Correlating instrumentation data to system states: A building block for automated diagnosis and control

Ira Cohen
Moises Goldszmidt
Terence Kelly
Julie Symons
HP Labs, Palo Alto, CA

Jeffrey S. Chase
Department of Computer Science
Duke University
Durham, NC
chase@cs.duke.edu

Abstract

This paper studies the use of statistical induction techniques as a basis for automated performance diagnosis and performance management. The goal of the work is to develop and evaluate tools for offline and online analysis of system metrics gathered from instrumentation in Internet server platforms. We use a promising class of probabilistic models (Tree-Augmented Bayesian Networks or TANs) to identify combinations of system-level metrics and threshold values that correlate with high-level performance states—compliance with Service Level Objectives (SLOs) for average-case response time—in a three-tier Web service under a variety of conditions.

Experimental results from a testbed show that TAN models involving small subsets of metrics capture patterns of performance behavior in a way that is accurate and yields insights into the causes of observed performance effects. TANs are extremely efficient to represent and evaluate, and they have interpretability properties that make them excellent candidates for automated diagnosis and control. We explore the use of TAN models for offline forensic diagnosis, and in a limited online setting for performance forecasting with stable workloads.

1 Introduction

Networked computing systems continue to grow in scale and in the complexity of their components and interactions. Today's large-scale network services exhibit complex behaviors stemming from the interaction of workload, software structure, hardware, traffic conditions, and system goals. Pervasive instrumentation and query capabilities are necessary elements of the solution for managing complex systems [31, 21, 32, 12]. There are now many commercial frameworks on the market for coordinated monitoring and control of large-scale systems: tools such as HP's OpenView and IBM's Tivoli aggregate information from a variety of sources and present it graphically to operators. But it is widely recognized that the complexity of deployed systems surpasses the ability of humans to diagnose and respond to problems rapidly and correctly [15, 25]. Research on automated diagnosis and control—beginning with tools to analyze and interpret instrumentation data—has not kept pace with the demand for practical solutions in the field.

Broadly there are two approaches to building self-managing systems. The most common approach is to incorporate *a priori* models of system structure and behavior, which may be represented quantitatively or as sets of event-condition-action rules. Recent work has explored the uses of such models in automated performance control [13]. This approach has several limitations: the models and rule bases are themselves difficult and costly to build, are likely to be incomplete or inaccurate in significant ways, and inevitably become brittle when systems change or encounter conditions unanticipated by their designers.

The second approach is to apply techniques from automated learning and reasoning to build the models automatically. These approaches are attractive because they assume little or no domain knowledge; they are therefore generic and have potential to apply to a wide range of systems and to adapt to changes in the system and its environment. For example, there has been much recent progress on the use of statistical analysis tools to infer component relationships from histories of interaction patterns (e.g., from packet traces) [8, 7, 1, 23, 9]. But it is still an open problem to identify techniques that are powerful enough to induce effective models, and that are sufficiently efficient, accurate, and robust to deploy in practice.

The goal of our work is to automate analysis of instrumentation data from network services in order to forecast, diagnose, and repair failure conditions. This paper studies the effectiveness and practicality of *Tree-Augmented Naive* Bayesian networks [16], or TANs, as a basis for performance diagnosis and forecasting from

system-level instrumentation in a three-tier network service. TANs comprise a subclass of Bayesian networks, recently of interest to the systems community as potential elements of an Internet "Knowledge Plane" [10]. TANs are less powerful than generalized Bayesian networks (see Section 3), but they are simple, compact and efficient. TANs have been shown to be promising in diverse contexts including financial modeling, medical diagnosis, text classification, and spam filtering, but we are not aware of any previous study of TANs in the context of computer systems.

To explore TANs as a basis for self-managing systems, we analyzed data from 124 metrics gathered at a fine time granularity from a three-tier e-commerce site under synthetic load. The induced TAN models select combinations of metrics and threshold values that correlate with high-level performance states—compliance with Service Level Objectives (SLO) for average response time—under a variety of conditions. The experiments support the following conclusions:

- Combinations of metrics are significantly more predictive of SLO violations than individual metrics. Moreover, different combinations of metrics and thresholds are selected under different conditions. This implies that even this relatively simple problem is too complex for simple "rules of thumb" (e.g., just monitor CPU utilization).

- Small numbers of metrics (typically 3–8) are sufficient to predict SLO violations accurately. In most cases the selected metrics yield insight into the cause of the problem and its location within the system. This property of *interpretability* is a key advantage of TAN models (Section 3.3). While we do not claim to solve the problem of root cause analysis, our results suggest that TANs have excellent potential as a basis for diagnosis and control. For example, we may statically associate metrics with control variables (actuators) to restore the system to a desired operating range.

- Although multiple metrics are involved, the relationships among these metrics are relatively simple in this context. Thus TAN models are highly accurate: in typical cases, the models yield a *balanced accuracy* of 90%–95% (see Section 2.1).

- The TAN models are extremely efficient to represent and evaluate. Model induction is efficient enough to adapt to changes in workload and system structure by continuously inducing new models.

Of the known techniques from automated learning and statistical pattern recognition, the TAN structure and algorithms are among the most promising for deployment in real systems. They are based on sound and well-developed theory, they are computationally efficient and robust, they require no expertise to use, and they are readily available in open-source implementations [22, 33, 3]. While other approaches may prove to yield comparable accuracy and/or efficiency, Bayesian networks and TANs in particular have important practical advantages: they are interpretable and they can incorporate expert knowledge and constraints easily, as described below. Although our primary emphasis is on *diagnosing* performance problems after they have occurred, we illustrate the versatility of TANs by using them to *forecast* problems in advance. However, we emphasize that our methods discover correlations rather than causal connections, and that our results do not yet show that a robust "closed loop" diagnosis is practical at this stage. Even so, the technique can sift through a large amount of instrumentation data rapidly and focus the attention of a human analyst on the small set of metrics most relevant to the conditions of interest.

This paper is organized as follows: Section 2 defines the problem and gives an overview of our approach. Section 3 gives more detail on TANs and the algorithms to induce them, and outlines the rationale for selecting this technique for computer systems diagnosis and control. Section 4 describes the experimental methodology and Section 5 presents results. Section 8 concludes.

2 Overview

Figure 1 depicts the experimental environment. The system under test is a three-tier Web service: the Web server (Apache), application middleware server (BEA WebLogic), and database server (Oracle) run on three different servers instrumented with HP OpenView to collect a set of system metrics. A load generator (`httperf` [27]) offers load to the service over an execution period divided into a sequence of intervals. An SLO compliance indicator processes the Apache logs to determine SLO compliance over each interval, based on the average server response time for requests in the interval.

This paper focuses on the problem of constructing an analysis engine to process the metrics and indicator values. The goal of the analysis is to induce a *classifier*, a function that predicts whether or not the system is in compliance over any interval, based on the values of the metrics collected over that interval. The classifier may be useful to predict SLO violations before they occur. If the classifier is interpretable, then it may also be useful for diagnostic forensics or control. One advantage of our approach is that it is based on a feature selection: it identifies sets of metrics and threshold values that correlate with SLO violations. Since specific metrics are associ-

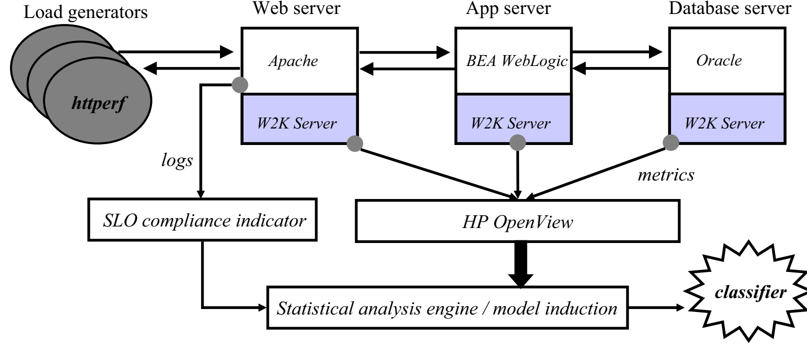


Figure 1: This study explores the use of TAN classifiers to diagnose a common type of network service: a three-tier Web application with a Java middleware component, backed by a database.

ated with specific components, resources, processes, and events within the system, the classifier indirectly identifies the system elements that are most likely to be involved with the failure or violation. In essence, the classifier has both explanatory and predictive power. Even so, the analysis process itself is general because it uses no a priori knowledge of the system's structure or function.

