Using Link Gradients to Predict the Impact of Network Latency on Multitier Applications

Shuyi Chen, *Student Member, IEEE,*, Kaustubh R. Joshi, *Member, IEEE,* Matti A. Hiltunen, *Member, IEEE,* Richard D. Schlichting, *Fellow, IEEE,* and William H. Sanders, *Fellow, IEEE*

***Abstract*—Managing geographically dispersed deployments of complex multitier applications involves dealing with the substantial effects of network latency. However, the effects of network latency on an application's end-to-end performance can be far from obvious, thus making it difficult to predict the true impact of infrastructure changes such as network upgrades or server relocation on the users of an application. In this paper, we propose a new metric to quantify this impact called the *link gradient*. We develop a novel noise-resistant, non-intrusive technique to measure the link gradients in running systems without requiring knowledge of the system structure by using a combination of run-time delay injection and spectral analysis. We evaluate the intrusiveness and accuracy of our approach using micro-benchmarks and a deployment of two benchmark multitier web applications on PlanetLab. Using these results, we show that link gradients can be used to accurately predict the impact of network latency changes on the end-to-end responsiveness of individual application transactions, even in new application configurations and without requiring a dedicated test environment.**

I. Introduction

The performance of modern distributed programs such as multitier enterprise applications is often significantly affected by their *application topology*, that is, the network overlay defined by which application components communicate with which other components, together with the characteristics of these intercomponent links. For example, consider a geographically dispersed 3-tier application in which a web server forwards a transaction to an application server, which in turn exchanges messages with a back-end database to execute the transaction. Even in such a simple scenario, the response times of a transaction can vary substantially depending on the topology, such as having the application server co-located with the web server versus the database. In this case, the intercomponent links remain the same, but the characteristics of the underlying network connections would differ if the web server and database are separated by a wide geographic distance for instance.

In such multitier applications, a key element in determining transaction end-to-end response time *logical link latency*, that is, the network latency in the overlay from one component to another, possibly across multiple physical links. To a degree, this is obvious, but as we discuss below in section II, the actual relationship between logical link latencies and transaction performance can be surprisingly complex and difficult to predict. Moreover, the need to anticipate how changes in link latencies will impact response times is only increasing given the emergence of improved infrastructures that can support distributed and decoupled applications on a global scale. Such improvements include networks with higher bandwidths and connectivity, data centers that are geographically distributed around the world, in-network services such as cloud computing, and software paradigms such as cross-enterprise service-oriented architectures (SOAs).

This paper introduces the *link gradient*, a new metric that captures the impact of changes in logical link latencies on end-to-end transaction response times in distributed multitier applications. Once the link gradients for an application have been determined on a single given topology—one per link per transaction type—they can be used to predict responsiveness in any arbitrary topology without having to go through the time and effort to actually deploy and test the application. Thus, knowing the link gradients of a given application provides guidance that can be used in a variety of ways to configure and manage the system. For example, they can be used a) to determine server placements that improve availability while ensuring that responsiveness remains within a tolerable threshold, b) to help choose between competing services in service-oriented architectures, and c) to evaluate the cost versus benefit tradeoffs of planned network upgrades by network operators. We demonstrate the accuracy of performance predictions based on link gradients experimentally on PlanetLab using RUBiS and RUBBoS, two well-known examples of multitier applications [1], [2].

In addition to the metric itself, we also develop an innovative technique for measuring the link gradient in a running system based on delay injection and spectral analysis. This minimally-intrusive approach allows link gradient measurements to be performed continuously on running production systems, and allows the link gradient for each transaction to isolated from other transactions and applications even if they use the same hardware resources. We demonstrate the non-intrusiveness of the measurement technique experimentally using a collection of micro-benchmarks.

II. Overview

We begin by motivating the need to predict the impact of latency on end-to-end response time in multitier applications,

Shuyi Chen and William H. Sanders are with the Information Trust Institute, University of Illinois at Urbana-Champaign, Urbana, IL 61801. Email: {schen38,whs}@illinois.edu

Kaustubh R. Joshi, Matti A. Hiltunen, and Richard D. Schlichting are with AT&T Labs Research, Florham Park, NJ 07932. Email: {kaustubh,hiltunen,rick}@research.att.com

This material is based upon work supported by the National Science Foundation under Grant Nos. 06-15372 and 04-06351 and by AT&T.

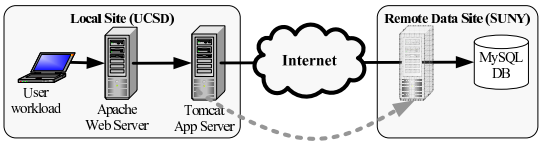


Fig. 1. Sample RUBiS configuration

discuss how link gradients address the need, and provide an overview of our proposed techniques to measure them. To do so, we consider the example of a simple but classic multitier application that implements a JavaBeans-based online auction site (RUBiS [1]) as shown in Figure 1. It consists of three tiers, a front-end Apache-based web server tier, a middle Tomcat-based application server tier, and a back-end MySQL-based database tier across which the application logic is distributed. Now consider a scenario in which the application, having been built and tested on a local development network, is to be deployed for widespread use. Today, there are many different ways to do so including hosting in single or multiple data-centers, a CDN-based proxy architecture, cloud-bursting setups, or a hybrid of these techniques.

Transaction	San Diego (sec)	New York (sec)	Change
PutBidAuth	0.03	0.17	466%
SearchItemsByReg.	5.70	0.16	-97%
SellItemForm	0.04	0.14	250%
ViewBidHistory	2.31	0.19	-92%
ViewUserInfo	4.13	0.20	-95%

TABLE I
AVERAGE RESPONSE TIME FOR RUBIS TRANSACTIONS

While these different deployment models offer an application a wide range of cost, availability, and scalability properties, it is not always easy to determine their precise impact on an application's responsiveness. Each model introduces significant changes in the network latency between the various elements of the target system. And because different transactions often stress different parts of an application, the impact of deployment architectures on end-to-end responsiveness can vary dramatically on a per-transaction basis.

For example, consider RUBiS application deployments in which the web server is located in San Diego, California, the back-end database server in New York city, and the application server is co-located with either the front-end or the back-end servers. Table I shows the response times of a few RUBiS transactions for both deployments. As can be seen, even with such a simple application, the impact of network latency changes on end-to-end response time is dramatic, and not trivial to predict without a detailed understanding of the application. For production systems, the architectures become more complex, and elements such as load balancers, content caches, firewalls, network accelerators, authentication servers, and network file systems all interject onto the application communication paths, and their placement can have significant impacts on the application's end-to-end responsiveness. Moreover, these impacts can change dynamically, as a system's workload changes, and impacts queuing and call flow behavior.

However, being able to predict those impacts is useful not only for optimizing an application's deployment architecture either statically or dynamically, in response to changing workload conditions, but also for predicting the impact that changes in the network, that affect path latency, would have on the applications running on it. An additional challenge is to predict the impacts for black-box systems, i.e., those for which the code, or any significant design expertise is not available, and to be able to do so cheaply, and in a push-button way.

Our proposed metric, the link gradient, provides exactly such a capability in a concise and straightforward manner. For any given application architecture and deployment, the link gradient quantifies the point derivative of end-to-end transaction response time, i.e., the rate at which each application transaction's response time varies as a function of the link latency of each logical communication link between nodes in the system. One can then measure the system's transaction response times in its current configuration, and using the point derivative along with certain linearity assumptions (see Section III-A), predict the per-transaction response times in a new configuration in which everything else remains the same, but with *different* latencies between each of the system's components.

To measure the link gradients, we propose an online technique in which the basic idea is to inject small delays into all the network packets flowing across each of the application's logical links at runtime. Then, we measure the impact of these perturbations on the end-to-end response time of each transaction and use these measurements to compute the gradients. By carefully choosing the pattern of perturbations - a square wave - and appropriate spectral analysis techniques based on Fourier transforms, we show how measurement noise can be significantly reduced, and good measurement accuracy can be achieved even with very small perturbations that do not significantly alter the user's end-to-end experience. Our proposed technique satisfies all the desired properties described above. It provides predictions that capture all relevant variables such as application architecture, configuration, and intermediate components. Because it is a measurement-based approach, it does not require any detailed application knowledge, and can thus work on black-box systems in a push-button manner. Because the required perturbations are very small, the approach can be continuously deployed while the system is running, and can recompute the link gradients whenever they change, e.g., due to workload changes. Finally, it is able to isolate the impacts of network latency on a per-transaction basis even if multiple transactions share the same components, or if multiple components share the same physical resources.

III. TECHNICAL APPROACH

In this section, we define the link gradient of a system, explain how spectral analysis can allow its computation with only small perturbations to the system, and derive the required formulae.

A. The Link Gradient

Consider a multitier application consisting of multiple software nodes and let $C = \{c_1, c_2, \cdots, c_n\}$ be the set of n logical one-way communication links between them[1] with each link

[1]Two-way communications are represented by two independent links, one for each direction.

c_i parameterized by a link latency l_i. Further, let the system's performance metric be quantified using its end-to-end mean response time $\bar{rt}$. If the system provides a number of different services (e.g., an e-commerce site with multiple transactions, such as browse and buy), then the response time can be either a per-transaction response time, or the system's overall mean response time.

