics of these classifiers and the best suited classifier.

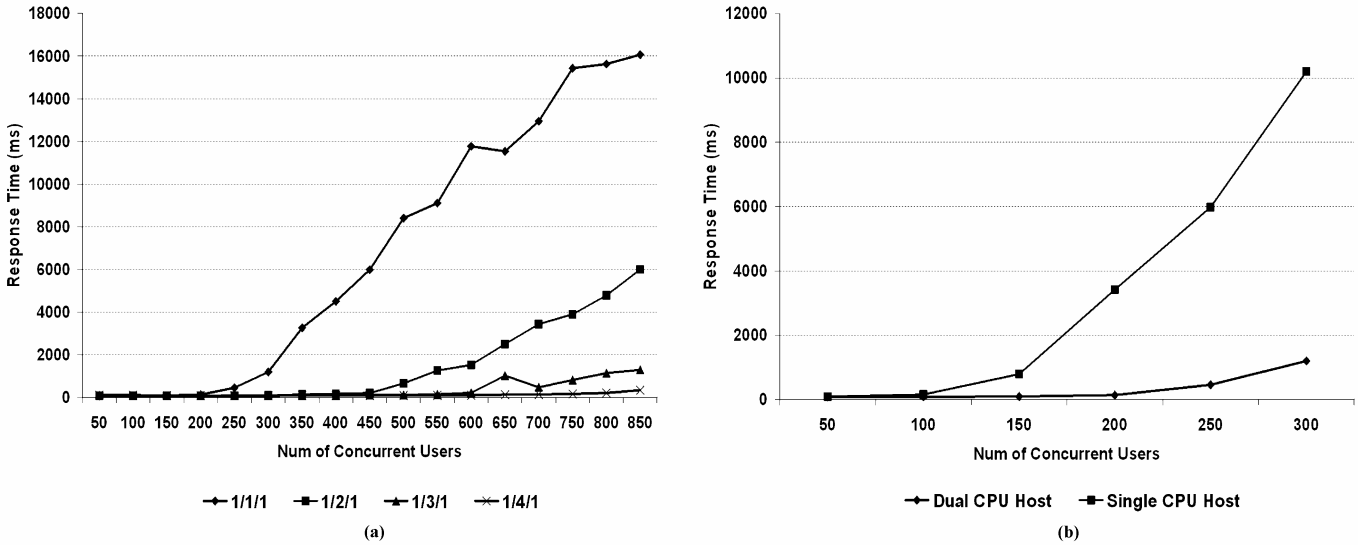

Fig. 2. Comparison response times (a) among four topologies, (b) two different CPU hardware configurations.

