

# Detecting Bottleneck in $n$ -Tier IT Applications Through Analysis

Gueyoung Jung<sup>1</sup>, Galen Swint<sup>1</sup>, Jason Parekh<sup>1</sup>, Calton Pu<sup>1</sup>, and Akhil Sahai<sup>2</sup>

<sup>1</sup>CERCS, Georgia Institute of Technology  
801 Atlantic Drive, Atlanta, GA 30332

{gueyoung.jung, galen.swint, jason.parekh, calton}@cc.gatech.edu

<sup>2</sup>HP Laboratories

Palo-Alto, CA  
akhil.sahai@hp.com

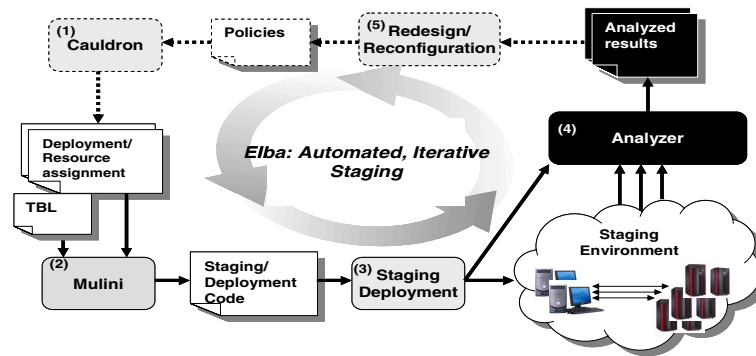
**Abstract.** As the complexity of large-scale enterprise applications increases, providing performance verification through staging becomes an important part of reducing business risks associated with violating sophisticated service-level agreement (SLA). Currently, performance verification during the staging process is accomplished through either an expensive, cumbersome manual approach or ad hoc automation. This paper describes an automation approach as part of the Elba project supporting monitoring and performance analysis of distributed multi-tiered applications that helps in bottleneck detection. We use machine-learning to determine service-level objectives (SLOs) satisfaction and locate bottlenecks in candidate deployment scenarios. We evaluate our tools with TPC-W, an on-line bookstore, and RUBiS, an on-line auction site.

**Keywords:** Bottleneck detection,  $n$ -tier application, Decision tree, SLOs, Elba.

## 1 Introduction

The increasing complexity of enterprise applications has emphasized the importance of verifying and validating the configuration performance prior to production use. While functional properties are typically verified during system integration and testing, performance as specified in SLOs of SLA is verified and validated by a pre-production process referred to as *staging*. Since a failure to fulfill SLA requirements results in business losses, staging has the critical role of verifying and validating the deployment plan to cover a wide range of system configurations and workloads. Current staging processes have been largely manual, augmented occasionally with ad hoc automation scripts, and these processes have become increasingly error-prone and costly in terms of time and effort. To reduce costs and increase the coverage of staging, the Elba project seeks to automate staging and tuning for  $n$ -tier applications in a distributed environment [4]. Automated staging and tuning uses high-level requirement specifications and translates them into both staging deployment and workload parameter settings which are then used for execution in a staging environment. Monitors first collect performance data, followed by analysis which automatically identifies performance deficiencies. Generating and analyzing performance data to uncover SLOs satisfaction and performance bottlenecks are significant challenges to realizing tiered, self-tuning applications.

Our prior work in the Elba project, represented in Figure 1, reported on the challenges of automatically mapping design specifications into deployment tool specifications for production and provided a solution using a code generation and translation tool [4][5].



**Fig. 1**<sup>1</sup>. Staging in Elba is an automated and iterative process. (1) Cauldron converts the policy documents into resource and deployment assignments. (2) Mulini re-maps resource assignments and application staging test guideline (TBL) to generate three types of code: instrumented application code, deployment code, and monitoring/analysis code. (3) A deployment tool installs and configures the application and then executes it. (4) Monitoring data is fed into analysis tools, and (5) the result of the analysis is handed to an engine to generate recommendations for policy changes. In this paper, we focus on (4), highlighted by black boxes, for the automated monitoring and analysis using generated code from Mulini code generator. Note that dashed boxes are on progress.