In this study we limit our attention to system-level metrics gathered from a standard operating system (Windows 2000 Server) on each of the servers. Of course, the analysis engine may be more effective if it considers application-level metrics; on the other hand, analysis and control using system-level metrics can generalize to any application. Table 1 lists some specific system-level metrics that are often correlated with SLO violations in our experiments.

2.1 Formalizing the Problem

This problem is a pattern classification problem in supervised learning. Let S_t denote the state of the SLO at time t. In this case, S can take one of two states from the set $\{compliance, violation\}$: let $\{0,1\}$ or $\{s^+, s^-\}$ denote these states. Let $\vec{M}_t$ denote a vector of values for n collected metrics $[m_0, ..., m_n]$ at time t (we will omit the subindex t when the context is clear). The pattern classification problem is to induce or learn a classifier function $\mathcal{F}$ mapping the universe of possible values for $\vec{M}_t$ to the range of system states S [14, 5].

The input to this analysis is a training data set of the form $< \vec{M}_t, S_t >$. In this case, the training set is a log of observations from the system in operation. The learning is supervised in that the SLO compliance indicator identifies the value of S_t corresponding to each observed $\vec{M}_t$ in the training set, providing preclassified instances for the analysis to learn from.

We emphasize four premises that are implicit in this problem statement. First, it is not necessary to predict system behavior, but only to identify system states that correlate with particular failure events (e.g., SLO violations). Second, the events of interest are defined and identified externally by some failure detector. For example, in this case it is not necessary to predict system performance, but only to classify system states that comply with an SLO defined over average response time, as specified by an external indicator for SLO compliance. Third, there are repeated patterns among the collected metrics that correlate with SLO compliance; in this case, the metrics must be well-chosen to capture system states relating to the behavior of interest. Finally, the analysis must observe a sufficiently large set of event instances in the training set to correlate it with the observed metrics. Thus this approach is not useful to diagnose failure conditions that are rare or unanticipated; however, Bayesian classifiers may also be used for anomaly detection.

The key measure of success of the analysis is the accuracy of the resulting classifier. A common metric is the *classification accuracy*, which in this case is defined as the probability that $\mathcal{F}$ correctly identifies the SLO state S_t associated with any $\vec{M}_t$. This measure can be misleading when violations are uncommon: for example, in a system that violates SLO in 10% of the intervals, a trivial classifier that always guesses compliance ($\mathcal{F}(\vec{(\mathcal{M}})) = \int^+$) yields a classification accuracy of 90%. Instead, our preferred metric is *balanced accuracy* (BA), which averages the probability of correctly identifying compliance with the probability of detecting a violation. Formally:

$$BA = \frac{P(s^- = \mathcal{F}(\vec{M})|s^-) + P(s^+ = \mathcal{F}(\vec{M})|s^+)}{2} \quad (1)$$

To achieve the maximal BA of 100%, $\mathcal{F}$ must perfectly classify both SLO violation and SLO compliance. The trivial classifier in the example above has a BA of

Metric	Description
mean_AS_CPU_1_USERTIME	CPU time spent in user mode on the application server.
var_AS_CPU_1_USERTIME	Variance of user CPU time on the application server.
mean_AS_DISK_1_PHYSREAD	Number of physical disk reads for disk 1 on the application server, includes file system reads, raw I/O and virtual memory I/O.
mean_AS_DISK_1_BUSYTIME	Time in seconds that disk 1 was busy with pending I/O on the application server.
var_AS_DISK_1_BUSYTIME	Variance of time that disk 1 was busy with pending I/O on the application server.
mean_DB_DISK_1_PHYSWRITEBYTE	Number of kilobytes written to disk 1 on the database server, includes file system reads, raw I/O and virtual memory I/O.
var_DB_GBL_SWAPSPACEUSED	Variance of swap space allocated on the database server.
var_DB_NETIF_2_INPACKET	Variance of the number of successful (no errors or collisions) physical packets received through network interface #2 on the database server.
mean_DB_GBL_SWAPSPACEUSED	Amount of swap space, in MB, allocated on the database server.
mean_DB_GBL_RUNQUEUE	Approximate average queue length for CPU on the database server.
var_DB_NETIF_2_INBYTE	Variance of the number of KBs received from the network via network interface #2 on the database server. Only bytes in packets that carry data are included.
var_DB_DISK_1_PHYSREAD	Variance of physical disk reads for disk 1 on the database server.
var_AS_GBL_MEMUTIL	Variance of the percentage of physical memory in use on the application server, including system memory (occupied by the kernel), buffer cache, and user memory.
numReqs	Number of requests the system has served.
var_DB_DISK_1_PHYSWRITE	Variance of the number of writes to disk 1 on the database server.
var_DB_NETIF_2_OUTPACKET	Variance of the number of successful (no errors or collisions) physical packets sent through network interface #2 on the database server.

Table 1: A sampling of system-level metrics that are often correlated with SLO violations in our experiments, as named by HP OpenView. "AS" refers to metrics measured on the application server; "DB" refers to metrics measured on the database server.

only 50%. In some cases we can gain more insight into the behavior of a classifier by considering the false positive rate and false negative rate separately.

2.2 Inducing Classifier Models

There are many techniques for pattern classification in the literature (e.g., [5, 29]). Our approach first induces a *model* of the relationship between $\vec{M}$ and S, and then uses the model to decide whether any given set of metric values $\vec{M}$ is more likely to correlate with an SLO violation or compliance. In our case, the model represents the conditional distribution $P(S|\vec{M})$—the distribution of probabilities for the system state given the observed values of the metrics. The classifier then uses this distribution to evaluate whether $P(s^+|\vec{M}) > P(s^-|\vec{M})$.

Thus, we transform the problem of pattern classification to one of statistical fitting of a probabilistic model. The key to this approach is to devise a way to represent the probability distribution that is compact, accurate, and efficient to process. Our approach represents the distribution as a form of Bayesian network (Section 3). An important strength of this approach is that one can interrogate the model to identify specific metrics that affect the classifier's choice for any given $\vec{M}$. This *interpretability* property makes Bayesian networks attractive for diagnosis and control, relative to competing alternatives such as neural networks and support vector machines. One other alternative, decision trees [29], can be interpreted as a set of if-then rules on the metrics and their values. Bayesian networks have an additional advantage of *modifiability*: they can incorporate expert knowledge or constraints into the model efficiently. For example, a user can specify a subset of metrics or correlations to include in the model, as discussed below. Section 3.3 outlines the formal basis for these properties.