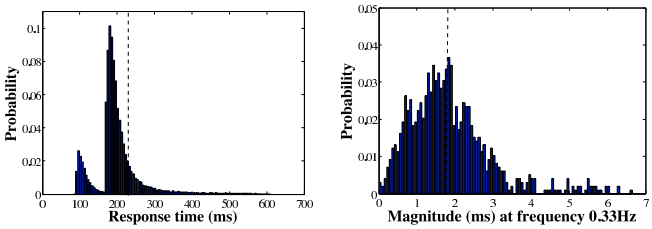
Then, the *link gradient* $\vec{\nabla}\bar{rt}$ quantifies how a change in the link latency for each link affects the response time, and is defined as a vector $\vec{\nabla}\bar{rt} = \left(\frac{\partial\bar{rt}}{\partial l_1}, \ldots, \frac{\partial\bar{rt}}{\partial l_n}\right)$, where each element $\nabla\bar{rt}_i = \frac{\partial\bar{rt}}{\partial l_i}$ is the link gradient of link c_i. Intuitively, the link gradient of a link is a partial derivative that specifies the rate at which the system's response time changes per unit change in the link latency of communication link c_i, assuming that the latencies of all other links remain constant.

The link gradient can be used to approximate how the response time of the system would be affected by a change in link latencies (e.g., due to reconfiguration of the system). Specifically, if the vector $\vec{\Delta l} = \{\Delta l_1, \ldots, \Delta l_n\}$ represents the amounts by which each link latency changes, the new response time of the system can be approximated by its Taylor expansion, i.e.,

$$\bar{rt}(\vec{l} + \vec{\Delta l}) \approx \bar{rt}(\vec{l}) + \vec{\nabla}\bar{rt} \cdot \vec{\Delta l} + O(\vec{\Delta l}^2)$$

This equation makes two simplifying assumptions. First, it assumes that the response time (as a function of link latency) is differentiable. Second, the equation only captures first-order effects. If a system's response time has nonlinear dependencies on the link latency, the equation is accurate only as long as the change in latency Δl is small and the higher-order terms can be ignored. For linear relationships, the higher-order terms vanish, making the equation exact. Although there are some exceptions, we show experimentally under realistic conditions that many systems have large regions of continuous linear relationships in which the higher-order effects can be ignored.

To understand why linearity is observed, we consider an approximate interpretation of link gradients in terms of message crossings. To illustrate, let node a call another node b over a link that has a latency of l, and consider how the mean response time of a would change if the (one-way) link latency increases by Δl under different types of communication scenarios. If a sends a message to b and waits for a reply before continuing, the response time could be expected to increase by Δl, and the link gradient would be $\lim_{\Delta l \to 0} \frac{\bar{rt}(l+\Delta l) - \bar{rt}(l)}{\Delta l} = 1$. The link gradient would remain the same if a sent m messages to b in a pipelined fashion before waiting for a response (e.g., over a TCP link). However, if a were to send m messages in series such that it waited for a response from b before sending the next message, the increased latency would affect the response time for each of the m messages, and the link gradient would be m. Conversely, if a did not wait for a reply from b, an increase in link latency would not affect the response time at all, and the link gradient would be 0. Drawing upon these observations, we can loosely interpret the link gradient as the "mean number of message crossings in the critical path of the system response," which, for many communication patterns is a constant function of node behavior, and thus gives rise to a linear relationship between response time and latency. Factors such as timeouts and queuing can affect linearity when very large latency changes are considered, but as we show in Section VI the linearity assumption holds quite well in practice even across transcontinental latency changes. Moreover, the measurement techniques we propose can easily detect any violations.



(a) Truncated distribution (b) Distribution at freq. 0.33Hz

Fig. 2. Response time distributions

Following the linearity assumption, one obtains the following *Link Gradient Equation*, which predicts the end-to-end response time rt^b in a configuration b based on the response time rt^a in another configuration a, and the logical link latencies for each link c_i in the two configurations, i.e., l_i^a and l_i^b.

$$rt^b = rt^a + \sum_{\{i|c_i \in C\}} (l_i^b - l_i^a) \cdot \vec{\nabla}\bar{rt}_i \quad (1)$$

B. Link Latency Perturbation

Conceptually, link gradients can be computed at run-time by active delay injection. A target communication link can be chosen, and a delay Δl_i can be systematically injected in the packets traversing the link. The end-to-end response time of the system $\bar{rt}'$ can then be measured while the delay is being injected and compared with the nominal response time $\bar{rt}$. The ratio $\frac{\bar{rt}' - \bar{rt}}{\Delta l_i}$ is then an approximation to the link gradient $\vec{\nabla}\bar{rt}_i$. As long as the relationship between response time and link latency is linear, the approximation is exact.

However, in practice, the complicating factor is that production systems, especially those running across wide-area networks and/or on shared resources, typically have response time distributions with long tails, high variances, and non-stationary noise patterns due to periodic events such as garbage collection. For example, Figure 2(a) shows a truncated response time distribution for a single transaction of RUBiS (*PutBid*) running on PlanetLab. The distribution is constructed using 27648 samples, with a mean of 229 ms and a standard deviation of 269 ms, and has a long tail (not shown) reaching up to 12s. Despite the large number of samples, the 95% confidence interval for the mean is ± 3.17 ms, implying that to detect a change in the mean within 10% error, the mean would have to be shifted by at least 63.4 ms (i.e., by 27%). That would have to be done by adding a low-variance constant delay to every transaction, since attempts to manipulate the mean by introducing larger delays to only a few transactions would increase the variance, and thus the required shift as well. Injecting such large delays into a running system for long periods of time is intrusive, and does not lend itself to dynamic on-the-fly techniques. Moreover, it is also not trivial to inject low-variance delays into communication links.

Therefore, additional noise can also be introduced by the measurement apparatus during delay injection.

To solve that problem of excessive noise, we propose a unique approach based on the observation that most of the noise found in typical environments is not periodic. Moreover, if periodic noise does exist (e.g., garbage collection), it occurs only at a few narrow frequencies. Therefore, by injecting perturbations in the form of periodic waveforms at carefully chosen frequencies and performing spectral analysis to extract the effect on the system's response time, one can obtain a level of precision that is significantly superior to that offered by a time domain method. To illustrate, consider Figure 2(b), which shows a distribution of the magnitude of an arbitrarily chosen frequency component (0.33Hz) from the frequency spectrum of the same response time data represented in Figure 2(a). At this frequency, not only is the mean response time much smaller (1.86 ms), but the distribution is much tighter (with no truncated tail), is more symmetric, and has a standard deviation of 1.1 ms. The corresponding 95% confidence interval of ± 0.07 ms allows a change of 1.4 ms to be detected with 10% error.

We use the Fast Fourier Transform (FFT) of the response time series to compute is power spectrum. The power spectrum shows the energy present in the waveform as a function of frequency. Because of the nonperiodicity of the noise, the energy content at each frequency except the 0 frequency is very low. Therefore, if the delay is injected in a periodic manner at any of those frequencies, its magnitude does not have to be very large. We introduce a periodic delay into the link by using a square wave pattern. When the square wave is high, a constant delay is injected into all the messages that traverse the link. When it is low, no delay is injected. Then, we compute the energy difference with and without the delay injection at the frequency component where we inject the waveform. Finally, we derive the link gradient from the difference as presented in the following subsection.

We use a sliding window to estimate several Fourier coefficients for each frequency. In doing so, we convert a distribution of samples from the raw data in the time domain to a distribution of samples in the spectral domain at a particular frequency. Subsequently, we use the mean value of this distribution to compute the shift in the distribution due to the injection of delays and the link gradients. If more sophisticated filtering techniques to estimate distribution shifts are available in the time domain, they can also be applied to the frequency domain distribution to increase the sensitivity of the approach even further. Thus the square-wave injection and spectral estimation is orthogonal to improvements in analysis methods.

C. Spectral Link Gradient Computation

Next, we develop formulae to show how the power spectrums with and without delay injection can be used to compute the link gradient of the link. To do so, we consider a response time series x_i, $i = 0 \ldots N-1$ of length N measured at uniform intervals of time ΔT_s. The Discrete Time Fourier Transform for this time series at all frequencies $f_k = \frac{k}{N \cdot \Delta T_s}, k = 0 \cdots N-1$ (i.e., those with an integral number k of cycles over the duration of the time series) is given by $\mathrm{F}^0(f_k) = \sum_{i=0}^{N-1} x_i e^{\frac{-2\pi j}{N} ik}$.

Now consider a delay time series x_i^d with a square-wave pattern of magnitude A_d and frequency $f_d = \frac{k_d}{N \cdot \Delta T_s}$ for some k_d added to the original response time series. Then, the Fourier Transform $\mathrm{F}^d(f_{k_d})$ of the resulting time series can be expressed as:

$$
\begin{aligned}
F^d(f_{k_d}) &= \sum_{i=0}^{N-1} (x_i^d + x_i) e^{-\frac{2\pi j}{2n} i} = \sum_{i=0}^{N-1} x_i^d e^{-\frac{2\pi j}{2n} i} + F^0(f_{k_d}) \\
&= \sum_{m=0}^{k_d} \sum_{i=0}^{2n-1} x_{2nm+i}^d e^{-\frac{2\pi j}{2n}(2nm+i)} + F^0(f_{k_d}) \\
&= \sum_{m=0}^{k_d} \sum_{i=0}^{2n-1} x_{2nm+i}^d e^{-\frac{2\pi j}{2n} i} + F^0(f_{k_d}) \\
&= k_d A_d \sum_{i=0}^{n-1} e^{-j(\delta + \frac{\pi}{n} i)} + F^0(f_{k_d}) \\
&= k_d A_d e^{-j\delta} \frac{1 - e^{-j\pi}}{1 - e^{-j\frac{\pi}{n}}} + F^0(f_{k_d}) \Big| \textstyle\sum_{i=0}^{N} a^i = \frac{1-a^N}{1-a} \\
&= \frac{2 \cdot k_d A_d e^{-j\delta} e^{j\frac{\pi}{2n}}}{e^{j\frac{\pi}{2n}} - e^{-j\frac{\pi}{2n}}} + F^0(f_{k_d}) \Big| 1 - e^{-jx} = \frac{e^{j\frac{x}{2}} - e^{-j\frac{x}{2}}}{e^{j\frac{x}{2}}} \\
F^d(f_{k_d}) &= \frac{k_d A_d e^{j(\frac{\pi}{2n} - \delta)}}{j \sin\frac{\pi}{2n}} + F^0(f_{k_d}) \Big| \sin x = \frac{e^{jx} - e^{-jx}}{2j}
\end{aligned}
$$

