Modellus: Automated Modeling of Complex Internet Data Center Applications

Peter Desnoyers,
Northeastern University
pjd@ccs.neu.edu

Timothy Wood, Prashant Shenoy,
University of Massachusetts Amherst
{twood,shenoy}@cs.umass.edu

Sangameshwar Patil, and Harrick Vin
Tata Research Development and Design Centre (TRDDC)
{sangameshwar.patil,harrick.vin}@tcs.com

1. INTRODUCTION

Distributed server applications have become commonplace in today's Internet and business environments. The data centers hosting these applications—large clusters of networked servers and storage—have in turn become increasingly complex. Some of this is due to complexity of the applications themselves, which may have multiple tiers and share resources across applications. Another factor contributing to data center complexity, however, is evolution and change as hardware and software components are added or replaced, often resulting in unknown or unforeseen system interactions.

These systems, which must be managed to meet service level agreements (SLAs) and to minimize operating and capital costs, have become too complex to be comprehended by a single human. This paper proposes a new approach for conquering this complexity, using statistical methods from data mining and machine learning. These methods create predictive models which capture interactions within a system, allowing the user to relate input (i.e. user) behavior to interactions and resource usage. Data from existing sources (log files, resource utilization) is collected and used for model training, so that models can be created "on the fly" on a running system. From this training data we then infer models which relate events or input at different tiers of a data center application to resource usage at that tier, and to corresponding requests sent to tiers further within the data center. By composing these models, we are able to examine relationships across multiple tiers of an application, and to isolate these relationships from the effects of other applications which may share the same components.

The nature of modern web-based applications, however, makes this analysis and modeling difficult. To create models of inputs and responses, we must classify them; yet they are typically unique to each application. Classifying inputs by hand will not scale in practice, due to the huge number of unique applications and their rate of change. Instead, if we are to use modeling as a tool in data center management, we must *automatically* learn not only system responses but input classifications themselves.

The benefits of an automated modeling method are several. It relieves humans from the tedious task of tracking and modeling complex application dependencies in large systems. The models created may be used for the higher-level task of

analyzing and optimizing data center behavior itself. Finally, automated modeling can keep these models up-to-date as the system changes, by periodic testing and repetition of the learning process.

We have designed and implemented *Modellus*[1], a system that implements our automated modeling approach. Modellus incorporates novel techniques that "mine" the incoming web workload for features that best predict the observed resource usage or workload at a downstream component. Specifically, model inference in Modellus is based on step-wise regression—a technique used in statistical data mining—for determining features that best correlate with empirical observations taken from monitoring of application request logs and OS-level resource usage measurements. Derived models can then be composed to capture dependencies between interacting applications. Modellus also implements automated model testing to verify that derived models remain valid, and triggers relearning of a new model upon model failure.

We implement a host of optimizations to ensure that these statistical methods are practical in large distributed systems. A fast, distributed model testing algorithm performs frequent coarse-grain testing at local nodes, triggering full model testing only when these tests fail. This improves scalability, while reducing the latency of detecting model failures. Techniques for estimating prediction errors are used to prevent excessive errors due to the composition of a large number of models. Finally, Modellus implements back-off heuristics to avoid scenarios where transient phase changes in the workload or inherently "noisy" workloads cause frequent model failures, triggering wasteful relearning.

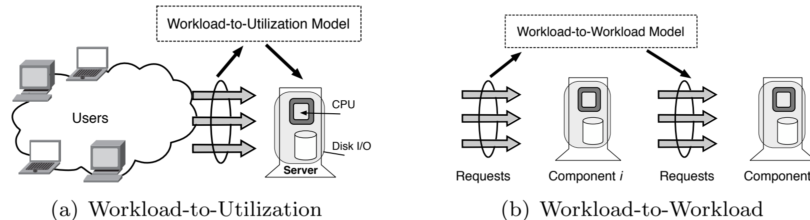
We have implemented a prototype of Modellus, consisting of both a nucleus running at the monitored systems and a control plane for model generation and testing. We conduct detailed experiments on a prototype data center running a mix of realistic web-based applications. Our results show that in many cases we predict server utilization within 5% or less based on measurements of the input to either that server or upstream servers. In addition, we demonstrate the utility of our modeling techniques in predicting responses to future traffic loads and patterns for use in capacity planning.

The remainder of this paper is structured as follows. We present background and formulate the modeling problem in Section 2, and describe our automated modeling approach in Sections 3–5. Section 6 presents the Modellus implementation, while Sections 7 and 8 present our experimental results. Finally, we survey related work in Section 9, and conclude in Section 10.

2. BACKGROUND AND PROBLEM FORMULATION

Consider an Internet data center consisting of a large collection of computers, running a variety of applications and accessed remotely by clients via the Internet. The data center will have resources for computation (i.e. the servers themselves), storage, local communication (typically a high-speed LAN), and remote communication with end-users. Software and hardware components within the data center will interact with each other to implement useful services or *applications*.

[1] *Latin.* Root of 'model'



(a) Workload-to-Utilization (b) Workload-to-Workload

Fig. 1: Application models

As an example, a web-based student course registration system might be implemented on a J2EE server, passing requests to a legacy client-server backend and then to an enterprise database. Some of these steps or tiers may be shared between applications; for instance our example backend database is likely to be shared with other applications (e.g. tuition billing) which need access to course registration data. In addition, in many cases physical resources may be shared between different components and applications, by either direct co-location or through the use of virtual machines.

For our analysis we can characterize these applications at various tiers in the data center by their requests and the responses to them.[2] In addition, we are interested in both the computational and I/O load incurred by an application when it handles a request, as well as any additional requests it may make to other-tier components in processing the request. (e.g. database queries issued while responding to a request for a dynamic web page) We note that a single component may receive inter-component requests from multiple sources, and may generate requests to several components in turn. Thus the example database server receives queries from multiple applications, while a single application in turn may make requests to multiple databases or other components.

In order to construct models of the operation of these data center applications, we require data on the operations performed as well as their impact. We obtain request or event information from application logs, which typically provide event timestamps, client identity, and information identifying the specific request. Resource usage measurements are gathered from the server OS, and primarily consist of CPU usage and disk operations over some uniform sampling interval.

Problem Formulation: The automated modeling problem may formulated as so. In a system as described, given the request and resource information provided, we wish to automatically derive the following models:[3]

(1) A *workload-to-utilization model*, which models the resource usage of an application as a function of its incoming workload. For instance, the CPU utilization and disk I/O operations due to an application μ_{cpu}, μ_{iop} can be captured as a function of its workload λ:

$$\mu_{cpu} = f_{cpu}(\lambda),\ \mu_{iop} = f_{iop}(\lambda)$$

(2) A *workload-to-workload model*, which models the outgoing workload of an application as a function of its incoming workload. Since the outgoing workload of

[2]This restricts us to request/response applications, which encompasses many but not all data center applications.

[3]Workload-to-response time models are an area for further research.

an application becomes the incoming workload of one or more downstream components, our model derives the workload at one component as a function of another: $\lambda_j = g(\lambda_i)$

We also seek techniques to compose these basic models to represent complex system systems. Such model composition should capture *transitive behavior*, where pair-wise models between applications i and j and j and k are composed to model the relationship between i and k. Further, model composition should allow pair-wise dependence to be extended to n-way dependence, where an application's workload is derived as a function of the workloads seen by all its n upstream applications.

3. DATA CENTER MODELING: BASICS

In this section, we present the intuition behind our basic models, followed by a discussion on constructing composite models of complex data center applications.

3.1 Workload-to-utilization Model

Consider an application component that sees an incoming request rate of λ over some interval τ. We may model the CPU utilization as a function of the aggregate arrival rate and mean service time per request:

$$\mu = \lambda \cdot s \tag{1}$$

where λ is the total arrival rate, s is the mean service time per request, and μ is the CPU usage[4] per unit time, or utilization. By measuring arrival rates and CPU use over time, we may estimate $\hat{s}$ for the service time, allowing us to predict utilization as arrival rate changes.

If each request takes the same amount of time and resources, then the accuracy of this model will be unaffected by changes in either the rate or request type of incoming traffic. However, in practice this is often far from true. Requests typically fall into classes with very different service times: e.g. a web server might receive requests for small static files and computationally-intensive scripts. Equation 1 can only model the average service time across all request types, and if the mix of types changes, it will produce errors.