This paper presents our work on collecting data and analyzing bottlenecks for a significant number of performance metrics. It helps answers two questions:

- *Does the application configuration meet performance requirements?* This question is answered by observing metrics that correspond to policy objectives in SLOs.
- *If the requirements are not met, then where in the configuration is the bottleneck?* This question should be answered by examining metrics data establishing, first, what metrics are relevant, and second, which metrics best represent the bottleneck.

Neither task is trivial, but both are valuable for staging and production. Of course, over-provisioned systems can meet SLOs, but this entails additional capital outlays, maintenance, and sometimes over-engineering of the software itself [8]. Systematic, automated staging mitigates the risks of under- and over-provisioning and can provide valuable application behavior information applicable to the production application.

The contribution of this paper is an approach to support automated monitoring, analysis, and reporting by applying machine-learning in the context of staging. This automated approach will assist service providers in answering the previous two questions while preventing resource wastage through over-provisioning. With Mulini code generator improved from [4], our approach uses policy documents to generate metrics monitoring and performance analysis code and hooks into a machine-learning tool for

<sup>1</sup> We have slightly improved the figure of Elba used in [4] and [12].

automated bottleneck detection. We compared different classifiers and decided that the decision tree classifier (J48) was more robust in detecting bottlenecks [12]. In this article, we evaluate the accuracy of our bottleneck detection approach by analyzing two well-known benchmarking applications<sup>2</sup> that have differing bottleneck profiles, TPC-W and RUBiS.

The remainder of this paper is organized as follows. Section 0 presents the challenges and our approach to providing analysis support in the Elba project. Section 0 describes the evaluation environment, and Section 0 presents evaluation results for TPC-W and RUBiS. Section 0 discusses related work, and Section 0 presents our conclusions.

## 2 Automated Staging and Analysis

### 2.1 Challenges

For a distributed  $n$ -tier application, staging is an important, complex task that entails repeated tests over an extended period of time; the system and configuration are refined until they meet performance requirements. If performed early in the application development cycle, staging can provide crucial feedback that helps steer application development by identifying bugs, performance shortfalls, “hotspots,” and resource waste. Increasing application complexity makes staging worth automating to enable faster, earlier testing.

Staging may share some tools and techniques with production, but three important factors differentiate application staging from production. First, the hardware available in a staging environment mirrors but may not replicate exactly the production environment. Perfect duplication would provide higher application assurance once staged, but involves high costs in terms of initial expenditures and ongoing maintenance. To best utilize an approximate environment that minimizes the costs requires staging the application multiple times to establish predictive performance trends. Each staging iteration tests one particular application configuration and may involve multiple staging trials under varying staging parameters. A second differentiator is that applications may require additional fine-grained implementation to ascertain bottlenecks accurately which must be removed from the production code. Finally, staging requires the generation of synthetic workloads that stress the application similarly to production environments in a limited time period.

Automated analysis adds challenges to the staging process. First, automated analysis entails the orchestration of several tasks and may drive multiple executions with slightly different staging parameterizations. Second, automated analysis requires system and application instrumentation derived from performance requirements to record metrics data. Third, it requires the construction of an analysis, decision, and detection process which can answer the two questions presented in the introduction.

Automated analysis tools must translate policy-level documents into functional artifacts that become part of the staging process. Service-level indicators (SLIs) are

<sup>2</sup> In this paper, they are used as exemplar distributed multi-tiered e-commerce applications with defined metrics rather than as benchmarks; our results can not be used for performance comparisons outside this paper.

obtained from the SLA and its components, SLOs; these are translated into metrics and staging parameters. Furthermore, administrator/operator policies that may govern aspects such as acceptable resource usage must also be mapped into metrics. Test-specific information, such as machine locations, testing times, and workload, must be incorporated. Once recorded, a custom analysis engine automatically processes the data and compares its results against performance goals set forth in policy specifications.

