Online Capacity Identification of Multi-tier Websites Using Hardware Performance Counters

Jia Rao, *Student Member, IEEE* and Cheng-Zhong Xu, *Senior Member, IEEE*

Abstract—Understanding server capacity is crucial to system capacity planning, configuration and QoS-aware resource management. Conventional stress testing approaches measure server capacity offline in terms of application-level performance metrics like response time and throughput. They are limited in measurement accuracy and timeliness. In a multi-tier website, resource bottleneck often shifts between tiers as client access pattern changes. This makes the problem of online capacity measurement even more challenge. This paper presents an online measurement approach based on low-level hardware performance metrics such as instructions execution rate and cache access behavior. Such metrics together define a system internal running state. The measurement approach uses machine learning techniques to infer application-level performance at each tier from a set of selected hardware performance counters. A coordinated predictor is induced over individual tier-wide models to make global system performance prediction and identify the bottleneck when the system becomes overloaded. Experiments were conducted on a two-tier Tomcat/MySQL-configured website using TPC-W benchmarks. Experimental results demonstrated that this approach was able to achieve an overload prediction accuracy of higher than 90% for *a priori* known input traffic mix and over 85% accuracy even for traffic causing frequent bottleneck shifting. It costs less than 0.5% runtime overhead for data collection and no more than 50 ms for each online decision-making.

Index Terms—Multi-tier website, machine learning, hardware performance counter.

1 Introduction

Understanding of server capacity is crucial to server capacity planning, configuration and QoS-aware resource management. It is known that a server can be run in one of the three states: underloaded, saturated, and overloaded. When the server is underloaded, its throughput grows with the increase of input traffic rate until a saturation point is reached. The saturated throughput may not stay unchanged when the input rate continues to increase. It may drop sharply due to resource contention and algorithmic overhead for load management [15]. Knowledge about the server capacity could help measurement-based admission controller in the front-end to regulate the input traffic rate so as to prevent the server from running in an overloaded state. Moreover, for input traffic of multi-class requests, server capacity information can also be used by a back-end scheduler to calculate the portion of the capacity to be allocated to each class for service differentiation and QoS provisioning [12], [31], [35].

An industry standard approach to server capacity measurement is offline stress-testing using benchmarks [6]. It views the server as a blackbox and observes the change of server performance in terms of application-level metrics like response time and throughput with the increase of input load. It approximates server capacity to be the saturated throughput or the system throughput when the observed response time starts to rise abruptly. In [2], the authors proposed an S-client approach to generate excessive workload efficiently for measuring the capacity of web servers. In [6], the authors focused on the construction of benchmarks for measuring the basic capacities of streaming servers. In [12], the authors suggested to measure the capacity of a server in terms of execution units. They used a method of incremental steps [15] to find out the server capacity by incrementally probing the saturated throughput.

These offline profiling approaches are limited to systems with static resource configuration. They cannot be applied to today's highly reliable and available servers that are capable of dynamic resource configuration through techniques like hot-swapping and dynamic frequency/voltage scaling [39].

Application-level performance metrics like response time and throughput are good intuitive measures. However, they have limitations in accuracy and timeliness when they are used for *fine-grained* QoS-aware resource management. It is known that requests of an e-commerce transaction have very different processing times and the times also tend to change with server load condition. As a result, request-specific response time becomes an ill-defined performance measure in stress-testing of server capacity. There were studies on the use of mean response time to characterize the server load change in statistics. Welsh and Culler showed that 90th-95th percentile response time represented the shape of response time curve more accurately, in comparison with average or maximum time [33]. However, setting a request-specific

• *The authors are with the Department of Electrical and Computer Engineering, Wayne State University, 5050 Anthony Wayne Drive, Detroit, MI 48202. E-mail: {jrao,czxu}@wayne.edu*

response time value for admission control remains non-trivial. In [18], Mogul presented a case that a misconfiguration of the response time threshold could possibly cause the system to enter a live-lock state. In practice, the threshold is often set conservatively. For example, Blanquer *et al.* [16] set a threshold to a half of the target response time of the most restrictive requests for the admission controller to regulate the incoming traffic rate. Such a conservative estimation of the server capacity by setting a low threshold value is equivalent to resource over-provisioning.

Besides the limitation in accuracy, server processing capability measured in application-level response time may not be a timely measure for fine-grained resource management. The observed response time of past requests may mislead the front-end admission controller to wrong decisions because of the presence of long deadtime of requests in a multi-tier website. That is, there is a non-negligible delay from the time a request is admitted to the time its response is observed. Processing tasks of the request could be queueing blocked in many places, particularly when a system is heavily loaded.

In a multi-tier website, processing of a request often involves multiple system components in different tiers. Saturation of the system in the processing of one type of requests may not necessarily mean it cannot handle other requests. Bottleneck may also shift dynamically. Response-time based server capacity measurement provides little insight into constrained resources.

These motivated us to develop an online capacity measurement approach, based on low-level system running statistics. Modern processors are all equipped with a set of performance monitoring registers to record detailed hardware-level system information during the execution time of each application. The information includes a large group of parameters like instruction mix, rate of execution, memory access behaviors and branch prediction accuracy [27]. Together, they define a system internal running state and reflect aggregated effects of the requests in concurrent execution. Questions are how to define a small group of relevant parameters to characterize the system load condition accurately, how to map them onto a high level system overload/underload status, and more importantly how to identify bottleneck resource when the system becomes overloaded.

In this paper, we present effective and efficient solutions to these questions. Specifically, we develop models involving a small set of hardware performance counter metrics to characterize the system state of each server. We further develop a two-tier coordinated real-time classification approach to infer system overload/underload state and identify resource bottleneck. We evaluated the approach in a two-tier Tomcast/MySQL website using TPC-W benchmark. Experimental results demonstrated its effectiveness and efficiency.

The rest of the paper is organized as follows. Section 2 gives examples to show limitations of response time-based approaches for server capacity measurement. Section 3 and Section 4 present the details of the hardware performance counters-based capacity measuring approach. Section 5 and Section 6 give the evaluation methodology and experimental results. Related work is presented in Section 7. Section 8 concludes this paper.

2 Limitations of Response Time Metric

Response time is an application-level intuitive metric for understanding server capacity and user-experienced service quality. However, it is insufficient for the design of request-specific admission control and fine-grained QoS-aware resource management, particularly in multi-tier websites. In this section, we give a brief overview of the dynamics of websites and show limitations of the metric.

2.1 Dynamics of a Multi-tier Website

In its simplest form, a website consists of a web server in the front-end, a database server in the back-end, and an application server in the middle to implement the application logic. A configuration example is a Tomcat servlet engine [30] for combined web and application servers and a MySQL [19] for the database server.

Processing of transactional requests often goes through four phases: web protocol parsing, application servlet execution, database connection establishment, and SQL query processing. They are synchronous in the sense that one phase is not finished without the completion of the subsequent phases.

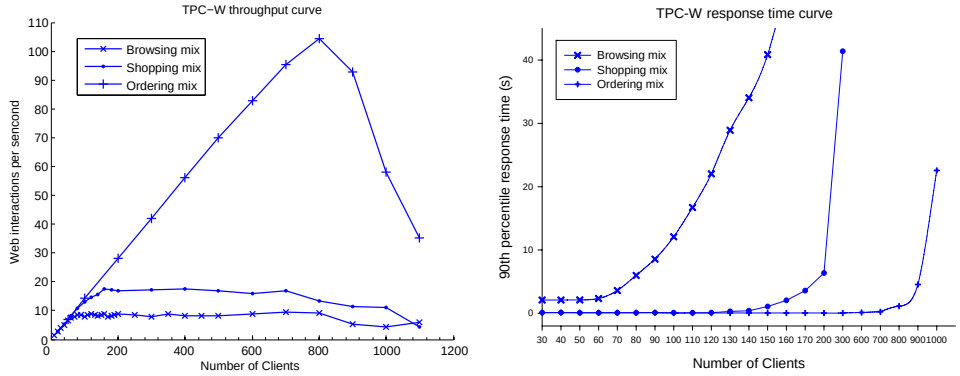
Servers often deploy a multi-threaded processing model to process multiple requests simultaneously. In Tomcat, concurrency is set by a group of system configuration parameters regarding of the maximum number of threads to be run in parallel. Requests will be queueing blocked, if there are no available threads. Requests in processing could also be blocked, waiting for connections to the database server.