Let us suppose, that the input stream consists of k distinct classes of requests, where requests in each class have similar service times—in the example above: static files and cgi-bin scripts. Let $\lambda_1, \lambda_2, \ldots \lambda_k$ denote the observed rates for each request class, and let $s_1, s_2, \ldots s_k$ denote the corresponding mean service time. Then the aggregate CPU utilization over the interval τ is a linear sum of the usage due to each request type:

$$\mu = \lambda_1 \cdot s_1 + \lambda_2 \cdot s_2 + \ldots + \lambda_k \cdot s_k + \epsilon \tag{2}$$

where ϵ is a error term assumed random and independent.

If the request classes are well-chosen, then we can sample the arrival rate of each class empirically, derive the above linear model from these measurements, and use it to yield an estimate $\hat{\mu}$ of the CPU utilization due to the incoming workload λ. Thus in our example above, λ_1 and λ_2 might represent requests for small static files and scripts; s_2 would be greater than s_1, representing the increased cost of script processing. The value of this model is that it retains its accuracy when the request

[4] Or alternately, number of disk operations.

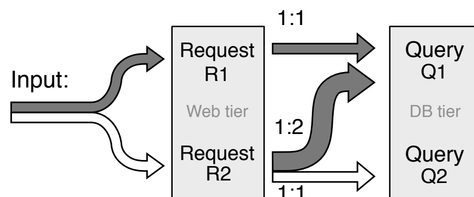


Fig. 2: Example request flow for a 2-tier web application

mix changes. Thus if the overall arrival rate in our example remained constant, but the proportion of script requests increased, the model would account for the workload change and predict an increase in CPU load.

3.2 Workload-to-workload Model

In addition to examining requests at a single component, we next consider two interacting components as shown in Figure 1(b), where incoming requests at i trigger requests to component j. For simplicity we assume that i is the source of all requests to j; the extension to multiple upstream components is straightforward. Let there be k request classes at components i and m classes in the workload seen by j. Let $\lambda_I = \{\lambda_{i1}, \lambda_{i2}, \ldots\}$ and $\lambda_J = \{\lambda_{j1}, \lambda_{j2}, \ldots\}$ denote the class-specific arrival rates at the two components.

To illustrate, suppose that i is a front-end web server and j is a back-end database, and web requests at i may be grouped in classes $R1$ and $R2$. Similarly, SQL queries at the database are grouped in classes $Q1$ and $Q2$, as shown in Figure 2. Each $R1$ web request triggers a $Q1$ database query, while each $R2$ web request triggers two $Q1$ queries and a single $Q2$ query.

We can thus completely describe the workload seen at the database in terms of the web server workload:

$$\lambda_{Q1} = \lambda_{R1} + 2\lambda_{R2}; \quad \lambda_{Q2} = \lambda_{R2} \tag{3}$$

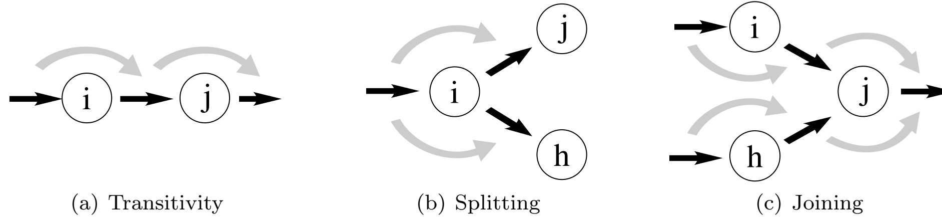
More generally, each request type at component j can be represented as a weighted sum of request types at component i, where the weights denote the number of requests of this type triggered by each request class at component i:

$$\begin{aligned} \lambda_{j1} &= w_{11}\lambda_{i1} + w_{12}\lambda_{i2} + \ldots + w_{1k}\lambda_{ik} + \epsilon_1 \\ \lambda_{j2} &= w_{21}\lambda_{i1} + w_{22}\lambda_{i2} + \ldots + w_{2k}\lambda_{ik} + \epsilon_2 \\ \lambda_{jm} &= w_{m1}\lambda_{i1} + w_{m2}\lambda_{i2} + \ldots + w_{mk}\lambda_{ik} + \epsilon_m \end{aligned} \tag{4}$$

where ϵ_i denotes an error term. Thus, Equation 4 yields the workload at system j, $\lambda_J = \{\lambda_{j1}, \lambda_{j2}, \ldots\}$ as a function of the workload at system i, $\lambda_I = \{\lambda_{i1}, \lambda_{i2}, \ldots\}$.

3.3 Model Composition

The workload-to-utilization (W_2U) model yields the utilization due to an application j as a function of its workload: $\mu_j = f(\lambda_J)$; the workload-to-workload (W_2W) model yields the workload at application j as a function of the workload at application i: $\lambda_J = g(\lambda_I)$. Substituting allows us to determine the utilization at j directly as a function of the workload at i: $\mu_j = f(g(\lambda_I))$. Since f and g are both linear equations, the composite function, obtained by substituting Equation 4 into 2, is also a linear equation. This composition process is transitive: given cascaded



(a) Transitivity (b) Splitting (c) Joining

Fig. 3: Composition of the basic models.

components i, j, and k, it can yield the workload and the utilization of the downstream application k as a function of the workload at i. In a three-tier application, for instance, this lets us predict behavior at the database back-end as a function of user requests at the front-end web server.

Our discussion has implicitly assumed a linear chain topology, where each application sees requests from only one upstream component, illustrated schematically in Figure 3(a). This is a simplification; in a complex data center, applications may both receive requests from multiple upstream components, and in turn issues requests to more than one downstream system. Thus an employee database may see requests from multiple applications (e.g., payroll, directory), while an online retail store may make requests to both a catalog database and a payment processing system. We must therefore be able to model both: (i) "splitting" − triggering of requests to multiple downstream applications, and (ii) "merging", where one application receives request streams from multiple others. (see Figure 3(b) and (c))

To model splits, consider an application i which makes requests of downstream applications j and h. Given the incoming request stream at i, λ_I, we consider the subset of the outgoing workload from i that is sent to j, namely λ_J. We can derive a model of the inputs at i that trigger this subset of outgoing requests using Equation 4: $\lambda_J = g_1(\lambda_I)$. Similarly by considering only the subset of the outgoing requests that are sent to h, we can derive a second model relating λ_H to λ_I: $\lambda_H = g_2(\lambda_I)$.

For joins, consider an application j that receives requests from upstream applications i and h. We first split the incoming request stream by source: $\lambda_J = \{\lambda_J | src = i\} + \{\lambda_J | src = h\}$. The workload contributions at j of i and h are then related to the input workloads at the respective applications using Equation 4: $\{\lambda_J | src = i\} = f_1(\lambda_I)$ and $\{\lambda_J | src = h\} = f_2(\lambda_H)$, and the total workload at j is described in terms of inputs at i and h: $\lambda_J = f_1(\lambda_I) + f_2(\lambda_H)$. Since f_1 and f_2 are linear equations, the composite function, which is the summation of the two—$f_1 + f_2$—is also linear.

By modeling these three basic interactions—cascading, splitting, and joining—we are able to compose single step workload-to-workload and workload-to-utilization models to model any arbitrary application graph. Such a composite model allows workload or utilization at each node to be calculated as a linear function of data from other points in the system.

4. AUTOMATED MODEL GENERATION

We next present techniques for automatically learning models of the form described above. In particular, these models require specification of the following parameters: (i) request classes for each component, (ii) arrival rates in each class, λ_i, (iii) mean service times s_i for each class i, and (iv) rates w_{ij} at which type i requests trigger type j requests. In order to apply the model we must measure λ_i, and estimate s_i and w_{ij}.

If the set of classes and mapping from requests to classes was given, then measurement of λ_i would be straightforward. In general, however, request classes for a component are not known *a priori*. Manual determination of classes is impractical, as it would require detailed knowledge of application behavior, which may change with every modification or extension. Thus, our techniques must *automatically determine an appropriate classification of requests* for each component, as part of the model generation process.

Once the request classes have been determined, we estimate the coefficients s_i and w_{ij}. Given measured arrival rates λ_i in each class i and the utilization μ within a measurement interval, Equations 2 and 4 yield a set of linear equations with unknown coefficients s_i and w_{ij}. Measurements in subsequent intervals yield additional sets of such equations; these equations can be solved using linear regression to yield the unknown coefficients s_i and w_{ij} that minimize the error term ϵ.