Here, δ is the phase shift of the square wave in comparison with the time series interval, and $2n = N/k_d$ is the total number of data points in each square wave cycle. However, we also know from Equation 1 that if the link latency l of a link is increased by Δl, then the change in response time is given by $\bar{rt}(l + \Delta l) - \bar{rt}(l) \approx \frac{\partial \bar{rt}}{\partial l} \cdot \Delta l$. Equating this change in response time to the delay A_d and noting that $|e^{j(\frac{\pi}{2n} - \delta)}/j| = 1$, we obtain the link gradient $\frac{\partial \bar{rt}}{\partial l}$ as a function of the Fourier transforms of the response time before and after a square wave latency increase of magnitude Δl:

$$
\frac{\partial \bar{rt}}{\partial l} = \frac{|\mathrm{F}^d(f_{k_d}) - \mathrm{F}^0(f_{k_d})| \cdot sin(\frac{\pi}{2n})}{\Delta l \cdot k_d} \tag{2}
$$

Since the phase shift δ of the square wave does not appear in the final derivation, the delay injection, which is done locally at each link, does not have to be synchronized with the end-to-end response time measurement, which is done at the entry point of the system. This eliminates the need for synchronized clocks or the need to know the time taken for propagation of latency increases into the end-to-end response, and makes the technique attractive for wide-area settings.

IV. Architecture and Implementation

Using the basic approach described in the previous section, we have implemented a distributed active monitoring framework that automatically calculates the link gradients for a distributed application.

A. Monitoring Framework

The framework is shown in Figure 3 in the context of a single application. It consists of a central coordinator and a

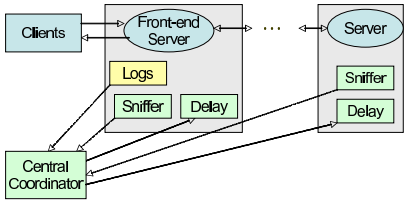


Fig. 3. Monitoring architecture

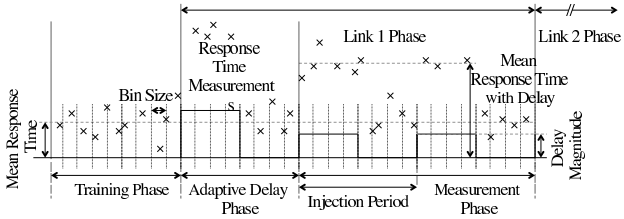


Fig. 4. Timeline for the measurement process

set of *sniffer* and *delay* daemons. The sniffer daemon provides the central coordinator with a list of all communication links, identified by the source and destination IP and port number, on which traffic was exchanged.

The delay daemon can inject delays in a square wave pattern with a specified frequency and magnitude on each node in the system based on commands from the coordinator. To do so, it requires redirection of application packets through its process. Our prototype implements two such mechanisms. The first is based on the loadable packet filter kernel module `ip_queue` in the standard Linux kernel. Packets are intercepted and their headers routed to the delay daemon through the installation of `iptables` rules. The second mechanism, used for PlanetLab machines, utilizes the `tap` device and UDP tunnelling. Applications are configured to use a network interface implemented by the `tap` device. Designated packets sent through the `tap` device are then intercepted by the local delay daemon. Once the packets have been passed to the delay daemon process, it uses a timer-wheel implementation [3] to delay each packet by a fixed amount during the "on" period of the square wave, and then sends the packet using a UDP connection to the delay daemon running in the remote node. Upon receipt of the packet, the remote delay daemon reinjects the packet into the remote `tap` device for the remote application to receive. To correct for inaccuracies, the delay daemon measures the amount of delay actually injected into each packet, and reports the mean for use in the link gradient computation.

The central coordinator orchestrates the daemons and executes the link gradient measurement algorithm. It requires a list of the application's nodes (host, process) and the location of the end-to-end response time data. Currently, the framework supports applications that have web interfaces, and computes link gradients for each transaction. It parses Apache access logs to extract a response time series—i.e., timestamp, response time pairs—for each transaction's URL. No additional workload beyond the application's normal workload is required for measurement purposes, thus ensuring minimal interference with a running system.

B. Measurement Algorithm

The process of measuring the link gradients consists of two phases: a) the training phase, and b) a set of per-link phases. In the training phase, the coordinator builds a list of communication links, and passively collects the system response times for the transactions in question. It uses the response time series to determine the parameters for the delay square wave that is to be injected into the communication links. The per-link phase is conducted once for every communication link, and is the active phase during which a delay is introduced into the target link and the system response is measured. Figure 4 presents a timeline of the entire process.

The measurement algorithm assumes that the system's communication patterns and the distribution of noise in the response time remain unchanged over the duration of the training and measurement phases. These assumptions are reasonable because the time taken to measure link gradients is only a few minutes for practical systems, and we do not expect the measurement to be done during periods of rapidly changing traffic. Nevertheless, if the models mispredict the response time due to a change in the environment, we require rerunning the measurement algorithm in the new environment to recalibrate the gradients.

Training Phase: The training phase involves estimating the following parameters.

1. Bin Size: To obtain a periodic time series of end-to-end response times from web access logs that contain requests arriving at random times, the framework divides the measurement period into equal-sized bins as shown in Figure 4. It then uses the mean of all the data points within a bin as a single sample point in the time series. Prior to binning, we remove outliers by discarding samples whose response times are more/less than one inter-quartile range of the upper/lower quartiles of the collected response time measurements. The discarded samples are replaced with the mean value of their adjacent sample points. To choose the bin size ΔT_s, the framework records the mean $\bar{m}_t$ and standard deviation $\sigma(m)_t$ of the time within k consecutive requests. ΔT_s is then set to $\bar{m}_t + 3\sigma(m)_t$. This heuristic ensures that at least k points are averaged in each bin with a high probability (0.999 under the assumption of a normal inter-request distribution).

2. Normal Fourier Transforms: Next, the framework computes the frequency domain characteristics of the system's normal response time so that it can choose a good delay injection frequency and magnitude. To do so, the coordinator divides the normal response time series into M different chunks of N sample points each. M is chosen as a trade-off between the measurement length, intrusiveness, and accuracy as described below. The parameter N is a user-specified parameter that indicates the number of bins used in each link gradient computation. The Fourier Transform for each of the M chunks is then computed, and these transforms are then averaged to obtain the mean $\bar{\mathrm{F}^0}(f)$ and standard deviation $\sigma(\mathrm{F}^0)(f)$ (both complex numbers) of the Fourier Transform at each frequency $f = \frac{k}{N \cdot \Delta T_s}$. These are used to select both an appropriate delay magnitude and a frequency.

3. Delay Frequency: Measurement noise, although significantly reduced in the frequency domain, still occurs when computing the link gradients using Equation 2 because the Transforms with and without delay injection (i.e., $\mathrm{FFT}^d(f_k)$

and $\mathrm{FFT}^0(f_k)$) are calculated separately. To minimize the effects of this noise, we choose the frequency f_d (from $\frac{1}{N\cdot\Delta T_s},\ldots,\frac{N-1}{N\cdot\Delta T_s}$) at which the standard deviation of the response time $\sigma(\bar{\mathrm{F}^0})(f)$ as calculated above has the lowest value. In this manner, we can also avoid noise due to periodic events present in the system (e.g., garbage collection or cron jobs).

4. Delay Magnitude: Finally, to ensure that measurement accuracy is maintained irrespective of whatever noise remains, we select a delay magnitude Δl that is proportional to the standard deviation, i.e., $\Delta l = A_d = d * \sigma(\mathrm{FFT}^0)(f_d)$ at injection frequency chosen in the previous step. The constant of proportionality d is called the *delay scale factor* and can be chosen so as to ensure a user-specified level of accuracy in the gradient measurements. For gradients with 5% accuracy, we first compute the 95% confidence interval for the FFT value at the injection frequency, i.e., $\bar{\mathrm{F}^0}(f_d) \pm 1.96 \cdot \frac{|\sigma(\mathrm{F}^0)(f_d)|}{\sqrt{M}}$. Then, the delay scale factor is set high enough that the size of the confidence interval, i.e. the possible error, is less than 5% of the injected delay. Putting together the above equations, we get $\mathrm{ConfInterval}/\Delta l = \frac{2\times 1.96\times|\sigma(\mathrm{F}^0)(f_d)|}{\sqrt{M}}/(d\cdot\sigma(\mathrm{FFT}^0)(f_d)) \leq 5\%$. This equation is used to balance the measurement time (which depends on M) against intrusiveness (which depends on d). If measurement time is more important that intrusiveness, one specifies M and allows the framework to pick d. Conversely, one can limit the delay injected by specifying d, and let the measurement run for as long as is needed. Using $d = 30$ from the micro-benchmarks in Section V, we get $M \geq 7$ and use $M = 9$ to leave sufficient margin.