In the database server, SQL queries generated by different servlets are not necessarily executed in the same order as they arrive. Because they are executed as a batch of *interleaved* queries, requests in processing may be even been blocked inside the database server.

2.2 Website Capacity Identification

In general, a transaction processing system has a saturation point (upper bound) of the throughput the system could produce, as its load increases. After the "upper bound" is reached, the system throughput will either drop because of thrashing or maintain at a saturation level, but with decreased service quality [15]. In order to fully utilize the system resource, admission control must be applied before the system reaches this saturation bound.

We conducted experiments in a typical Tomcat/MySQL website setting, using TPC-W benchmark [29]; Please see Section 5 for details of



(a) Throughput (b) Response time

Fig. 1. Performance of a website in different TPC-W traffic mixes.

the test-bed and TPC-W benchmark. TPC-W defines three input traffic mixes: browsing, shopping and ordering, with different request profiles. Table 1 summarizes the profile of each mix.

TABLE 1
Request composition in TPC-W.

	Browsing	Shopping	Ordering
Browsing request	95%	80%	50%
Ordering request	5%	20%	50%

Figure 1(a) shows the throughput curve of these three mixes, in terms of web interactions per second (WIPS), as the number of concurrent clients increases. It is expected that the website would have different processing capacities in different input traffic mixes. With an input of ordering mix, the system throughput drops sharply after it goes beyond the system capacity. In contrast, the throughput stabilizes for the browsing and shopping mixes. This is because browsing related requests tend to put more pressure on the back-end database server, while ordering requests more likely cause CPU overload on the front-end application server. The figure also demonstrates that the bottleneck tends to shift with the change of input patterns.

Figure 1(b) shows the 90th percentile response time under different input traffic patterns and different load conditions. The figure shows that for inputs in a browsing mix, the response time increases sharply when the input load goes beyond the system capacity, although the throughput remains unchanged.

Measurement-based admission control needs an online system performance metric to represent the current system load condition. Request response time is a widely used intuitive system load indicator and the metric is easy to monitor online. However, response time-based approach has limitations:

1) It is hard to find "good" thresholds that differentiate "underload" and "overload" system states, because different requests have a large variety of response times and their execution times varies in different load conditions.
2) Response time of a request can not be measured until the request is completed. It reflects the system status in past time windows, rather than the system's current load condition. Figure 2 shows the change of response time in an experiment, in which we injected load spikes at time of 1800th and 5400th second. From the microscopic view of the plot, we can see that the spike at the 5400th second can not be detected in 10 seconds based on response time.
3) Request processing involves many resources in different tiers. The response time metric provides little insight into the bottleneck tier or constrained resources. Since different types of requests put pressure on different tiers of the system, it is possible that, under heavy load, the system's resource bottleneck shifts from one tier to another when the input traffic pattern changes. Using request response time as a system load indicator masks the underlying system load dynamics, and hinders the efficiency of admission control.

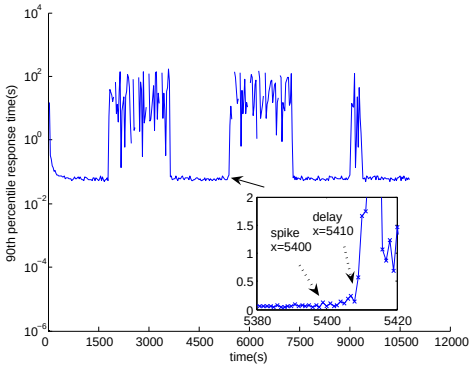


Fig. 2. Response times in transient spike

Response time measures the system processing capacity based on application-level observation. An alternative is to measure the capacity in lower level system performance metrics.

3 Lower Level System Performance Metrics

A system provides a rich set of performance metrics in both hardware and Operating System (OS) levels. Their statistics represent the internal performance states at run-time. Each internal state contributes to a high-level "underload" or "overload" state. Identifying the system state using lower level performance metrics poses three challenges: (1) What metrics should be used to characterize the high level performance state; (2) How to infer high level performance states such as "underload" and "overload" from the statistics of the metrics; (3) How to identify the bottleneck tier in a multi-tier website, based on the runtime statistics of each tier. We will discuss the first two challenges in this section and leave the third in Section 4.

3.1 Revisit of the Concept of Capacity

System capacity often refers to the maximum amount of work that can be completed during a certain period of time. We refer to the amount of completed work as *yield* and the amount of resource consumed during the time as *cost*. An overloaded system means that its cost keeps increasing but with stagnated or compromised yield. We define a metric of *productivity index* (PI) as the ratio of yield to cost and use it to measure the system processing capability:

$$PI = \frac{Yield}{Cost}.$$

This is a generic concept and *yield* and *cost* can be defined at different system abstractions or under different workload scenarios. By defining *yield* to be the number of requests and *cost* as the wall time, *PI* becomes equivalent to application-level throughput. Today's modern processors are all equipped a number of Hardware Performance Counters (HPC) that provide a rich source of statistical information on application execution. This information includes but not limited to memory bus access pattern, cache reference and pipeline execution information. By defining *yield* as instructions-per-cycle and *cost* the stalled CPU cycles or cache miss rate, the *PI* metric reflects the instruction-level productivity.

The concept of productivity can also be defined at OS level. We argue that OS level metrics like CPU utilization may not be a good metric for system performance. The following example shows that CPU utilization fails to reflect application level performance. Consider the following two code segments on a 2.0GHz Pentium 4 machine with 512 KB L2 cache and 512 MB memory.

```
#define NUM_ITERS 10000
double matrix[65536*8];
int stride=8;

void Sequential(void) {
    for(line = 0; line < 65536*8; line += stride)
        for(offset = 0; offset < stride; offset++)
            for(i = 0; i < NUM_ITERS; i++) {
                temp += matrix[line+offset];
}
```

TABLE 2
Different low-level performance under different workloads.

Workload	IPC	L2 (%)	CPU%	User%	Time (s)
Sequential	0.31	0.03	100%	99.7%	41.4
Stride	0.13	0.92	100%	99.5%	230.5

```
void Stride(void) {
  for(i = 0; i < NUM_ITERS; i++)
    for(offset = 0; offset < stride; offset++)
      for(line = 0; line < 65536*8; line += stride)
        temp += matrix[line+offset];
}
```

`Sequential()` accesses consecutive memory locations while `Stride()` visits memory in strides. Table 2 shows the execution time for each segment and the statistics reported by OS metrics (CPU% and User%) and hardware counters level metrics (IPC (Instruction Per Cycle) and L2 cache miss rate). The OS metrics shows no performance difference between the two programs. In comparison, hardware-level metrics, IPC and L2 miss rate, reflect application-level performance more accurately. More studies about the selection and effectiveness of hardware-level PI will be given in Section 6. In the following, we use hardware-level PI to measure the system capacity.

For online identification, the single PI metric is not enough to identify system state because any change of PI can be either due to the system capacity or the input load change. For example, a decrease in the incoming workload can lead to a smaller value of PI. But a decrease in PI with sustained or increasing workload can only be due to a system overload. During offline classification, we keep increasing client traffic and label the system state as "underload" until PI begins to drop, from which we label the system state as "overload". During the above process, we take snapshots of hardware counter metrics and develop an online model to correlate them to each high-level system state in a machine learning approach. The model makes it possible for online prediction of system state, for a given set of hardware statistics.

In the following, we give the details of the modeling and learning approach.