A key contribution of our automated model generation is to combine determination of request classes with parameter estimation, in a single step. We do this by mechanically enumerating possible request classes, and then using statistical techniques to select the classes which are predictive of utilization or downstream workload. In essence, the process may be viewed as "mining" the observed request stream to determine features (classes) that are the best predictors of the resource usage and triggered workloads; we rely on step-wise regression—a technique also used in data mining—for our automated model generation.

In particular, for each request we first enumerate a set of possible features, primarily drawn from the captured request string itself. Each of these features implies a classification of requests, into those which have this feature and those which do not. By repeating this over all requests observed in an interval, we obtain a list of candidate classes. We also measure arrival rates within each candidate class, and resource usage over time. Step-wise regression of feature rates against utilization is then used to select only those features that are significant predictors of utilization and to estimate their weights, giving us the workload-to-utilization model.

Derivation of W_2W models is an extension of this. First we create a W_2U model at application j, in order to determine the significant workload features. Then we model the arrival rate of these features, again by using stepwise regression. We model each feature as a function of the input features at i; when we are done we have a model which takes input features at i and predicts a vector of features $\hat{\lambda}_J$ at j.

1: the entire URL:
`/test/PHP/AboutMe.php?name=user5&pw=joe`
2: each URL prefix plus extension:
`/test/.php`, `/test/PHP/.php`
`/test/PHP/AboutMe.php`
3: each URL suffix plus extension:
`AboutMe.php`, `PHP/AboutMe.php`
4: each query variable and argument:
`/test/PHP/AboutMe.php?name=user5`
`/test/PHP/AboutMe.php?pw=joe`
5: all query variables, without arguments:
`/test/PHP/AboutMe.php?name=&pw=`

Fig. 4: HTTP feature enumeration algorithm.

4.1 Feature Enumeration

For this approach to be effective, classes with stable behavior (mean resource requirements and request generation) must exist. In addition, information in the request log must be sufficient to determine this classification. We present an intuitive argument for the existence of such classes and features, and a description of the feature enumeration techniques used in Modellus.

We first assert that this classification is possible, within certain limits: in particular, that in most cases, system responses to identical requests are similar, across a broad range of operating conditions. Consider, for example, two identical requests to a web server for the same simple dynamic page—regardless of other factors, identical requests will typically trigger the same queries to the back-end database. In triggering these queries, the requests are likely to invoke the same code paths and operations, resulting in (on average) similar resource demands.[5]

Assuming these request classes, we need an automated technique to derive them from application logs—to find requests which perform similar or identical operations, on similar or identical data, and group them into a class. The larger and more general the groups produced by our classification, the more useful they will be for actual model generation. At the same time, we cannot blindly try all possible groupings, as each unrelated classification tested adds a small increment of noise to our estimates and predictions.

In the cases we are interested in, e.g. HTTP, SQL, or XML-encoded requests, much or all of the information needed to determine request similarity is encoded convention or by syntax in the request itself. Thus we would expect the query `'SELECT * from cust WHERE cust.id=105'` to behave similarly to the same query with `'cust.id=210'`, while an HTTP request for a URL ending in `'images/map.gif'` is unlikely to be similar to one ending in `'browse.php?category=5'`.

Our enumeration strategy consists of extracting and listing *features* from request strings, where each feature identifies a potential candidate request class. Each enumeration strategy is based on the formal or informal[6] syntax of the request

[5]Caching will violate this linearity assumption; however, we argue that in this case behavior will fall into two domains—one dominated by caching, and the other not—and that a linear approximation is appropriate within each domain.

[6]E.g. HTTP, where hierarchy, suffixes, and query arguments are defined by convention rather than standard.

```
1: database: TPCW
2: database and table(s): TPCW:item,author
3: query "skeleton":
   SELECT * FROM item,author WHERE item.i_a_id=author.a_id AND i_id=?
4: the entire query:
   SELECT * FROM item,author WHERE item.i_a_id=author.a_id AND i_id=1217
5: query phrase:
   WHERE item.i_a_id=author.a_id AND i_id=1217
6: query phrase skeleton:
   WHERE item.i_a_id=author.a_id AND i_id=?
```

Fig. 5: SQL Feature Enumeration

and it enumerates the portions of the request which identify the class of operation, the data being operated on, and the operation itself, which are later tested for significance. We note that the feature enumeration algorithm must be manually specified for each application type, but that there are a relatively small number of such types, and once algorithm is specified it is applicable to any application sharing that request syntax.

The Modellus feature enumeration algorithm for HTTP requests is shown in Figure 4, with features generated from an example URL. The aim of the algorithm is to identify request elements which may identify common processing paths; thus features include file extensions and URL prefixes, and query skeletons (i.e. a query with arguments removed), each of which may identify common processing paths. In Figure 5 we see the feature enumeration algorithm for SQL database queries, which uses table names, database names, query skeletons, and SQL phrases (which may be entire queries in themselves) to generate a list of features. Feature enumeration is performed on all requests present in an application's log file over a measurement window, one request at a time, to generate a list of candidate features.

4.2 Feature Selection using Stepwise Linear Regression

Once the enumeration algorithm generates a list of candidate features, the next step is to use training data to learn a model by choosing only those features whose coefficients s_i and w_{ij} minimize the error terms in Equations 2 and 4. In a theoretical investigation, we might be able to compose benchmarks consisting only of particular requests, and thus measure the exact system response to these particular requests. In practical systems, however, we can only observe aggregate resource usage given an input stream of requests which we do not control. Consequently, the model coefficients s_i and w_{ij} must also be determined as part of the model generation process.

One naïve approach is to use *all* candidate classes enumerated in the previous step, and to employ least squares regression on the inputs (here, arrival rates within each candidate class) and outputs (utilization or downstream request rates) to determine a set of coefficients that best fit the training data. However, this will generate spurious features with no relationship to the behavior being modeled; if included in the model they will degrade its accuracy, in a phenomena known as *over-fitting* in the machine learning literature. In particular, some will be chosen due to random correlation with the measured data, and will contribute noise to future predictions.

This results in a data mining problem: out of a large number of candidate classes

```
1: Let Λ_model = {}, Λ_remaining = {λ_1, λ_1, ...}, e = μ
2: while Λ_remaining ≠ φ do
3:     for λ_i in Λ_model do
4:         e(λ_i) ← error (Λ_model − λ_i)
5:     end for
6:     λ_i ← min_i e(λ_i)
7:     if TEST(e(λ_i)) = not in model then
8:         Λ_model ← Λ_model − λ_i
9:     end if
10:    for λ_i in Λ_remaining do
11:        e(λ_i) ← error (Λ_model + λ_i)
12:    end for
13:    λ_i ← min_i e(λ_i)
14:    if TEST(e(λ_i)) = in model then
15:        Λ_model ← Λ_model ∪ λ_i
16:    else
17:        return Λ_model
18:    end if
19: end while
```

Fig. 6: Stepwise Linear Regression Algorithm

and measurements of arrival rates within each class, determining those which are predictive of the output of interest, and discarding the remainder. In statistics this is termed a *variable selection* problem [Draper and Smith 1998], and may be solved by various techniques which in effect determine the odds of whether each input variable influences the output or not. Of these methods we use Stepwise Linear Regression, [M. A. Efroymson 1960] due in part to its scalability, along with a modern extension—the Foster and George's risk inflation criteria [Foster and George 1994].

A simplified version of this algorithm is shown in Figure 6, with input variables λ_i and output variable μ. We begin with an empty model; as this predicts nothing, its error is exactly μ. In the first step, the variable which explains the largest fraction of μ is added to the model. At each successive step the variable explaining the largest fraction of the remaining error is chosen; in addition, a check is made to see if any variables in the model have been made redundant by ones added at a later step. The process completes when no remaining variable explains a statistically significant fraction of the response.

To summarize, the full process to collect data, enumerate features, and build a model is as follows:

(1) Collect log data and resource utilization over N intervals of t seconds each. (e.g. $N = 200, t = 10$)

(2) Enumerate features, identifying candidate features $F_i | i = 1 \dots p$

(3) Determine arrival rate λ_{ij} for each feature i over each measurement interval $j = 1 \dots N$.

(4) Perform stepwise regression on λ_{ij} and utilization measurement, resulting in a model β.

