Bottleneck Detection Using Statistical Intervention Analysis

Simon Malkowski[1], Markus Hedwig[1], Jason Parekh[1], Calton Pu[1], and Akhil Sahai[2]

[1]CERCS, Georgia Institute of Technology,
266 Ferst Drive, Atlanta, GA 30332
{simon.malkowski, markus.hedwig, jason.parekh, calton}@cc.gatech.edu
[2]HP Laboratories, Palo-Alto, CA
akhil.sahai@hp.com

Abstract. The complexity of today's large-scale enterprise applications demands system administrators to monitor enormous amounts of metrics, and reconfigure their hardware as well as software at run-time without thorough understanding of monitoring results. The Elba project is designed to achieve an automated iterative staging to mitigate the risk of violating Service Level Objectives (SLOs). As part of Elba we undertake performance characterization of system to detect bottlenecks in their configurations. In this paper, we introduce our concrete bottleneck detection approach used in Elba, and then show its robustness and accuracy in various configurations scenarios. We utilize a well-known benchmark application, RUBiS (Rice University Bidding System), to evaluate the classifier with respect to successful identification of different bottlenecks.

Keywords: Bottleneck detection, statistical analysis, enterprise systems, perforance analysis

1 Introduction

Pre-production configuration testing of complex n-tier enterprise application deployment, or *staging*, can be as demanding and complex as the production system itself. System analysts and administrators monitor and analyze a large number of application-specific metrics such as the number of active threads and the number of EJB entity bean instances, along with system-level metrics like CPU usage and disk I/O rate. Any of these resources may cause the system to violate performance service level objectives (SLO), usually specified as service level agreements (SLA). Significantly lower cost and results at higher confidence levels may be produced by automated staging. The latter is an iterative process in the Elba project [11] whereby an application configuration is gradually refined.

The main contribution of this paper is an automated bottleneck detection scheme based on a statistical intervention model. This approach is distinct from our previous work that used machine learning [4]. We introduce a deterministic algorithm, which has proven to be very effective in the Elba environment. The process automatically examines and analyzes the entire metric data derived from the staging experiment trials. A limited set of interesting metrics is identified very fast without the need of an extensive training or configuration phase. This set is then ordered according to the degree of correlation with the high-level system performance. We also show that we are able to accurately determine potential bottlenecks in different scenarios. Moreover, the resulting output is easily interpretable due to the intuitive model structure.

The remainder of this paper is organized as follows. Section 2 describes our approach to bottleneck detection using intervention analysis. Section 3 presents the evaluation environment and our results utilizing RUBiS. Section 4 discusses related work, followed by the conclusion.

2 Intervention Analysis

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. So far our bottleneck detection approaches consisted of a multi-step analysis. If a SLA was not met (SLO-satisfaction drops significantly) in a certain scenario, a three-step detection process began: staging the system with varying workloads and collecting performance data from system-level and application-specific metrics, training a machine learning classifier with the data, and finally querying the trained machine learning classifier to identify potential bottlenecks. Please refer to [4] for more details. While our three-step methodology proved to be successful, it mainly relies on machine learning algorithms to execute the final performance modeling and classification. This implies two typical shortcomings that lie in the nature of the modeling scheme. Firstly, the machine learning classifiers require a training phase. This can be cost-intensive since certain accuracy and robustness levels might be defined *a priori*. Secondly machine learning classifiers produce a model that is not necessarily interpretable in a trivial manner. We discussed suitable interpretations in [4]. Nevertheless, this led to a residual degree of uncertainty in the interpretation of the analysis results.

In this article we propose a novel approach based on statistical techniques, which results in an improvement of our bottleneck detection process in a consistent manner. We introduce an intuitive statistical model, which eliminated the need of machine learning on the one hand. And on the other, we observe that our approach achieves the same high accuracy level at a lower cost (fewer staging trials). Therefore we greatly increase the efficiency of the detection process and enhance the clarity of the final results at the same time.

2.1 Assumptions

The following assumptions form the basis of our automated bottleneck methodology. They emphasize the general issues that need to be addressed by any satisfactory detection method and are reflected in previous Elba efforts.