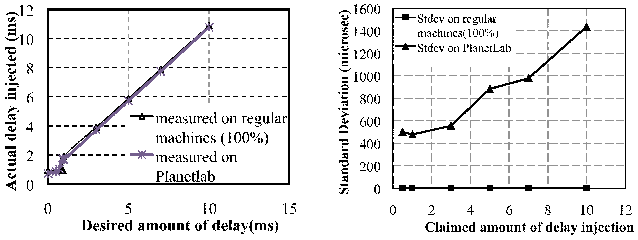
Per-link Phase: To perform delay injection on a link, the per-link phase uses the bin size and injection frequency computed during training directly. However, the delay magnitude A_d, chosen, as above, to be large enough for accurate measurement, can be too large. Specifically, if the unknown link gradient $\nabla\bar{rt}_i$ of the link is large (e.g., for heavily used links), then the impact of delay injection on the end-to-end response time, which is proportional to $\nabla\bar{rt}_i \cdot A_d$, could be unacceptably large. To compensate, the framework executes an adaptive delay phase in which it first obtains an estimate of the link gradient by injecting only a single cycle of the smallest delay that is accurately injectable by the injection mechanism (see Section V). Using this rough estimate $\nabla\bar{rt}'_i$, a delay magnitude of $A'_d = A_d/\nabla\bar{rt}'_i$ is used in the measurement phase so that the overall impact on the end-to-end response time is approximately equal to the value A_d needed for accuracy.

In the measurement phase, the coordinator injects a delay of magnitude A'_d with frequency f_d into the system, collects enough response time measurements to compute a time series of N bins, and computes the final value of the link gradient using the measured response times, the mean value of the actual delay injected as reported by the delay daemon, and Equation 2.

V. Micro Benchmarks

In this section, we examine how the accuracy of the algorithms changes as a function of the delay injection method, the algorithm inputs (i.e., the total number of sample points or bins, N, the delay scale factor d, and the number of raw data points per bin k), and the target system's communication patterns. The goal of these experiments is to provide an understanding of the sources of error during measurement, and to determine how to minimize the error resulting from them.

To eliminate the effects of external factors other than the algorithm parameters, most of the experiments were conducted under ideal conditions on a testbed running on a 100Mbps local Ethernet network connected to a single switch. The hosts used had identical configurations running Fedora Linux with kernel 2.6.18 on single core 2.0 GHz Athlon XP 2400+ processors with 1GB RAM. For measuring the accuracy of the `tun/tap` injection mechanism, we used PlanetLab machines.



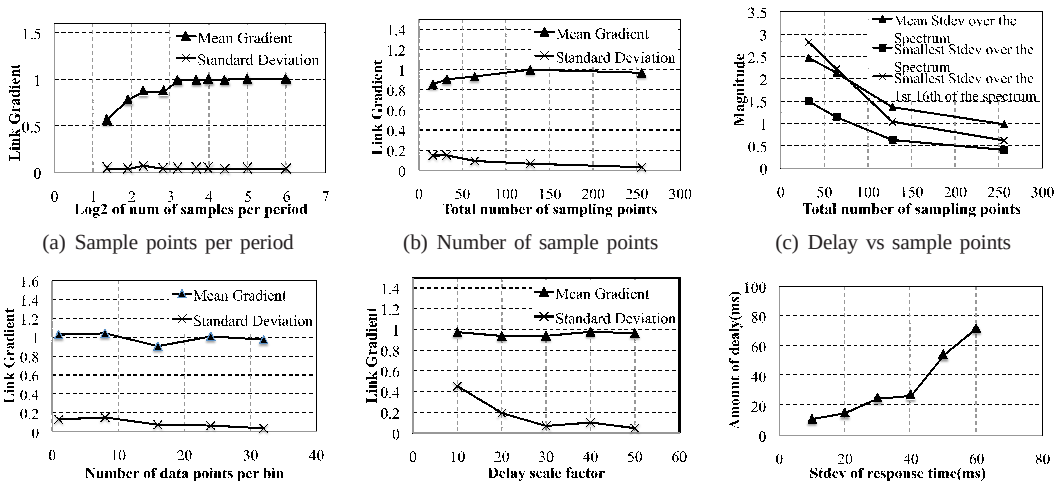
(a) Mean (b) Deviation

Fig. 5. Accuracy of delay injection mechanisms

Accuracy of Delay Injection: The first set of experiments measured the amount of actual delay injected as a function of the desired delay. The experiments were conducted under 100% CPU load on the testbed, and conditions of heavy load on the PlanetLab machines. The actual delays injected were measured using the *gettimeofday* system call, and the mean values are shown in Figure 5(a). As can be seen from the figure, for desired delays greater than 1 ms, the mean delay injected is accurate except for a constant shift of 1 ms, and for delays smaller than 1 ms, the actual injected delay is on the average 1 ms. This is due to the heavy loads and scheduling granularity of 1 ms on the Linux kernels used in our experiments. To correct this difference between desired and actual delay injected, the framework measures the actual delay injected during runtime in the delay daemon, and uses the measured value for the link gradient computation.

While this technique suffices for correcting a consistent error during injection, any variations in the error over successive injection cannot be corrected in this manner. The standard deviation of the error is a metric of the variability of the error, and is shown in Figure 5(b). For the `ip_queue` injector, the deviation is less than 5 μs even under 100% CPU workload. That shows that this daemon is able to inject delays as small as 1 ms reliably even under heavy workloads. However, the standard deviation of the error introduced by the `tap` delay daemon is as large as 1.5 ms when the desired delay is 10 ms. This results in additional noise during the link gradient measurement, and causes the injected delay magnitude chosen by the algorithm to be higher when the `tap` injector is used.

Next, we consider the effect of the algorithm inputs on its accuracy. To do so, we deployed a simple test application with a front-end server, which, on receiving a client request, calls a back-end server that processes the request for a normally distributed random amount of time with a mean of 100 ms



(a) Sample points per period (b) Number of sample points (c) Delay vs sample points

(d) Bin size (e) Delay scale factor (f) Delay vs RT std. dev.

Fig. 6. Sensitivity study of link gradient

and a standard deviation of 30 ms. When the back-end server returns, the front-end server generates a reply back to the client. A client node generates requests one at a time. Then, as a measure of accuracy, we compared the measured link gradient for the link between the front- and back-end servers with its expected value of 1. Each experiment was replicated 10 times to compute the mean and standard deviation.

Sample Points per Period: In this experiment, we plotted accuracy vs. the number of sample points per injection period. To do so, we fixed the number of data points per bin (k) to 8, the injected delay to 30 ms, and the total number of sample points (N) to 128. In the rest of this section, we use these as the default values. The number of sample points per injection period was changed from 2 (the Nyquist rate) to 64, corresponding to only 2 cycles during the entire measurement phase. Figure 6(a) shows that the accuracy is poor at an injection frequency close to the Nyquist frequency, but improves rapidly as the number of data points per injection period increases and the number of cycles decreases, and approaches 1 when there are more than 8 data points per period (corresponding to frequencies between 0 and 1/4 of the Nyquist frequency in this case). Additional experimentation showed that any increase in the length of measurement phase, and correspondingly, the number of injection periods, does not help compensate for the error introduced by decreasing number of sample points per period.

On further investigation, we found that the error when the number of sample points per period is small is due to the smoothing of the square wave because of binning. Bins that fall on a transition may have values that are averaged between the low and high values, thus distorting the shape of the wave. The relative number of the affected bins can be minimized by reducing the number of transitions, and in our implementation, we achieve this by only considering relative frequencies between 2 and $N/8$ when choosing an injection frequency during the training phase. We have discovered that this restriction does not significantly reduce the number of "good" frequencies available for delay injection, and in practice provides very good results.

Total Number of Sample Points: Next, we investigate how the total number of sample points affects the accuracy, stability, and intrusiveness of the measurements by varying the total number of sample points from 16 to 256. Figure 6(b) shows both the mean link gradient and the corresponding standard deviation (which provides a metric of stability of the link gradient measurement) across 10 experiments. As expected, as the number of sample points increases, the accuracy increases until it reaches and remains close to the correct value (1.0), and the standard deviation decreases monotonically, indicating improving stability.

Figure 6(c) shows how the standard deviation of the response time series without any delay injection $\sigma(\mathrm{F}^0)(f_d)$ (i.e., the noise) changes as the number of sample points in the measurement interval increases. The figure shows that the mean standard deviation across all the frequencies in the FFT, the smallest standard deviation amongst all the frequencies, and the smallest standard deviation amongst the first $N/8$ frequencies (the set of frequencies used to choose the injection frequency) all decrease monotonically. The reason is that, as the number of sample points increases, the corresponding increase in the number of injection cycles causes a reduction in any periodic noise components. Since the amplitude of the injected delay is set to be a multiple of the frequency with the smallest standard deviation ($A_d = d * \sigma(\mathrm{FFT}^0)(f)$), the amount of delay injected into the link and its associated perturbation decrease as well.

From those results, it is clear that increasing the total number of sample points is always good if everything else remains the same. However, this leads to an increase in the length of the measurement phase, which may be dictated by external factors (e.g., how quickly the results are needed).

Data Points per Bin: The next experiment evaluated the sensitivity of the link gradient measurement to the number of data points per bin (k) used to compute the bin size. Recall that binning acts as an averaging filter and performs noise filtering functions. Figure 6(d) shows how the accuracy of the link gradient computation changes as k is changed from 1 to 32 with the total number of sample points fixed at 64 and the number of sample points per injection period fixed at 32.

Although the number of data points per bin does not affect the mean link gradient value much, the standard deviation reduces slowly, indicating, as expected, that increasing the number of data points per bin increases the stability of the result. However, if additional data points are indeed available, then comparison with previous results indicates that it is better to increase the total number of sample points rather than increase the number of data points per bin, since the former not only leads to an increase in the stability, but also requires a smaller delay to be injected in the process.

