Online Measurement of the Capacity of Multi-tier Websites Using Hardware Performance Counters

Jia Rao and Cheng-Zhong Xu
Department of Electrical & Computer Engineering
Wayne State University, Detroit, Michigan 48202
{jrao, czxu}@wayne.edu

Abstract

Understanding server capacity is crucial for system capacity planning, configuration, and QoS-aware resource management. Conventional stress testing approaches measure the server capacity 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 capacity measurement even more challenging. This paper presents a measurement approach based on hardware performance counter metrics. The approach uses machine learning techniques to infer application-level performance at each tier. A coordinated predictor is induced over individual tier models to estimate system-wide performance and identify the bottleneck when the system becomes overloaded. Experimental results demonstrate that this approach is able to achieve an overload prediction accuracy of higher than 90% *for* a priori *known input traffic patterns 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 on-line decision.*

I. Introduction

Understanding of server capacity is crucial for 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 remain unchanged when the input rate continues to increase. It may drop sharply due to resource contention and algorithmic overhead for load management [11]. Knowledge about the server capacity can help a 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 [9], [17], [20].

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; see [18], [17] for examples. However, setting a request-specific response time value for admission control is non-trivial. In [13], 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.* [12] set a threshold to be half of the most restrictive request response time guarantee 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 dead-time 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 can be observed, particularly when a system is heavily loaded. In a multi-tier e-commerce 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. In [15], we quantified the delay in various

1063-6927/08 $25.00 © 2008 IEEE
DOI 10.1109/ICDCS.2008.97

input patterns and revealed that response-time based capacity measurement lack in accuracy and timeliness.

In our previous work [15], we developed an online capacity measurement approach, based on operating system (OS) level running statistics. The proposed statistical learning approach achieved better prediction accuracy than single performance metrics in the presence of workload changes. However, OS level metrics lack accuracy in reflecting detailed application performance which is useful for request scheduling and access control. Modern processors like Intel Xeon and AMD Operon are all equipped with a set of performance monitoring counters to record detailed hardware-level system information. The information includes a large group of parameters like instruction mix, rate of execution, memory access behaviors and branch prediction accuracy [16]. Together, they define a more accurate system internal running state and reflect aggregated effects of the requests in concurrent execution.

In this paper, we present effective and efficient solutions to the capacity measurement problem. In [15], we determined the capacity of the server based on application-level healthiness. This approach is sometimes problematic due to the limitations of application-level metrics discussed above. In this work, we define a metric of *productivity index* as a quantitative indicator of system healthiness and develop models over a small set of hardware performance counter metrics to characterize the system state of each server. We further develop a two-level coordinated real-time classification framework to infer system overload/underload state and identify resource bottleneck. We evaluated the approach in a two-tier Tomcat/MySQL website using TPC-W benchmark. Experimental results demonstrated its effectiveness and efficiency.

The rest of the paper is organized as follows. Sections II and III show the details of hardware-level capacity measuring approach. Sections IV and V give the evaluation method and experimental results. Related work is presented in Section VI. Section VII concludes this paper.

II. Lower Level System Performance Metrics

A system provides a rich set of performance metrics in both hardware performance counters and OS levels. Their statistics represent the internal performance states at runtime. Identifying a system state using lower level performance metrics involves 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 III.

A. 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* as the ratio of yield to cost and use it to measure the system processing capability:

$$PI = \frac{Yield}{Cost}. \quad (1)$$

This is a generic concept. By defining yield to be the number of completed requests and cost as the wall time, PI is equivalent to throughput in application level. Today's modern processors are all equipped with 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 stall cycles or cache miss rate, the PI metric reflects the instruction-level productivity.

The concept of productivity can also be defined at OS level. By defining yield and cost as user-mode and system-mode execution times, PI reflects CPU utilization. We argue that OS level metrics like CPU utilization may not be a good metric for system performance. For example, two programs reading data from memory in different ways, one sequentially and the other in a stride of 8, may have similar OS level resource utilization. In comparison, hardware-level metrics such as L2 cache miss rate can reflect application-level performance more accurately.

Hardware-level PI provides a good measure of system processing capability. An application-level "overload" and "underload" state can be identified by setting thresholds for PI. The thresholds can be determined empirically in offline stress-testing. 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. With the offline classification of "overload" and "underload" states in terms of PI, 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.