- A single experiment trial is not sufficient to record a conclusive metric vector and thus several trials of varying workloads are required.
- Non-obvious interactions between resources make observation based bottleneck detection a hard problem. Nontrivial correlations have to be examined and the detection method needs to be able to produce a probabilistic result ranking.
- The number of recorded monitoring metrics is very high. It is critical to device an approach that is able to sort through copious metric data automatically.
- The nature and appearance of metrics can vary significantly and they are typically categorized as either system-level or application-specific.
- High utilization of a resource implies high demand from an application while it may not necessarily be indicative of a bottleneck. A detection mechanism has to be capable of distinguishing bottlenecking behavior in terms of resource saturation.
- Especially trend changes in metric graphs are of high importance. In fact we found in our previous work that it was highly effective to examine first derivative approximations instead of the actually recorded values.

2.2 The Detection Model

These assumptions together with observations from empirical data analysis suggest a simple performance model. We formulate the latter in terms of statistical intervention analysis, which allows us to formalize the characteristic bottleneck behavior of the system accurately.

First we need to define an exogenous crossover point ($c \in WS$). This specific number of concurrent user sessions can be seen as an intervention point that divides our workload span (WS) into two disjunctive intervals:

$$I := \{w \in WS : w < c\} \qquad (1) \qquad\qquad I' := \{w \in WS : w \geq c\} \qquad (2)$$

In this notation I represents the set of workloads that result in high levels of SLO satisfaction of the system, whereas the satisfaction levels drop significantly when exposed to workloads in I' (intervention effect).

For our purposes we also need to adapt the standard transfer functional model formulation [1]. For any workload $w \in WS$ an impact assessment model for the first difference of any metric value Y_w can be formulated in terms of Equation 3. Note that we use the first difference as approximation of the first derivative consistently with our findings in [4].

$$\nabla Y_w := f(I_w) + N_w + \mu \qquad (3) \qquad\qquad I_w := \begin{cases} 1 & \text{for } w \geq c \\ 0 & \text{else} \end{cases} \qquad (4)$$

In this formulation N_w is the noise component, and μ denotes the constant term. The effect of the intervention variable I_w on the metric trend is defined as $f(I_w)$. Following the standard notation, I_w is defined as an indicator function (Equation 4). We can now subtract the noise component from both sides of Equation 1. Since we only have to deal with abrupt and permanent intervention effects we can assume linearity in the metric values. Based on this linearity assumption we introduce δ as the constant term of the intervention effect, which yields the following formulation:

$$\nabla Y_w - N_w = \delta I_w + \mu \qquad (5)$$

In order to characterize the final model in a convenient manner, we define μ' in Equation 6 which leads to the final model formulation in Equation 7.

$$\mu' := \mu + \delta \qquad (6) \qquad\qquad \nabla \tilde{Y}_w := \nabla Y_w - N_w = \begin{cases} \mu & \text{for } w < c \\ \mu' & \text{for } w \geq c \end{cases} \qquad (7)$$

This notation emphasizes the importance of the potential change in the trend of the metric value Y_w as the system progresses from I to I' with increasing workload. Moreover, it allows us to establish causality between the model parameters of the low level metric and the high level system performance in an intuitive manner.

2.3 Determining an Intervention Point

Since the crossover point (c) between I and I' needs to be defined *a priori*, we define an iterative algorithm for our automated analysis scheme. The main idea is to asses the workload when the SLO-satisfaction (SAT_w) looses its stability and starts to deteriorate significantly upon further workload increase (i.e. we assume Property 8 and 9). Although the model formulation requires an exact transition point, it is sufficient for our method to approximate c in a qualitative manner (refer to Table 4).

$$\forall_{i \in I} : SAT_i \approx \text{const} \qquad (8) \qquad\qquad \forall_{i \in I'} : SAT_i << \frac{1}{|I|} \sum_{j \in I} SAT_j \qquad (9)$$

We start at the lowest workload in our dataset and iteratively increase the value by the smallest possible step. In every iteration we calculate a simple heuristic approximation of the ninety-five percent confidence interval of the SLO satisfaction values seen so far. We consider n_0 values which resulted from a workload smaller or equal to $w_0 \in WS$ (the workload currently examined).