Delay Scale Factor: Figure 6(e) shows how the delay scale factor d affects the accuracy and stability of the result. In practice, a large perturbation is only feasible in off-line evaluation, and in any on-line measurement, the perturbation should be as low as possible while still allowing accurate measurement. Fortunately, the results of changing the delay scale factor from 10 through 50 while keeping N at 64 and k at 12 show that low delay scale factors do not affect the accuracy of the link gradient, just its variance. Moreover, the variance decreases rapidly at first, and much more slowly after the scale factor increases beyond 30. The result is expected since a lower scale factor implies that the injected delay is small in comparison with the natural variance of the response time spectrum. Based on these results, our implementation uses a default delay scale factor of 30 as a good trade-off between intrusiveness and stability.

Standard Deviation of Response Time: Next, we examined how the amount of delay required for a fixed delay scale factor (set to 30) varies as a function of the variance of the system's response time. Figure 6(f) shows a plot of the amount of delay A_d computed by the training phase as the standard deviation of the system's response time is changed from 10 to 60 (by changing of the variance of the normally distributed response time of the back-end server). As seen in the figure, the required delay magnitude increases almost linearly with a slope of 1. This result indicates that our technique only requires a delay comparable to the standard deviation of the system itself, and thus achieves its objective of low intrusiveness and high accuracy for our micro-benchmarks.

Based on those results, the best way to improve the accuracy and stability of the results while minimizing intrusiveness is to increase the number of sample points per measurement phase. However, as increasing the measurement duration may not be possible, the delay scale factor can be increased as long as it does not trigger recovery mechanisms within the application.

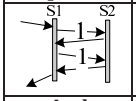
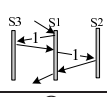
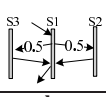
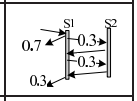
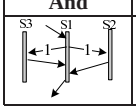
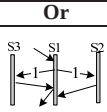
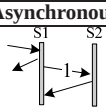
Sequential	Multiple	Load Balancing	Cache
			
And	**Or**	**Asynchronous**	
			

TABLE II
COMMUNICATION PATTERNS

Communication Patterns: Finally, we measured the accuracy and intrusiveness of the link gradient algorithm across different communication patterns commonly found in distributed systems. These patterns are shown in Table II, and represent interactions between a front-end server S1 and two back-end servers S2 and S3 when S1 receives a client request. They include repeated sequential calls to a single server, sequential calls to multiple servers, load balancing, caching, AND parallel (i.e., S1 waits for the replies from both S2 and S3 before continuing), OR parallel (i.e., S1 waits for the first reply from either S2 or S3 before continuing), and asynchronous communication.

Transactions are generated at the client using a Poisson arrival process. The service times for S2 and S3 are normally distributed and their values are set roughly to the same order of magnitude typical of web applications. For S2, the mean and standard deviation are 100 ms and 30 ms respectively, while for S3, these parameters depend on the pattern. They are 100ms and 30 ms for the Multiple pattern, 150 ms and 45 ms for the Load Balancing pattern, and 200 ms and 60 ms for the And and Or pattern. The behavior of S3 in the load-balancing pattern is made different from S2 to test whether the gradients will generate the correct weighted average for each load-balanced link. In the And and Or communication patterns, the service times are constructed so that S3 will reply after S2 with a high probability. Thus, S1 is bottlenecked by S3 in the And pattern in which it waits for both S2 and S3 to reply, and by S2 in the OR pattern in which it returns after the first response. For the cache pattern, the miss rate is 0.3. Inputs to the framework are the same across all scenarios, based on the microbenchmarks conducted earlier: the number of sample points per experiment is set to 128, the delay scale factor is set to 30 and the number of data points per bin is set to 12.

Table III shows the expected values of the link gradient calculated as described in Section III, and the measured values for the call and return path for all the communication patterns. The expected value for the multiple pattern is 1 because the delayed link (S1-S2 or S1-S3) is used only once. For the cache pattern, it is 0.6 because there are up to two calls to the cache, each with a probability of 0.3. Finally, the expected values for the And and Or patterns are the probabilities that the response time of the delayed server is either maximum or the minimum of the two back-end servers, respectively. For example, the expected gradient for the link S1-S2 for the And pattern is $\mathrm{Prob}[N(100,30) > N(200,60)]$.

Pattern	S1↔S2			S1↔S3		
	Exp.	Call	Ret.	Exp.	Call	Ret.
Seq.	2	1.92	1.96	N/A	N/A	N/A
Multiple	1	1.00	1.01	1	0.99	1.00
Load Bal.	0.5	0.51	0.47	0.5	0.53	0.49
Cache	0.6	0.60	0.64	N/A	N/A	N/A
And	0.07	0.14	0.16	0.93	0.95	0.96
Or	0.93	0.92	0.93	0.07	0.08	0.10
Async.	0	0.07	0.07	N/A	N/A	N/A

TABLE III
RESULTS

The results show that the link gradient measurements are accurate in cases where the communication pattern produces a linear relationship between the link latency and end-to-end response time. The only two patterns with a nonlinear relationship are the parallel And and Or patterns because

Pattern	Delay	rt w/o delay	rt w/ delay
Seq.	36 ms	202.03 (42.25)	233.54 (55.67)
Multiple	46 ms	200.16 (42.99)	222.34 (48.67)
Load Bal.	42 ms	124.77 (44.51)	135.4 (43.37)
Cache	82 ms	58.33 (94.27)	77.76 (134.34)
And	61 ms	201.2 (54.61)	228.91 (63.71)
Or	30 ms	98.63 (28.75)	112.91 (33.64)
Async.	27 ms	100.20 (30.55)	101.14 (30.31)

TABLE IV
DELAY INJECTION AND PERTURBATION

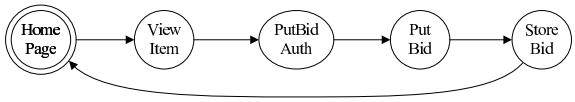


Fig. 7. Client transition diagram

they give rise to the nondifferentiable max and min functions. Even in such cases, the results show that the gradients are still accurate point estimators of the system and can provide valuable insight. For instance, the main impact on response time in the And pattern is due to S1-S3 latency rather than S2-S3 latency showing the bottleneck due to the slower S3 server. In contrast, in the Or scenario, the link between S1 and the faster server (S2) has more impact on response time, since the front-end server only needs the first reply to continue.

Finally, we quantify the measurement overhead in Table IV by comparing the mean response times with and without delay injection for all scenarios. The standard deviations of the response time are shown in parenthesis. The results illustrate that our framework imposes relatively low perturbation comparable to the intrinsic variability of the system itself, and does not increase the variance of the response time significantly in the process.

VI. EVALUATION

In this section, we evaluate the approach and its predictive power in a realistic setting. Predictive power is evaluated in terms of the link gradient's ability to predict the application's response time in a configuration *different* from the one that was used to compute the link gradient. We show that although the relationship between latency and end-to-end response time changes across varying workloads, configurations, communication models, and load-balancing policies, the link gradient captures those effects.

A. Experimental Setup

We used a deployment of RUBiS on PlanetLab. RUBiS [1] is a well-known eBay-like auction application benchmark. Although small, it is representative of many multi-transaction, multitier web applications and can be configured using many settings (e.g., load balancing, connection pooling, and replication) that make it hard to predict the effects of changing logical link latency. To minimize the effect of hardware differences, we chose nodes from PlanetLab with identical configurations. Deployment on PlanetLab nodes distributed across the United States ensured that the measurements were made in a highly-shared, high-variance, and challenging wide-area environment.

We used the 3-tier Java-servlet version of RUBiS with a front-end Apache server (WS), middle-tier Tomcat application servers (AS), and a back-end MySQL database server (DB). We used the standard RUBiS workload generator with randomly generated TPC-W client think times [4], and instantiated each client with the bidding-oriented workflow shown in Figure 7. *ViewItem* (VI) returns information about an item, *PutBidAuth* (PBA) returns a user authentication page, *PutBid* (PB) performs authentication and returns detailed bidding information, and *StoreBid* (SB) stores a bid in the database. Of these, *PutBidAuth* is application-server-centric, while the other transactions are application- and database-server-oriented.