5. ACCURACY, EFFICIENCY AND STABILITY

We have presented techniques for automatic inference of our basic models and their composition. However, several practical issues arise when implementing these techniques into a system:

- —*Workload changes:* Although our goal is to derive models which are resilient to shifts in workload composition as well as volume, some workload changes will cause model accuracy to decrease — for instance, the workload may become dominated by requests not seen in the training data. When this occurs, prediction errors will persist until the relevant models are re-trained.
- —*Effective model validation and re-training:* In order to quickly detect shifts in system behavior which may invalidate an existing model, without un-necessarily retraining other models which remain accurate, it is desirable to periodically test each model for validity. The lower the overhead of this testing, the more frequently it may be done and thus the quicker the models may adjust to shifts in behavior.
- —*Cascading errors:* Models may be composed to make predictions across multiple tiers in a system; however, uncertainty in the prediction increases in doing so. Methods are needed to estimate this uncertainty, so as to avoid making unreliable predictions.
- —*Stability:* Some systems will be difficult to predict with significant accuracy. Rather than spending resources repeatedly deriving models of limited utility, we should detect these systems and limit the resources expended on them.

In the following section we discuss these issues and the mechanisms in Modellus which address them.

5.1 Model Validation and Adaptation

A trained model makes predictions by extrapolating from its training data, and the accuracy of these predictions will degrade in the face of behavior not found in the training data. Another source of errors can occur if the system response changes from that recorded during the training window; for instance, a server might become slower or faster due to failures or upgrades.

In order to maintain model accuracy, we must retrain models when this degradation occurs. Rather than always re-learning models, we instead test predictions against actual measured data; if accuracy declines below a threshold, then new data is used to re-learn the model.

In particular, we sample arrival rates in each class ($\hat{\lambda_i}$) and measure resource utilization $\hat{\mu}$. Given the model coefficients s_i and w_{ij}, we substitute $\hat{\lambda_i}$ and $\hat{\mu}$ into Equations 2 and 4, yielding the prediction error ϵ. If this exceeds a threshold ϵ_T in k out of n consecutive tests, the model is flagged for re-learning.

A simple approach is to test all models at a central node; data from each system is collected over a testing window and verified. Such continuous testing of tens or hundreds of models could be computationally expensive. We instead propose a fast, distributing model testing algorithm based on the observation that *although model derivation is expensive* in both computation and memory, *model checking is cheap*. Hence model validation can be distributed to the systems being monitored themselves, allowing nearly continuous checking.

In this approach, the model—the request classes and coefficients—is provided to each server or node and can be tested locally. To test workload-to-usage models, a node samples arrival rates and usages over a short window, and compares the usage predictions against observations. Workload-to-workload models are tested

similarly, except that communication with the downstream node is required to obtain observed data. If the local tests fail in k out of n consecutive instances, then a full test is triggered at the central node; full testing involves a larger test window and computation of confidence intervals of the prediction error. Distributed testing over short windows is fast and imposes minimal overheads on server applications; due to this low overhead, tests can be frequent to detect failures quickly.

5.2 Limiting Cascading Errors

In the absence of errors, we could monitor only the external inputs to a system, and then predict all internal behavior from models. In practice, models have uncertainties and errors, which grow as multiple models are composed.

Since our models are linear, errors also grow linearly with model composition. This may be seen by substituting Equation 4 into Equation 2, yielding a composite model with error term $\sum s_i \cdot \epsilon_i + \epsilon$, a linear combination of individual errors. Similarly, a “join” again yields an error term summing individual errors.

Given this linear error growth, there is a limit on the number of models that may be composed before the total error exceeds any particular threshold. Hence, we can no longer predict all internal workloads and resource usages solely by measuring external inputs. In order to scale our techniques to arbitrary system sizes, we must to measure additional inputs inside the system, and use these measurement to drive further downstream predictions.

To illustrate how this may be done, consider a linear cascade of dependencies, and suppose ϵ_{max} is the maximum tolerable prediction error. The first node in this chain sees an external workload that is known and measured; we can compute the expected prediction error at successive nodes in the chain until the error exceeds ϵ_{max}. Since further predictions will exceed this threshold, a new measurement point must inserted here to measure, rather than predict, its workload; these measurements drive predictions at subsequent downstream nodes.

This process may be repeated for the remaining nodes in the chain, yielding a set of measurement points which are sufficient to predict responses at all other nodes in the chain. This technique easily extends to an arbitrary graph; we begin by measuring all external inputs and traverse the graph in a breadth-first manner, computing the expected error at each node. A measurement point is inserted if a node's error exceeds ϵ_{max}, and these measurements are then used to drive downstream predictions.

5.3 Stability Considerations

Under certain conditions, however, it will not be possible to derive a useful model for predicting future behavior. If the system behavior is dominated by truly random factors, for instance, model-based predictions will be inaccurate. A similar effect will be seen in cases where there is insufficient information available. Even if the system response is deterministically related to some attribute of the input, as described below, the log data may not provide that that attribute. In this case, models learned from random data will result in random predictions.

In order to avoid spending a large fraction of system resources on the creation of useless models, Modellus incorporates backoff heuristics to detect applications which fail model validation more than k times within a period T. (e.g. 2 times

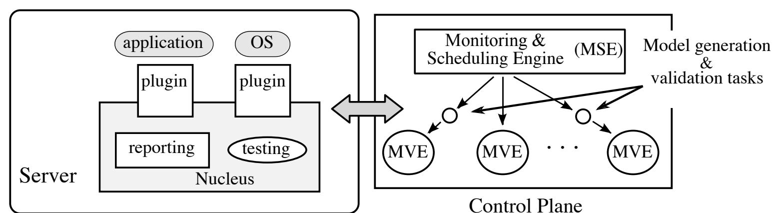


Fig. 7: Modellus components

within the last hour) These "mis-behaving" applications are not modeled, and are only occasionally examined to see whether their behavior has changed and modeling should be attempted again.

6. MODELLUS DESIGN

Our Modellus system implements the statistical and learning methods described in the previous sections. Figure 7 depicts the Modellus architecture. As shown, Modellus implements a *nucleus* on each node to monitor the workload and resource usage, and to perform distributed testing. The Modellus *control plane* resides on one or more dedicated nodes and comprises (i) a *Monitoring and Scheduling Engine (MSE)* which coordinates the gathering of monitoring data and scheduling of model generation and validation tasks when needed, and (ii) one or more *Modeling and Validation Engines (MVE)* which implements the core numerical computations for model derivation and testing. The Modellus control plane exposes a front-end allowing derived models to be applied to data center analysis tasks; the current front-end exports models and sampled statistics to a Matlab engine for interactive analysis.

The Modellus nucleus and control plane are implemented in a combination of C++, Python, and Numerical Python [Ascher et al. 2001], providing an efficient yet dynamically extensible system. The remainder of this section discusses our implementation of these components in detail.

6.1 Modellus Nucleus

The nucleus is deployed on each target system, and is responsible for both data collection and simple processing tasks. It monitors resource usage, tracks application events, and translates events into rates. The nucleus reports these usages and rates to the control plane, and can also test a control plane-provided model against this data. A simple HTTP-based interface is provided to the control plane, with commands falling into the following groups: (i) monitoring configuration, (ii) data retrieval, and (iii) local model validation.

Monitoring: The nucleus performs *adaptive monitoring* under the direction of the control plane—it is instructed which variables to sample and at what rate. It implements a uniform naming model for data sources, and an extensible plugin architecture allowing support for new applications to be easily implemented.

Resource usage is monitored via standard OS interfaces, and collected as counts or utilizations over fixed measurement intervals. Event monitoring is performed by plugins which process event streams (i.e. logs) received from applications. These plugins process logs in real time and generate a stream of request arrivals; class-

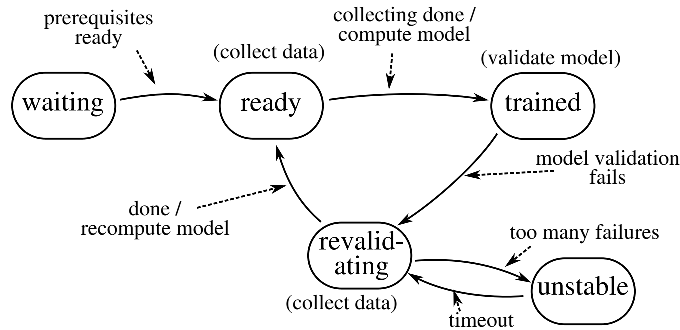


Fig. 8: Model training states

specific arrival rates are then measured by mapping each event using application-specific feature enumeration rules and model-specified classes.

The Modellus nucleus is designed to be deployed on production servers, and thus must require minimal resources. By representing feature strings by hash values, we are able to implement feature enumeration and rate monitoring with minimal overhead, as shown experimentally in Section 7.5. We have implemented plugins for HTTP and SQL, with particular adaptations for Apache, MySQL, Tomcat, and XML-based web services.