The key challenge for our approach is that it is intractable to induce the optimal Bayesian network classifier. Heuristics may guide the search for a good classifier, but there is also a risk that a generalized Bayesian network may overfit data from the finite and possibly noisy training set, compromising accuracy. Instead, we restrict the form of the Bayesian network to a TAN (Section 3) and select the optimal TAN classifier over a heuristically selected subset of the metrics of size at most k, for some parameter k. This approach is based on the premise (which we have validated empirically in our domain) that a relatively small subset of metrics and threshold values is sufficient to approximate the distribution accurately in a TAN encoding relatively simple dependence relationships among the metrics. Although the effectiveness of TANs is sensitive to the domain, TANs have been shown to outperform generalized Bayesian networks and other alternatives in both cost and accuracy in a variety of con-

texts [16]. This paper evaluates the efficiency and accuracy of the TAN algorithm in the context of SLO maintenance for a three-tier Web service, and investigates the nature of the induced models.

2.3 Using Classifier Models

Before explaining the approach in detail, we first consider the significance of the study and its potential impact in practice. We are interested in using classifiers to diagnose a failure or violation condition, and ultimately to repair it.

The technique can be used for diagnostic forensics as follows. Suppose a developer or operator wishes to gain insight into a system's behavior during a specific execution period for which metrics were collected. Running the algorithm yields a classifier for any event—such as a failure condition or SLO threshold violation—that occurs a sufficient number of times to induce a model (see Section 3). In the general case, the event may be defined by any user-specified predicate (indicator function) over the metrics. The resulting model gives a list of metrics and ranges of values that correlate highly with the event, selected from the metrics that do not appear in the definition of the predicate.

The user may also "seed" the induced models by preselecting a set of metrics that must appear in the models, and the value ranges for those metrics. This causes the algorithm to determine the degree to which those metrics and value ranges correlate with the event, and to identify additional metrics and value ranges that are maximally correlated subject to the condition that the values of the specified metrics are within their specified ranges. For example, a user can ask a question of the form: "what percentage of SLO violations occur during intervals when the network traffic between the application server and the database server is high, and what other metrics and values are most predictive of SLO violations during those intervals"?

The models also have high potential to be useful for online forecasting of failures or SLO violations. For example, Section 5 shows that it is possible to induce models that identify system states that are predictive of SLO violations in the near future, when the characteristics of the workload and system are stable. An automated controller may invoke such a classifier directly to identify impending violations and respond to them, e.g., by shedding load or adding resources.

Because the models are cheap to induce, the system may refresh them periodically to track changes in the workload characteristics and their interaction with the system structure. In more dynamic cases, it is possible to maintain multiple models in parallel and select the best model for any given period. The selection criteria may be based on recent accuracy scores, known cyclic behavior in the workload, or recognizable attributes of the workloads.

3 Approach

This section gives more detail on the TAN representation and algorithm. First we summarize why it is useful to provide this detail. We also discuss the advantages of this approach relative to its alternatives.

As stated in the previous section, we use TANs to obtain a compact, efficient representation of the model underlying the classifier. The model approximates a probability distribution $P(S|\vec{M})$, which gives the probability that the system is in any given state S for any given vector of observed metrics $\vec{M}$. It has been shown that inducing a model of this form is equivalent to the problem of fitting the distribution $P(\vec{M}|S)$—the probability of observing a given vector $\vec{M}$ of metric values when the system is in a given state S. Multidimensional problems of this form are subject to challenges of robustness and overfitting, and require a large number of data samples [14, 19]. We can simplify the problem by making some assumptions about the structure of the distribution P. Our approach considers a subclass of Bayesian networks: it assumes that most of the metrics are independent, and that only a relatively small number of relationships are important to represent the distribution accurately.

3.1 Bayesian networks and TANs

Bayesian networks [28] offer a well-developed mathematical language to represent structure in probability distributions. Furthermore, there is a set of well-understood algorithms and methods for inducing these models statistically from data [20], and these are available in open-source software [22, 33, 3].

A Bayesian network is an annotated directed acyclic graph encoding a joint probability distribution. The vertices in the graph represent the random variables of interest in the domain to be modeled, and the edges represent direct influences between the random variables. In our case, each system-level metric m_i is a random variable represented in the graph. Each vertex in the network encodes a probability distribution on the values that the random variable can take, given the state of its predecessors. This representation encodes a set of (probabilistic) independence statements of the form: each random variable is independent of its non-descendants, given that the state of its parents is known.

In a naive Bayesian network, the state variable S is the only parent of all other vertices. Thus a naive Bayesian network assumes that all the metrics are fully independent given S. A tree-augmented naive Bayesian network

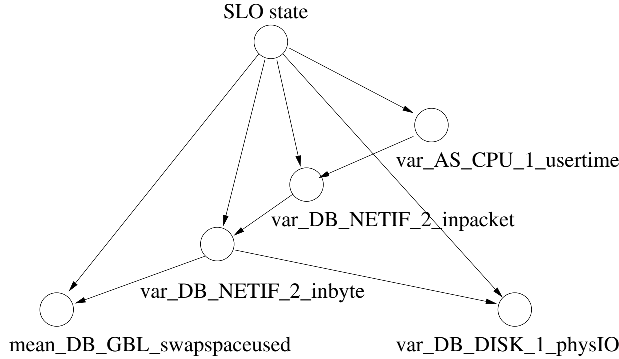


Figure 2: Example TAN induced to fit SLO violations in a three-tier Web service. Table 1 defines the metrics used.

(TAN) extends this structure to consider relationships among the metrics themselves, with the constraint that each metric m_i has at most one parent m_{p_i} in the network other than S. Thus a TAN imposes a tree-structured dependence graph on a naive Bayesian network; this structure is a *Markov tree*. The TAN for a set of observations and metrics is defined as the Markov tree that is *optimal* in the sense that it has the highest probability of having generated the observed data [16]. TANs have been extensively tested against other classifiers in the literature, and have proven to be highly effective in a variety of domains [16].

Figure 2 illustrates a TAN obtained for one of our experiments (see the STEP workload in Section 4.1.2). This model has a balanced accuracy (BA) score of 94% for the system, workload, and SLO in that experiment. The metrics selected are the variance of the CPU user time at the application server, network traffic (packets and bytes) from the application server to the database tier, and the swap space and disk activity at the database. The tree structure relating the metrics captures the following assertions: (1) once we know the network traffic between the tiers, the CPU activity in the application server is irrelevant to the swap space and disk activity at the database tier; (2) the network traffic is correlated with CPU activity, so common increases in the values of those metrics are not anomalous.

3.2 Selecting a TAN model

Given a basic understanding of the classification approach and the models, we now outline the methods and algorithms used to select the TAN model for the classifier (derived from [16]). For efficiency, our approach induces a TAN from a selected subset $\vec{M}^*$ of the metrics in $\vec{M}$. For each possible $\vec{M}^*$, there is exactly one TAN representing the optimal Markov tree over the metrics in $\vec{M}^*$. The goal of the algorithm is to select the $\vec{M}^*$ whose TAN yields the most accurate classifier. The algorithm considers only vectors $\vec{M}^*$ that incorporate at most k metrics, for some parameter k. We used $k = 10$ for the experiments presented in Section 5. In our experience, choosing $k > 10$ yields negligible improvements in accuracy.

The problem of selecting the best $\vec{M}^*$ is known as *feature selection*. Most solutions use some form of heuristic search given the combinatorial explosion of the search space in the number of metrics in $\vec{M}$. We use a greedy strategy: at each step select the metric that is not already in the vector $\vec{M}^*$, and that yields maximum improvement in balanced accuracy (BA) of the resulting TAN over the sample data. To do this, the algorithm computes the optimal Markov tree for each candidate metric, computes the BA score for the tree against the observed data, then selects the metric that yields the highest BA score. The cost is $O(kn)$ times the cost to induce and evaluate the Markov tree, where n is the number of metrics. The algorithm to compute the optimal Markov tree is based on a minimum spanning tree computation among the metrics in $\vec{M}^*$.