Users/ λ	Link	View Item	PutBid Auth	Put Bid	Store Bid	λ_{lg}
20/	WS-AS	1.92	1.78	2.04	1.05	11.05
11.15	AS-DB	15.14	0.63	18.03	18.76	11.00
30/	WS-AS	1.47	1.46	1.75	0.68	16.55
17.00	AS-DB	16.99	0.51	19.13	18.90	16.60
40/	WS-AS	1.86	1.78	1.89	1.02	22.05
22.05	AS-DB	16.28	0.83	18.77	18.19	21.95
50/	WS-AS	1.70	2.08	1.58	0.82	27.70
27.80	AS-DB	18.24	0.46	20.40	20.37	27.40
60/	WS-AS	1.48	2.05	1.97	0.87	33.10
33.75	AS-DB	21.74	0.87	23.15	22.80	32.65

TABLE V
LINK GRADIENTS

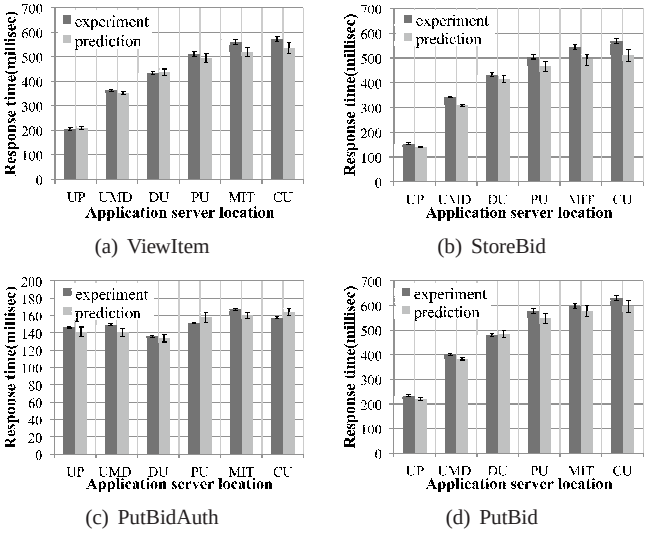
B. Link Gradient Computation

We used a setup of RUBiS identical to the one shown in Figure 1 with a single server of each type and the default configuration settings. The workload generator and WS were located in San Diego (UCSD), while AS and DB were located in Pittsburgh (CMU). In all experiments, the link gradient algorithm was set to use 1 data point per bin, 3456 sampling points per experiment, and a delay scale factor of 30. Table V shows the measured link gradients for all the transactions as the workload was varied from 20 to 60 concurrent clients. It also shows the normal throughput of the system for each workload (λ) and the modified throughput during link gradient measurement (λ_{lg}) for each link. Although we measured the link gradients for both directions of each link (e.g., WS-AS and AS-WS), the table shows the mean value, because the results were very similar. However, as we show later, that may not always be the case, and the ability of our approach to measure link gradient in both directions independently is useful in asymmetric setups such as ADSL lines and satellite links.

Comparing the link gradients for the AS-DB link, one can clearly see the difference between the small gradient for the web-server-oriented *PutBidAuth* transaction and the large gradients for the others (which are database-oriented). The magnitude of the link gradients provides guidance for targeted application optimization, e.g., moving of components with high link gradients closer together or increasing of cache sizes across such links. The table also shows that the link gradient is not a static metric and changes with workload, possibly due to queuing effects. Therefore, a minimally intrusive runtime technique is desirable for link gradient measurement so that the gradients can be recomputed on-the-fly as the system dynamics change. Finally, the throughput measurements show that the system throughput changes by less than 5% in all cases, thus demonstrating the low intrusiveness of the technique.

Link	Delay (ms)	View Item	Store Bid	PutBid Auth	Put Bid
Std. Dev.		210.78	160.5	82.46	151.07
AS-WS	24	28.10	28.01	53.82	39.25
DB-AS	6	65.95	94.32	18.9	86.62

TABLE VI
RESPONSE TIME PERTURBATION



(a) ViewItem (b) StoreBid

(c) PutBidAuth (d) PutBid

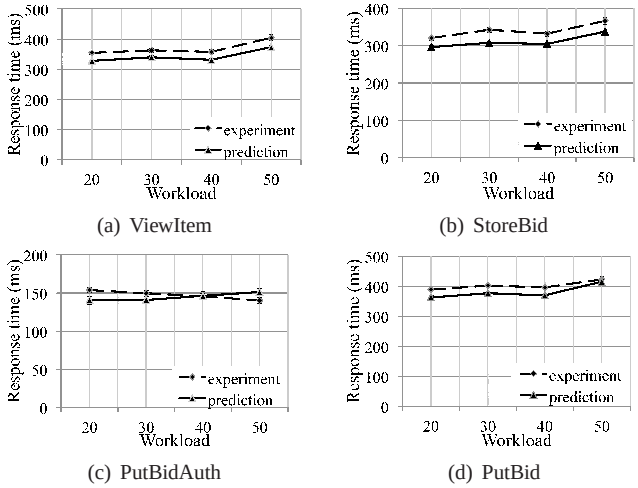
Fig. 8. Predicting the effects of component placement

Another measure of intrusiveness is the increase in response time due to the delays injected during measurement. Those results are shown for a workload of 30 clients in Table VI. The first row indicates the standard deviation of each transaction's response time during normal system operation, while the other rows show both the injected per-message delay and the change in response time during the measurement process. All numbers are in milliseconds. Although the noisy PlanetLab environment requires much higher delays for some links than an exclusive environment would, the impact is still within the system's normal behavior. In all cases, one can see that the additional delay is (sometimes significantly) less than the system's normal standard deviation. One reason for the larger delays is the high variance Tun/Tap injection mechanism on PlanetLab. Based on our experience in local experiments, the injection mechanism based on the standard Linux `ip_queue` facility requires much less delay injection, and thus less perturbation to the running system.

C. Predictive Power

Figure 8 demonstrates the link gradient's predictive power. We use the Link Gradient Equation from Section III-A to predict a transaction's response time in a new configuration based on its current response time and link latencies, the link gradient, and the latencies in the new configuration. The prediction is compared against a measured response time obtained by actual deployment. To estimate one-way link latency, we compute the mean of 1000 TCP round-trip times, and divide it by two. If application-specific effects such as network throttling are a concern, it might be necessary to compute port-specific TCP/UDP latencies using the application's own ports.

In the first set of experiments, we kept the workload constant at 30 clients and the locations of our WS and DB fixed at UCSD and CMU respectively, but placed the AS on similar



(a) ViewItem (b) StoreBid

(c) PutBidAuth (d) PutBid

Fig. 9. Prediction results under different workloads

hardware in 6 geographically dispersed locations: the U. of Pittsburgh (UP), the U. of Maryland (UMD), Duke U. (DU), Princeton U. (PU), MIT, and Cornell U. (CU). Figure 8 shows the predicted and measured response times for each transaction in each different configuration along with 95% confidence intervals. The confidence interval for the predicted response time rt^b in a new configuration b takes into account the errors introduced due to both the response time rt^a in the original configuration a and the latency measurements in both a and b, and is calculated using the equation $\rho(rt^b) = \rho(rt^a) + \sum_{\{i|c_i \in C\}}(\rho(l_i^b) - \rho(l_i^a)) \cdot \vec{\nabla}\bar{rt}(l_i)^2$, where $\rho(rt^a)$, $\rho(rt^b)$, $\rho(l_i^a)$, and $\rho(l_i^b)$ are the variances in the response time and latency measurements in configurations a and b, respectively. The results show good agreement between the predicted and measured response times across all transactions even across large response time changes (more than 3x in some cases). Since the hardware and workload in all of the locations were similar, the results demonstrate that link latency plays a dominant role in the application's response time, and that the link gradient is able to accurately capture the effect of link latency changes on the said response time.

In the next set of experiments, we placed WS, AS, and DB at UCSD, UMD, and CMU, respectively, and measured the predictive ability of the link gradients (computed in Table V) as the workload was changed from 20 to 50 concurrent clients. The results, presented in Figure 9, show that as the workload increases, the link gradient is able to track the changes in response time across all the transactions to within the limits of experimental error. Note that because separate link gradients are computed independently for the various workloads, the predictions do not suffer from any systemic errors due to higher-order effects as a result of increasing workloads.

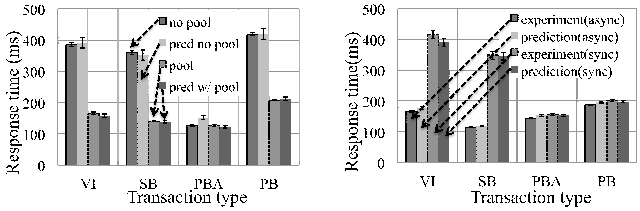
The results show that the link gradient is a useful predictive tool that can be used by system administrators to evaluate alternative component placements under different workloads.

D. Communication Pattern Variations

Real enterprise systems typically have many different types of communication patterns, such as synchronous and asynchronous calls, load balancing, and connection pooling. To

Link	ViewItem	StoreBid	PutBidAuth	PutBid
AS-WS	2.20	0.98	1.69	1.93
DB-AS	2.87	5.08	0.66	5.87

TABLE VII
LINK GRADIENTS WITH CONNECTION POOLING



(a) Connection pooling (b) Replication mode

Fig. 10. Communication pattern effects

be useful for predicting end-to-end response times in new configurations, the link gradients must be able to automatically capture the effects of such variations. Next, we examine the link gradients' ability to do so.

Connection Pooling. *Connection pooling* is a technique used to optimize networked applications by recycling connections shared among different requests, thus altering the application's communication patterns. With connection pooling enabled, requests can use existing connections between the AS and DB rather than initialize a new connection every time. When we computed the link gradients for a workload of 60 clients after enabling connection pooling, we obtained the link gradients shown in Table VII. Comparing the gradients with those for the default setup in Table V, we see, as predicted, that while the gradient on the AS-DB link for the web- and application-server-centric *PutBidAuth* transaction remains relatively unchanged, the link gradients for the other transactions are substantially reduced. Figure 10(a) shows the predicted vs. measured response time results in a new configuration (with AS moved from CMU to UMD) both with and without connection pooling. As can be seen from the figure, the predictions match the measured response times to within error tolerances.

State Replication. Next, we constructed a scenario with two application servers configured to perform passive session-state replication for fault tolerance purposes. The server AS1 is designated as the primary, and WS forwards all the requests to it. The application server AS2 is designated as the backup and only receives state updates from AS1. Tomcat allows replication to proceed either synchronously, such that requests do not return to the caller until state transfer to the backup is complete, or asynchronously, such that requests can return before state transfer is complete. Since RUBiS does not use session state, we modified the *ViewItem* and *PutBid* transactions to exercise this facility, but left the other transactions untouched. Then, we computed link gradients using both synchronous and asynchronous replication modes with WS in UCSD and DB in CMU as before, but with both application servers placed at the same site (Cornell) because of the replication modes' multicast requirements.