Data retrieval: A monitoring agent such as the Modellus nucleus may either report data asynchronously (*push*), or buffer it for the receiver to retrieve (*pull*). In Modellus data is buffered for retrieval, with appropriate limits on buffer size if data is not retrieved in a timely fashion. Data is serialized using Python's *pickle* framework, and then compressed to reduce demand on both bandwidth and buffering at the monitored system.

Validation and reporting: The nucleus receives model validation requests from the control plane, specifying input classes, model coefficients, output parameter, and error thresholds. It periodically measures inputs, predicts outputs, and calculates the error; if out of bounds k out of n times, the control plane is notified. Testing of workload-to-workload models is similar, except that data from two systems (upstream and downstream) is required; the systems share this information without control plane involvement.

6.2 Monitoring and Scheduling Engine

The main goal of the Modellus control plane is to to generate up-to-date models and maintain confidence in them by testing. Towards this end, the monitoring and scheduling engine (MSE) is responsible for (i) initiating data collection from the nuclei for model testing or generation, and (ii) scheduling testing or model re-generation tasks on the modeling and validation engines (MVEs).

The monitoring engine issues data collection requests to remote nuclei, requesting sampled rates for each request class when testing models, and the entire event stream for model generation. For workload-to-workload models, multiple nuclei are involved in order to gather upstream and downstream information. Once data collection is initiated, the monitoring engine periodically polls for monitored data, and disables data collection when a full training or testing window has been received.

The control plane has access to a list of workload-to-utilization and workload-to-workload models to be inferred and maintained; this list may be provided by configuration or discovery. These models pass through a number of states, which may be seen in Figure 8: *waiting* for prerequisites, *ready* to train, *trained*, *re-validating*, and *unstable*. Each W_2W model begins in the *waiting* state, with the downstream W_2U model as a prerequisite, as the feature list from this W_2U model is needed to infer the W_2W model. Each W_2U model begins directly in the *ready* state. The scheduler selects models from the ready pool and schedules training data collection; when this is complete, the model parameters may be calculated. Once parameters have been calculated, the model enters the *trained* state; if the model is a prerequisite for another, the waiting model is notified and enters the *ready* state.

Model validation as described above is performed in the *trained* state, and if at any point the model fails, it enters *revalidating* state, and training data collection begins. Too many validation failures within an interval cause a model to enter the *unstable* state, and training ceases, while from time to time the scheduler selects a model in the unstable state and attempts to model it again. Finally, the scheduler is responsible for distributing computation load within the MVE, by assigning computation tasks to appropriate systems.

6.3 Modeling and Validation Engine

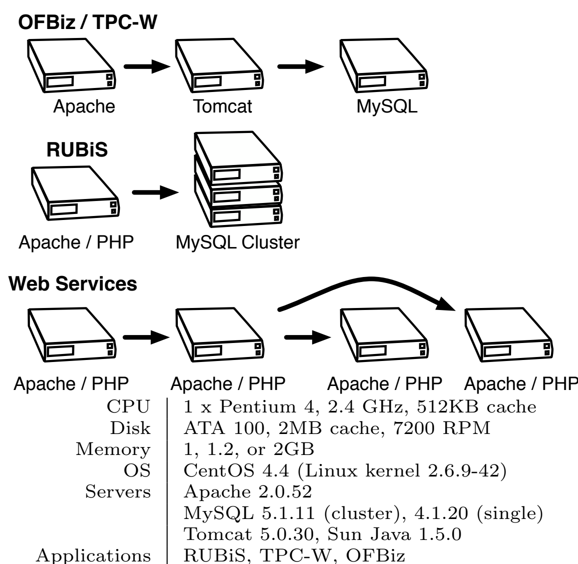
The modeling and validation engine (MVE) is responsible for the numeric computations at the core of the Modellus system. Since this task is computationally demanding, a dedicated system or cluster is used, avoiding overhead on the data center servers themselves. By implementing the MVE on multiple systems and testing and/or generating multiple models in parallel, Modellus will scale to large data centers, which may experience multiple concurrent model failures or high testing load.

The following functions are implemented in the MVE:

Model generation: W_2U models are derived directly; W_2W models are derived using request classes computed for the downstream node's W_2U model. In each case step-wise regression is used to derive coefficients relating input variables (feature rates) to output resource utilization (W_2U models) or feature rates (W_2W models).

Model validation: Full testing of the model at the control plane is similar but more sophisticated than the fast testing implemented at the nucleus. To test an W_2U model, the sampled arrival rates within each class and measured utilization are substituted into Equation 2 to compute the prediction error. Given a series of prediction errors over successive measurement intervals in a test window, we compute the 95% one-sided confidence interval for the mean error. If the confidence bound exceeds the tolerance threshold, the model is discarded.

The procedure for testing an W_2W model is similar. The output feature rates are estimated and compared with measured rates to determine prediction error and a confidence bound; if the bound exceeds a threshold, again the model is invalidated. Since absolute values of the different output rates in a W_2W model may vary widely, we normalize the error values before performing this test, by using the downstream model coefficients as weights, allowing us to calculate a scaled error magnitude.



CPU	1 x Pentium 4, 2.4 GHz, 512KB cache
Disk	ATA 100, 2MB cache, 7200 RPM
Memory	1, 1.2, or 2GB
OS	CentOS 4.4 (Linux kernel 2.6.9-42)
Servers	Apache 2.0.52
	MySQL 5.1.11 (cluster), 4.1.20 (single)
	Tomcat 5.0.30, Sun Java 1.5.0
Applications	RUBiS, TPC-W, OFBiz

Fig. 9: Data Center Testbed and system specifications.

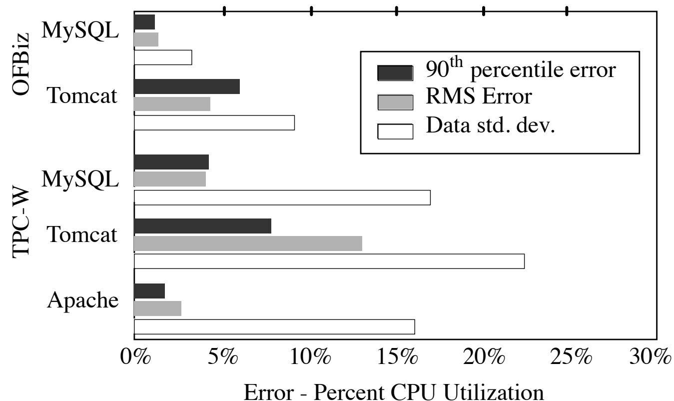


Fig. 10: Workload-to-Utilization prediction errors

7. EXPERIMENTAL RESULTS

In this section we present experiments examining various performance aspects of the proposed methods and system. To test feature-based regression modeling, we perform modeling and prediction on multiple test scenarios, and compare measured results with predictions to determine accuracy. Additional experiments examine errors under shifting load conditions and for multiple stages of prediction. Finally, we present measurements and benchmarks of the system implementation, in order to determine the overhead which may be placed on monitoring systems and the scaling limits of the rest of the system.

7.1 Experimental Setup

The purpose of the Modellus system is to model and predict performance of real-world web applications, and it was thus tested on results from a realistic data center testbed and applications. The testbed is shown in Figure 9, with a brief synopsis of the hardware and software specifications. Four web applications were tested:

(1) TPC-W [Smith ; Cain et al. 2001]: an e-commerce benchmark, implemented as a 3-tier Java servlet-based application, consisting of a front-end server (Apache) handling static content, a servlet container (Tomcat), and a back-end database (MySQL).
(2) Apache Open For Business (OFBiz) [openforbiz 2007]: an ERP (Enterprise Resource Planning) system in commercial use. Uses the same Apache, Tomcat, MySQL setup as TPC-W.
(3) RUBiS [Cecchet et al. 2003]: a simulated auction site running as a 2-tier LAMP[7] application; application logic written in PHP runs in an Apache front-end server, while data is stored in a MySQL database cluster.
(4) Web Services Benchmark: a custom set of Apache/PHP based components that can be connected in arbitrary topologies through a RESTful web API.

Both RUBiS and TPC-W have associated workload generators which simulate varying numbers of clients; the number of clients as well as their activity mix and think times were varied over time to generate non-stationary workloads. A load generator for OFBiz was created using JWebUnit [jwebunit 2007], which simulated multiple users performing shopping tasks from browsing through checkout and payment information entry. The httperf [Mosberger and Jin 1998] tool was used as a load generator for the Web Services Benchmark.