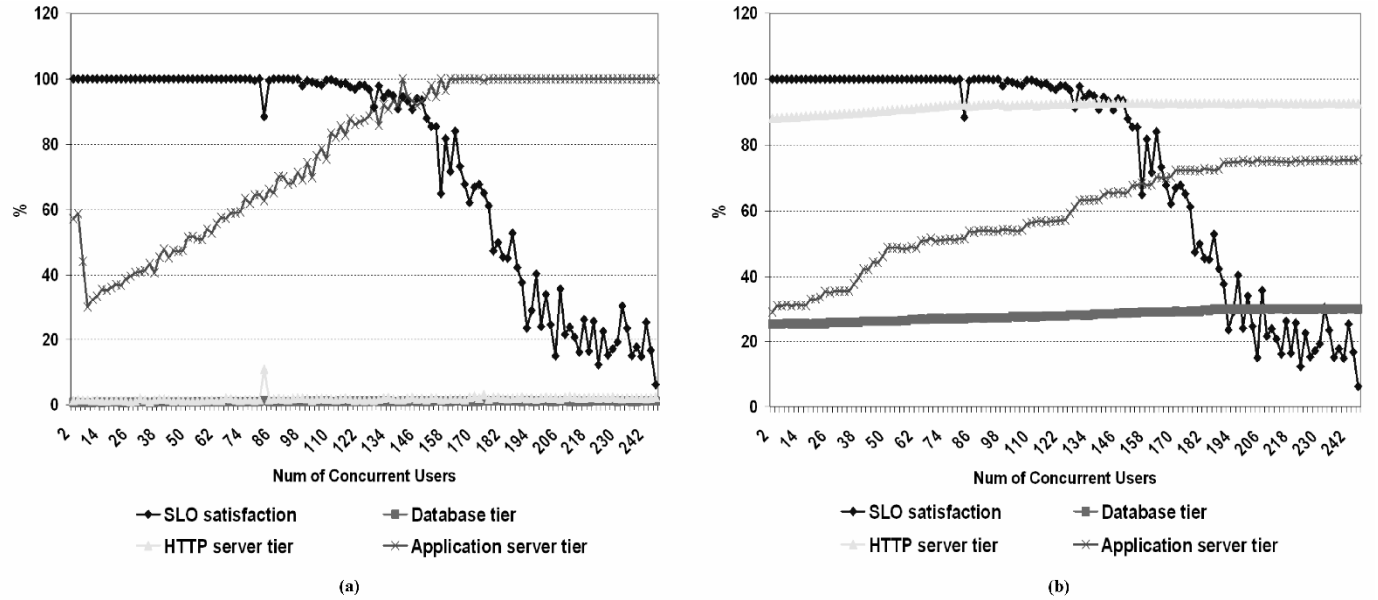

Fig. 3. SLO satisfaction against (a) CPU usage metric and (b) memory usage metric in all tiers.

A. Bottleneck Pattern of RUBiS

In this section, we show the performance characteristics of RUBiS in terms of which tiers cause bottleneck limitations and which metrics make the bottleneck inside these tiers by executing RUBiS with several topologies and two different CPU configurations. Fig. 2 (a) illustrates the application server tier mainly influences response time. That is, when we increase the number of machines for the application server tier (JOnAS-Tomcat), response time drastically decreases. When we use different topologies, which increase the number of machines for HTTP server tier and/or database tier, such as 2/1/1, 2/1/2, and 1/1/2, the results are almost identical to 1/1/1. Fig. 2 (b) shows one of the main bottleneck metrics is CPU of the application server tier since the machine equipped with hyper-threading enabled dual-CPU performs much faster than single CPU machine. The gap gets worse as the number of concurrent users increases.

Fig. 3 (a) shows the CPU usage of application server radically increases and saturates at 100% when the SLO satisfaction goes down to around 85%. In contrast, while the SLO satisfaction starts to decrease, the trends of other tiers' CPU usages are almost flat. In Fig. 3 (b), the memory usages of both the application server and database server are under utilized. The memory usage of HTTP server is somewhat high, but its trend is almost flat.

We may use this result as criterion to figure out classifiers' accuracy. In other words, to be a suitable classifier for bottleneck detection, it should capture the CPU usage as a metric causing bottleneck limitation.

B. Correlation between Metrics

Fig. 4 illustrates the correlations of some metrics monitored from application server tier to response time. Fig. 4 (a) and (b) show the number of active threads and the actual memory usage of JVM are more correlated to response time than cache size metric. However, the actual memory usage of JVM utilized at most 0.1% of overall total memory taken by JVM. Therefore, this metric should be excluded from candidate bottleneck metrics. Table II shows the list of metrics, which are most correlated to response time. We used the correlation coefficient function introduced in [7]. For the next