From Eq. 1 it is clear that to compute a candidate's BA score we must estimate the probability of false positives and false negatives for the resulting model. The algorithm must approximate the real BA score from a finite set of samples. To ensure the robustness of this score against variability on the unobserved cases in the data, the following procedure called *ten-fold cross validation* is used [19]. Randomly divide the data into two sets, a training set and a testing set. Then, induce the model with the training set, and compute its score with the testing set. Compute the final score as the average score over ten trials. This reduces any biases or overfitting effects resulting from a finite data set.

Given a data set with N samples of the n metrics, the overall algorithm is dominated by $O(n^2 \cdot N)$ for small k, when all N samples are used to induce and test the candidates. In our experiments, $n = 124$ and $N = 2400$. A computation involving a complete characterization of this log, including metric selections, over a range of 30 SLO definitions takes less than ten minutes on Matlab code running on a 1.8GHz Pentium 4 (~ 20 seconds per SLO definition). Each experiment required the induction of about 40,000 candidate models, for a rough average of 15 ms per model. Once the model is selected, evaluating it to classify a new interval sample takes 1-10 ms. These operations are cheap enough to train a new model for every violation interval and even to maintain and evaluate multiple models in parallel.

3.3 Interpretability and Modifiability

In addition to their efficiency in representation and inference, TANs (and Bayesian networks in general) present two key advantages relative to competing approaches: *interpretability* and *modifiability*. These properties are especially important in the context of diagnosis and control.

The influence of each metric on the violation of an SLO can be established and quantified in a sound probabilistic model. Mathematically, we arrive at the following functional form for the classifier:

$$\sum_i \log[\frac{P(m_i|m_{p_i}, s^-)}{P(m_i|m_{p_i}, s^+)}] + \log\frac{P(s^-)}{P(s^+)} > 0 \qquad (2)$$

A value greater than zero indicates a violation. Note that this value is computed by a linear combination of likelihood tests on each individual metric (each term in the summation of the inequality above). Thus, each metric is essentially subjected to a likelihood test comparing the probability that the observed value occurs during compliance to the probability that the value occurs during violation. This analysis catalogs each type of SLO violation according to the metrics and values that correlate with the violation. We contend that this gives insight into the causes of the violation or even how to repair it. For example, if violation of a temperature threshold is highly correlated with an open window, then one potential solution may be to close the window. Of course, any correlation is merely "circumstantial evidence" rather than proof of causality; much of the value of the analysis is to "exonerate" the metrics that are not correlated with the failure rather than to "convict the guilty".

Because these models are interpretable and have clear semantics in terms of probability distributions, we can enhance and complement the information induced directly from data with expert knowledge of the domain or specific system under study [20]. This knowledge can take the form of explicit lists of metrics to be included in the model, information about correlations and dependencies among the metrics, or prior probability distributions. Blake & Breese [6] give examples, including an early use of Bayesian networks to discover bottlenecks in the Windows operating system. Sullivan [30] applies this approach to tune database parameters.

4 Methodology

We considered a variety of approaches to empirical evaluation before eventually settling on the testbed environment and workloads described in this section. We rejected the use of standard synthetic benchmarks, e.g., TPC-W, because they typically ramp up load to a stable plateau in order to determine peak throughput subject to a constraint on mean response time. Such workloads are not sufficiently rich to produce the wide range of system conditions that might occur in practice. Traces collected in real production environments are richer, but production systems rarely permit the controlled experiments necessary to validate our methods. For these reasons we constructed a testbed with a standard three-tiered Web server application—the well-known Java PetStore—and subjected it to synthetic stress workloads designed to expose the strengths and limitations of our approach.

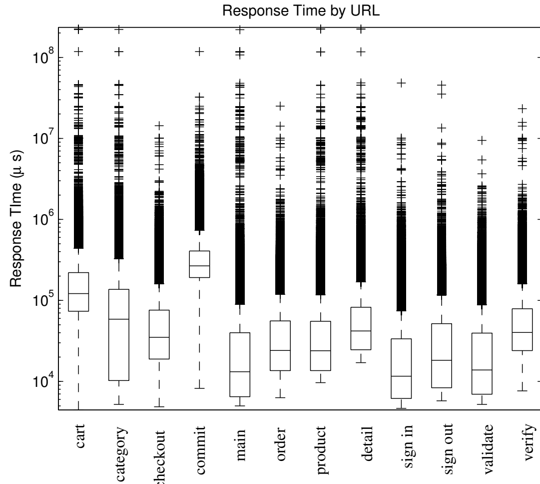


Figure 3: Observed distributions of response time for PetStore operations. Each box marks the quartiles of the distribution; the horizontal line inside each box is the median. Outliers are shown as crosses outside each box.

The Web, application, and database servers were hosted on separate HP NetServer LPr systems configured with a Pentium II 500 MHz processor, 512 MB of RAM, one 9 GB disk drive and two 100 Mbps network cards. The application and database servers run Windows 2000 Server SP4. We used two different configurations of the Web server: Apache Version 2.0.48 with a BEA WebLogic plug-in on either Windows 2000 Server SP4 or RedHat Linux 7.2. The application server runs BEA WebLogic 7.0 SP4 over Java 2 SDK Version 1.3.1 (08) from Sun. The database client and server are Oracle 9iR2. The cluster network is a fully-switched 100 Mbps full-duplex network; traffic to and from each server is distributed between the two network cards, with the exception of the Windows Web server, which runs with a single network card. For example,one link on the Linux Web server carries traffic to and from the clients, and the other carries traffic to and from the application server.

The experiments use a version of the Java PetStore obtained from the Middleware Company in October 2002. We tuned the deployment descriptors, config.xml, and startWebLogic.cmd in order to scale to the transac-

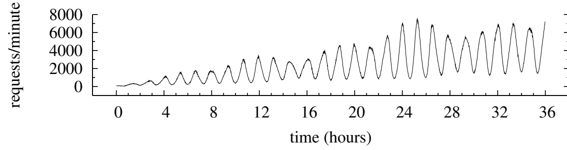


Figure 4: Requests per minute in RAMP workload.

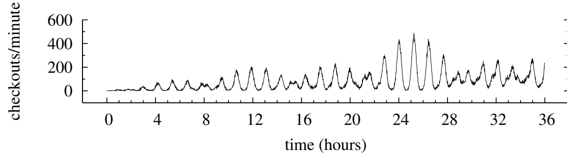


Figure 5: Purchases per minute in RAMP workload.

tion volumes reported in the results. In particular, we modified several of the EJB deployment descriptors to increase the values for `max-beans-in-cache`, `max-beans-in-free-pool`, `initial-bean-in-free-pool`, and some of the timeout values. The `concurrency-strategy` for two of the beans in the Customer and Inventory deployment descriptors was changed to "Database." Other changes include increasing the execute thread count to 30, increasing the initial and maximum capacities for the JDBC Connection pool, increasing the `PreparedStatementCacheSize`, and increasing the JVM's maximum heap size. The net effect of these changes was to increase the maximum number of concurrent sessions from 24 to over 100.

Each server is instrumented using the HP OpenView Operations Embedded Performance Agent, a component of the OpenView Operations Agent, Release 7.2. We configured the agent to sample and collect values for 124 system-level metrics (e.g., CPU & disk utilization and the metrics listed in Table 1) at 15 second intervals.

4.1 Workloads

We designed the workloads to exercise our model-induction methodology by providing it with a wide range of $(\vec{M},\vec{P})$ pairs, where $\vec{M}$ represents a sample of values for the system metrics and $\vec{P}$ represents a vector of application-level performance measurements (e.g., response time & throughput). Of course, we cannot directly control either $\vec{M}$ or $\vec{P}$; we control only the exogenous workload submitted to the system under test. We vary several characteristics of the workload, including

1. aggregate request rate,
2. number of concurrent client connections, and
3. fraction of requests that are database-intensive (e.g., checkout) vs. app-server-intensive (e.g., browsing).