3.2 Definition of Performance Synopsis

We define a *performance synopsis* data structure to represent the correlation between a set of lower-level performance metrics and their corresponding high-level system states. Formally, let $U = \{A_1, ..., A_n\}$ be a set of attribute variables, in which A_i can be any individual hardware counter performance metric such as number of L2 cache misses. Adding a class variable C into U, we have $U^* = \{A_1, ..., A_n, C\}$. The class variable can be any type of system state. In capacity identification, it is a binary variable, taking value of 1 ("overload") or 0 ("underload"). Each attribute A_i can be instantiated by assigning a measured value a_i during each sampling interval. Instantiating all variables in U^* results in an instance u^*.

For a training set $D = \{u_1^*, ..., u_N^*\}$ with N instances, we build a synopsis to capture the relationship between the group of attributes $A_1, \ldots, A_n$ and class C. We denote it by $SYN(\{A_1, ..., A_n\}, C)$.

3.2.1 Attribute Selection

A system often contains a large number of low-level performance metrics that can be measured online. For example, Linux provides over 100 OS-level metrics; Intel CPU contains hardware performance counters for more than 20 parameters. Including too many attributes in a synopsis could be time complexity prohibitive.

Furthermore, irrelevant attributes in a synopsis may even cause a loss of prediction accuracy. It is desirable to select most relevant attributes in the training set to reduce computing complexity and avoid noises. We apply the concept of *information gain* in information theory to evaluate the relevance between each attribute and the class variable. Information gain is the reduction of entropy about the classification of a test class based on the observation of a particular attribute. For an attribute A_i, its information gain in any class of C can be calculated as follows:

$$\begin{aligned} G(C, A_i) &= H(C) - H(C|A_i) \\ &= -\sum_{c \in C} Prob(c) \log_2 Prob(c) + \\ &\quad \sum_{a \in A_i} \sum_{c \in C} Prob(a, c) \log_2 Prob(c|a), \end{aligned}$$

where $H(C)$ is the entropy of class variable and $H(C|A_i)$ is the conditional entropy of class variable given the attribute variable A_i. Attribute selection is an iterative process, in which the most relevant attribute is added to the attribute set each time only if its addition improves synopsis accuracy. The overall accuracy of a synopsis is evaluated by a *10-fold cross validation* method [17].

3.2.2 Construction of Synopsis and Prediction

A synopsis builder is essentially a machine learning algorithm that generates a synopsis from a training set. In the following, we first present the overview of four representative algorithms that are to be used for synopsis construction. Impact of the algorithms on the prediction accuracy will be discussed in Section 6.2.

Linear regression (LR): *Linear regression* is a regression method that models the linear relationship between a dependent variable C, independent variables $A_1, ..., A_n$, and a random term ϵ. To build the LR model is to estimated the coefficients of each A_i and ϵ that best fit in the training set D.

Naive bayes (Naive): Bayesian network is a powerful tool to represent joint probability distributions over a set of random variables. It is often made up of two components: a directed acyclic graph B_s and a set of conditional probability tables B_p. *Naive bayes* is one of the most effective bayesian classifiers. It makes a strong independence assumption: all attributes A_i are conditionally independent given the value of class C.

Tree augmented naive bayes (TAN): Unlike *Naive bayes*, *TAN* allows the generated B_s to represent correlations between attributes $A_1, ..., A_n$ [14]. The correlations between attributes are captured by imposing a tree structure on the naive Bayesian structure.

Support vector machine (SVM): *SVM* performs classification by constructing an $n-1$ dimensional hyperplane that optimally separates the data into two categories. Different from other classifiers, *SVM* is able to find out the maximum separation between the two classes.

For a synopsis trained from a set D, we consider a set of testing instances p^*, each with a similar structure with the instances in D. For each instance p^*, the same training algorithm of the synopsis is re-applied to generate a prediction C' with respect to the class variable C of the instance. We represent the prediction algorithm as function $Predict()$. That is, $C' = Predict(SYN, p^*)$. If $C' = C$, the prediction is correct, otherwise incorrect.

4 Two-Level Coordinated Website Capacity Identification

The preceding section shows the modeling and learning processes to correlate lower-level performance metrics to high level system state in a single server. In a multi-tier website, each server has a PI reference for "underload" and "overload" states. Because the bottleneck may shift between tiers, there are two challenges in the website capacity identification: (1) which PI reference should be used to identify the entire system state offline; (2) which synopsis should be used to predict system state online? We give an overview of the two issues and a solution to the first issue in Section 4.1. The rest of this section is about our coordinated learning approach to the second challenge.

4.1 Website Capacity Identification Framework

It is expected that the metrics from a bottleneck tier have the strongest correlation to high-level performance. We select the corresponding PI reference as a measure of the website capacity. We define a correlation index $Corr(PI, r)$, in a way similar to [27], between the PI and high level performance metric r (e.g. throughput) over a time period t:

$$Corr(PI, r) = \frac{Cov(PI, r)}{\sigma_{PI} \cdot \sigma_r} = \frac{\sum_{j=1}^{q} (PI_j - \overline{PI})(r_j - \overline{r})}{q \cdot \sigma_{PI} \cdot \sigma_r}$$

where q is the number of (PI, r) pairs sampled during the time t. The correlation index between PI and r is calculated using their means $\overline{PI}$, $\overline{r}$ and standard deviations σ_{PI}, σ_r in the q samples. The PI with the largest *Corr(PI,r)* value will be selected as the measure of the entire system capacity.

Internet traffic contains many different types of requests (e.g. browsing and ordering) and their mix may

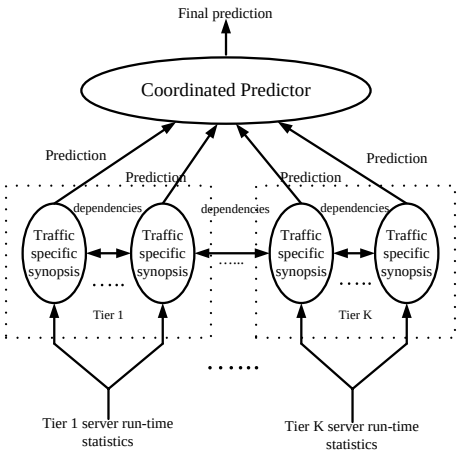


Fig. 3. The two-level coordinated prediction framework.

change with time. Variations of the request composition would affect the performance of a multi-tier website and may even lead to bottleneck shift between tiers. Recall that synopses on each tier are constructed based on specific traffic patterns. Intuitively, a synopsis due to a specific workload is unlikely to be accurate for traffic whose bottleneck lies in another tier. We build synopses on each tier for representative workloads. The workload selection will be discussed in Section 5.

For a given set of runtime statistics under a traffic pattern, each workload-specific synopsis will be used to make a prediction. To make a global system state prediction, we propose a two-level coordinated learning scheme which dynamically selects the best synopsis for the given traffic pattern. Following are the details of the scheme.

The capacity measurement framework employs a two-level hierarchical architecture, a group of *performance synopses* in the bottom and a *coordinated predictor* at the top. Figure 3 shows the structure. The two-level coordinated prediction architecture takes runtime statistics on each tier machine as inputs. Based on these inputs, individual synopses generate their predictions in regard to system high-level states. Final state prediction will be made in the coordinated predictor by combining these individual predictions.

Although a synopsis is specific to tiers and traffic patterns, the relationship between low level metrics and system state defined by the synopsis remains valid in the presence of workload changes, as long as the bottleneck remains in the same tier. When the workload changes make the bottleneck shifting to another tier, a new synopsis should be selected. The coordinated predictor selects the best synopses dynamically by studying the spatial (synopsis-wise) and temporal patterns among predictions of individual synopses.

Note that a synopsis with less accuracy with regard to certain workloads does not mean that it provides no information for the global system state. Given a workload, predictions from synopses have spatial patterns. For example, synopses might make consistent predictions for certain workloads although the predictions are not correct. Many performance problems manifest not as a single major shift in system behavior but rather as a series of subtle changes. In addition to spatial prediction patterns, temporal patterns among consecutive predictions are also observed in the coordinated predictor.