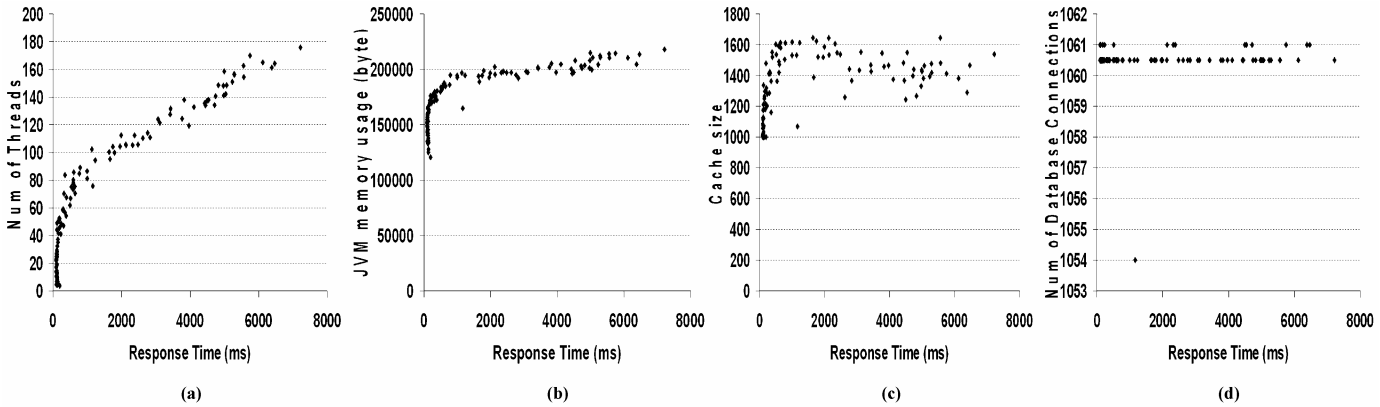

Fig. 4. Correlations of (a) the number of threads, (b) JVM actual memory usage, (c) cache size, and (d) the number of database connections to response time in the application server.

Metrics	Correlation Coefficient	Metrics	Correlation Coefficient
DB_CPU_Usage	0.612	**HTTP_Bytes_NW**	0.714
App_CPU_Usage	0.683	**App_Bytes_NW**	0.533
DB_Mem_Usage	0.829	**App_Item_Cache**	0.511
HTTP_Mem_Usage	0.498	**App_Num_Item_Instances**	-0.345
App_Mem_Usage	0.767	**App_JVM_Mem_Usage**	0.809
App_Paging	-0.841	**App_Num_DB_Conn**	0.499
DB_Bytes_Sent	0.569	**App_Num_Tx_Commits**	0.366
DB_Bytes_Rcv	0.572	**App_Num_Threads**	0.921

Table II. Correlation coefficients of metrics, which are mostly correlated to response time.

classification process, some metrics such as DB_CPU_Usage, DB_Mem_Usage, App_Mem_Usage, and network throughput metrics in the table are excluded because of their under utilization even though their correlation coefficient numbers are high enough to be included as candidate bottleneck metrics.

C. Bottleneck Detection Process

We applied our bottleneck detection process described in Section III to RUBiS in order to automatically identify the CPU bottleneck discovered manually earlier. We describe the accuracy of each classifier both in terms of cross-validation and bottleneck identification accuracy, and the convergence speed of each.

1) Accuracy of Classifiers

We define the accuracy of each machine learning classifier in two ways: the cross-validation accuracy, which describes its accuracy in predicting the SLO satisfaction levels of a partitioned training set, and the bottleneck identification accuracy, which describes its accuracy in identifying only the actual bottlenecks of our experimental system. The former accuracy determines the accuracy in a pure machine learning sense, oblivious to the actual application of the classifier, by testing the trained classifier's generated model on how well it can predict unseen instances. The latter accuracy determines the accuracy as applied to our bottleneck detection process.

Cross-validation involves partitioning the training set into *k*-folds and generating *k* different trained models, with each model trained on a unique set of *k-1* folds. Each trained model is then tested against an independent set of data from the training data – the 1 fold left out while training the model. Based on the sum of all the positively classified instances with the total number of instances from the combined testing sets, we form our prediction accuracy. For *k*-fold cross-validation, *k* must be less than the number of training instances for which our lowest was 11 workloads. In our experiments, we use 10-fold cross-validation as it provided results comparative to 100-fold cross-validation (which we tested using our maximum number of workload, 115), while allowing us to stay consistent in prediction accuracy measurement throughout all our workload trials.

Fig. 5 (a) shows each classifier's prediction accuracy using 10-fold cross-validation for the 115 workload experiment. LogitBoost has the highest prediction accuracy among the three at 60.87%, followed by J48 at 60.12%, and finally TAN at 57.39%. While these accuracies could be stronger, our experiments show that the classifiers' bottleneck identification accuracies are high.

Fig. 5 (b) shows each classifier's bottleneck identification accuracy on three different workload experiments. The first workload experiment consists of 92 workloads. In this experiment, the TAN and LogitBoost classifiers both identified only the application-tier CPU as being a potential bottleneck, whereas the J48 classifier identified the application-tier CPU and HTTP-tier memory as being potential bottlenecks. However, the difference in predicted and actual SLO satisfaction for the application-tier CPU is 33.0%, whereas for the HTTP-tier memory it is only 3.5%. Hence, to calculate the bottleneck identification accuracy for J48 the process divides 33.0% (the actual bottleneck) by the sum of 33.0% and 3.5%, which equals 90.4%. In the 104 workload experiment, the J48 and LogitBoost classifiers' bottleneck identification accuracy is at 100% bottleneck identification accuracy whereas the TAN classifier is at 93.6% (TAN's difference in SLO prediction and actual SLO for application-tier CPU is 51.4%, and for HTTP-tier memory it is 3.5%).