Even automating bottleneck detection from gathered data requires the recognition and resolution of several problems. First, a single trial may not provide enough information to determine bottlenecks – a metric may appear “maxed” out even though it really reflects normal operating levels. In such cases, several trials of varying workloads are required to establish operating baselines and trends. Interactions between metrics can also make bottleneck detection difficult. For example, CPU usage and network throughput may trend in parallel, but only the CPU is the bottleneck. Finally, bottleneck detection requires sorting through copious metrics data. The total number of metrics varies with the number of both hardware and software components in the system, and they can be categorized generally as either application-level (e.g., the number of threads, the number of database connections, and elapsed query time) or system-level (e.g., CPU and memory utilization) metrics.

To detect bottlenecks and sort through the myriad metrics produced during monitoring, we employ an automated classifier. The input to the classifier is the first derivative of the metrics, since we are interested in trends. It is first trained by inputting the metrics with the result of a SLO-evaluator, a tool generated for deciding the violation of the SLOs. The output of the classifier is metrics whose derivatives correlate strongly to SLOs violation. From these identified metrics, we discover the bottlenecks of the system.

## 2.2 Automating Monitoring and Analysis

Our approach to automated staging and analysis occurs within the context of the Elba project [4]. The project goal is to first realize iterative staging to determine the inadequacies of application performance, then evaluate the results, and finally enable automated tuning of the system to meet the expected performance objectives. In particular, to integrate our automation for monitoring and analysis, we extended Elba’s Mulini code generator which employs XML/XSLT techniques with Aspect Oriented Programming (AOP) paradigm to create the necessary code for monitoring and analysis, including the instrumentation of source code for application-level metrics, and for the generation of the analysis code. Interested readers for code generation and Mulini can refer to [4][9] for more details. The metrics data can then be fed to a machine-learning tool to identify performance bottlenecks. The Analysis addresses the two questions posed in the introduction, namely,

### **Does the application configuration meet its performance requirements?**

This must be answered for each trial. Mulini generates an SLO-evaluator with policy-specific code that computes the individual satisfaction of the component SLOs. The SLO-evaluator uses data collected by the synthetic workload generators and computes application-specific throughput and average response time. Once the SLOs satisfaction is determined, overall SLA satisfaction can be determined.

**If the requirements are not met, then where in configuration is the bottleneck?**

This question is answered with aggregated data from multiple trials. If SLA is not met, the tools begin a three-step bottleneck detection process to correlate performance shortfalls and metrics. The detection process requires performance data from a series of trials. The first trial subjects the application to a low synthetic workload, and each subsequent trial increments the workload until consecutive trials fail the SLA. For example, a trial for a retail store application may begin with 10 concurrent simulated users, and then in each subsequent trial the number of concurrent simulated users increases by 10 over the previous trial until the SLA is violated for 70, 80, and 90 users.

In subsequent analysis, the first step is to determine the bottleneck tier. For each tier, the average duration spent by each service request is computed, and we identify the bottleneck tier as tier with the fastest growing duration (change in duration divided by the change in synthetic workload). The second step is to select, from the metrics of the bottleneck tier, the highly utilized metrics as candidate indicators. The assumption is that high utilization of a resource implies high demand from the application and a potential bottleneck. This also helps distinguish between highly-correlated metrics, such as bandwidth and CPU usage. To be considered a candidate indicator, a metric must either surpass 90% utilization or some threshold value as specified by a policy document, heuristic, or system administrator. The third step in bottleneck detection is to discover the metrics indicating bottlenecks using the aggregated performance data from all trials. For our applications and metrics, we have found that using the change in a metric from trial to trial provides a reliable indicator for correlating a metric to SLOs violation. The change, effectively a first derivative of the metric, will drop from some positive factor (utilization increases) towards around zero (utilization constant) when the underlying resource is fully utilized. In comparison, a non-bottleneck metric can continue to increase – constant growth does not correlate a change in SLOs satisfaction. In other words, our bottleneck detection searches out the metric that best correlates to reduced SLOs satisfaction (i.e., greater SLOs violation).