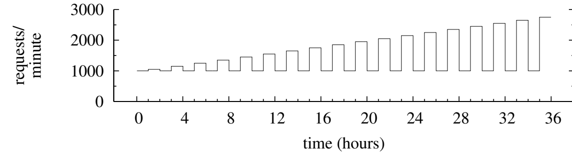


Figure 6: Requests per minute in STEP workload.

Figure 3 presents box plots depicting the response time distributions of the twelve main request classes in our PetStore testbed. Response times differ significantly for different types of requests, hence the request mix is quite versatile in its effect on the system.

We mimic key aspects of real-world workload, e.g., varying burstiness at fine time scales and periodicity on longer time scales. However we design experiments to run in 1–2 days, so the periods of workload variation are shorter than in the wild. We wrote simple scripts to generate session files for the `httperf` workload generator [27], which allows us to vary the client think time and the arrival rate of new client sessions.

4.1.1 RAMP: Increasing Concurrency

In this experiment we gradually increase the number of concurrent client sessions. We add an emulated client every 20 minutes up to a limit of 100 total sessions, and terminate the test after 36 hours. *Individual* client request streams are constructed so that the *aggregate* request stream resembles a sinusoid overlaid upon a ramp; this effect is depicted in Figure 4, which shows the ideal throughput of the SUT. The *ideal* throughput occurs *if all requests are served instantaneously*; because `httperf` uses a closed client loop with think time, the actual rate depends on response time.

Each client session follows a simple pattern: go to main page, sign in, browse products, add some products to shopping cart, check out, repeat. Two parameters indirectly define the number of operations within a session. One is the probability that an item is added to the shopping cart given that it has just been browsed. The other is the probability of proceeding to the checkout given that an item has just been added to the cart. These probabilities vary sinusoidally between 0.42 and 0.7 with periods of 67 and 73 minutes, respectively. The net effect is the ideal time-varying checkout rate shown in Figure 5.

4.1.2 STEP: Background + Step Function

In this experiment we use two workload generators running on different client hosts. The first `httperf` creates a steady background traffic of 1000 requests per minute generated by 20 clients. The second provides an on/off workload consisting of hour-long bursts with one hour

```
 1 For each Experiment
 2   For SLO threshold = 60,...,90 percentile
     of average response time
 3       Identify intervals violating SLO.
 4       Select k metrics using greedy search.
 5       Induce TAN model with top k metrics.
 6       Evaluate with 10-fold cross validation
            for balanced accuracy, false alarm
              and detection rates.
 7       Evaluate using only Application server
            CPU usertime metric (''CPU'').
 8       if (SLO threshold == 60 percentile)
             Save model as ''MOD''.
10        else
             Evaluate MOD on current SLO.
12        Record metric attribution of current
          TAN for each interval violating SLO.
```

Table 2: The testing procedure.

between bursts. Successive bursts involve 5, 10, 15, etc. client sessions; each session generates 50 requests per minute. The test lasts 36 hours. Figure 6 summarizes the ideal request rate for this pattern, omitting fluctuations at fine time scales.

The intent of this workload is to mimic sudden, sustained bursts of increasingly intense workload against a backdrop of moderate activity. Each "step" in the workload produces a different plateau of workload level, as well as transients during the beginning and end of each step as the system adapts to the change.

4.1.3 BUGGY: Numerous Errors

BUGGY was a five-hour run with 25 client sessions. Aggregate request rate ramped from 1 request/sec to 50 requests/sec during the course of the experiment, with sinusoidal variation of period 30 minutes overlaid upon the ramp. The probability of add-to-cart following browsing an item and the probability of checkout following add-to-cart vary sinusoidally between 0.1 and 1 with periods of 25 and 37 minutes, respectively. This run occurred before the Petstore deployment was sufficiently tuned as described previously. The J2EE component generated numerous Java exceptions, hence the title "BUGGY."

5 Experimental Results

This section evaluates our approach using the system and workloads described in Section 4. For each workload, we trained and evaluated a TAN classifier for each of 31 different SLO definitions, given by varying the threshold on the average response time such that the percentage of intervals violating the SLO varies from 40% to 10% in increments of 1%. As a baseline, we also evaluated the accuracy of the 60-percentile SLO classifier (MOD) and a simple "rule of thumb" classifier using application server CPU utilization as the sole indicator metric. Table 2 summarizes the testing procedure.

In these experiments we varied the SLO threshold to explore the effect on the induced models, and to evaluate accuracy of the models under varying conditions. In practice, the SLO threshold is determined externally, and is driven by business objectives; this test procedure can characterize the system behavior to guide the choice of the SLO threshold.

Table 3 summarizes the average accuracy of all models across all SLO thresholds for each workload. Figure 7 plots the results for all 31 SLO definitions for STEP. We make several observations from the results:

1. Overall balanced accuracy of the TAN model is high, ranging from 87%-94%. In a breakdown of false alarms to detection rates we see that detection rates are higher than 90% for all experiments, with false alarms at about 6% for two experiments and 17% for BUGGY.

2. Simply using a single metric (CPU in this case) is not sufficient to capture the patterns of SLO violations. While CPU has a BA score of 90 for RAMP, it does very poorly for the other two workloads. To illustrate, Figure 8 plots average response time for each interval in the STEP run as a function of its average CPU utilization. The plot shows that while CPU usage correlates with average latency when latency is low, the correlation is not apparent for intervals with high average latency. Indeed, Figure 7 shows that the low BA score stems from a low detection rate for the less stringent SLO thresholds.

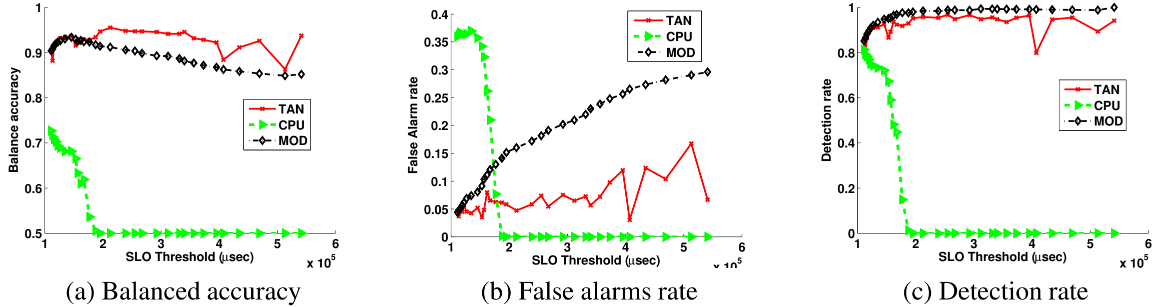
3. A small number of metrics is sufficient to capture the patterns of SLO violations. The number of metrics in the TAN models ranges from 3 to 8.

4. The models are sensitive to the workload and SLO definition. For example, the accuracy of MOD (the TAN model for the most stringent SLO) always has a high detection rate on the less stringent SLOs (as expected), but generates false alarms at an increasing rate as the SLO threshold increases.