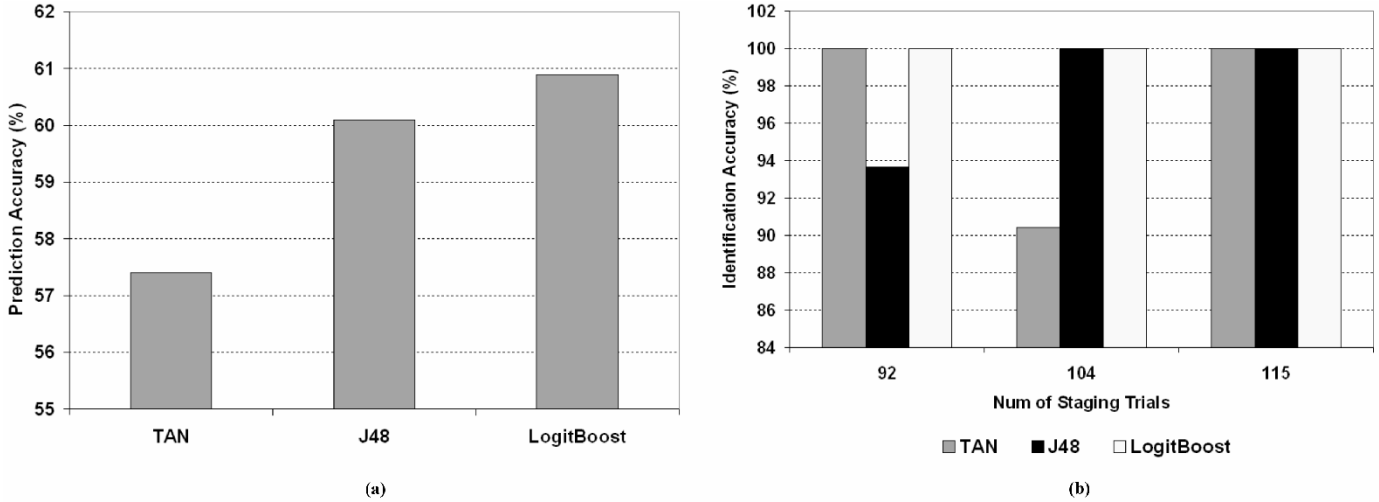

Fig. 5. The accuracies of the machine learning classifiers. (a) illustrates the prediction accuracy using 10-fold cross-validation. (b) illustrates the bottleneck identification accuracy over three staging trials: 92, 104, 115.