B. 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, for a given workload pattern. It is built based on previous correlations. 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 miss. 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 measuring, it is a binary variable, taking value of 1 ("overload") or 0 ("underload"). Each attribute A_i, $1 \leq i \leq n$, can be instantiated by assigning a measured value a_i during a sampling interval. Instantiating each variable 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 attributes $A_1, \ldots, A_n$ and class C. We denote it by $SYN(\{A_1, ..., A_n\}, C)$. The following subsection shows the construction in detail.

1) Construction of Synopsis and Prediction: A synopsis builder is essentially a set of algorithms that generate a synopsis from a training set. Generally speaking, there are two types of synopsis builders: linear and non-linear. In the following, we consider the following four linear and non-linear machine learning algorithms for synopsis construction: Linear regression (LR), Naive Bayes (Naive), Tree augmented naive Bayes (TAN), Support vector machine (SVM).

For a synopsis trained from a set D, we consider a testing set $P = \{p_1^*, ..., p_N^*\}$ with a similar structure as D. For each instance $p_i^* \in P$, the same training algorithm of the synopsis is re-applied to generate a prediction C' with respect to the class variable $C \in p_i^*$. We represent the prediction algorithm as function $Predict()$. That is, $C' = Predict(SYN, p_i^*)$. If $C' = C$, the prediction is correct, otherwise incorrect.

2) Attribute Selection: We borrow the concept of *information gain* in information theory to evaluate the relevance between each attribute and the class variable and only include the most relevant metrics in a synopsis. 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.

III. Two-Level Coordinated Website Capacity Measurement

The preceding section defines PI index and performance synopsis to correlate lower level 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 measurement: (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?

A. Issues in Multi-tier Website

We assume 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 measure $Corr$, in a way similar to [16], between the PI and high level performance metric r (e.g. throughput) over a time period:

$$Corr = \frac{Cov(pi, r)}{\sigma_{pi} \cdot \sigma_r} = \frac{\sum_{j=1}^{q}(pi_j - \overline{pi})(r_j - \overline{r})}{q \cdot \sigma_{pi} \cdot \sigma_r}, \qquad (2)$$

where q is the number of (pi, r) pairs sampled during the time t. The correlation measure 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 $Corr$ 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 mixes may change with time. 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. 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.

B. Framework

The two-level coordinated capacity measurement employs a similar hierarchical architecture as in [15]. It consists of a group of *performance synopses* and a *coordinated predictor*, which can be found in figure 1. The two-level coordinated prediction architecture takes runtime statistics on each tier as inputs. Based on these inputs, individual synopsis generates its prediction in regard to system high-level states. Final state prediction will be made in the coordinated predictor by combining these individual predictions.

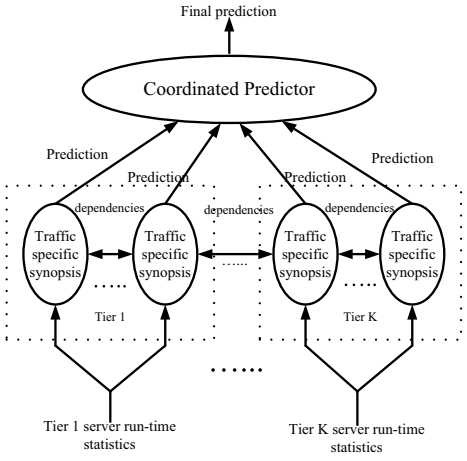


Fig. 1. The two-level coordinated prediction framework.

Although a synopsis is built on specific workload, its correlation remains valid in the presence of workload changes, as long as the bottleneck remains in the same tier. Once the

workload changes make the bottleneck shifting to another tier, the corresponding synopsis should be selected. The coordinated predictor selects the best synopses dynamically by studying the spatial (synopsis-wise) and temporal (time-wise) patterns among predictions of individual synopsis.

C. Coordinated Two-level Predictor

The coordinated predictor is designed as a two-level predictor to capture spatial and temporal patterns in synopses predictions. Its structure, as shown in Figure 2, is similar to a branch predictor of superscalar processors [22].

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), each bit R_i is 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 of 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.