$$90\% \leq \frac{1}{n_0} \sum_{0 \leq i \leq w_0} SAT_i - \frac{1.96}{\sqrt{n_0 - 1}} \sqrt{\sum_{0 \leq i \leq w_0} (SAT_i - \frac{1}{n_0} \sum_{0 \leq j \leq w_0} SAT_j)^2} \qquad (10)$$

We continue to the next iteration as long as the lower bound of the confidence interval is not below ninety percent (Equation 10). Thus we characterize the satisfaction level for the first interval in a binary fashion as suggested by our observations. Once the lower bound of the confidence interval drops below ninety percent we exit the algorithm. The exit point $w^* \in WS$ is a heuristic approximation of the crossover point c. We can assume that the SLO satisfaction has deteriorated significantly from its stable level for all workloads greater or equal to w^*, which yields the following formulation:

$$\hat{I} := \{ w \in WS : w < w^* \} \qquad (11) \qquad\qquad \hat{I}' := \{ w \in WS : w \geq w^* \} \qquad (12)$$

2.4 Metrics Selection Scheme

We can now turn to the process of selecting a set of potential bottleneck metrics and discarding all metrics that do not indicate a high resource saturation level. Given a known intervention (SLO begins to deteriorate) we identify all metrics that show evidence of a corresponding plateau (i.e. significant and permanent shift in average value) and a variability change in their first derivative (further evidence for a saturated resource). To identify the candidate set we perform a basic hypothesis-testing scheme adapted from [10]. We define a rule-based analysis process for testing the null hypothesis (13) of constant mean μ and variance σ between the two intervals.

$$H_0 : \ \hat{\mu} \approx \hat{\mu}' \ \wedge \ \hat{\sigma} \approx \hat{\sigma}' \qquad (13)$$

Empirical testing revealed that we have to account for the high variability of the metric data as well as adjust the analysis to specifically detect abrupt plateau shifts. Thus we deviate from the traditional intervention analysis methodology and devise a different testing scheme. We calculate representative quantiles for each interval and metric. The latter characterize the filtered behavior of the data in a more stable manner. We proceed to apply two selection rules in order to limit the group of candidate bottleneck metrics.

$$q_{0.5} > q'_{0.5} \ \wedge \ |q_{0.2} - q_{0.8}| > |q'_{0.1} - q'_{0.9}| \qquad (14)$$

Rule 14 accounts for all limited metrics that will saturate at a level of hundred percent. We choose all metrics where the median has decreased as well as where the distance between ten- and ninety-quantile in the second interval is smaller than the distance between twenty- and eighty-quantile in the first interval. If this rule is satisfied we have significant evidence to reject the null hypothesis and assign the metric to a set of potential bottlenecks.

$$q_{0.9} < q'_{0.1} \ \wedge \ q_{0.9} < q'_{0.5} \ \wedge \ q_{0.9} < q'_{0.9} \qquad (15)$$

Rule 15 accounts for all metrics that are not limited and show an exponential behavior when the resource saturates. We select all metrics, where all three quantiles of the second interval have increased above the ninety quantile of the first one. Again we reject the H_0 if the rule applies. In this manner we have eliminated all metrics that do not show strong indications of bottlenecking behavior near the intervention point and narrowed our attention to potentially interesting resources. Note that the complete empirical derivation of the two decision rules is omitted due to space restrictions. Nevertheless it is based on standard statistical methods and our analysis experience.

2.5 Impact Assessment

Once we have identified the set of candidate bottlenecks we can perform a ranking to describe the magnitude of the change. The magnitude reveals the correlation with the intervention and specifically accounts for the exact time when the change in the metric occurred. Hence we design a normalizing ranking function R by calculating the quotient of the absolute mean values of the two intervals:

$$R := \left| \frac{\hat{\mu}}{\hat{\mu}'} \right| \qquad (16)$$

This mechanism has two implications. Firstly, we assess how well the crossover point was chosen for each particular metric (temporal ranking). If the split is not exact, the resulting quotient will have a value closer to one. Furthermore, we have to rank how large the relative shift in plateau levels is for each particular metric. We expect bottlenecked metrics that were chosen with Rule 14 (limited metric) to display a very high-ranking value potentially approaching infinity. The slope of the metric values drops from a linear increase to a stable value near zero. Metrics chosen by Rule 15 (unlimited metrics) will show a very low ranking value that is close to zero on the other hand. This means that a moderate positive slope changes to a very strong (exponential) growth. In the following evaluation we will subdivide the candidate set into set one and two. This will simplify the analysis for limited and unlimited metrics, respectively.