Sync	ViewItem	StoreBid	PutBidAuth	PutBid
AS1-AS2	4.18	4.03	0.15	0.11
AS2-AS1	4.68	5.24	0.12	0.35
Async	**ViewItem**	**StoreBid**	**PutBidAuth**	**PutBid**
AS1-AS2	0.14	0.05	0.10	0.11
AS2-AS1	0.03	0.03	0.02	0.04

TABLE VIII
LINK GRADIENTS FOR REPLICATION MODES

Table VIII shows the computed link gradients for the link between the primary and backup Tomcat servers under both synchrony settings and in both directions. As shown, the link gradient can clearly distinguish between the two replication modes for the modified transactions. Furthermore, as expected, the synchronous mode has much larger link gradients than the asynchronous mode. However, the asymmetry between the forward and reverse link gradients on the primary-backup link in the asynchronous mode is puzzling, as it is the only such asymmetry we discovered in the entire application. We speculate that the slight dependency of the response time on the AS1-AS2 link is lock interference between the threads handling the request and the communications thread responsible for sending the session state to the backup replica.

Figure 10(b) shows the predicted vs. measured response times when the link latency between the two application servers was increased by 20 ms. (We could not move the servers to a different location due to multicast requirements.) As can be seen, the predictions match the experimental results quite well for all the transactions, showing that the link gradient is able to accurately capture the effects of different communication patterns without requiring any prior information about them.

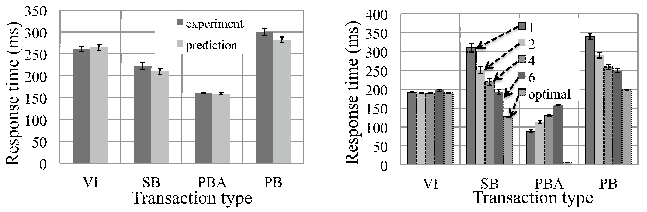
Link	ViewItem	StoreBid	PutBidAuth	PutBid
AS1-WS	1.07	0.60	1.03	0.97
AS2-WS	1.00	0.55	0.76	0.76
DB-AS1	7.17	9.31	0.09	8.58
DB-AS2	7.13	8.96	0.25	8.49

TABLE IX
LINK GRADIENTS WITH LOAD BALANCING

Load Balancing. The last communication pattern we consider is uniform load balancing using Apache's *mod_jk* module across two identical application servers without any state replication. We computed the link gradients for this scenario with WS in UCSD, and with AS1, AS2, and DB at CMU. The results in Table IX show that, compared with the non-replicated case in Table V, link gradients are roughly halved for *all* the links, not just the ones between the WS and ASes. The reason is that the link gradient measures the average effect of changes over time rather than measuring on a per-flow basis. Therefore, the reduction in transaction flows on a link due to upstream load-balancing is reflected as a reduction of the link's impact on the mean response time, thus reducing the link's gradient. Figure 11(a) shows that the predicted vs. measured response times show excellent agreement for all transactions when one of the ASes is moved from CMU to UMD.

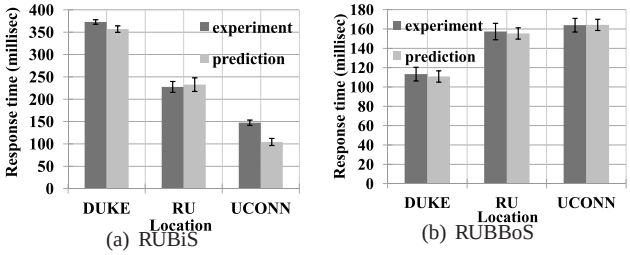
E. Per-Transaction Optimization

Link gradients can be used to optimize a system for responsiveness at transaction granularity without detailed application knowledge. Although different transaction types have different optimal locations for the AS (i.e., close to the clients vs. close to the database), the WS can route transaction types differently.



(a) Uniform load balancing (b) Per transaction load balancing

Fig. 11. Load Balancing Effects



(a) RUBiS (b) RUBBoS

Fig. 12. Predicting the effects of database placement for RUBiS and RUBBoS. (RU is Rutgers and UConn is U. Connecticut)

Therefore, we can obtain an optimized configuration by having *both* local and remote AS copies, and using the link gradient to choose which transactions are to be routed to each server. Transactions whose WS-AS gradients are higher than their AS-DB gradients are routed to the local server (the *PutBidAuth* transaction), while all other transactions are routed to the remote server.

We conducted experiments using such a setup, with the WS and AS1 at UCSD, and the DB and AS2 at CMU. For comparison, we also conducted experiments in which all requests were load-balanced between the two servers without regard to the transaction type, but using various load-balancing ratios (the best that can be done without application-specific information). Figure 11(b) shows the measured response time when all the configurations were subjected to a 60-user workload. In the figure, a configuration *n* means that the requests were routed to the local and remote servers with a ratio of 1:*n*. The results show that no constant load-balancing ratio can achieve the optimal results that are achieved through the per-transaction load-balancing based on link gradients. In particular, the link gradients revealed that *PutBidAuth* transactions only use the WS and the AS. Hence, all such transactions could be routed to the local AS, thus resulting in a dramatically smaller response time compared to any other static load-balancing ratio.

F. Optimization for Multiple Applications

Link gradients can also be used to optimize deployments consisting of multiple applications that share components. We consider the example of ACDNs [5]. ACDNs are a proposed extension of Content Distribution Networks (CDNs), which "cache" dynamic content by moving application components closer to end users in order to improve responsiveness. Consider a scenario in which a company has two applications that share a common back-end database. To optimize the response time for static content, it is willing to use third-party cloud computing solutions to host the front-ends close to the clients, but for privacy and security reasons, it wants to host the database in its own data centers. The decision on where to place the database can be affected by how each application uses the database (e.g., synchronous, asynchronous, stored procedure calls, caching or connection pooling) and the application workloads relative to each other. Link gradients can be used to determine at which of many possible locations the back-end should be placed so as to maximize the responsiveness across both applications.

To demonstrate that scenario, we used RUBiS and the PHP version of RUBBoS [2], a bulletin board benchmark, as the two applications. We deployed the front-ends for RUBiS (WS and AS) at MIT and for RUBBoS (WS only) at USF (University of South Florida), and considered 3 different locations for the back-end database. The response time at each of these possible locations was predicted using the link gradients measured on each application running independently, with a workload of 20 clients each. The link latencies associated with the candidate configurations were measured in the field. Figure 12 shows the measured average response time over all transactions and the predicted response time over all transactions for each application, with the database at each of the possible locations.

The results show that the Java-servlet-based RUBiS application is much more sensitive to database server placement than the PHP-based RUBBoS application is (because PHP processing in the web server is a bottleneck for RUBBoS). The average predicted response times using the relative application importance as weights can be used to make a decision on the best location. More importantly, the results also show that link gradient measurements conducted in isolation for each application can be used to predict, with good accuracy, the response times in shared scenarios. The only experimental measurement that did not fall within the confidence interval for the prediction was for RUBiS when the DB was located at the University of Connecticut. The reason is that the PlanetLab node used at Connecticut was a much slower machine, with a processor speed of 2.0 GHz compared to the 3.0 GHz machines used in all the other nodes.

By predicting per-transaction response times, the gradients also support reconfiguration of the system if user behavior (i.e., the fractions of the transaction types) changes. Although the opportunities for optimization in these applications are limited, the results pave the way for link gradients to be used for response time optimization in larger systems together with search algorithms that allow a systematic exploration of the space of a large number of deployment configurations.

G. Discussion

Although the link gradient is an important factor when predicting the impact of network changes on end-to-end application responsiveness, it is not the only one. When other factors, like machine speed, bandwidth, and loss rates, are drastically changed, link gradients alone might not be enough. We believe that the techniques we have proposed to measure the impact of link latency can be readily extended to these other factors as well, and used to define a class of similar metrics, such as CPU and loss rate gradients. We will do so in future work.

In addition, as factors such as timeouts and increased queuing delay can come into play due to large changes in link latency in highly congested systems, the linearity assumption of the technique might not hold. However, such conditions are usually exceptions rather than the norm in typical well-provisioned enterprise systems. Furthermore, by increasing the magnitude of the square wave delay injected into the links during the measurement phase and looking for changes in the link gradient (compared to the normal delay injection), our approach could detect the latency range over which the linearity assumptions hold and the response time predictions are likely to be accurate. If large-magnitude square waves are to be injected into the system, perturbation can be minimized by reducing the duty cycles of the waveforms (i.e., injecting short spikes of delay rather than square waves).

In spite of those caveats, the results show that link gradient by itself has excellent predictive value even in a highly heterogeneous and dynamic environment such as PlanetLab (compared to which most enterprise environments are far more stable), and thus we believe that the tool is an invaluable one for system and network administrators in its current form.

VII. Related Work

While we are not aware of any work directly comparable to ours, there has been work on measuring and profiling different aspects of distributed applications for debugging, optimization, modeling, and failure diagnosis. For example, [6] evaluates the impacts of communication overhead and network latency on the performance of parallel applications by running benchmark applications in a controlled experimental platform. [7] examine the impact of variations in WAN conditions on different topologies for web-service-based service oriented architectures and use a bandwidth-based network performance model to choose the best topology. [8] examine the impact of WAN latency on various remote desktop protocols using slow-motion benchmarking. The conclude that latency considerations are more important than bandwidth in those types of systems.

Critical path analysis for parallel program execution was introduced in [9] and extended to an on-line version in [10]. While the critical path can be used to guide debugging and performance optimization in parallel programs, it cannot realistically be used to predict the impact of network latency change on the response time of multitier services.