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 defined as $\lambda_b(b_K...b_1) = \arg\max_i(b_i)$. It chooses the tier having the largest value in its corresponding bit in $b_K...b_1$ as the bottleneck tier.

D. Training and Prediction

To exploit the spatial (synopsis-wide) 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 $R_{m-1}...R_0$, is used to address the LHTs and find corresponding LHT for the GPV.
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:

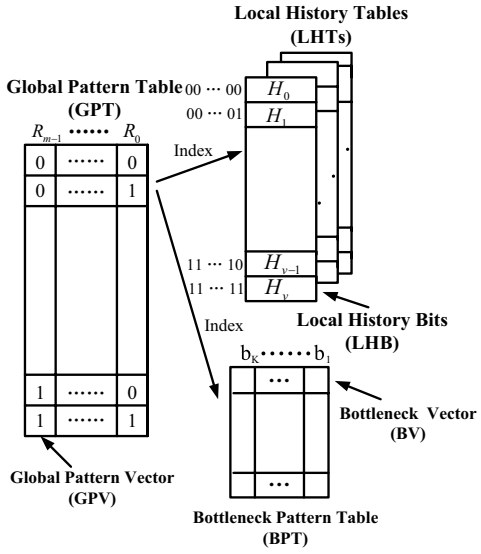


Fig. 2. The structure of the two-level predictor.

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 selected. 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 identification. The bottleneck predictor is invoked only when the system state is predicted to be overloaded. The system state 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 $C'' = \lambda(H_c)$, and

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

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 has the restriction relaxed. For any $\delta > 0$ there exists an interval $[-\delta, \delta]$, in which the predictor is not sure what prediction to make. We implemented two heuristic schemes for $\phi(H_c)$: An optimistic scheme set $\phi(H_c) = 0$ (underload) when $H_c \in [-\delta, \delta]$, while a pessimistic scheme sets $\phi(H_c) = 1$ (overload).

IV. Evaluation Methodology

To evaluate the two-level coordinated website capacity measurement, we built a test-bed of multi-tier e-commerce website. It consists of two tiers: front-end application server and back-end database server. The website was tested using workloads conforming TPC-W specifications. During execution, hardware counter level runtime statistics were collected. For comparison, OS level metrics were also reported.

A. TPC-W and Workload Selection

TPC-W is a transactional web e-commerce benchmark (`www.tpc.org/tpcw`). Its specification defines 14 different types of requests for an online bookstore service. In our test-bed, we deployed a free Java implementation of TPC-W benchmark from Rice University. TPC-W defines three traffic mixes: Browsing, Shopping, and Ordering, which have the percentages of browsing and ordering requests: (95%, 5%), (80%, 20%), and (50%, 50%), respectively. 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 e-commerce websites. 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 was found to put more pressure on database than on application server. For ordering mix, front-end would become bottleneck.

We assumed that the incoming traffic had a request mix in between the above two extremes: Browsing and Ordering. The bottleneck may change with the request mix in input traffic. Thus we selected the browsing and ordering mix as the representative workloads for training synopses and the coordinated predictor. The workloads were 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 was controlled by the number of Emulated Browsers (EBs).

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: Ramp-up workload and Spike workload. In ramp-up workloads, we gradually increased concurrent client sessions until overloaded. Spike workload refers to occasional extreme traffic burst. See [21] for workload configuration details. 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.

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. We defined a Balanced Accuracy (BA) metric to evaluate the prediction of induced synopses. It is an average of the probabilities of true positive and true negative.

B. Experiment Settings

Our test-bed consisted of a client machine, an application server and a database server. The front-end and back-end machines 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. The servers were interconnected by a fast Ethernet network.

The machines ran Linux 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 through a kernel patch `PerfCtr` (`www.user.it.uu.se/~mikpe/linux/perfctr.`) We wrote a lightweight tool to read hardware counter metrics in all physical CPUs using the global mode in `PerfCtr`. Event counter maintenance in hardware requires no runtime overhead [16] and we limited our tool to minimum functionalities that just initialize and read hardware counters to reduce runtime overheads. The machine learning algorithms used in our experiments were adapted from WEKA (`www.cs.waikato.ac.nz/ml/weka`) data mining software.