Apache, Tomcat, and MySQL were configured to generate request logs, and system resource usage was sampled using the `sadc(8)` utility with a 1-second collection interval. Traces were collected and prediction was performed off-line, in order to allow re-use of the same data for validation. Cross-validation was used to measure prediction error: each trace was divided into training windows of a particular length (e.g. 30 minutes), and a model was constructed for each window. Each model was then used to predict all data points outside of the window on which it was trained; deviations between predicted and actual values were then measured.

7.2 Model Generation Accuracy

To test W_2U model accuracy, we use the OFBiz, TPC-W and RUBiS applications and model each of the tiers individually. Using traces from each application we compute models over 30-minute training windows, and then use these models to predict utilization $\hat{\mu}$ for 30-second intervals, using cross-validation as described above. We report the root mean square (RMS) error of prediction, and the 90th percentile absolute error ($|\mu - \hat{\mu}|$). For comparison we also show the standard deviation of the measured data itself, $\sigma(y)$.

In Figure 10 we see results from these tests. Both RMS and 90th percentile prediction error are shown for each server except the OFBiz Apache front end,

[7] Linux/Apache/MySQL/PHP

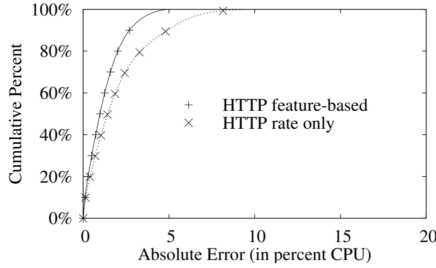


Fig. 11: Error CDF : RUBiS Apache

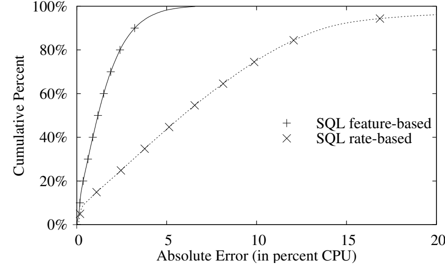


Fig. 12: Error CDF: RUBiS MySQL

which was too lightly loaded ($< 3\%$) for accurate prediction. In addition we plot the standard deviation of the variable being predicted (CPU utilization), in order to indicate the degree of error reduction provided by the model. In each case we are able to predict CPU utilization to a high degree of accuracy—less than 5% except for the TPC-W Tomcat server, and in all cases a significant reduction relative to the variance of the data being predicted.

We examine the distribution of prediction errors more closely in Figures 11 and 12, using the RUBiS application. For each data point predicted by each model we calculate the prediction error ($|\hat{y} - y|$), and display a cumulative histogram or CDF of these errors. From these graphs we see that about 90% of the Apache data points are predicted within 2.5%, and 90% of the MySQL data points within 5%.

In addition, in this figure we compare the performance of modeling and predicting based on workload features vs. predictions made from the aggregate request rate alone. Here we see that CPU on the RUBiS Apache server was predicted about twice as accurately using feature-based prediction, while the difference between naïve and feature-based prediction on the MySQL server was even greater.

7.3 Model Composition

The results presented above examine performance of single workload-to-utilization (W_2U) models. We next examine prediction performance when composing workload-to-workload (W_2W) and W_2U models. We show results from the multi-tier experiments described above, but focus on cross-tier modeling and prediction.

As described earlier, the composition of two models is done in two steps. First, we train a W_2U model for the downstream system (e.g. the database server) and its inputs. Next, we take the list of significant features identified in this model, and for each feature we train a separate upstream model to predict it. For prediction, the W_2W model is used to predict input features to the W_2U model, yielding the final result. Prediction when multiple systems share a single back-end resource is similar, except that the outputs of the two W_2W models must be summed before input to the W_2U model.

In Figure 13 we compare learning curves for both composed and direct W_2U models of the RUBiS application. In the composed model (HTTP→SQL→DB CPU), the W_2W model translates from HTTP features to SQL features, then the W_2U model is in turn used to predict the CPU utilization of the database; the model achieves less than 10% RMS error for training windows over eight minutes.

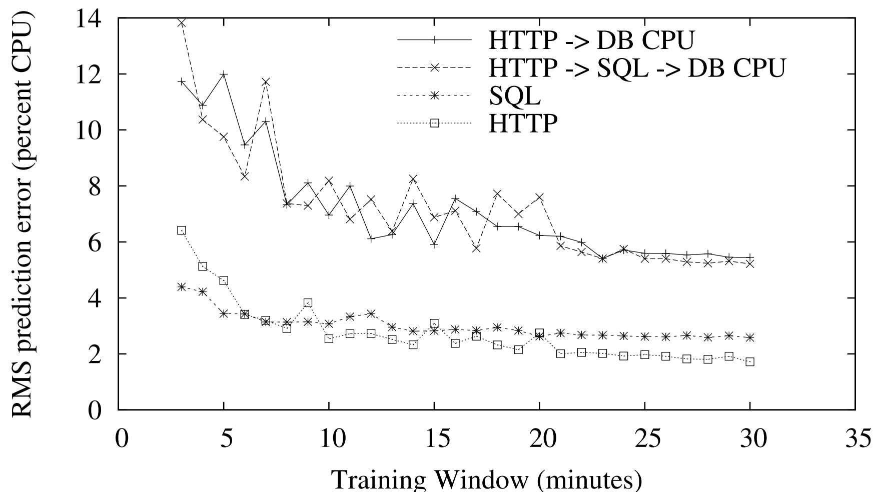


Fig. 13: Composed vs. and direct prediction: learning curves for error of MySQL CPU from RUBiS HTTP

Alternatively, the HTTP features from the front tier can be used for a single W_2U model that predicts CPU usage of the database tier (HTTP→DB CPU), which has comparable error and requires a similar training period.

In addition we validate our model composition approach by comparing its results to two direct models made by training the HTTP or SQL server CPU utilization directly from their own inputs. While the direct models have lower error, it may not always be possible to observe all input features at every tier. Using model composition can lower monitoring overhead by allowing measurements from one tier to predict resource needs at downstream tiers.

7.4 Cascading Errors

We measured prediction performance of the Web Services application in order to investigate the relationship between model composition and errors. This benchmark allows application components to be defined and connected in arbitrary topologies. Three separate topologies were measured, corresponding to the model operations in Figure 3—cascade, split, and join— and prediction errors were measured between each pair of upstream and downstream nodes. In Figure 14 we see results for the cascade topology, giving prediction errors for model composition across multiple tiers; errors grow modestly, reaching at most about 4%.

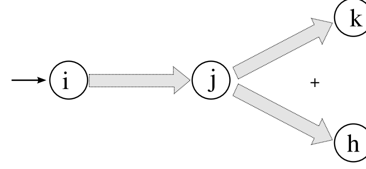
In Figure 15 we see results for the split topology, and the join case in Figure 16. In each case prediction errors are negligible. Note that in the join case, downstream predictions must be made using both of the upstream sources. This does not appear to affect accuracy; although the final prediction contains errors from two upstream models, they are each weighted proportionally.

7.5 System Overheads

We have benchmarked both the Modellus nucleus and the computationally intensive portions of the control plane. The nucleus was benchmarked on the testbed

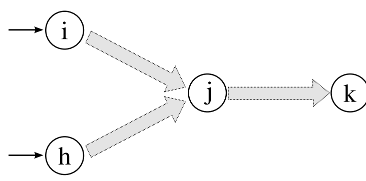
	Prediction target			
	I	J	H	K
I	0.47%	4.28%	3.45%	4.12%
J		0.82%	0.97%	1.21%
H			0.50%	0.70%
K				0.64%

Fig. 14: Cascade errors. Entry (**x**,**y**) gives RMS error predicting at **y** given inputs to **x**.



	Prediction target			
	I	J	H	K
I	0.35%	0.47%	0.43%	0.57%
J		0.47%	0.43%	0.57%
H			0.39%	
K				0.53%

Fig. 15: Split errors - composition with multiple downstream servers.



	Prediction target			
	I	H	J	K
I	0.55%			
H		0.48%		
I+H				0.59%
J			0.59%	
K				0.82%

Fig. 16: Join error: composition by summing multiple upstream models.