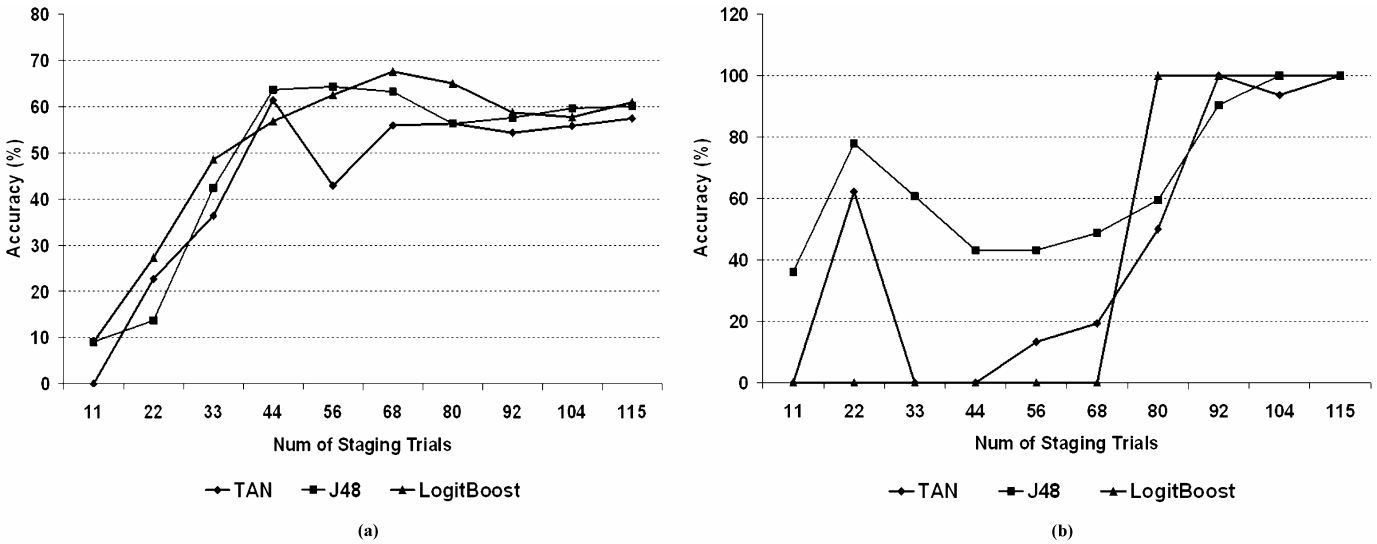

Fig. 6. The convergence speed of a classifier reveals how lengthy of a staging phase is needed to obtain accurate results from a classifier. Each staging trial uses a new workload which is then translated to an instance in the training set for the classifiers. (a) shows the prediction accuracy (based on 10-fold cross-validation) of each classifier as the number of trials increases. (b) shows the final bottleneck identification accuracy as the number of trials increases.

2) Convergence Speed

The convergence speed of a classifier defines how many trials of staging (with each trial varying in workload and thus training set size) is needed to obtain strong accuracy. This speed becomes relevant as we measure the efficiency of each classifier, which is the minimal amount of staging required for the classifier to identify bottlenecks in the system.

Fig. 6 (a) displays the convergence speed for the prediction accuracy, which shows the minimal number of staging trials needed to obtain good prediction accuracy. The period from experiments using 10 to 50 staging trials shows a low accuracy at start, but heading toward stabilization at around 90 staging trials. The period from 50 to 90 staging trials shows a slight decrease in accuracy, which can be explained from each of these being independent experiments consisting of random transition tables for RUBiS, which affects metric values that comprise the training set. Each of the classifiers in the graph show similar behavior, although once stabilized LogitBoost has a slightly higher prediction accuracy, immediately followed by J48 and finally by TAN.

Fig. 6 (b) displays the convergence speed for the bottleneck identification accuracy, which shows the minimal number of staging trials needed for the bottleneck detection process to have strong results. The period from 10 staging trials to 40 staging trials shows some erratic behavior which can be explained by the under developed classifiers (which can be seen by looking at the convergence speed for prediction accuracy graph, Fig. 6 (a)). The period from 40 staging trials onwards shows positive results as each classifier increases toward 100% bottleneck identification accuracy. The J48 classifier remains above the TAN classifier (aside from the 94 staging trial experiment), whereas the LogitBoost classifier shows interesting behavior of having no bottleneck identification until the 77 staging trials experiment, from which onwards it has 100% bottleneck identification accuracy. From our experiments with all of the classifiers, we deduce the minimum number of staging trials for positive results is 70, whereas 90 staging trials provides even stronger results. While all the classifiers performed well, the LogitBoost classifier seemed to excel in terms of convergence speed by reaching 100% bottleneck detection accuracy first. However, in terms of overall reliability the J48 classifier seemed to provide better results due to its steady increase in bottleneck detection accuracy toward 100% and its higher bottleneck detection accuracy throughout a majority of the variations in number of staging trials.

VI. Related Work

Cohen *et al* [5] apply a tree-augmented Naïve Bayesian network (TAN) to discover correlations between system-level metrics and performance states, such as SLO satisfaction and SLO failure. Similarly, we utilize TAN to investigate performance patterns, however we differ on three aspects. First, we perform a comparative study of classifiers beyond TAN and include the J48 decision tree and LogitBoost, two well known machine learning algorithms that have yet to be applied to performance analysis. Our goal is to compare the performance of classifiers in terms of bottleneck detection, and finally identify the classifier that best detects bottlenecks in multi-tier applications. Second, in addition to correlating metrics to performance states, we focus on the detection of actual performance-limiting bottlenecks by employing a unique change in metric training procedure. Finally, our set of metrics for bottleneck detection includes 193 application-level metrics as well as system-level metrics.