Causal paths indicate how end-user requests flow through system components, and have been used to understand and analyze distributed applications' performance and to identify bottlenecks [11]. A number of techniques for determining causal paths have been proposed [11], [12], [13], [14], [15], [16], [17], each with its own advantages and disadvantages in terms of assumptions on communication patterns, accuracy, and execution cost. Link gradients intrinsically capture behavior that is very different from causal paths. For example, load balancing causes only local changes in a probabilistic causal graph, while it affects the link gradients of all downstream nodes as shown in Section VI. It is conceivable that placement of some restrictions on communication patterns might make it possible to use causal paths to compute the "mean number of message crossings in the path of the system response" (described in Section III), and thus approximate link gradients. However, causal paths cannot capture the effects of increased link latency on other parts of the system (e.g., queuing) and thus cannot measure link gradients exactly. Of the literature on the determination of causal paths, only Magpie [13] collects enough information about resource usage along paths that detailed response time modeling might be attempted. However, the need for extensive (albeit lightweight) instrumentation precludes Magpie's use by hosting providers, such as AT&T, that often do not have the required access to the applications they host.

Methods for identifying dependencies between system components have been developed to help in intrusion detection, failure diagnosis and fault localization in complex systems. Service-level dependencies in large enterprise networks were identified by use of passive observation and statistical techniques [18], [19]. Failure dependencies for multitier systems were addressed in [20] and [21] using instrumentation techniques. However, failure dependencies are largely orthogonal to link gradients. The fact that a strong failure dependency exists between components does not imply anything about the link gradient. For example, a server's response time is typically not affected by its DHCP server, even though it may have a failure dependency on it. Conversely, nonzero link gradients do not necessarily imply failure dependencies. For example, a server with an active backup may have a non-zero gradient to the backup, but no failure dependency.

Signal-injection-based techniques have been used by others, mostly for determining failure dependencies. The ADD (Active Dependency Discovery) technique determines failure dependencies by active perturbation of system components and observation of their effects [20]. The ADD approach is generic and does not specify the perturbation and effect measurement methods. In [21], the ADD approach is used with fault injection as the perturbation method. The Automatic Failure-Path Interference (AFPI) technique combines pre-deployment failure injection with runtime passive monitoring [22]. While our technique could be seen as a special case of ADD, our technique is far less disruptive to the service provided and can thus be used in running production systems. Delay injection for disk and network access events is used in [15] to verify causal dependencies between such events in a component-based system. Specifically, [15] uses this technique to determine the object read and write policies in a commodity-based commercial storage cluster. However, the technique is strictly off-line and requires full control of the system workload (including message sizes, types, and frequency).

Finally, Fourier analysis has been used by others to detect periodic behavior in network routing updates [23], to reduce random noise from delay distribution [19], and to assist in the fields of system identification [24] and network tomography [25]. However, we are not aware of any other work that uses the specific combination of signal injection and Fourier analysis to improve measurement accuracy in software systems.

VIII. Conclusions

In this paper, we introduced the link gradient, a new metric that can be used to approximate how the application response time changes with changes in logical link latencies. We presented a novel technique for computing link gradients using signal injection and Fast Fourier Transforms, and described a framework for computing the link gradients of a running multitier enterprise application in a push-button manner without requiring any information about the structure of the application. Finally, we also demonstrated the efficiency and accuracy of our technique using distributed deployments of the RUBiS and RUBBoS applications on PlanetLab and showed that link gradients can be used to predict response times for new and untested configurations.

References

[1] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of EJB applications," in *Proc. OOPSLA'02*, 2002, pp. 246–261.

[2] ObjectWeb Consortium, "RUBBoS: Bulletin board benchmark," http://-jmob.objectweb.org/rubbos.html, Feb 2005.

[3] G. Varghese and A. Lauck, "Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility," *IEEE/ACM Trans. on Networking*, vol. 5, no. 6, pp. 824–834, 1997.

[4] Transaction Processing Performance Council, "TPC benchmark W (web commerce) specification, v.1.8," www.tpc.org/tpcw/spec/tpcw_V1.8.pdf, Feb 2002.

[5] M. Rabinovich, Z. Xiao, and A. Aggarwal, "Computing on the edge: A platform for replicating internet applications," in *Proc. 8th Int. Workshop on Web Content Caching and Distribution*, Sept 2003.

[6] R. Martin, A. Vahdat, D. Culler, and T. Anderson, "Effects of communication latency, overhead, and bandwidth in a cluster architecture," in *Proc. ISCA'97*, 1997, pp. 85–97.

[7] G. Chafle, S. Chandra, N. Karnik, V. Mann, and M. Nanda, "Improving performance of composite web services over wide area networks," in *Proc. 2007 IEEE Congress on Services (SERVICES 2007)*, July 2007, pp. 292–299.

[8] A. Lai and J. Nieh, "On the performance of wide-area thin-client computing," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 2, pp. 175–209, May 2006.

[9] C.-Q. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Proc. 8th Int. Conf. on Distributed Computing Systems*, 1988, pp. 366–373.

[10] J. Hollingsworth, "Critical path profiling of message passing and shared-memory programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 1029–1040, 1998.

[11] B. Miller, "DPM: A measurement system for distributed programs," *IEEE Trans. on Computers*, vol. 37, no. 2, pp. 243–248, 1988.

[12] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. DSN'02*, June 2002, pp. 595–604.

[13] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in *Proc. OSDI'04*, Dec 2004, pp. 259–272.

[14] M. Aguilera, J. Mogul, J. Weiner, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *Proc. SOSP*, Oct 2003, pp. 74–89.

[15] H. Gunawi, N. Agrawal, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and J. Schindler, "Deconstructing commodity storage clusters," in *Proc. ISCA'05*, June 2005, pp. 60–71.

[16] P. Reynolds, J. Wiener, J. Mogul, M. Aguilera, and A. Vahdat, "WAP5: Black-box performance debugging for wide-area systems," in *Proc. Int. WWW Conf.*, May 2006, pp. 347–356.

[17] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, "E2EProf: Automated end-to-end performance management for enterprise systems," in *Proc. DSN'07*, June 2007, pp. 749–758.

[18] P. Bahl, R. Chandra, A. Greenberg, S.Kandula, D. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *Proc. SIGCOMM'07*, Aug 2007.

[19] X. Chen, M. Zhang, M. Mao, and P. Bahl, "Automating network application dependency discovery: Experience, limitations, and new solutions," in *Proc. OSDI'08*, Dec 2008, pp. 117–130.

[20] A. Brown, G. Kar, and A. Keller, "An active approach to characterizing dynamic dependencies for problem determination in a distributed environment." in *Proc. 7th IFIP/IEEE Int. Symp. on Integrated Network Management (IM 2001)*, May 2001, pp. 377–390.

[21] S. Bagchi, G. Kar, and J. Hellerstein, "Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment," in *Proc. 12th Int. Workshop on Distributed Systems: Operations & Management*, Oct 2001.

[22] G. Candea, M. Delgado, M. Chen, and A. Fox, "Automatic failure-path inference: A generic introspection technique for internet applications," in *Proc. 3rd IEEE Workshop on Internet Applications (WIAPP)*, Jun 2003.

[23] C. Labovitz, R. Malan, and F. Jahanian, "Internet routing instability," in *Proc. SIGCOMM '97*, 1997, pp. 115–126.

[24] R. Pintelon and J. Schoukens, *System Identification: A Frequency Domain Approach*. IEEE, Inc, 2001.

[25] A. Chen, J. Cao, and T. Bu, "Network tomography: Identifiability and fourier domain estimation," in *Proc. INFOCOM'07*, May 2007, pp. 1875–1883.

Shuyi Chen is Ph.D. student at the University of Illinois at Urbana Champaign. He received the B.S. degree from the Peking University, and the M.S. degree from University of Illinois at Urbana Champaign. Shuyi Chen is an IEEE student member. His research interests include application tomography and adaptive systems.

Kaustubh R. Joshi is a researcher at AT&T Labs-Research in Florham Park, NJ. He received the B.Eng degree from the University of Pune, and the M.S. and Ph.D. degrees from the University of Illinois at Urbana-Champaign. Dr. Joshi is a member of the ACM and the IEEE Computer Society. His research interests include adaptive systems, dependable computing, cloud computing, and the use of online probabilistic models to enhance system performance, dependability, and security.

Matti Hiltunen is a researcher at AT&T Labs-Research in Florham Park, NJ. He received the M.S. degree in computer science from the University of Helsinki and the Ph.D. degree in computer science from the University of Arizona. Dr. Hiltunen is a member of the ACM, the IEEE Computer Society, and IFIP Working Group 10.4 on Dependable Computing and Fault-Tolerance. His research interests include dependable distributed systems and networks, cloud computing, and adaptive systems.

William H. Sanders is a Donald Biggar Willett Professor of Engineering, the Director of the Information Trust Institute, and Acting Director of the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. He is a professor in the Department of Electrical and Computer Engineering and Affiliate Professor in the Department of Computer Science. He is a Fellow of the IEEE and the ACM. Dr. Sanders' research interests include performance/dependability evaluation, dependable computing, and reliable distributed systems, and he has published more than 200 technical papers in these areas. He is currently the Director and PI of the NSF/DOE/DHS Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) Center.

Richard D. Schlichting is Executive Director of Software Systems Research at AT&T Labs-Research. He received the B.A. degree from the College of William and Mary, and the M.S. and Ph.D. degrees from Cornell University. Dr. Schlichting is an ACM Fellow and an IEEE Fellow, and is the current chair of IFIP Working Group 10.4. His research interests include distributed systems, highly dependable computing, and networks.