V. Experimental Results

A. Effectiveness of Productivity Index

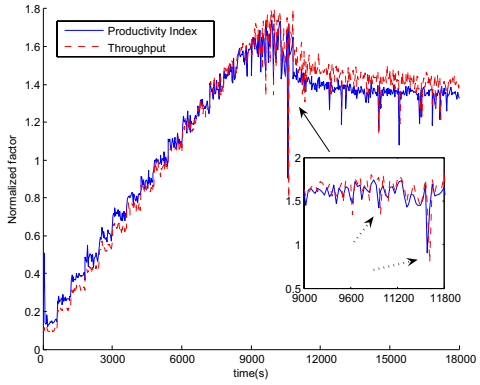


Fig. 3. Effectiveness of PI in reflecting high-level performance.

The first experiment was conducted to show the effectiveness of PI in (1) 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 $Corr$ in (2). For an ordering mix input, the front-end server turned out to be the bottleneck; accordingly we defined IPC as yield and L2 cache miss rate in this tier as cost. For a browsing mix input, database IPC and stalled CPU cycle metrics were considered.

Figure 3 shows the effectiveness of PI as an indicator of high-level throughput. Due to space limit, only Ordering mix is plotted here. 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. The figure suggests that 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.

B. 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 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 I summarizes the accuracy results in different input mixes.

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 I(a) due to TAN algorithm. But, even with the same learning algorithm, other synopses observed poor accuracy.
2) 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. The result was different from the one we observed in [15] due to the more accurate way we monitored system healthiness here. A possible reason is that the OS level metrics are insufficient to represent system internal state under the browsing mix input. More specifically, an ordering mix input would saturate the application server because there were too many threads in concurrent execution. But for a browsing mix input, system overload was due to a small percentage of heavy requests in the database server. OS level metrics might not be able to identify whether the overload state is due to excessive load (i.e., too many requests) or excessive work (i.e., heavy requests).
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 each metric.

The execution time required to build a synopsis and make a single decision using LR, Naive, SVM and TAN are 90ms, 10ms, 1710ms, and 50ms, respectively. Considering the accuracy and runtime overhead, TAN is the best choice for synopsis construction.

C. 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 $\delta = 5$.

Figure 4 presents the results based on both OS level and hardware counter level metrics. For overload prediction in Figure 4(a), similar to individual synopsis accuracy, OS level metrics had poor accuracy in a browsing mix input. Hardware counter metrics have consistent good 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 is still acceptable.

For the bottleneck identification in Figure 4(b), the hardware counter level metrics also have consistent good accuracy. It is interesting that the bottleneck prediction accuracy has a similar trend as overload prediction in Figure 4(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 4 were obtained under an assumption of optimistic scheme and a 3-bit history. We also evaluated the impact of these two factors. Results suggest that the schemes had little impact on the coordinated accuracy and the prediction accuracy would be increased by approximately 10% if a single history bit is used. However,

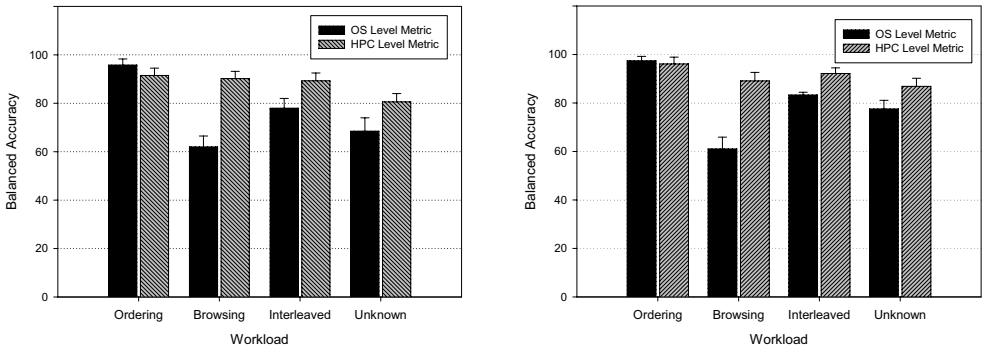
TABLE I. Prediction accuracy of individual synopsis.

(a) Browsing Mix Input

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

(b) Ordering Mix Input

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



(a) Overload prediction accuracy.

(b) Bottleneck prediction accuracy.

Fig. 4. Coordinated prediction accuracy under different workloads.

any further history information would lead to only a marginal improvement.