Urgaonkar *et al* [11] introduce a dynamic queuing model combined with predictive and reactive provisioning. Their contribution allows an enterprise system to increase capacity in bottleneck tiers during flash crowds in production. Elba, in addition to being oriented towards avoiding in-production performance shortfalls, emphasizes fine-grained reconfiguration. By identifying specific limitations such as low-level system metrics (CPU, memory, etc.) and higher

level application parameters (pool size, cache size, etc.) configurations are tuned to the particular performance problem at hand.

Powers *et al* [12] similarly use machine learning techniques to analyze performance. However, rather than detecting bottlenecks in the current system, they predict whether the system will be able to withstand load in the following hour. The machine learning classifiers they use differ from ours (we apply a J48 decision tree and LogitBoost) as does the approach for classification. Rather than predicting immediate failures, our paper addresses the performance of each classifier in terms of detecting bottlenecks in multi-tier applications. We also differ in service level objectives, which they formulate as target values for underlying system-level metric values, such as CPU utilization exceeding 75%. Elba, by addressing policy level SLOs, targets business-level objectives, such as response time, translates these objective into system and tier-specific objectives, and then uses low-level and application-level metrics to meet these goals.

VII. Conclusion

In this paper, we explore the performance of various machine learning classifiers with regard to bottleneck detection in enterprise, multi-tier applications governed by service level objectives. This builds on our previous work which used a J48 decision tree to assist tuning the TPC-W application. Specifically, in this paper, we demonstrate the effectiveness of three classifiers, a tree-augmented Naïve Bayesian network, a J48 decision tree, and LogitBoost, using our bottleneck detection process, which delves into a new area of performance analysis based on the trends of metrics (first order derivative) rather than the metric value itself. Furthermore, we illustrate the efficiency of each classifier by measuring the convergence speed, or the number of staging trials required in order to provide positive results. Using RUBiS, we test our bottleneck detection process on a set of 220 combined system-level (CPU, memory, etc.) and application-level metrics (open database connections, EJB pool size). Finally, we show the effectiveness of the classifiers used in our bottleneck detection process as each classifier strongly identifies the enterprise system bottleneck.

Acknowledgment

We would like to thank Sharad Singhal of HP Labs for his valuable insight and comments during the development of this paper.

References

[1] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic Web content," in *2003 Proc. of the International Middleware Conf.*, Rio de Janeiro, Brazil, June 2003.

[2] G. S. Swint, G. Jung, C. Pu, and A. Sahai, "Automated staging for built-to-order application systems," in *2006 Proc. Network Operations and Management Symposium (NOMS 2006)*, Vancouver, Canada, April 2006.

[3] WEKA distribution. http://www.cs.waikato.ac.nz/ml/weka.

[4] RUBiS distribution. http://forge.objectweb.org/project/showfiles.php?group_id=44.

[5] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in *Proc 6th Operating System Design and Implementation (OSDI)*, San Francisco, CA, USA, Dec. 2004.

[6] Elba project. http://www-static.cc.gatech.edu/systems/projects/Elba.

[7] M. Raghavachari, D. Reimer, and R. D. Johnson, "The deployer's problems: configuring application servers for performance and reliability", in *Proc. 25th International Conference on Software Engineering (ICSE)*, Portland, OR, USA, May 2003.

[8] J. R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers Inc, 1993.

[9] N. Friedman, and M. Goldszmidt, "Building classifiers using Bayesian networks", in *Proc. Thirteenth National Conference on Artificial Intelligence (AAAI96)*, 1996.

[10] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression: a statistical view of boosting," Dept. of Statistics, Stanford University Technical Report, 1998.

[11] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, "Dynamic provisioning of multi-tier Internet applications," in *Proc. Second International Conference on Autonomic Computing (ICAC)*, Seattle, WA, USA, 2005.

[12] R. Powers, M. Goldszmidt, I. Cohen, "Short term performance forecasting in enterprise systems," in *Proc. 11th conf. on Knowledge Discovery in Data (KDD '05)*, Chicago, IL, USA, 2005.