For the third step mentioned above, we apply machine-learning to form a decision tree where tree nodes embody if/then decisions based on growth in a metric (the delta metric value) and whose leaves embody overall SLOs satisfaction. After training, the set of nodes traced from a leaf (SLOs satisfaction) to the root will be a set of inequalities that is able to distinguish the leaf prediction attribute from the other prediction attributes. We categorize the satisfaction levels as quintiles since the decision tree classifier must have nominal types for prediction attributes. Five categories balance enough categories to allow correlations with each category to still collect multiple trials from the training set. By inspecting the generated decision tree, the bottleneck detection process is able to find the metric that was identified to have the highest correlation to SLOs. In situations where the decision tree consists of multiple metrics, the metric that appears most often in the tree will be selected as the highest potential bottleneck. To illustrate the bottleneck detection process, we present a sample scenario where CPU is the bottleneck metric and memory is shown to be high but not considered a bottleneck as it is cached data that accounts for most of the memory (the cached data will be replaced if an application requires additional memory). In Figure 2 (b), the utilization for both CPU and memory is shown along with the SLOs satisfaction at each workload trial. The first step for bottleneck detection is taking the difference of each metric's utilization across the change in workload trials. The trends

resulting from this are shown in Figure 2 (c) where the delta CPU utilization is somewhat linear until flattening out at 0% (in which case its utilization reaches 100%), and the delta memory utilization remains mostly constant around 0%-1%. By feeding this data to a decision tree classifier, we obtain a sample tree similar to Figure 2 (a). In this case, the CPU metric was chosen at each node as its delta is most correlated to the different SLOs satisfaction categories. Memory was not chosen as it is not possible to use the delta memory utilization to differentiate each of the SLOs satisfaction categories.

### 3 Evaluation Environment

We evaluated the described automation approach by using TPC-W, an on-line bookstore application for a transactional web-based e-commerce benchmark [2][3], and RUBiS, e-commerce application implementing the core features of an online auction site [2]. These applications have differing performance characteristics, as described in [2]. In both, customer interaction is simulated by remote simulated browsers that send and receive HTTP messages. Each simulated browser starts from a home interaction and executes another interaction after “thinking” for a random period of time. The visitation path is governed by a chosen transition matrix which encodes probabilities for visiting the next page according to current visiting page. For our tests, we chose shopping transition and bidding transition models for TPC-W and RUBiS, respectively, since these are the most representatives of the workload of these applications as described in [2][4].

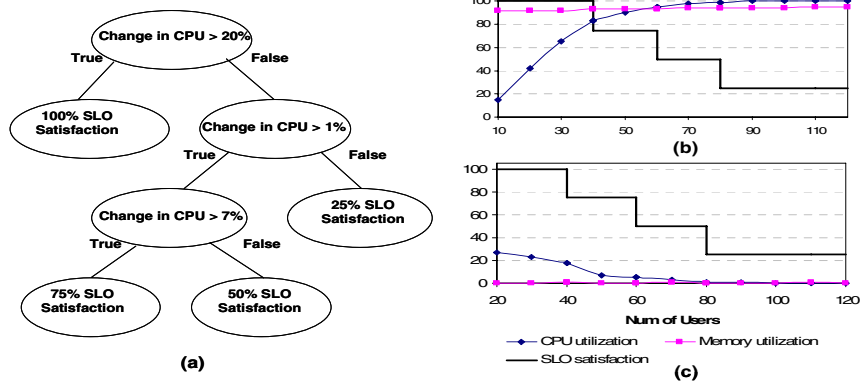


Fig. 2. (a) decision tree, (b) metric utilization, and (c) delta graph

In our evaluation, we used two software architectures common in the e-commerce domain: Java servlets for TPC-W and Enterprise Java Beans for RUBiS. A minimum configuration of the TPC-W in our evaluation consists of a web server (Apache), a servlet engine (Apache Tomcat), and a database server (MySQL) each running on a dedicated host; a minimal TPC-W installation requires three machines. A minimal RUBiS configuration comprises a web server (Apache), a servlet engine (Tomcat), an EJB server (JONAS), and a database server (MySQL). The servlet engine and EJB server (the application tier) share a single machine. Beginning with these minimal

configurations, we iterate through more complex configurations by employing higher-performance machines or adding new machines to each bottleneck tier until a configuration satisfies the given SLA.