4.2 Coordinated Two-level Predictor

The coordinated predictor is designed as a two-level predictor to capture spatial and temporal patterns in synopses predictions. The coordinated predictor is similar in structure to a branch predictor in superscalar processors [36]. Figure 4 shows the structure of the two-level predictor.

The first level is a Global Pattern Table (GPT) which represents synopsis-wise patterns. Each entry in GPT is a Global Pattern Vector (GPV). A GPV is an m-bit vector (m is the number of synopses), and each bit R_i represents the prediction result of corresponding synopsis during a sampling interval τ. That is, $R_i = Predict(SYN_i, p^*_\tau)$. The GPT enumerates all the possible patterns of GPV, thus it has 2^m entries.

The second level are Local History Tables (LHTs) that record the last h prediction results of the specific pattern in GPT. For each of these 2^m patterns, there is a corresponding LHT in the second level which contains the occurrences of different temporal patterns. Each entry of a LHT is referred to as Local History Bits (LHB), denoted by H_c. It is used for making the coordinated prediction. The coordinated prediction is $C'' = \lambda(H_c)$, where λ is the prediction decision function. The length of LHB determines the size of the LHT table. For example, if LHB contains v bits, which records the last v prediction results ($h = v$), the corresponding LHT has 2^v entries.

Along with the two-level predictor for the system state prediction, we also include a simple bottleneck predictor in the coordinated predictor. The bottleneck predictor is implemented by adding an extra Bottleneck Pattern Table (BPT) to the second level. Each entry in the BPT is a Bottleneck Vector (BV) which is indexed by GPV, as well. The bottleneck prediction is $B' = \lambda_b(b_K, ..., b_1)$, where λ_b is the bottleneck decision function.

4.3 Training and Prediction

To exploit the spatial and temporal prediction patterns, the coordinated predictor needs to be trained. The training process is to determine the values of LHB H_c in each LHT. Initially, all H_c are set to 0. The values of H_c are learned from all the instances from which each individual synopsis is built. The training process includes the following steps:

1) Given an instance u^*_i, generate predictions from each synopsis. Combining these predictions forms a GPV. Then the GPV, denoted as a binary sequence of $< R_{m-1}...R_0 >$, is used to find the corresponding LHT.

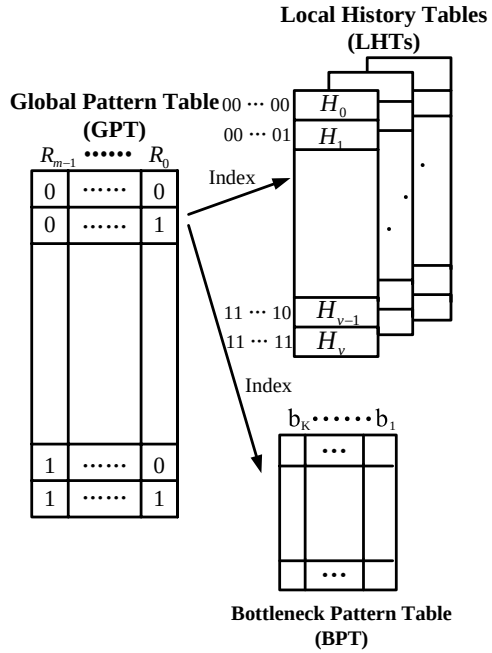


Fig. 4. The structure of the two-level predictor.

2) In the LHT, the local history bits H_c is indexed by last h prediction history. Update the value of the corresponding H_c for each instance u_i^* as follows: If the value of the class variable in u_i^* equals to 1, increase H_c by 1, otherwise decrement by 1.

The training of the bottleneck predictor is similar except that instead of learning H_c values, the values for each $b_K, ..., b_1$ should be trained. For bottleneck identification, we manually augment a training instance u_i^* with information about the bottleneck tier. For example, if the class variable in instance u_i^* has a value of 1 and tier i is the bottleneck for current workload, update b_i as $b_i = b_i + 1$, otherwise $b_i = b_i - 1$.

The coordinated predictor is used to make online global system state predictions as well as bottleneck tier predictions. The bottleneck predictor is only invoked when the system state prediction is 1. For system state prediction, the predictor finds the corresponding H_c according to the current value of GPV. During each sampling interval, the coordinated prediction is made using the prediction decision function $C'' = \lambda(H_c)$, where

$$\lambda(H_c) = \begin{cases} 1 & \text{if } H_c > \delta; \\ \phi(H_c) & \text{if } -\delta \leq H_c \leq \delta; \\ 0 & \text{if } H_c < -\delta, \end{cases}$$

where δ is a threshold for H_c which describes the confidence in H_c making a prediction.

A large δ prevents the predictor from making a prediction unless current spatial and temporal prediction patterns occur a large number of times in previous workloads. Setting δ to a small value relaxes the restriction. For any $\delta > 0$ there exists an interval $[-\delta, \delta]$, in which the predictor is not sure what prediction to make.

We develop two heuristic schemes to select a prediction: an optimistic scheme and a pessimistic scheme. These two schemes are different in function $\phi(H_c)$. The optimistic scheme always makes a prediction of 0 (underload) when $H_c \in [-\delta, \delta]$, while the pessimistic always predicts as 1 (overload).

For the bottleneck predictor, whenever the state predictor predicts as 1, it outputs bottleneck information. The bottleneck decision function is $\lambda_b(b_K, ..., b_1) = \arg\max_i(b_i)$. That is to choose the tier having the largest value in its corresponding bit in $b_K, ..., b_1$ as the bottleneck.

4.4 An Example

We use an example to illustrate the two-level predictor. Suppose the website has two tiers: application (AP) and database (DB) tiers, and it takes two different types of input: browsing (B) and ordering (O). There are altogether four ($m = 4$) synopses: AP-O, AP-B, DB-O and DB-B in the GPT, representing synopses for different inputs on different tiers. An coded GPV like (0101) means the predictions of the four synopses are "underload", "overload", "underload", and "overload", respectively. Assume that LHB records the last three overall system state predictions (*i.e.*, h=3), and they are "underload", "overload", "overload", respectively. The corresponding entry H_c in (0101)-indexed LHT is in the address of 110. Suppose the threshold δ is initially set to 5. For any H_c larger than δ, the predictor will forecast an "overload" state.

5 Evaluation Methodology

To evaluate the two-level coordinated website capacity measurement, we built a test-bed of multi-tier e-commerce website according to the classic TPC-W benchmark. In our test-bed, the multi-tier website consists of two tiers: front-end application server and back-end database server. Representative workloads conforming TPC-W specifications were thrown to the test-bed. During execution, hardware counter level runtime statistics were collected. For comparison, OS level metrics were also reported.

5.1 TPC-W and Workload Selection

TPC-W is a transactional web e-commerce benchmark. Its specification defines 14 different types of requests for an online bookstore service. In our test-bed, we deployed the free Java implementation of TPC-W benchmark from Rice University [25]. TPC-W defines three traffic mixes: Browsing, Ordering and Shopping, as shown in Table 1. It classifies web interactions as either Browse or Order depending on whether they involve browsing and searching on the site or whether they play an explicit role in the ordering process.

The primary TPC-W performance metric WIPS is based on the shopping mix, which is the most common workload in an e-commerce website. TPC-W also

considers the extreme cases in which the workload is either mostly composed of browsing requests or ordering requests. Experimented with our test-bed, browsing mix is found to put more pressure on database than on application server. For ordering mix, front-end becomes the bottleneck.

We assume that the incoming traffic to a multi-tier website ranges within the above two extreme workloads: Browsing and Ordering. As the characteristic of the workload changes, the bottleneck tier can be either the back-end or the front-end and bottleneck shifting exists. Thus we selected the browsing and ordering mix as the representative workloads for training synopses and the coordinated predictor. The workloads are generated using the Remote Browser Emulator (RBE) shipped with the Rice TPC-W implementation. We modified the RBE to generate the workload needed in training and testing sets. The number of concurrent clients is controlled by the number of Emulated Browsers (EBs).