3 Experimental Evaluation

Rice University Bidding System, is a multi-tiered e-commerce application with 26 interaction types, such as browsing, 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. In our experiments the write ratio is extended adding additional variability as explained in [6]. We utilize the bidding transition as well as neighboring write ratios of 10% and 20% in our evaluation since these transitions are better representatives of an auction site workload [2] and provide a more accurate picture [6]. Our system reuses and extends a recent version of RUBiS from ObjectWeb [13]. 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 [2].

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 the 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 an 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. Sysstat 7.0.2 was used for system resource utilization tracking.

The experimental setup was deployed on two different clusters. Our initial data set (used in Section 3.1) was collected on the Georgia Tech Warp Cluster. This cluster is comprised of 56 Intel Blade Servers with Red Hat Enterprise Linux 4, with Linux kernel version 2.6.9-34-i386 as operating system. Each server is equipped with two Xeon 64-bit 3.06 GHz CPUs, 1 or 2 GB main memory, 1 Gbps network adapter, and a 5400 RPM disk with 8 MB cache. The second cluster used for the data generation was the Emulab [12], which provides more than 200 servers of different types. Emulab also allows the physical separation of experiments by simulating a local network topology for each experiment. The results detailed in Section 3.3 incorporate two types of servers. Primarily we employed a high end system with one Xeon 3.0Ghz 64 bit CPU, 2 GB main memory , six 1 Gbps network adapters, and a 10000 RPM disk. In order to change the bottleneck pattern we also used low end machines with a Pentium P3 600 MHz processor, 256 MB main memory, five 100 Mbps network adapters, and 7200 RPM disk. Both server types ran with Red Hat Enterprise Linux 4, with Linux kernel version 2.6.9-34-i386.

3.1 Bottleneck Detection in the 1/1/1 configuration

In this section we present a walk-through of our bottleneck detection process for a 1/1/1 configuration (no server replication in the tiers). The RUBiS benchmark was set to use the bidding transition matrices and the workload was incremented in steps of two.

The graphs in Fig. 1 show two different representative metrics in all tiers. The SLO satisfaction is graphed against the CPU usage in (a) and against the memory usage in (b). The depicted satisfaction level needs to be calculated by for each trial. This is conveniently resolved by the policy specific SLO-evaluator that is generated by Mulini. Individual satisfaction levels are determined for each of the SLO components. Interested readers can refer to [4] and [11] for more details. For simplicity reasons we solemnly use the response time of the system from the SLO-evaluator in the presented analysis. It is clearly visible that the satisfaction level meets our assumptions. It remains very stable up to around one hundred and fifty users, and decreases rapidly once this point is crossed. The CPU usage of the application server increases linearly and saturates at 100% when the SLO satisfaction drops down to around 85%. Furthermore, it is evident that the variability of CPU usage strongly decreases at the same time, signifying the maximal saturation level. The trends of other CPU utilizations remain linear and stable on the contrary. In Fig. 1 (b) we can see that the memory of the application server and database server is underutilized. Both show a fairly stable linear trend. Although the memory usage of the HTTP server is somewhat high its trend is almost flat. The variability of all other metrics stays fairly constant throughout the whole experiment. Following the argument in [4] we can regard the application server CPU a typical representative of a single bottlenecked resource that will show a strongly non-stationary behavior in its delta values (first difference normalized by the step-width). All other delta series will retain a stable behavior throughout the entire workload span.