We employ two classes of hardware in our evaluation. A low-end machine, L, is a Pentium III 800MHz dual-processor with 512 MB memory, and a high-end machine, H, is a Xeon 2.8GHz dual-processor with 4GB memory. A configuration may combine these two classes of hardware. For instance, the L/2H/L configuration represents one low-end machine for a web server, two high-end machines as application servers, and one low-end machine for a database server. All machines are connected through 100 Mbps Ethernet. For basic cost accounting of the configurations, we assign low-end machines a cost of \$500 and high-end machines a cost of \$3500.

## 4 Evaluation Results

### 4.1 Automated Analysis for TPC-W

Consistent with [2], the SLO-evaluator indicates that the L/L/L configuration for TPC-W fails the SLA of 10.7 WIPS with average response times less than 500 ms at staging with 150 concurrent simulated users. This triggers the process of the automated bottleneck detection with aggregated monitoring results of the L/L/L configuration, which is adjusted, and then re-staged iteratively until a configuration satisfies the SLA.

From the first step of the automated bottleneck detection process, the automated bottleneck detection identifies the database server tier as the bottleneck tier. Figure 3 shows the results of application-level monitoring in the L/L/L configuration of TPC-W. Mulini weaves monitoring code with the TPC-W application source code to record response times elapsed in database

queries and execution for the presentation and business logic of “BestSeller” interaction, a representative TPC-W interaction. This figure shows that the duration of the database tier grows fastest. That is, the database server tier dominates the overall response time of the interaction. In fact, the average duration for executing database queries is about 89s while the average duration both for executing presentation and business logic and for executing requests forwarding at web server are about 1.3s when the synthetic workload generators run 150 concurrent simulated users. Once the database server tier is identified as a bottleneck, the bottleneck detection proceeds to the thresholding step, which focuses on metrics that indicate high-resource utilization.

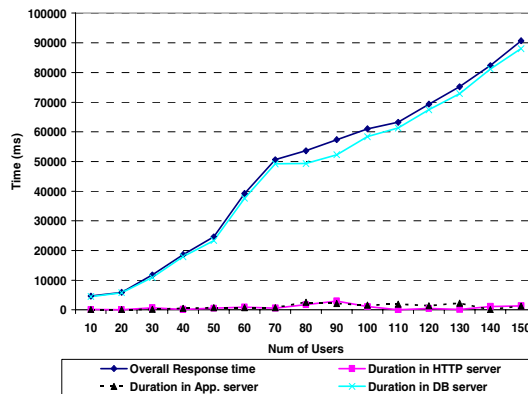


Fig. 3. Average response time and duration in each tier of BestSeller interaction

Figure 4 (a) displays each metric as a percentage of its maximum capability. Any metric not reaching 90% utilization is automatically dropped from bottleneck consideration. We can see from the figure that the metrics reaching the 90% threshold are the CPU and overall memory usage.

The final step of bottleneck detection is training the J48 decision tree classifier (WEKA toolkit’s implementation of the C4.5 decision tree [6]) to locate metrics that most influence the SLOs satisfaction. Our classifier is trained with the nine derivative metrics values and the first order derivatives of the metrics to identify trends rather than the values of metrics to SLOs satisfaction. In our experience using only metric values can lead to false conclusions about which metrics are the real bottlenecks as is illustrated in this case by overall memory utilization. In Figure 4 (a), we see that the overall memory usage value is about 98% under a load of 100 concurrent simulated users. Note that the memory usage of database processes is very low. If we turn our attention to the derivatives of the metrics in Figure 4 (b), our inquiries are guided a different direction. The trend of the memory usage derivative is nearly constant. In this case, it turns out that memory is not the bottleneck because the high utilization