5.2 Training and testing sets

In real scenarios, internet traffic can be either steady or bursty. To generate realistic workloads, we compose the workload generating the training runtime statistics as two parts:

1) **Ramp-up workload**. In ramp-up workloads, we gradually increased concurrent client sessions. Because the multi-tier website can serve different numbers of concurrent browsing clients and ordering clients, we increased the workload in different rates. For browsing mix, we started with 20 concurrent clients and incremented 20 clients every 10 minutes up to a limit of 600 concurrent sessions. For ordering mix, we started with 50 clients and added 50 more clients each time until a total of 1500 sessions. For each browsing and ordering mix, we ran the experiments for five hours.
2) **Spike workload**. Spike workload refers to occasional extreme traffic burst. We set the baseline traffic to 80 concurrent shopping clients for both browsing and ordering spikes. Every 30 minutes, we threw a spike workload to the baseline and kept the spike for 10 minutes. Each browsing spike contains 200 browsing clients and each ordering spike has 800 ordering clients. Each experiment also lasted for five hours.

We collected the hardware counter level and OS level runtime statistics on each tier every second. The average statistics over a 30 second interval combined with its corresponding high-level state formed an instance in a training set. The training sets were used to build synopses and tune the coordinated predictor.

Note that although all synopses were trained from the two extreme browsing and ordering mixes, we will show the coordinated predicator works well for traffic of unknown mixes as well. We designed the testing sets as four parts: browsing mix, ordering mix, interleaved mix, and unknown workload mix. The interleaved mix refers to a workload that continues to switch between browsing mix and ordering mix. For the unknown mix, we change the transition probability in RBE to generate workload different from either browsing or ordering mix.

5.3 Evaluation Metrics

The key measure of the effectiveness of coordinated predictor is its prediction accuracy in testing sets. Absolute prediction accuracy is the ratio of the number of correctly classified instances over the total number of instances. It depends on the ratio of each class. Instead, we use the Balanced Accuracy (BA) as the metric to evaluate the prediction accuracy. Formally, BA can be defined as:

$$BA = \frac{Prob(C'' = 0|C = 0) + Prob(C'' = 1|C = 1)}{2},$$

where C is the actual value of the class and C'' is the predicted value. Measured by BA, a good predictor should perform well in both classes, independent of the composition of testing sets. To evaluate the prediction accuracy of the two-level predictor, we designed the testing sets mentioned above. We injected approximately 40% to 50% overload instances in each testing set. Thus, any naive method is bounded by a prediction accuracy of 60% at most.

5.4 Experiment Settings

We followed the organization of dynamic websites in [1] to build our test-bed except that only one client machine was used to emulate the concurrent clients. The client machine featured a dual AMD 2.10 GHz CPU configuration and 2GB memory. We ensure that the client machine is not the bottleneck by comparing the one client machine experiment with a multiple clients setting. In both settings, the client machine(s) were lightly loaded and the TPC-W performance differences are within 1%. The front-end application server and the back-end database server were configured with Pentium 4 2.0 GHz CPU, 512 MB RAM and Pentium D 2.80 GHz CPU, 1 GB RAM respectively. The CPUs in the servers are based on Intel *NetBurst* architecture and without Hyperthreading technology. All the devices were interconnected by a fast Ethernet network.

The machines ran Fedora Core 6 Linux with kernel 2.6.18. We used Apache Tomcat version 5.5.20 as the application server. For the database server, MySQL standard version 5.0.27 was used. We used `Sysstat` version 7.0.3 to collect 64 OS level metrics. Hardware counter level metrics were recorded by a kernel patch `PerfCtr` [22]. There are software packages, such as OProfile[20], PAPI [21], and PerfSuite[23], which can be used to monitor hardware counter level runtime metrics. Because of their overhead concerns, we wrote a lightweight tool to read hardware counter metrics in all physical CPUs using the global mode in `PerfCtr`.

TABLE 3
Collected hardware counter metrics.

Performance counter event	Description
X87_FP_RETIRED	retired uops
X87_FP_UOP	x87 floating point uops
L2_REFERENCE	L2 cache access
L2_MISS	L2 cache miss
CPU_STALL	CPU stalled cycles on any resource
INS_RETIRED	retired instructions
ITLB_REFERENCE	translation lookaside buffer access
ITLB_MISS	translation lookaside buffer miss
RETIRED_MISPRED_BRANCH	retired mispredicted branch

Although the global mode in `Perfctr` only updates performance counter values at regular intervals which may not be accurate enough for small code regions, server programs always run for a long time and management operations are invoked in the granularity of several seconds or minutes. Event counter maintenance in hardware requires no runtime overhead [27] and we limited our tool to minimum functionalities that just initialize and read hardware counters to reduce runtime overheads.

There are 18 performance counter registers in Intel Pentium 4 CPU. Due to hardware restrictions only 9 registers can be used simultaneously. The performance counter metrics collected in our experiments are listed in Table 3.

The machine learning algorithms used in our experiments were adapted from WEKA [32] data mining software.

6 Experimental Results

6.1 Effectiveness of Productivity Index

The first experiment was conducted to show the effectiveness of *productivity index* in reflecting system high-level performance. We took Ordering and Browsing workloads as input and drove the test-bed into an overloaded state. We selected yield and cost metrics according to the correlation measure $Corr$. For an ordering mix input, the front-end server turned out to be the bottleneck and the PI defined as IPC over L2 cache miss rate had the most correlation with the high level performance. For a browsing mix input, database IPC and stalled CPU cycle metrics were selected as yield and cost, respectively.

Figure 5 shows the effectiveness of PI as an indicator of high-level throughput. In order to display PI and throughput curves in a similar scale, we normalized each of their values to their geometric means in different sampling intervals.

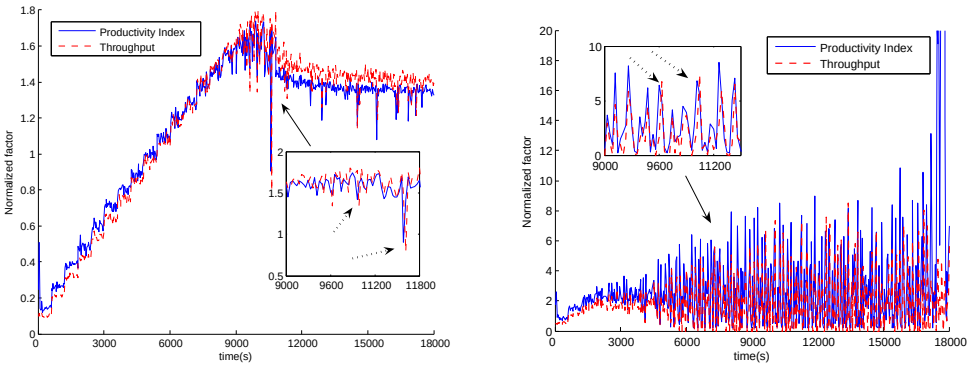
Figure 5 suggests that for both workloads, the PI and throughput metrics are in high agreement with each other. From the microscopic views, we can see that whenever there is a drop in PI, the corresponding throughput would decrease. Moreover, during some intervals, as pointed out by dotted arrows in the figure, the PI is more responsive than the throughput metric. PI also provides useful information about system-wide performance problems. For example, for the ordering mix input, our test-bed was overloaded due to the application server bottleneck. A drop in PI value suggests decreased IPC and increased L2 cache miss rate. The degraded performance may due to wasted work during context switching when there were too many threads in access to L2 cache in a time multiplexing way.

6.2 Individual Prediction Accuracy

The second experiment was designed to demonstrate the prediction accuracy of individual synopsis. A high synopsis accuracy means that the low-level metrics selected are sufficient in representing system internal states and the machine learning algorithm used is capable of correlating low-level metrics to a high-level state.