	CPU / event	Equiv. overhead	Output data
HTTP (TPC-W)	16.5μs	2.7%	9.00 bytes/s
HTTP (World Cup)	11.8μs	n/a	9.05 bytes/s
SQL	23.8μs	2.6%	

Table I: Overhead for Modellus log processing on trace workloads.

W_2U model	features	considered	
Training window	500	1000	2000
short (8 min)	0.06s	0.12	0.24
medium (15 min)	0.10	0.20	0.42
long (30 min)	0.16	0.33	0.72

Table II: Training times for Workload-to-Utilization models.

machines to determine both CPU utilization and volume of data produced. HTTP and SQL processing overheads were measured on log data from the TPC-W benchmark; in addition, HTTP measurements were performed for logfiles from the 1998 World Cup web site [Arlitt and Jin 1999].

Based on the request rate in the trace logs and the CPU utilization while they were being generated, we report the estimated overhead due to Modellus event processing if the server were running at 100% CPU utilization. Figures include overhead for compressing the data; in addition, we report the rate at which compressed data is generated, as it must be buffered and then transmitted over the network. Results may be seen in Table I; in each case Modellus incurs less than 3% overhead.

We measure the computationally intensive tasks of the Modeling and Validation

W_2W model	features	considered	
Training window	500	1000	2000
short (8 min)	0.4s	0.3	0.3
medium (15 min)	0.8	0.7	0.8
long (30 min)	1.1	1.0	1.1

Table III: Training times for Workload-to-Workload models.

Engine to determine the scalability of the system. Tests were run using two systems: a 2.8GHz Pentium 4 with 512K cache, and a 2.3GHz Xeon 5140 with 4M cache. Results are reported below for only the Xeon system, which was approximately 3 times faster on this task than the Pentium 4. Each test measured the time to train a model; the length of the training window and the number of features considered was varied, and multiple replications across different data sets were performed for each combination.

Results for training W_2U models are seen in Table II. For 30 minute training windows and 1000 features considered, a single CPU core was able to compute 3 models per second. Assuming that at most we would want to recompute models every 15 minutes—i.e. overlapping half of the previous training window— a single CPU would handle model computation for over 2500 monitored systems. W_2W model training is computationally more complex; results for the same range of model sizes are shown in Table III. These measurements showed a very high data-dependent variation in computation time, as complexity of computing the first-tier model is directly affected by the number of significant features identified at the second tier. We see that computation time was primarily determined by the training window length. For 30 minute windows our system took about a second per model computation; calculating as above, it could handle training data from nearly 1000 monitored systems.

Unlike model generation, model testing is computationally simple. Validation of a W_2U model across a window of 30 minutes of data, for example, required between 3 and 6 milliseconds on the system used above.

7.6 Limitations of our Approach

Our approach to modeling system performance based on input features has a number of limitations, which we explore in this section. As with any statistical process, the larger the random component of a given amount of measured data, the higher the resulting error will be. In Modellus, such errors may be due to almost purely random factors (e.g. scheduling and queuing delays) or to "hidden variables" - factors which may deterministically affect system response, but which we are unable to measure. In this section we demonstrate the effects of such errors by modification of testbed traces.

Simulated CPU traces were created from actual TPC-W data traces, by assigning weights to each of the TPC-W operations and then adding a lognormal-distributed random component to each processing time. Workload-to-utilization models were then trained on the original input stream and the resulting utilization data, and prediction results are reported. These may be seen in Figure 17, where prediction error for a fixed training window size may be seen to grow roughly linearly with the variability of the data. From this we see that increases in variability will result in either longer training windows, lower accuracy, or some combination of the two.

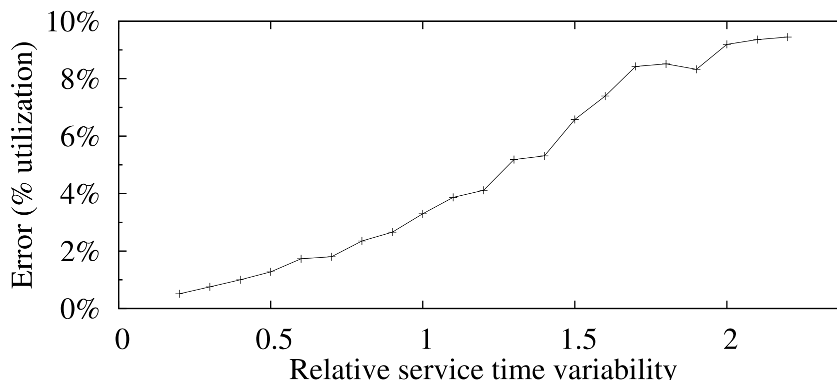


Fig. 17: Prediction error vs. scaled data variability

Application	Mix	Web reqs/sec	Predicted	Measured	Error
Apache	Browsing	90	32.00%	20.00%	+12.00%
	Browsing	114	40.53%	31.40%	+9.13%
	Ordering	90	43.00%	41.00%	+2.00%
Tomcat	Browsing	90	37.00%	32.00%	+5.00%
	Browsing	114	46.87%	45.00%	+1.87%
	Ordering	90	56.00%	51.00%	+5.00%
MySQL	Browsing	90	25.00%	17.30%	+7.70%
	Browsing	114	31.67%	26.00%	+5.67%
	Ordering	90	66.00%	69.00%	-3.00%

Table IV: Case study: Predicted impact of workload changes

8. DATA CENTER ANALYSIS

In this section we apply the Modellus methodology to actual and simulated real-world scenarios.

8.1 Online Retail Scenario

First we demonstrate the utility of our models for “what-if” analysis of data center performance.

Consider an online retailer who is preparing for the busy annual holiday shopping season. We assume that the retail application is represented by TPC-W, which is a full-fledged implementation of an 3-tier online store and a workload generator that has three traffic mixes: *browsing*, *shopping* and *ordering*, each differing in the relative fractions of requests related to browsing and buying activities. We assume that the shopping mix represents the typical workload seen by the application. Suppose that the retailer wishes to analyze the impact of changes in the workload mix and request volume in order to plan for future capacity increases. For instance, during the holiday season it is expected that the rate of buying will increase and so will the overall traffic volume. We employ Modellus to learn models based on the typical shopping mix and use it to predict system performance for various what-if scenarios where the workload mix as well as the volume change.

We simulate this scenario on our data center testbed, as described in Section 7.1.

Model training was performed over a 2 hour interval with varying TPC-W and RUBiS load, using the TPC-W "shopping" mixture. We then used this model to express utilization of each system in terms of the different TPC-W requests, allowing us to derive utilization as a function of requests per second for each of the TPC-W transaction mixes. The system was then measured with several workloads consisting of either TPC-W "browsing" or "ordering" mixtures.

Predictions are shown in Table IV for the three traffic mixes, on the three servers in the system: Apache, which only forwards traffic; Tomcat, which implements the logic, and MySQL. Measurements are shown as well for two test runs with the browsing mixture and one with ordering. Measured results correspond fairly accurately to predictions, capturing both the significant increase in database utilization with increased buying traffic as well as the relative independence of the front-end Apache server to request mix.

8.2 Financial Application Analysis

The second case study examines the utility of our methods on a large stock trading application at a real financial firm, using traces of stock trading transactions executed at a financial exchange. Resource usage logs were captured over two 72-hour periods in early May, 2006; in the 2-day period between these intervals a hardware upgrade was performed. The event logs captured 240,000 events pre-upgrade and 100,000 events after the upgrade occurred. We present the accuracy of Modellus modeling both CPU utilization and disk I/O rates before and after the upgrade; this demonstrates the importance of continuously validating models and adapting them to changes in hardware.

In contrast to the other experiments in this paper, only a limited amount of information is available in these traces. CPU utilization and disk traffic were averaged over 60s intervals, and, for privacy reasons, the transaction log contained only a database table name and status (success/failure) for each event. This results in a much smaller number of features to be used as inputs to the model. In Table V we see predicted values for three variables—CPU utilization, physical reads, and logical reads—compared to measured values and a naïve rate-based model. The poor performance of the naïve model indicates that Modellus can get significant accuracy gains even from the coarse grain features available in this scenario.

After the upgrade is performed, the prediction accuracy of the model falls significantly because it does not know that the underlying CPU and disk hardware has been changed. However, the Modellus validation engine is able to detect that the predicted CPU and disk utilizations have a high level of error, causing the system to recalculate the model. After the model has been retrained, the accuracy returns to the level seen prior to the upgrade.