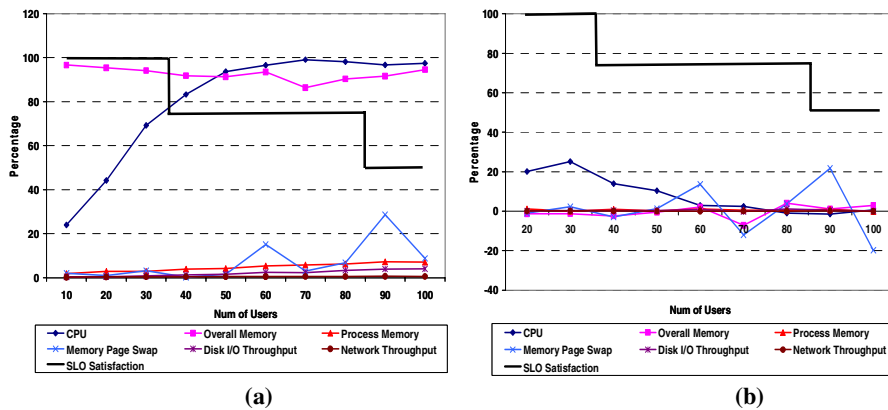


Fig. 4. For the TPC-W database tier, (a) metric values and (b) their derivatives

stems from OS caching. Taking the derivative screens out linearly increasing metrics, which correlate but have adequate headroom for growth. The metric with the highest correlation to SLOs satisfaction is the CPU usage since the inequality “change in CPU usage  $> 20$ ” can be used to differentiate the 100% SLOs satisfaction. The decision tree further differentiates the 75% SLOs satisfaction with the inequality “change in CPU usage  $> 2$  and change in CPU usage  $< 15$ ”.

Since the CPU of the database server tier limits performance, we set Elba to first increase the number of low-end database server machines. This approach is much cheaper than the approach employing a few high-end machines in terms of configuration cost. The results of several iterations are shown in Figure 5. We see that only the L/L/H2L configuration (cost \$5500), in which we use one high-end and two low-end machines as database servers, and the more-costly L/L/2H (cost \$8000) configuration satisfy the given SLOs (configurations arranged by increasing cost). To show that neither the web server nor the application server is the bottleneck, we set Elba to conduct



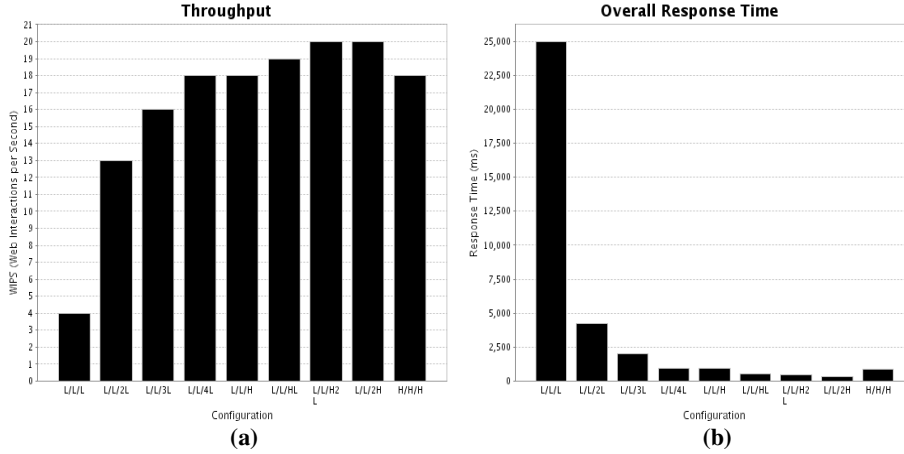


Fig. 5. TPC-W iterative staging results (a) WIPS and (b) overall average response time

extra staging with the H/H/H configuration. Fig. 5 shows that the performance results of both the L/L/H and the H/H/H configurations are almost identical in terms of WIPS and overall average response time even though we use high-end machines for both the web and the application servers. Therefore, H/H/H configuration is discovered as an over-provisioning.

Figure 6 breaks performance into per-interaction SLOs which must meet a 90% SLOs satisfaction level. Configurations that employ only cheap machines like L/L/3L cannot meet the SLOs in most interactions of the TPC-W. Using a single high-end as database server, L/L/H, also fails the SLOs for “BestSeller” and “BuyConf”. The L/L/H2L configuration narrowly meets these SLOs, and L/L/2H is clearly sufficient. From this staging result, the service provider can choose either a configuration at lower cost with less growth potential (i.e., L/L/H2L) or higher cost with high growth potential (i.e., L/L/2H).