We tested the prediction accuracy for different level of metrics (e.g. OS level and hardware counter level) and using different machine learning algorithms. Table 4 and Table 5 summarize the accuracy results. We make several observations from the results:

1) For each testing workload, only the synopsis from the bottleneck tier and built from a similar workload pattern would produce a high prediction accuracy. For example, the synopsis built from a browsing mix on the database server had an accuracy of 0.965 in Table 4 due to TAN algorithm. But, even with the same learning algorithm, other synopses led to low accuracy. For example, when tested by browsing mix the synopsis built from ordering mix on application server resulted in an accuracy as low as 0.5. By examining the prediction results, we found that this synopsis generated prediction 0 (underload) most of the time.
2) Overall, hardware counter level metrics produced a higher accuracy than OS level metrics. For an ordering mix input, they achieved an accuracy of 0.952 and 0.935, respectively. But for a browsing mix input, the accuracy of OS level metrics dropped down to 0.635. Note that we can not claim hardware level metrics always perform better than OS level metrics. It depends on the workload characteristics. For some workload (ordering mix)



(a) Ordering mix (b) Browsing mix

Fig. 5. Effectiveness of PI in reflecting high-level performance.

TABLE 4
Prediction accuracy of individual synopsis tested by Browsing mix.

Specific Synopsis		OS Level				HPC Level			
Workload	Tier	LR	Naive	SVM	TAN	LR	Naive	SVM	TAN
Ordering	APP	0.585	0.500	0.505	0.545	0.570	0.500	0.502	0.505
	DB	0.473	0.500	0.465	0.587	0.439	0.453	0.493	0.646
Browsing	APP	0.635	0.621	0.505	0.603	0.529	0.557	0.540	0.515
	DB	0.604	0.612	0.667	0.635	0.859	0.935	0.957	0.965

TABLE 5
Prediction accuracy of individual synopsis tested by Ordering mix.

Specific Synopsis		OS Level				HPC Level			
Workload	Tier	LR	Naive	SVM	TAN	LR	Naive	SVM	TAN
Ordering	APP	0.842	0.928	0.965	0.935	0.805	0.883	0.921	0.952
	DB	0.689	0.932	0.776	0.665	0.746	0.791	0.844	0.840
Browsing	APP	0.583	0.585	0.593	0.547	0.662	0.588	0.588	0.588
	DB	0.545	0.514	0.512	0.572	0.635	0.659	0.662	0.694

both of them are accurate, but hardware level metrics perform significantly better than OS level metrics in some others (browsing mix). The reason is that hardware level metrics provide more detailed performance information. However, OS level metrics should be considered for I/O intensive workloads because hardware level metrics provide little information on I/O events.

3) Among the machine learning algorithms, SVM and TAN gained highest accuracy in most of the test cases. *Linear regression* performed worst because it can only capture linear correlations. *Naive bayes* performed not as well as TAN. It is because of its strong assumption on the independence of low level parameters.

Table 6 lists the execution time required to build a synopsis and make a single prediction using different machine learning algorithms. Although SVM has good prediction accuracy, it is computational prohibitive in online performance monitoring. Considering the accuracy and runtime overhead, TAN becomes the best choice for synopsis construction.

In addition to prediction accuracy, *TAN* also provides insights on bottleneck resources. Figure 6 shows the TAN structure for the application server synopsis built from ordering workload. Recall that for the ordering mix the front-end CPU is the bottleneck and server overload is due to excessive concurrent requests. From the Bayesian network in Figure 6, we can see that hardware counter metrics such as ITLB_REFERENCE, L2_MISS and ITLB_MISS were highly correlated to high-level overload state.

TABLE 6
Execution time for each machine learning algorithm.

	LR	Naive	SVM	TAN
Execution time (ms)	90	10	1710	50

6.3 Coordinated Prediction Accuracy

The third experiment was to demonstrate the overload prediction accuracy and bottleneck identification accuracy of coordinated predictor under different workloads. We used TAN learning algorithm in each synopsis and set the length of history bits to 3. We assumed $optimistic$ scheme with a threshold $\delta = 5$.

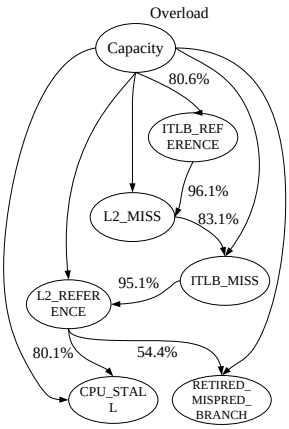


Fig. 6. Bayesian Network structure for hardware counter metrics.

Figure 7 presents the results based on both OS level and hardware counter level metrics. For overload prediction in Figure 7(a), similar to individual synopsis accuracy, OS level metrics led to poor accuracy in a browsing mix input. Hardware counter metrics resulted in consistently high prediction accuracy over all the workloads. For a priori known traffic (e.g. ordering mix), the prediction accuracy can be up to 90%. For interleaved workload, which consists of either browsing or ordering mix during any interval, the coordinated predictor still has an accuracy over 85%. The predictor is robust to workload changes and can maintain high accuracy even in the presence of bottleneck shifting.

It is expected that coordinated predictor would not be able to outperform the best individual synopsis for current workload. Based on spatial and temporal patterns in individual synopses, the predictor actually masks inaccurate synopses and selects the best synopsis for a workload. But for unknown workload, individual synopsis will have a degraded accuracy due to the limitation of supervised learning. Thus, the resulted coordinated accuracy decreased to approximately 80% in unknown workload input, which should be still acceptable.

For the bottleneck identification in Figure 7(b), the hardware counter level metrics also show consistently good accuracy. It is interesting that the bottleneck prediction accuracy has a similar trend as overload prediction in Figure 7(a). This may be due to the similar way the bottleneck identifier exploits the patterns in individual bottleneck prediction.

Recall that the results in Figure 7 were obtained under an assumption of optimistic scheme and a 3-bit history. In the following, we evaluated the impact of these two factors. Figure 8(a) shows that the schemes had little impact on the coordinated accuracy and there is no single scheme that performs consistently better than the other one. A possible reason is that the spatial and temporal patterns are obvious, the cases that the predictor is not sure are rare.

The length of the history bits controls how many steps the coordinated predictor looks back before making a prediction. Results in Figure 8(b) suggest there would be an increased accuracy when history bits be used. In most cases, a single history bit would improve the accuracy by approximately 10%. However, any further history information would lead to only a marginal improvement or even accuracy loss.

TABLE 7
Runtime overhead in collection of low-level metrics.

	Throughput loss	Latency increment
OS	2.64%	3.74%
hardware counter	0.49%	0.34%

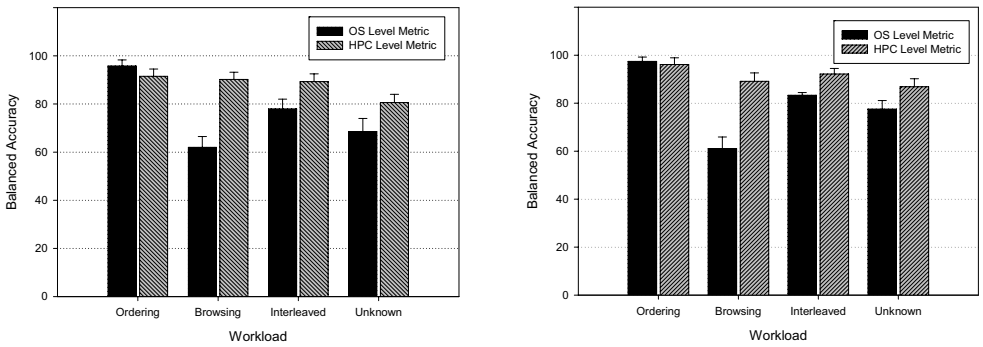
6.4 Runtime Overhead

The last experiment was to investigate the runtime overhead of the predictor. The cost for prediction in different machine learning algorithms was shown in Table 6. Table 7 lists the runtime overhead for OS and hardware counter level metrics collection. We normalized the throughput and request latency with respect to the values without metrics collection. The experiments takes an average of 5 executions and each execution lasted 30 minutes. The results show a much lower overhead for the hardware counter metrics collection. The throughput loss and latency increment are within 1% for hardware counter metrics collection.