Determining number of metrics. To illustrate the role of multiple metrics in the accuracy of the TAN models, Figure 9 shows the top three metrics (in order) as a function of average response time for the STEP workload with SLO threshold of 313 msec (20% instances of SLO violations). The top metric alone yields a BA score of 84%, which improves to 88% with the second metric. However, by itself, the second metric is not discriminative; in fact, the second metric alone yields a BA of just 69%. The TAN combines these metrics

experiment	SLO thresh (msec)	Avg # Metrics	TAN BA	MOD BA	CPU BA	TAN FA	MOD FA	CPU FA	TAN Det	MOD Det	CPU Det
RAMP	62 – 627	3	94 ±2.4	84 ±5	90 ±8	6.4 ±2	29 ±11	8.5 ±4	93 ±2	98 ±0.3	88.7 ±19
STEP	111 – 541	8	92.7 ±2	89.9 ±2.6	56 ±8.8	6.6 ±2.9	16 ±8.2	13 ±16	91.9 ±4.5	96 ±4.2	27 ±34
BUGGY	214 – 627	4	87.3 ±3.3	86.4 ±3.2	63.4 ±12.1	16.9 ±6.8	21.0 ±7.2	14.9 ±13.3	91.6 ±3.1	94.2 ±1.02	41.7 ±37.6

Table 3: Summary of accuracy results. BA is balanced accuracy, FA is false alarm and Det is detection.



(a) Balanced accuracy (b) False alarms rate (c) Detection rate

Figure 7: Accuracy results for STEPas a function of SLO threshold. The TAN trained for a given workload and SLO balances the rates of detection and false alarms for the highest balanced accuracy (BA).

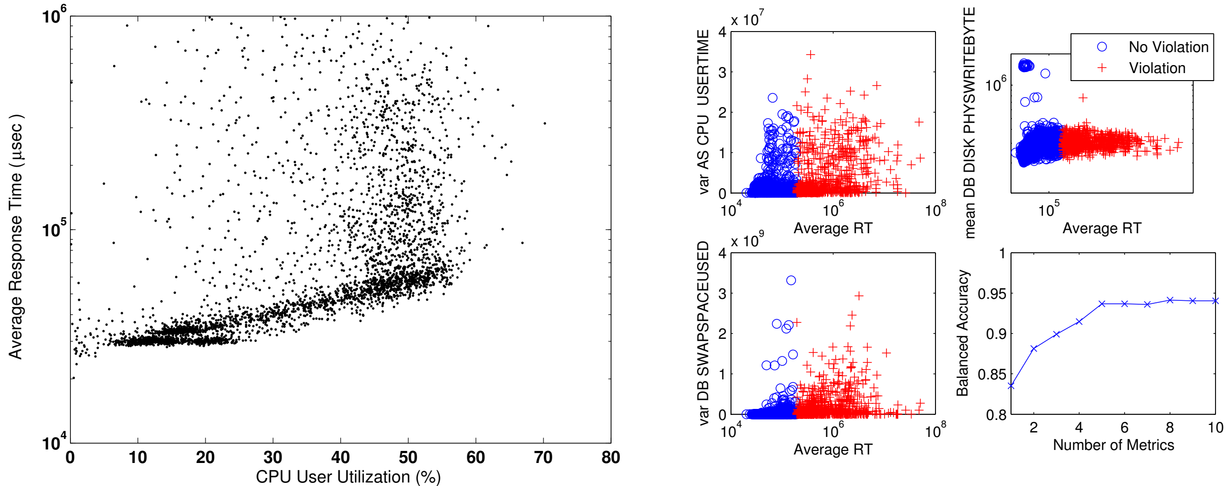


Figure 8: Average response time as a function of the application server CPU user utilization for STEP.

Figure 9: Plots of the top three metrics selected for a TAN model for the STEP workload. Modeling the correlations of five metrics yields a balanced accuracy of 93.6%, a significant improvement over any of the metrics in isolation.

for higher accuracy by representing their relationships. Adding two more metrics to the TAN increases the BA score to 93.6%.

Interaction between metrics and values. The metrics selected for a TAN model may have complex relationships and threshold values. The combined model defines decision boundaries that classifies the SLO state (violation/no violation) of an interval by relating the recorded values of the metrics during the interval. Figure 10 depicts the decision boundary learned by a TAN model for its top two metrics. The figure also shows the best decision boundary when these metrics are used in isolation. We see that the top metric is a fairly good predictor of violations, while the second metric alone is poor. However, the decision boundary of the model with both metrics takes advantage of the strength of both met-

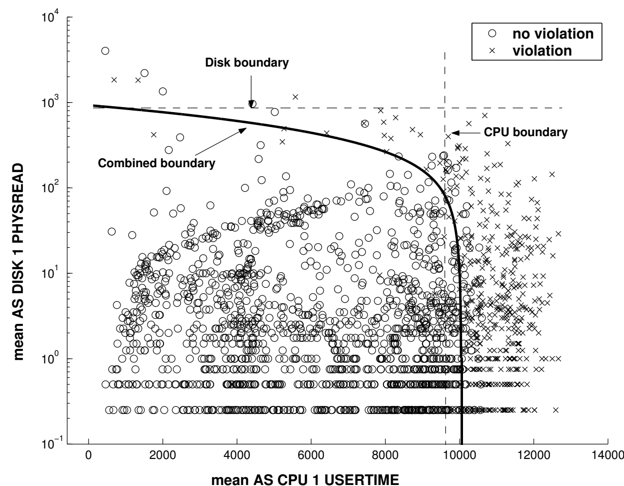


Figure 10: Decision boundary of a TAN model for the values of its top two metrics. Horizontal and vertical lines show decision boundaries for the individual metrics in isolation. Combining the metrics yields higher accuracy than either metric in isolation.

Metric/exper #	RAMP	STEP
mean_AS_CPU_1_USERTIME	27	7
mean_AS_DISK_1_PHYSREAD	14	0
mean_AS_DISK_1_BUSYTIME	6	0
var_AS_DISK_1_BUSYTIME	6	0
mean_DB_DISK_1_PHYSWRITEBYTE	1	22
var_DB_GBL_SWAPSPACEUSED	0	21
var_DB_NETIF_2_INPACKET	2	21
mean_DB_GBL_SWAPSPACEUSED	0	14
mean_DB_GBL_RUNQUEUE	0	13
var_AS_CPU_1_USERTIME	0	12
var_DB_NETIF_2_INBYTE	0	10
var_DB_DISK_1_PHYSREAD	0	9
var_AS_GBL_MEMUTIL	0	8
numReqs	0	7
var_DB_DISK_1_PHYSWRITE	0	6
var_DB_NETIF_2_OUTPACKET	5	6

Table 4: Counts of the number of times the most commonly selected metrics appear in TAN models for SLO violations in the RAMP and STEP workloads. See Table 1 for a definition of the metrics.

rics and "carves out" a region of value combinations that correlate with SLO violations.

Adaptation. Additional analysis shows that the models must adapt to capture the patterns of SLO violation with different response time thresholds. For example, Figure 7 shows that the metrics selected for MOD have a high detection rate across all SLO thresholds, but the increasing false alarm rate indicates that it may be necessary to adjust their threshold values and decision boundaries. However, it is often more effective to adapt the metrics as conditions change. To illustrate, Table 4 lists the metrics selected for at least six of the SLO definitions in either the RAMP or STEP experiments. We see that the most commonly chosen metrics differ significantly across the workloads, which stress the testbed in different ways. For RAMP, CPU usertime and disk reads on the application server are the most common, while swap space and I/O traffic at the database tier are most highly correlated with SLO violations for STEP. A third and entirely different set of metrics is selected for BUGGY: all of the chosen metrics are related to disk usage on the application server. Since the instrumentation records only system-level metrics, disk traffic is most highly correlated with the application server errors occuring during the experiment, which are logged to disk.