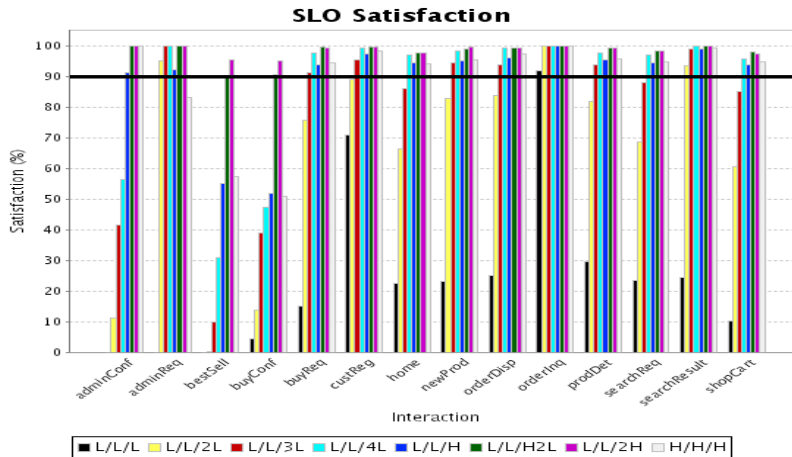


Fig. 6. TPC-W, per-interaction 90% SLOs satisfaction

## 4.2 Automated Analysis for RUBiS

Our SLO-evaluator indicates the L/L/L fails the target SLA of 25.7 WIPS and overall average response time of less than 500 ms under a load of 360 concurrent simulated users. This triggers the process of the automated bottleneck detection just as we have done with the TPC-W evaluation.

In the first step with “SearchItemsInCategory” interaction, we found that the application server tier dominates the overall response time of the interaction. In fact, the average duration for executing presentation and business logic (i.e., time spent in servlets and EJBs) is about 28s while the average duration in database server tier for executing database queries is about 30ms, and web server tier for forwarding requests and responses 104ms with 360 concurrent users.

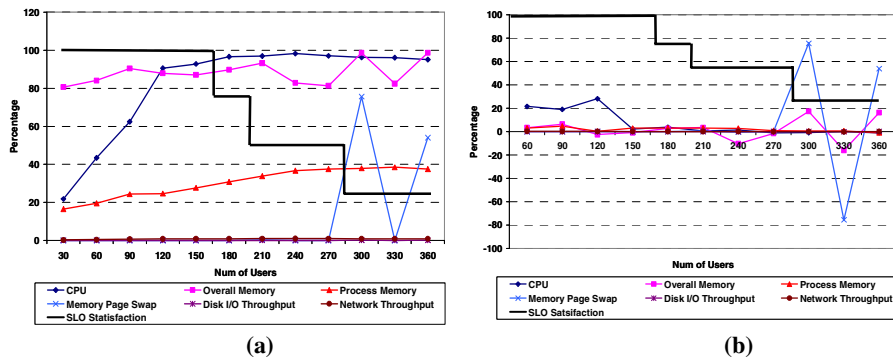


Fig. 7. For RUBiS app server tier, (a) metric values and (b) derivative values

Figure 7 (a) displays that the only metrics reaching the 90% threshold are the CPU and overall memory usage. In the final step, the decision tree classifier is trained using eleven instances. In Figure 7 (a), we see that the overall memory usage value is about 98% under a load of 360 concurrent users. However, its trend linearly increases. In Figure 7 (b), the trend of the memory usage derivative is first nearly constant and then erratic. Taking the derivative screens out the jittery metrics, which have no correlation, and linearly increasing metrics, which correlate but have adequate headroom for growth.

The metric with the highest correlation to the SLOs satisfaction is CPU usage since the inequality “change in CPU usage  $> 4$ ” can be used to differentiate the 100% SLOs satisfaction. The decision tree would be able to further differentiate SLOs satisfaction with a more fine-grained distinction since the CPU reaches its peak of near-100% utilization and the derivatives approach much smaller values. The inequality “change in CPU  $< 1.5$  and change in CPU  $> -1$ ” distinguishes the 50% SLOs satisfaction.

## 5 Related Work

Argo/MTE [1] uses automation and code generation via XSLT to evaluate middleware implementations. The Weevil framework supports the management of testing in

widely distributed systems again using a generative programming approach [11]. Weevil's focus has been on automating deployment and workload generation for applications utilizing overlay networks. Our work targets the enterprise  $n$ -tier IT environment and applications and emphasizes the re-use of existing policy-level specifications for automation of both performance testing and bottleneck identification with machine-learning technique.