7 Application of Capacity Prediction in Online Admission Control

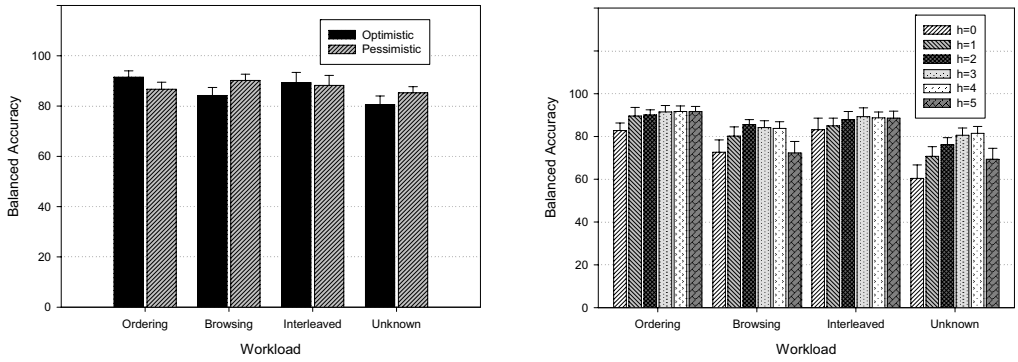
One application of multi-tier website capacity prediction is to guide an admission controller to reject excessive client requests when the incoming traffic exceeds the website capacity. Accurate predictions of the system capacity is crucial to the effectiveness of the admission control. We implemented the two-level capacity predictor in a standalone HTTP proxy residing on a separate machine. The proxy, on which admission controls can be applied, simply relayed the client requests to the front end of the multi-tier website. The proxy collected different levels of performance metrics in a specified interval (a 10-second interval in the remaining experiments), based on which the two-level predictor makes capacity predictions.

With online admission control, we verified that application level metrics like response time are not reliable for system capacity identification. Figure 9(a) plots the throughput of the multi-tier website under a transient spike due to different admission control mechanisms. The baseline traffic was 400 ordering clients. At the $150th$ second, a 600 ordering client spike was generated by another client machine. Figure 9(a) compares the hardware performance counter-based admission control with the response time-based one. To isolate the effect of admission control from the traffic variation, we simply instructed the proxy to reject the requests from the IP address generating the spike if an overload is detected.



(a) Overload prediction (b) Bottleneck prediction

Fig. 7. Coordinated prediction accuracy under different workloads.



(a) Impact of schemes (b) Impact of the length of history bits

Fig. 8. Impact of design parameters on accuracy.

In this way, the accuracy of the capacity prediction became the sole factor that affected the effectiveness of the admission control.

In Figure 9(a), we can see that hardware level metrics-based admission control can detect the overload immediately after the arrival of the spike and was able to maintain the throughput at a high level. In contrast, response time-based admission control failed to respond to the overload condition before the spike invaded the system. As a result, the website entered a churn state with up to 70% throughput loss and the overload remained for some time even after the spike's leave.

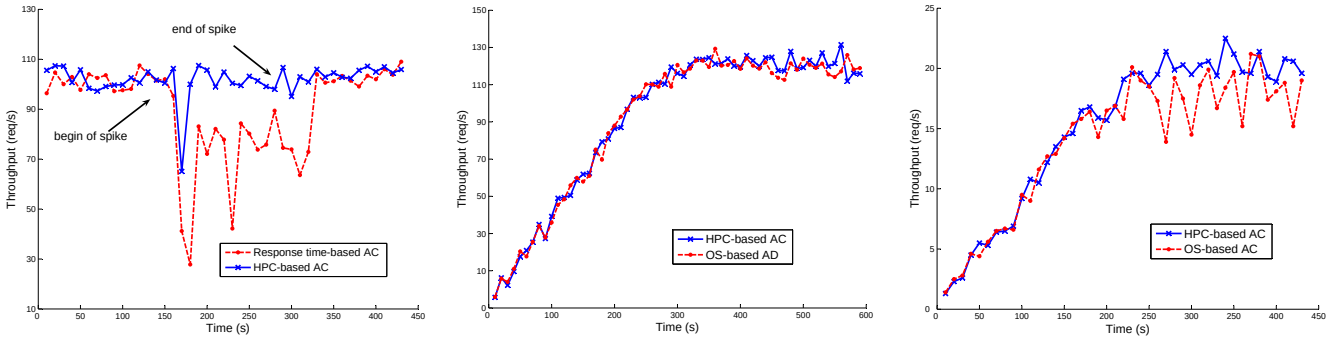
Figure 9(b) and Figure 9(c) compare the HPC level metrics-based admission control with the OS level metrics-based control under different traffic mixes. Instead of throwing transient spike to the website, we gradually increased the client traffic to overload the website in a step of 50 ordering clients and 10 browsing clients every 30 second. We implemented a simple adaptive rejection rate control based on the following heuristics: increase the rejection rate by 10% if the last system state (in the last 10-second interval) is "overload"; restore to the initial rejection rate if the last state is "underload". The initial rejection rates were set to 15% and 10% for ordering and browsing mixes respectively.

As discussed in Section 6.2, OS level metrics are accurate in determining the system capacity under the ordering mix. In Figure 9(b), we see similar admission control effects using HPC and OS level metrics: both effectively rejected excessive requests and stably maintained throughput. In contrast, as shown in Figure 9(c), OS level metrics failed to identify system overload as accurately as HPC level metrics under browsing mix, which results in large fluctuations in OS-based admission control.

8 Related Work

Server capacity determination is crucial to the problem of resource planning, configuration, and QoS-aware resource management. Early work on server capacity measurement [2] focused on how to generate synthetic workload to stress test the server capacity. Studies in [6] defined a set of benchmarks for stress testing the basic capacities of streaming servers. Unlike their offline measurement approaches, our approach focuses on online measuring the capability of multi-tier websites for the purpose of request-specific QoS-aware resource management.

Server capacity measurement is necessary for admission control and QoS-aware resource management. An admission controller should know when to turn away excessive requests, and the overload control mechanism should be invoked whenever the server capacity is reached. Most of the past work employed a single rule of



(a) Transient spike (b) Ordering mix (c) Browsing mix

Fig. 9. The effectiveness of admission control using different metrics.

thumb to measure server capacity based on application level metrics such as length of the web server request queue [26], incoming traffic density [3], [4], and request response time [33], [16], [12]. In [33], the authors employed a SEDA structure for response time-based admission control. The architecture has no mechanism for capacity measurement. Instead, it used the target response time to a conservative value so as to simplify the design of their admission controller. In [16], half of the most restrictive request response time guarantee was used as the threshold for controlling the incoming request rate. In [12], a measurement-based admission control approach was based on the execution time of requests. However, they assumed a non-preemptive shortest job first scheduling policy in the database server. As a result, the requests would have predictable processing time, independent of the server load condition. It makes it possible to estimate system utilization by monitoring admitted requests. Most of application servers are run in a processor sharing policy. In such servers, the processing time of a request is affected by other requests in concurrent execution. Even with a time-based server capacity estimate, request response time can no longer be used as a reference to calculate server utilization.

There were other QoS-aware resource management work that measured server capacity based on OS level metrics, such as server CPU utilization [10], [5], or hardware performance counter metrics [13]. However, in multi-tier servers, bottleneck resources may shift from tier to tier due to the dynamics of workload and it is difficult to set threshold values for capacity estimation. Our approach uses a combination of these metrics and does not require specifically setting the threshold values for each metric.

Our work is closely related to [7], [37], [8], [11] in that we use similar statistical models to capture underlying server characteristics. Duan and Batu proposed to use synopses in forecasting future event based on historic data [11]. Cohen et al. proposed to use TAN in computer systems [7] and Zhang et al. [37] improved the model accuracy by maintaining an ensembles of models, In [8], Cohen, et al. suggested to use the model to generate system signatures for the purpose of performance problem diagnosis. Our approach is different from theirs in the following aspects. First, we aim at real-time server capacity measurement, while theirs targeted at recursive problem identification. Second, they developed correlation for busy servers rather than overloaded systems. After determining the maximum concurrent level, they set steady state workload at 50-60% of the maximum level. Most importantly, we use multiple synopses for multi-tiers. The prediction results from the synopses are combined together to identify server capacity as well as the bottleneck tier. Wildstrom *et al.* also employed a similar idea using system level metrics for high level desicion making [34]. However, their goal was to maximize throughput by reconfiguring hardware under different traffics rather than overload prevention. We used a hardware metrics based index to monitor system health instead of simply using OS metrics for workload identification.