Metric "Attribution". The TAN models identify the metrics that are most relevant—alone or in combination—to SLO violations, which is a key step toward a root-cause analysis. Figure 11 demonstrates metric attribution for RAMP with SLO threshold set at 100msec (20% of the intervals are in violation). The model includes two metrics drawn from the application server: CPU user time and disk reads. We see that most SLO violations are attributed to high CPU utilization, while some instances are explained by the combination of CPU and disk traffic, or by disk traffic alone. For this experiment, violations occurring as sudden short spikes in average response time were explained solely by disk traffic, while violations occurring during more sustained load surges were attributed mostly to high CPU utilization, or to a combination of both metrics.

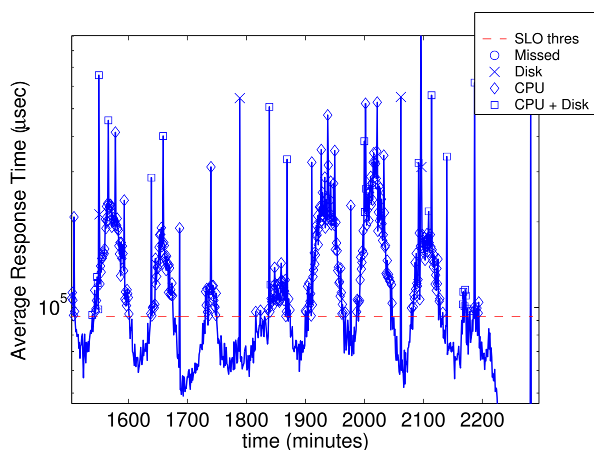


Figure 11: Plot of average response time for RAMP experiment with instances of violation marked as they are explained by different combinations of the model metrics. Horizontal line shows the SLO threshold.

exper.	TAN BA	TAN FA	TAN Det
RAMP	91.8 ±1.2	9.1 ±3	93 ±3.1
STEP	79.7 ±4.6	24 ±5.5	83 ±7.4
BUGGY	79.7 ±4.6	24 ±7.4	83.4 ±5.6

Table 5: Summary of forecasting results. BA is balanced accuracy, FA is false alarm and Det is detection.

Forecasting. Finally, we consider the accuracy of TAN models in forecasting SLO violations. Table 5 shows the accuracy of the models for forecasting SLO violations three sampling intervals in advance (sampling interval is 5 minutes for STEP and 1 minute for the others). The models are less accurate for forecasting, as expected, but their BA scores are still 80% or higher. Forecasting accuracy is sensitive to workload: the RAMP workload changes slowly, so predictions are more accurate than for the bursty STEP workload. Interestingly, the metrics most useful for forecasting are not always the same ones selected for diagnosis: a metric that correlates with violations as they occur is not necessarily a good predictor of future violations.

6 Validation with Independent Testbed

To further validate our methods, we analyzed data collected by a group of HP's OpenView developers on a different testbed. This data came from experiments that simulated performance problems due to external applications contending for system resources.

6.1 Testbed and workload description

The testbed consists of an Apache2 Web server running on a Linux system with a 2.4 kernel. The machine has 600 MHz CPU and 256 MB RAM. A total of 54 different system level metrics are collected using SAR at 15 second intervals.

Client Load The Apache Web server stores 2000 files of size 100 KB each. The client downloads a random file out of that set. The workload is constructed such that the Web server is forced to load files from the disk and cannot rely on the cache. The client machine exerts load on the system by downloading as many pages per second as possible. It also measures response times averaged over 15-second windows. This leads to an average response time for the system of around 70 msec, with a normal throughput of about 90 replies per second.

Tests We report results for three tests:

Disk Disk bottleneck test: Very large files are generated and copied to the Web server disk while the application is running and trying to access the disk. The test runs for 20 minutes, with 10 minutes of performance problems.

exper.	SLO th (msec)	TAN BA	TAN Det	TAN FA
Disk	204	92.5 ±12.9	88.3 ±15.3	3.3 ±10.5
Mem	98	99.5 ± 0.3	99.6 ±0.01	0.6 ±0.6
I/O	73	97.9 ± 1.4	97.8 ±2.4	1.9 ±0.4

Table 6: Summary of results on OpenView testbed. BA is balanced accuracy, FA is false alarm and Det is detection.

Mem Memory bottleneck test: An external application hogs memory. Test runs for 10.5 hours, out of which there are 2.5 hours of performance problems.

I/O I/O bottleneck test: A scheduled backup occurs during the run causing the application response time to go up and throughput to go do down. The test runs for 14.5 hours, with the backup taking over an hour.

We also had data in which the an external application causes a CPU bottleneck. We do not bring the results for that test as it was very easy to detect the bottleneck (our models obtained 100% accuracy, choosing the CPU metrics, but any glance at the CPU metrics would have indicated that it is the cause of the problem).

The main characteristic of these tests is that they exercise the system using an external mechanism to the application that causes a performance problem and not through changes in the workload hitting the application, or problems with the application itself. This is different compared to the tests described previously, where changes in the workload triggered performance problems for the application.

6.2 Results

For each test we used a single SLO threshold derived from the response time of the system under normal conditions. For each test we learn the TAN model as described in the previous sections. The accuracy of the TAN models is given in Table 6.

For each test, different metrics were chosen by the learning algorithm that correlated with the performance problem. Table 7 shows the metrics chosen for each test.

We see that for the Memory and I/O bottleneck tests, the metrics chosen point directly towards the bottleneck.

In the disk bottleneck experiment the chosen metrics do not offer a direct pointing towards the cause of the performance problem. One of the chosen metrics is the one

Disk	
ldavg-1:	System load average for the last minute
plist-sz:	Number of processes in the process list.
Mem	
pgpgout/s	Total number of blocks the system paged out to disk per sec.
txpck/s:	Total number of packets transmitted per sec (On the eth0 device)
I/O	
tps:	number of transfers per second that were issued to the device (I/O)
activepg:	Number of active (recently touched) pages in memory.
kbbuffers:	Amount of memory used as buffers by the kernel in kilobytes.
kbswpfree:	Amount of free swap space in kilobytes.
totsck:	Total number of used sockets.

Table 7: List of metrics chosen in the three experiments with their short description (from SAR man page)

minute average load (loadavg1), a metric which counts the number of runnable jobs in the system. This number includes the number of jobs waiting for I/O requests. Since disk queue metrics were not measured in this test, and since the file manipulation operations did not cause any CPU, network, memory or I/O load, the load average metric serves as a proxy for the disk queues. However, load average is usually considered a measure of CPU performance. To pinpoint the cause of the performance problem in this case, it is necessary to also look at the CPU utilization and see that it became lower (while load average went up). With both pieces of information it can be concluded that the bottleneck is the disk.

These results provide further evidence that the TAN models are able to point towards the cause of a performance problem, either directly or indirectly, depending on whether the metrics related to the bottlenecked component are available.

7 Related Work

Jain's classic text on computer systems performance analysis surveys a range of techniques including analytic models, simulation models, and controlled experiments [24]. It explains how to use sophisticated statistical techniques to reduce the otherwise prohibitive expense of experiments that relate design decisions (e.g., the choice of a slow, medium, or fast CPU) to system performance. The emphasis in this book is on design decisions rather than on performance diagnosis in live systems. Jain shows that appropriate statistical analysis of experiments that selectively sample a design space can yield insights as useful for design decision-making as experiments that exhaustively explore the design space. The book discusses bottleneck detection in the context of analytic models, defining a bottleneck device to be the most highly utilized queueing component in a system. Our work differs from the approaches presented in Jain's book in that we seek to develop methods of automatically inducing performance models of live production systems from passive measurements alone. Furthermore we seek to identify the system components whose behavior correlates most strongly with application-level performance, rather than bottleneck devices in the traditional sense.