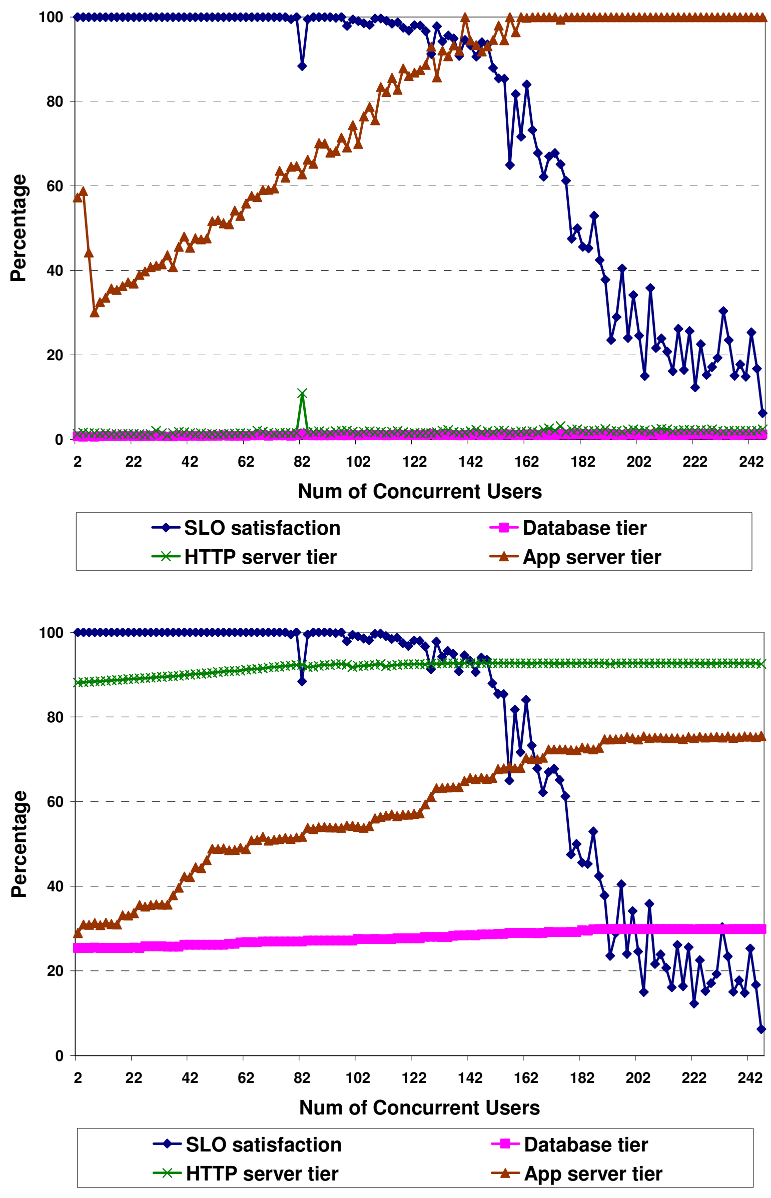

Fig. 1. SLO satisfaction against (a) CPU usage metric and (b) memory usage metric in all tiers.

Table 1. Heuristic approximation of the intervention point.

AVG [%]	ST-DEV	CI-LB	WL
98.39	2.37	93.75	148
98.31	2.46	93.49	150
98.17	2.72	92.84	152
98.09	2.81	92.58	154
97.79	3.83	90.28	156
97.63	**4.05**	**89.70**	**158**
97.40	4.53	88.52	160
97.26	4.66	88.12	162

We can now turn to the actual detection process. Table 1 contains the output of the algorithm used to determine the intervention as described in Section 2.3. We performed the analysis on a dataset consisting of one hundred and seventy-five staging trials. The number of concurrent users ranged from two to three hundred and fifty. The lower bound of the confidence interval drops below ninety percent for a workload of one hundred and fifty-eight. According to Section 2.3 this defines our heuristic approximation of the crossover point *c*.

The final outcome of our heuristic testing scheme (2.4) and the impact assessment (2.5) for a limited dataset (first one-hundred and twenty-five trials) are summarized in Table 2.

Table 2. Results of the bottleneck detection process.

Metric	$q_{0.1}$	$q_{0.5}$	$q_{0.9}$	$q'_{0.2}$	$q'_{0.5}$	$q'_{0.8}$	R
APP_CPU	-1.01	0.05	0.74	-0.62	-0.02	0.58	70.05
DB_KBCached	-3.80	4.63	12.39	-3.56	3.26	11.46	11.72
APP_KBBuffers	7.40	53.00	170.60	0.00	0.00	39.00	3.33
DB_CPU	-0.03	0.00	0.04	-0.03	0.00	0.03	2.81
APP_Memory	-0.15	0.08	1.04	-0.14	0.03	0.38	2.26
WWW_CPU	-0.07	0.00	0.06	-0.05	0.00	0.06	1.08

The two delta value intervals I and I' are [4-156] and [158-250] respectively. While Rule 14 returns six hits, Rule 15 results in an empty set of candidate bottlenecks. All other metrics are discarded automatically. The values in the last column reveal the application tier CPU as most likely bottleneck. Thus our model has correctly detected the bottleneck in this scenario. For further understanding of the table it is important to note that we map all related values to its resource for the final interpretation. Therefore the *R*-value of the APP_CPU can result from more than one metric (e.g. the overall usage or the system usage) for instance.