9. RELATED WORK

Application modeling: Server application models can be classified as either *black-box* or *white-box* models. Black-box models describe externally visible performance characteristics of a system, with minimal assumptions about the internal operations; white-box models, in contrast, are based on knowledge of these internals. Black-box models are used in a number of approaches to data center control via feedback mechanisms. MUSE [Chase et al. 2001] uses a market bidding mecha-

		cpu	preads	reads (·1000)
Pre-upgrade	Naive	38.95	6748	1151
	Feature-based	47.46	10654	1794
	Measured	47.20	8733	1448
Post-upgrade	Measured	31.03	6856	2061
	Outdated Model	71.18	4392	1369
	Recomputed Model	24.34	4819	1471

Table V: Trading system traces - feature-based and naïve rate-based estimation vs. measured values. The model must be retrained after the hardware upgrade.

nism to optimize utility, while Model-Driven Resource Provisioning (MDRP) [Doyle et al. 2003] uses dynamic resource allocation to optimize SLA satisfaction. Several control theory-based systems use admission control, instead, reducing the input to fit the resources available [Kamra et al. 2004; Parekh et al. 2002].

While black-box models concern themselves only with the inputs (requests) and outputs (measured response and resource usage), white-box models are based on causal links between actions. Magpie [Barham et al. 2004] and Project5 [Aguilera et al. 2003] use temporal correlation on OS and packet traces, respectively, to find event relationships. In a variant of these methods, Jiang *et al.* [Jiang et al. 2006] use an alternate approach; viewing events of a certain type as a *flow*, sampled over time, they find invariant ratios between event flow rates, which are typically indicative of causal links. Likewise, Menasce *et al.* [Menascé et al. 2000] characterized workloads using hierarchical models over multiple time scales.

Queuing models: Given knowledge of a system's internal structure, a queuing model may be created, which can then be calibrated against measured data, and then used for analysis and prediction. Stewart [Stewart and Shen 2005] uses this approach to analyze multi-component web applications with a simple queuing model. Urgaonkar [Urgaonkar et al. 2005] uses more sophisticated product-form queuing network models to predict application performance for dynamic provisioning and capacity planning.

Learning-based approaches: Other systems are predictive: NIMO [Shivam et al. 2006] uses statistical learning with active sampling to model resource usage and predict completion time for varying resource assignments. NIMO focuses on the completion time of long-running applications, and does not address model composition such as done by Modellus.

The work of Zhang *et al.* [Zhang et al. 2007] is closely related to ours; they use regression to derive service times for queuing models of web applications, but require manual classification of events and do not compose models over multiple systems. Other work learns classifiers from monitored data: in Cohen [Cohen et al. 2004] tree-augmented Bayesian Networks are used to predict SLA violations, and similarly in Chen [Chen et al. 2006] a K-nearest-neighbor algorithm is used to for provisioning to meet SLAs. Independent Component Analysis (ICA) has also been used to categorize request types based on service demand by Sharma *et al.* [Sharma et al. 2008].

Application models have been employed for a variety of reasons. Performance models can be used to guide provisioning engines that determine how many resources to allocate to an application [Bennani and Menasce 2005]. The R-Cappricio system uses regression based models for capacity planning as well as anomaly detec-

tion [Zhang et al. 2007]. SelfTalk/Dena [Ghanbari et al. 2010] is a query language and runtime system capable of describing interconnected application components and then evaluating queries about expected system behavior. In Modellus, we automate the process of building and updating application performance models, and demonstrate how these can be used to answer what if questions about resource consumption under different workloads and hardware configurations.

10. CONCLUSIONS

This paper argued that the rising complexity of Internet data centers has made manual modeling of application behavior difficult and proposed Modellus, a system to automatically model the resource usage and workload dependencies between web applications using statistical methods from data mining and machine learning. We proposed a number of enhancements to ensure that these statistical methods are practical in large distributed systems. We implemented a prototype of Modellus and deployed on a Linux data center testbed. Our experimental results show the ability of this system to learn models and make predictions across multiple systems in a data center, with accuracies in prediction of CPU utilization on the order of 95% in many cases; in addition, benchmarks show the overhead of our monitoring system to be low enough to allow deployment on heavily loaded servers. We further demonstrate the usefulness of Modellus in two case studies that illustrate the importance of adapting models after hardware upgrades and using model predictions to answer "what if" questions about changing workload conditions.

REFERENCES

Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. 2003. Performance debugging for distributed systems of black boxes. In *SOSP*.

Arlitt, M. and Jin, T. 1999. Workload Characterization, 1998 World Cup Web Site. Tech. Rep. HPL-1999-35R1, HP Labs.

Ascher, D., Dubois, P., Hinsen, K., Hugunin, J., Oliphant, T., et al. 2001. Numerical Python. *Available via the World Wide Web at* `http://www.numpy.org`.

Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. 2004. Using Magpie for request extraction and workload modeling. In *OSDI*. 259–272.

Bennani, M. N. and Menasce, D. A. 2005. Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*. IEEE Computer Society, Washington, DC, USA, 229–240.

Cain, H. W., Rajwar, R., Marden, M., and Lipasti, M. H. 2001. An Architectural Evaluation of Java TPC-W. In *HPCA*.

Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., and Zwaenepoel, W. 2003. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *Intl. Middleware Conf.*

Chase, J., Anderson, D., Thakar, P., Vahdat, A., and Doyle, R. 2001. Managing energy and server resources in hosting centers. *Operating Systems Review 35,* 5, 103–116.

Chen, J., Soundararajan, G., and Amza, C. 2006. Autonomic provisioning of backend databases in dynamic content web servers. In *ICAC*. 231–242.

Cohen, I., Chase, J. S., Goldszmidt, M., et al. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*. 231–244.

Doyle, R. P., Chase, J. S., Asad, O. M., et al. 2003. Model-Based Resource Provisioning in a Web Service Utility. In *USITS*.

Draper, N. R. and Smith, H. 1998. *Applied Regression Analysis*. John Wiley & Sons.

Foster, D. P. and George, E. I. 1994. The Risk Inflation Criterion for Multiple Regression. *The Annals of Statistics 22,* 4, 1947–1975.

Ghanbari, S., Soundararajan, G., and Amza, C. 2010. A query language and runtime tool for evaluating behavior of multi-tier servers. In *SIGMETRICS*. 131–142.

Jiang, G., Chen, H., and Yoshihira, K. 2006. Discovering Likely Invariants of Distributed Transaction Systems for Autonomic System Management. In *ICAC*. 199–208.

jwebunit 2007. JWebUnit. `http://jwebunit.sourceforge.net`.

Kamra, A., Misra, V., and Nahum, E. 2004. Yaksha: A Controller for Managing the Performance of 3-Tiered Websites. In *Proceedings of the 12th IWQoS*.

M. A. Efroymson, M. 1960. Multiple regression analysis. *Mathematical Methods for Digital Computers 1*, 191–203.

Menascé, D., Almeida, V., Riedi, R., Ribeiro, F., Fonseca, R., and Meira, Jr., W. 2000. In search of invariants for e-business workloads. In *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*. ACM, New York, NY, USA, 56–65.

Mosberger, D. and Jin, T. 1998. httperf – A Tool for Measuring Web Server Performance. In *Proceedings of the SIGMETRICS Workshop on Internet Server Performance*.

openforbiz 2007. The Apache "Open For Business" project. `http://ofbiz.apache.org`.

Parekh, S., Gandhi, N., Hellerstein, J., Tilbury, D., Jayram, T. S., and Bigus, J. 2002. Using control theory to achieve service level objectives in performance management. *Real-Time Systems 23,* 1, 127–141.

Sharma, A. B., Bhagwan, R., Choudhury, M., Golubchik, L., Govindan, R., and Voelker, G. M. 2008. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev. 36,* 2, 16–25.

Shivam, P., Babu, S., and Chase, J. 2006. Learning Application Models for Utility Resource Planning. In *ICAC*. 255–264.

Smith, W. TPC-W: Benchmarking An Ecommerce Solution. `http://www.tpc.org/information/other/techarticles.asp`.

Stewart, C. and Shen, K. 2005. Performance Modeling and System Management for Multi-component Online Services. In *NSDI*.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. 2005. An Analytical Model for Multi-tier Internet Services and Its Applications. In *SIGMETRICS*.

Zhang, Q., Cherkasova, L., Mathews, G., Greene, W., and Smirni, E. 2007. R-capriccio: a capacity planning and anomaly detection tool for enterprise services with live workloads. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 244–265.

Zhang, Q., Cherkasova, L., and Smirni, E. 2007. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *ICAC*.