D. Runtime Overhead

The last experiment was to investigate the runtime overhead of the predictor. The cost for prediction in different machine learning algorithms has been discussed above. We also measured the runtime overhead in metrics collection. We normalized the throughput and request latency with respect to the values without metrics collection. The experiments took the average of 5 executions and each execution lasted 30 minutes. The results show a much lower overhead for the hardware counter metrics collection. The performance loss due to hardware counter metrics collection is within 0.5%, compared with 4% for OS level metrics.

VI. Related Work

Early work on server capacity measurement [1] focused on how to generate synthetic workload to stress test the server capacity. Studies in [3] 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. Most of the past work employed a single rule of thumb to measure server capacity based on application level metrics such as length of the web server request queue [14], incoming traffic density [2], and request response time [18], [12], [9].

There were other QoS-aware resource management work that measured server capacity based on OS level metrics, such as server CPU utilization [7]. 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 previous work [15] uses a combination of OS metrics and does not require specifically setting the threshold values for each metric.

Our work is closely related to [4], [23], [5], [8] in that we use similar statistical models to capture underlying server characteristics. Our approach is different from theirs in the following aspects. Firstly, they developed correla-

tion for busy servers rather than overloaded systems. 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 [19]. However, their goal was to maximize throughput by reconfiguring hardware under different traffic rather than overload prevention.

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 [6], online workload modeling and job scheduling [24], [16], [10], and management of energy consumptions real-time embedded systems [25]. 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.

VII. 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 We would like to thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by U.S. NSF grants CCF-0611750, DMS-0624849, CNS-0702488, and CRI-0708232.

References

[1] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proc. of USITS*, 1997.

[2] X. Chen, P. Mohapatra, and H. Chen. An admission control scheme for predictable server response time for web accesses. In *Proc. of WWW*, 2001.

[3] L. Cherkasova and L. Staley. Measuring the capacity of a streaming media server in a utility data center environment. In *ACM Multimedia*, 2002.

[4] 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.

[5] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proc. of ACM SOSP*, 2005.

[6] 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 *ICS*, 2006.

[7] 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.

[8] S. Duan and S. Babu. Processing forecasting queries. In *VLDB*, 2007.

[9] 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.

[10] R. J. Fowler, A. L. Cox, S. Elnikety, and W. Zwaenepoel. Using performance reflection in systems software. In *Proc. of HotOS*, 2003.

[11] H.-U. Heiss and R. Wagner. Adaptive load control in transaction processing systems. In *Proc. of VLDB*, 1991.

[12] J.M.Blanquer, A.Batchelli, K.Schauser, and R.Wolsk. Quorum: Flexible quality of service for internet services. In *Proc. of NSDI*, 2005.

[13] J. C. Mogul. Emergent(mis) behavior vs. complex software systems. In *ACMSIGOPS Operating System Review*, 2006.

[14] V. T. R.Iyer and K. Kant. Overload control mechanisms for web servers. In *Proc. of Workshop on Performance and QoS of Next Generation Networks*, 2000.

[15] J. Rao and C.-Z. Xu. CoSL: A coordinated statistical learning approach to measuring the capacity of multi-tier websites. In *Proc. of IPDPS*, 2008.

[16] 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.

[17] J. Wei and C.-Z. Xu. eQoS: Provisioning of client-perceived end-to-end qos guarantees in web servers. *IEEE Trans. Computers*, 2006.

[18] M. Welsh and D. E. Culler. Adaptive overload control for busy internet servers. In *Proc. of USITS*, 2003.

[19] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin. Machine learning for on-line hardware reconfiguration. In *Proc. of IJCAI*, 2007.

[20] C.-Z. Xu. *Scalable and Secure Internet Services and Architecture*. Chapman and Hall/CRC Press, 2005.

[21] J. Rao and C.-Z. Xu. Online measurement of the capacity of multi-tier websites using hardware performance counters. Technical Report, Cluster and Internet Computing Laboratory, Wayne State University, October 2007.

[22] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. of ISCA*, 1992.

[23] 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.

[24] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *Proc. of HotOS*, 2007.

[25] X. Zhong and C.-Z. Xu. Frequency-aware energy optimization for real-time periodic and aperiodic tasks. In *Proc. of LCTES*, 2007.