Te-Kai et al. [10] have provided a capacity sizing tool to recommend cost-effective hardware configuration for integrated business processes; their tool is tailored to the WebSphere InterChange server. It assumes a prototype of the system is not available for system staging. Instead, it relies on similar previously benchmarked systems to predict capacity. The Elba project is geared towards staging an application that will be deployed to a production without pre-existing performance data. Our approach for bottleneck detection shares similarities with [7], but their work targets production systems to forecast problems; our work intervenes during application design to locate candidate bottleneck points, and our system also emphasizes automation support for testing alternative designs.

## 6 Conclusion

With the increasing complexity of large-scale enterprise applications, effective staging can ensure the SLA performance goals of complex application configurations. The goal of the Elba project is to automate iterative staging. The main contribution of this paper is the automated monitoring and performance analysis of large-scale applications through a decision tree approach for bottleneck detection assisted by code generation techniques. From declarative specifications of distributed  $n$ -tier applications, we generated the code to collect, process, and analyze performance data (e.g., SLOs satisfaction levels) to locate performance bottlenecks in configurations being staged.

Our evaluation results of TPC-W and RUBiS demonstrated the feasibility and effectiveness of automating the monitoring and performance analysis in the staging process. By generating and running various configurations, our tools analyzed the SLOs satisfaction levels, found potential bottlenecks, and guided the reconfiguration process towards the lowest cost solution. The analysis tool utilized simple machine-learning techniques to classify the resource consumption metrics and find potential bottlenecks.

## References

- [1] Cai, Y., Grundy, J., and Hosking, J.: Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool. Int. Conf. on Automated Software Engineering, Linz, Austria, Nov. 2004.
- [2] Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., and Zwaenepoel, W.: Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. Int. Middleware Conf., Rio de Janeiro, Brazil, June 2003.
- [3] García, D., and García, J.: TPC-W E-Commerce Benchmark Evaluation, *IEEE Computer*, Feb. 2003.

- [4] Swint, S. G., Jung, G., Pu, C., and Sahai, A.: Automated Staging for Built-to-Order Application Systems. Network Operations and Management Symposium, Vancouver, Canada, April 2006.
- [5] Sahai, A., Pu, C., Jung, G., Wu, Q., Yan, W., and Swint, S. G.: Towards Automated Deployment of Built-to-Order Systems, Distributed Systems; Operation and Management, Barcelona, Spain, Oct. 2005.
- [6] WEKA distribution. <http://www.cs.waikato.ac.nz/ml/weka>.
- [7] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J.: Correlating Instrumentation Data to System States: A building block for automated diagnosis and control. Operating System Design and Implementation, San Francisco, CA, USA, Dec. 2004.
- [8] Sauvé, J., Marques, F., Moura, A., Sampaio, M., Jornada, J., and Radziuk, E.: SLA Design from a Business Perspective, Distributed Systems: Operation and Management, Barcelona, Spain, Oct. 2005.
- [9] Swint, S. G., Pu, C., Consel, C., Jung, G., Sahai, A., Yan, W., Koh, Y., and Wu, Q.: Clearwater - Extensible, Flexible, Modular Code Generation. Int. Conf. on Automated Software Engineering, Long Beach, CA, USA, Nov. 2005.
- [10] Te-Kai, L., Hui, S., and Kumaran, S.: A capacity sizing tool for a business process integration, Int. Middleware Conf., Toronto, Ontario, Canada, Oct. 2004.
- [11] Wang, Y., Rutherford, M., Carzaniga, A., Wolf, A.: Automating Experimentation on Distributed Testbeds, Int. Conf. on Automated Software Engineering, Long Beach, CA, USA, Nov. 2005.
- [12] Parekh, J., Jung, G., Swint, S. G., Pu, C., and Sahai, A.: Comparison of Performance Analysis Approaches for Bottleneck Detection in Multi-Tier Enterprise Applications, Int. Workshop on Quality of Service, Yale University, New Haven, CT, USA, June, 2006.