Finally, we remark that there are recent work on the utilization of hardware counter metrics for application performance tuning and debugging. Examples include works for identification of parallel program execution phases [9], online workload modeling and job scheduling [38], [27], and management of energy consumptions in virtualized environment [28]. Their focus was on the hardware counter events occurred within application codes. In contrast, our work uses system-wide hardware counter metrics to estimate high-level system state. System-wide hardware counter events provide useful information on the health of the system and bottleneck resources.

9 Conclusion

In this paper, we proposed a two-level coordinated machine learning approach to measuring the multi-tier website capacity based on hardware performance counters. We developed performance synopses to correlate low-level hardware counter metrics with high level system states of each tier. A coordinated predictor was then used to infer system-wide overload/underload state and identify resource bottleneck. Experiments results demonstrate the effectiveness of our approach at less than 0.5%

overhead even in the presence of workload changes and bottleneck shifting.

Our current model cannot reflect I/O related system performance. There is also room for accuracy improvement when the input traffic pattern is unknown. This work can be further extended to combine hardware counter level metrics with OS level metrics to capture I/O related performance problems.

Acknowledgement

This research was supported in part by U.S. NSF grants CCF-0611750, DMS-0624849, CNS-0702488, CRI-0708232 and CNS-0914330. A preliminary version of this paper appeared in [24].

REFERENCES

[1] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Specification and implementation of dynamic web site benchmarks," in *Proc. of WWC*, 2002.
[2] G. Banga and P. Druschel, "Measuring the capacity of a web server." in *Proc. of USITS*, 1997.
[3] H. Chen and P. Mohapatra, "Session-based overload control in qos-aware web servers." in *Proc. of INFOCOM*, 2002.
[4] X. Chen, P. Mohapatra, and H. Chen, "An admission control scheme for predictable server response time for web accesses." in *Proc. of WWW*, 2001.
[5] L. Cherkasova and P. Phaal, "Session based admission control: a mechanism for improving the performance of an overloaded web server," HP Labs, Tech. Rep. HPL-98-119, 1998.
[6] L. Cherkasova and L. Staley, "Measuring the capacity of a streaming media server in a utility data center environment." in *Proc. of ACM Multimedia*, 2002.
[7] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states: A building block for automated diagnosis and control." in *Proc. of OSDI*, 2004.
[8] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history." in *Proc. of SOSP*, 2005.
[9] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Proc. of ICS*, 2006.
[10] Y. Diao, N. Gandhi, J. L. H. S. Parekh, and D. M. Tilbury, "Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server." in *Proc. of NOMS*, 2002.
[11] S. Duan and S. Babu, "Processing forecasting queries," in *Proc. of VLDB*, 2007.
[12] S. Elnikety, E. M. Nahum, J. M. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in e-commerce web sites." in *Proc. of WWW*, 2004.
[13] R. J. Fowler, A. L. Cox, S. Elnikety, and W. Zwaenepoel, "Using performance reflection in systems software." in *Proc. of HotOS*, 2003.
[14] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine Learning*, vol. 29, no. 2-3, 1997.
[15] H.-U. Heiss and R. Wagner, "Adaptive load control in transaction processing systems," in *Proc. of VLDB*. Morgan Kaufmann, 1991.
[16] J.M.Blanquer, A.Batchelli, K.Schauser, and R.Wolsk, "Quorum: Flexible quality of service for internet services," in *Proc. of NSDI)*, G. M. Lohman, A. Sernadas, and R. Camps, Eds., 2005.
[17] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proc. of IJCAI*, 1995.
[18] J. C. Mogul, "Emergent(mis) behavior vs. complex software systems," in *ACM SIGOPS Operating System Review*, 2006.
[19] MySql, http://www.mysql.com.
[20] OProfile, http://oprofile.sourceforge.net/.
[21] PAPI, http://icl.cs.utk.edu/papi.
[22] PerfCtr, http://user.it.uu.se/ mikpe/linux/perfctr.
[23] PerfSuite, http://perfsuite.ncsa.uiuc.edu.
[24] J. Rao and C.-Z. Xu, "Online measurement the capacity of multi-tier websites using hardware performance counters." in *Proc. of ICDCS*, 2008.
[25] Rice University Computer Science Department, http://www.cs.rice.edu/CS/Systems/DynaServer.
[26] V. T. R.Iyer and K. Kant, "Overload control mechanisms for web servers," in *Proceeding of Workshop on Performance and QoS of Next Generation Networks*, 2000.
[27] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang, "Hardware counter driven on-the-fly request signatures," in *Proc. of ASPLOS*, 2008.
[28] J. Stoess, C. Lang, and F. Bellosa, "Energy management for hypervisor-based virtual machines," in *Proc. of the USENIX Annual Technical Conference*, 2007.
[29] The Transaction Processing Council (TPC)., http://www.tpc.org/tpcw.
[30] Tomcat, http://tomcat.apache.org/.
[31] J. Wei and C.-Z. Xu, "eqos: Provisioning of client-perceived end-to-end qos guarantees in web servers," *IEEE Trans. Computers*, vol. 55, no. 12, 2006.
[32] WEKA, http://www.cs.waikato.ac.nz/ml/weka.
[33] M. Welsh and D. E. Culler, "Adaptive overload control for busy internet servers." in *Proc. of USITS*, 2003.
[34] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin, "Machine learning for on-line hardware reconfiguration." in *Proc. of IJCAI*, 2007.
[35] C.-Z. Xu, *Scalable and Secure Internet Services and Architecture*. Chapman and Hall/CRC Press, 2005.
[36] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proc. of ISCA*, 1992.
[37] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems." in *Proc. of DSN*, 2005.
[38] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen, "Processor hardware counter statistics as a first-class system resource," in *Proc. of HotOS*, 2007.
[39] X. Zhong and C.-Z. Xu, "Energy-aware modeling and scheduling for dynamic voltage scaling with statistical real-time guarantee," *IEEE Trans. on Computers*, vol. 56, no. 3, 2007.

Jia Rao received the B.S. and M.S. degrees from Wuhan University in 2004 and 2006, respectively. He is currently a PhD student in the Department of Electrical and Computer Engineering at Wayne State University. His research interests are in the areas of distributed systems, resource auto-configuration in virtualized data center and machine learning. He is a student member of the IEEE.

Cheng-Zhong Xu received B.S. and M.S. degrees from Nanjing University in 1986 and 1989, respectively, and a Ph.D. degree in Computer Science from the University of Hong Kong in 1993. He is currently a Professor in the Department of Electrical and Computer Engineering of Wayne State University and the Director of Sun's Center of Excellence in Open Source Computing and Applications. His recent research interests are mainly in distributed and parallel systems, particularly in scalable and secure Internet services, autonomic cloud management, energy-aware task scheduling in wireless embedded systems, and high performance cluster and grid computing. He has published more than 150 articles in peer-reviewed journals and conferences in these areas. He is the author of Scalable and Secure Internet Services and Architecture (Chapman & Hall/CRC Press, 2005) and the leading co-author of Load Balancing in Parallel Computers: Theory and Practice (Kluwer Academic/Springer, 1997). He serves on five journal editorial boards including IEEE Trans. on Parallel and Distributed Systems and J. of Parallel and Distributed Computing. He was a program chair or general chair of a number of conferences, including Infoscale'08, EUC'08, and GCC'07. He is a recipient of the Faculty Research Award of Wayne State University in 2000, the President's Award for Excellence in Teaching in 2002, and the Career Development Chair Award in 2003. He is a senior member of the IEEE.