We now proceed to demonstrate the robustness of our method when subjected to various configuration settings. We show that our approach is highly robust with regard to variations in the width of the intervals (Table 3), the position of the crossover point (Table 4), and the step-width between the different workloads (Table 5). Table 3 shows the value of the ranking function depending on the length of the two input intervals. Our model predicted the bottleneck correctly each time. We see that the width of the second interval influences the magnitude of the *R*-value strongly.

Table 3. Ranking function value against interval width.

I	*I'*	*R**	Pred Acc
[4;156]	[158;206]	6.94	1
[4;156]	[158;256]	68.32	1
[4;156]	[158;350]	13.92	1
[58;156]	[158;206]	3.83	1
[58;156]	[158;256]	7.66	1
[58;156]	[158;350]	15.01	1
[108;156]	[158;206]	4.33	1
[108;156]	[158;256]	8.67	1
[108;156]	[158;350]	17.11	1

We now examine the impact of different choices of crossover point values, which are summarized in Table 4. Within certain intuitive limits the model predicts correctly. This is of special importance since the determination of the intervention point is the result of the algorithm in Section 2.3. We see that its heuristic character is justified in the nature of the data.

Table 4. Ranking function value against different crossover points.

I	*I'*	*R**	Pred Acc
[16;114]	[116;214]	-	0
[26;124]	[126;224]	-	0
[36;134]	[136;234]	5.09	1
[46;144]	[146;244]	6.86	1
[56;154]	[156;254]	7.93	1
[66;164]	[166;264]	∞	1
[76;174]	[176;274]	59.43	1
[86;184]	[186;284]	0.93	0

Finally we turn to Table 5 and the analysis of the robustness of the choice of step-width. The table contains the ranking function values for different step-widths.

Table 5. Ranking function value against the step width.

I	*I'*	Step	# Trials	*R**	Pred Acc
[4;156]	[158;350]	2	174	13.92	1
[4;156]	[160;348]	4	87	12.58	1
[4;156]	[164;348]	8	44	188.60	1
[4;148]	[164;340]	16	22	112.17	1
[4;132]	[164;324]	32	11	∞	1

At first it looks surprising that the ranking values increase almost monotonically as the step-width increases. Nevertheless, this is due to the stochastic nature of our data and the fact that with increased step width the two intervals are separated farther apart. By increasing the observable change in the bottleneck metric the results become clearer. This proves the robustness of our method and reveals its effectiveness when exposed to data of higher step-width.

3.2 Performance Comparison of the Analysis

In this section we present the results of our experiments across a wide range of configurations to show how the method evaluates a changing bottleneck pattern. This data was collected on Emulab [12] with different write ratios (WR0.1 and WR0.2) and server configurations (H and L).

Table 6 lists the top candidate bottleneck metrics and their *R*-value in the last two columns. We also applied a simple heuristic filtering mechanism to discard uninteresting ranking values. The latter eliminates the problematic behavior of some utilization values for instance, which was detailed in our previous work. We automatically discard values if the utilization does not cross a certain threshold (90% in our case) [4]. The table shows that our methodology is able to detect the shifting bottleneck as we progress to higher replication levels of the application server tier.

Table 6. Top candidate bottleneck metrics.