More recent books aimed at system administrators and performance analysts working in the field consider goals closer to ours but pursue them using different approaches. Cockcroft & Pettit cover a wide range of system performance measurement facilities and describe an equally wide variety of system performance debugging techniques, ranging from engineering principles to rules of thumb [11]. They also describe SymbEL, a programming language designed for performance monitoring and diagnosis, and the "Virtual Adrian" performance diagnosis package written in SymbEL. Virtual Adrian (named after the book's first author) encodes human expert knowledge in the form of a large number of rules. For instance, the "RAM rule" applies heuristics to the virtual memory system's scan rate and reports RAM shortage if page residence times are too low.

Our approach differs from Cockcroft's in several important respects. Whereas Virtual Adrian examines only system metrics, e.g., CPU utilization, we correlate system metrics with application-level performance and use the latter as a conclusive measure of whether performance is acceptable or not. Our approach would *not* report a problem if application-level response times are acceptable even if, e.g., the virtual memory system suffered from a RAM shortage. Similarly, Virtual Adrian might report that the system is healthy even if application-level performance is unacceptable. Whereas Virtual Adrian relies on hand-crafted rules that must be created, validated, and maintained by a human expert, we induce performance models automatically. We believe that our approach can adapt more rapidly and at lower expense to changes in system design, e.g., a redesign of the virtual memory system.

Recent research has attempted to replace human expert knowledge with relatively knowledge-lean analysis of passive measurements. Aguilera *et al.* consider the problem of performance debugging in distributed systems of "black box" components [1]. In this approach, passive observations are collected of communications among a system's components, e.g., hosts corresponding to different tiers of a multi-tiered Web server or Enterprise Java Beans in a J2EE environment. If communi-

cations follow RPC-style request/reply patterns, a "nesting" algorithm is used to infer causal paths along which messages related to a single high-level transaction travel. If arbitrary non-RPC-like communications occur, a "convolution" algorithm is used to infer causal paths. In many respects our goals are similar: to aid performance debugging of complex black-box systems by automatically and quickly directing the human analyst's attention to likely trouble spots. One difference between the Aguilera *et al.* approach and ours is that we consider metrics collected within hosts rather than communications among components; in this sense the two approaches are complementary. Another difference is that we attempt to report possible problems at finer granularity ("the disk on the app server is strongly correlated with performance" vs. "the app server appears to be slow").

Several approaches have been proposed that involve tagging individual requests and tracking them as they propagate through multi-tiered servers or software component platforms like .Net or J2EE. Examples include WebMon [18], Magpie [2], and Pinpoint [9]. We refer the reader to Aguilera *et al.* [1] for an excellent review of these and related research efforts. These projects all differ from our work in that they require instrumentation of software platforms and related infrastructure. Furthermore they complement our approach because they relate application-level performance to hosts or software components rather than to physical resources as in our work.

Advanced model-induction techniques developed for machine learning have begun to be applied to a variety of systems problems. Mesiner *et al.* [26], for instance, apply decision-tree classifiers to the problem of predicting properties of files (e.g., access patterns) based on creation-time attributes (e.g., names and permissions). They report that accurate models can be induced for this classification problem, but that an adaptive model-induction methodology is necessary because a model that works well in one production environment may not be well suited to other environments.

8 Conclusion

The conclusion of our study is that the properties of TANs make them an attractive basis for self-managing systems. TANs are one of several competing techniques for building system models automatically with no a priori knowledge of system structure or workload characteristics. These statistical learning techniques can generalize to a wide range of systems, and can adapt to changing conditions—if model induction is sufficiently cheap. Our experimental results show that TANs are powerful enough to capture important correlations efficiently, and that they can capture the performance behavior of complex systems by sifting through instrumentation data to "zero in" on the small number of metrics that capture the important effects. TANs are extremely efficient to represent and evaluate, and they are interpretable. This combination of properties make them attractive candidates relative to alternatives, such as neural networks, decision trees, and generalized Bayesian networks.

Our ongoing work focuses on the online adaptation of these models. The theory and practice of adapting Bayesian networks to incoming data has been researched in enough detail to provide practical approaches [20, 4, 17] for adaptation. In general, the problem is separated into the adaptation the parameters of the model and the adaptation of the structure of the model. Strategies for the adaptation of the parameters rely on known statistical techniques for sequential update. The problem of adapting structure is more complicated as it requires a search over the space of models [17]. Note however, that in TAN models this problem is alleviated by the constrained tree structure.

Given the efficiency of both induction and inference of these models, continuous instrumentation and online adaptation may well be feasible. This would be extremely valuable in a utility computing infrastructure. In some cases, it may be helpful to build models for a range of behaviors and traffic patterns offline, then select among pre-induced models at runtime. We believe that ultimately the most successful approach will combine a priori models (e.g., from queuing theory) with automatically induced models. Bayesian networks are an promising technology to achieve this fusion of domain knowledge with automatically induced models for self-managing systems.

9 Acknowledgments

Joern Schimmelpfeng and Klaus Wurster conducted the experiments described in Section 6. Sharad Singhal provided useful comments on a previous version of the paper. Finally we would like to thank the anonymous reviewers and our shepherd Sam Madden; their comments and insights greatly improved the quality of this paper.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[2] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[3] Bayesian network classifier toolbox. `http://jbnc.sourceforge.net/`.

[4] J. Binder, D. Koller, S. Russell, and K. Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, 29:213–244, 1997.

[5] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford, 1995.

[6] R. Blake and J. Breese. Automating computer bottleneck detection with belief nets. In *Proc. 11th Conference on Uncertainty in Artificial Intelligence (UAI'95)*, Aug. 1995.

[7] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer. Using runtime paths for macro analysis. In *Proc. HotOS-IX*, Kauai, HI, May 2003.

[8] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic systems. In *Proc. 2002 Intl. Conf. on Dependable Systems and Networks*, pages 595–604, Washington, DC, June 2002.

[9] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of the First Symposium on Networked System Design and Implementation (NSDI'04)*, Mar. 2004.

[10] D. Clark, C. Partridge, J. C. Ramming, and J. Wroclawski. A knowledge plane for the Internet. In *Proceedings of ACM SIGCOMM*, August 2003.

[11] A. Cockcroft and R. Pettit. *Sun Performance and Tuning*. Prentice Hall, 1998.

[12] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, August 2001.

[13] R. P. Doyle, O. Asad, W. Jin, J. S. Chase, and A. Vahdat. Model-based resource provisioning in a Web service utility. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[14] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York, 1973.

[15] A. Fox and D. Patterson. Self-repairing computers. *Scientific American*, May 2003.

[16] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.

[17] N. Friedman and M. Goldszmidt. Sequential update of Bayesian network structure. In *UAI*, 1997.

[18] P. K. Garg, M. Hao, C. Santos, H.-K. Tang, and A. Zhang. Web transaction analysis and optimization. Technical Report HPL-2002-45, Hewlett-Packard Labs, Mar. 2002.

[19] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[20] D. Heckerman, D. Geiger, and D. Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.

[21] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Sept. 2003.

[22] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[23] R. Isaacs and P. Barham. Performance analysis in loosely-coupled distributed systems. In *7th CaberNet Radicals Workshop*, Bertinoro, Italy, Oct. 2002.

[24] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[25] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[26] M. Mesnier, E. Thereska, G. R. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Proceedings of the First International Conference on Autonomic Computing (ICAC-04)*, May 2004.

[27] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. Technical Report HPL-98-61, Hewlett-Packard Laboratories, Mar. 1998.

[28] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[29] J. Quinlan. *C4.5 Programs for machine learning*. Morgan Kaufmann, 1993.

[30] D. Sullivan. Using probabilistic reasoning to automate software tuning, 2003.

[31] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[32] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *Proceedings of ACM HotNets-II*, November 2003.

[33] I. Witten and E. Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, 2000.