Config	WR	*WS*	# Trials	*c*	S1/S2-Sz	Result Metric	*R**
H/2H/H	20%	100-600	51	440	11/2	APP_CPU	73.02
H/4H/H	20%	540-1040	51	844	4/1	APP_CPU	16.61
H/6H/H	20%	1040-1448	51	1264	7/0	APP_CPU	13.10
H/8H/L	10%	1300-1820	51	1490	4/0	DB_Memory	10.22
H/8H/2L	10%	1400-1920	51	1640	5/1	APP_CPU	7.82

In order to make the performance limitation appear faster we employed a lower write ratio and lower end DBs in the last two data sets. At a replication level of eight application servers the bottleneck has shifted to the database tier. Our algorithm identifies the DB memory as a potentially saturated resource. Now we examine the effect of replicating the bottlenecked DB. This again results in a shift of the bottleneck towards the application tier and is successfully detected by our algorithm. It is also evident that we are able to perform our detection process accurately with a significantly lower number of trials than other approaches.

4 Related Work

The area of performance modeling in multi-tier enterprise systems has been subjected to substantial research efforts in the recent time. Many of the well-documented approaches use machine learning or queuing theory.

Cohen *et al* [3] apply a tree-augmented Naïve Bayesian network to discover correlations between system-level metrics and performance states, such as SLO satisfaction and SLO failure. Powers *et al* [5] also 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. Similarly, we have performed a comparative study of machine learning classifiers to investigate performance patterns [11]. Our goal was to compare the performance of several well-known machine learning algorithms as classifiers in terms of bottleneck detection, and finally to identify the classifier that best detects bottlenecks in multi-tier applications. Several other studies are based on dynamic queuing models combined with predictive and reactive provisioning as in [9]. 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. Another fundamental difference of our work is that in addition to correlating metrics to performance states, we focus on the detection of actual performance-limiting bottlenecks. We employ a unique procedure to analyze the change in trends of metrics. Finally, our set of metrics for bottleneck detection includes over two hundred application-level metrics as well as system-level metrics.

5 Conclusion

Our detection scheme based on intervention analysis has proven to be very effective with our experimental data. The method is able to characterize the potential change in metric graph trends automatically and assess its correlation with SLO violations. Our statistical modeling approach eliminates the previously necessary data filtering (e.g. correlation analysis) and model calibrating phases (e.g. classifier training). The results are clear and intuitive in the interpretation. We showed that our new method yields these general as well as practical advantages in our evaluation. Potential bottlenecks are identified accurately in different scenarios. As future work this method could be extended with a maximization/minimization scheme for the ranking function. This would allow a more thorough root-cause analysis in the case of multiple bottlenecks. We also plan to employ our results as input for a regression model that will be able to predict actual SLO satisfaction levels.

Acknowledgment

This research has been partially funded by National Science Foundation grants CISE/IIS-0242397, ENG/EEC-0335622, CISE/CNS-0646430, AFOSR grant FA9550-06-1-0201, IBM SUR grant, Hewlett-Packard, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment.

References

1. P. Brockwell and A. Davis: Introduction to Time Series and Forecasting. Springer Inc., NY, 1996.
2. E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel: Performance comparison of middleware architectures for generating dynamic Web content. *Middleware 2003.*
3. 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. *OSDI 2004*.
4. G. Jung, G. Swint, J. Parekh, C. Pu, and A. Sahai. Detecting Bottlenecks in n-Tier IT Applications through Analysis. *DSOM 2006* (LNCS vol. 4269).
5. R. Powers, M. Goldszmidt, and I. Cohen: Short Term Performance Forecasting in Enterprise Systems. *KDD 2005*.
6. C. Pu, A. Sahai, J. Parekh, G. Jung, J. Bae, Y. Cha, T. Garcia, D. Irani, J. Lee, and Q. Lin: Observation-Based Approach to Performance Characterization of Distributed n-Tier Applications. Submitted for publication.
7. J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc., CA, 1993.
8. M. Raghavachari, D. Reimer, and R. D. Johnson: The Deployer's Problems: Configuring Application Servers for Performance and Reliability. *ICSE 2003*.
9. B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal: Dynamic Provisioning of Multi-tier Internet Applications. *ICAC 2005*.
10. C. Wu, and M. Hamada: Experiments: Planning, Analysis, and Parameter Design Optimization. Wiley & Sons Inc., NY, 2000.
11. Elba project. http://www-static.cc.gatech.edu/systems/projects/Elba.
12. Emulab/Netlab. http://www.emulab.net/.
13. RUBiS distribution. http://forge.objectweb.org/project/showfiles.php?group_id=44.