

# Link Gradients: Predicting the Impact of Network Latency on Multi-Tier Applications

Shuyi Chen<sup>†</sup>   Kaustubh R. Joshi<sup>‡</sup>   Matti A. Hiltunen<sup>‡</sup>  
William H. Sanders<sup>†</sup>   Richard D. Schlichting<sup>‡</sup>

<sup>†</sup>Coordinated Science Laboratory  
University of Illinois at Urbana-Champaign  
Urbana, IL, USA

{schen38, whs}@crhc.uiuc.edu

<sup>‡</sup>AT&T Labs Research  
180 Park Ave.  
Florham Park, NJ, USA

{kaustubh, hiltunen, rick}@research.att.com

## Abstract

Geographically dispersed deployments of large and complex multitier enterprise applications introduce many challenges, including those involved in predicting the impact of network latency on end-to-end transaction response times. Here, a measurement-based approach to quantifying this impact using a new metric called the *link gradient* is presented. A nonintrusive technique for measuring the link gradient in running systems using delay injection and spectral analysis is presented, along with experimental results on PlanetLab that demonstrate that the link gradient can be used to predict end-to-end responsiveness, even in new and unknown application configurations.

## 1 Introduction

The emergence of utility and cloud computing services has made it easy for companies of all sizes to have a global presence by distributing services and applications in multiple data centers geographically dispersed around the world. While offering many benefits, such distributed deployments introduce many challenges [13], including management of end-to-end response time in an environment where network latencies between application components can be significant. The main issue here is not so much in determining the raw latency of the logical link from site A to site B, which can easily be measured using existing techniques, but rather in understanding the way in which end-to-end application behavior is affected by that latency. The relationship is complex and can vary depending on the components that the link connects, workloads, configuration settings, and transaction types.

To illustrate the challenge, consider a 3-tier deployment of the RUBiS web service benchmark [8] on PlanetLab, as shown in Figure 1. In this simple yet representative configuration, the web server is located near users in California, while the database is located in New

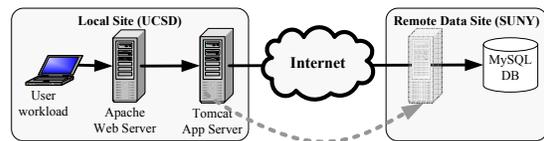


Figure 1: Sample RUBiS configuration

York. Table 1 shows the impact on transaction end-to-end response time associated with moving the application server from being colocated with the web server (*local*) to being colocated with the database (*remote*). As can be seen, the impact on even this simplest of configurations is not only significant, it varies dramatically based on the transaction type.

Transaction	Local (s)	Remote (s)	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 1: Average response time for RUBiS transactions

While the behavior of RUBiS in this specific instance might seem obvious based on knowledge of how transactions access the database, such insights are much more difficult in large and complex multitier enterprise applications of the type hosted and managed by AT&T for Fortune 500 companies. Such applications are significantly more complicated, since they may have, for example, replicated web, application, and authentication servers and databases, as well as load balancers, caches, firewalls, and traffic filters. The relationship between link latency and end-to-end performance may therefore be even more complex and transaction-specific, depending not only on the pattern of message exchange between components, but also on factors such as caching policies, communication semantics (e.g., synchronous vs. asynchronous), replication strategies (e.g., active vs. primary-backup), connection pooling strategies, and workload. Moreover, alternative approaches, such as those based on

queuing models, are largely untenable, given the inherent difficulty in constructing models of the application at the requisite level of detail. Quantifying the relationship between link latency and end-to-end performance is an important step towards improving overall system management, since it provides crucial information that can be used for prediction, evaluation, and optimization.

In this paper, we present a measurement-based approach to quantifying the network’s impact on end-to-end response time for distributed applications, with a special focus on multitier enterprise applications. The key concept behind this approach is the *link gradient*, a new metric for quantifying the rate at which an application’s transaction response time changes with changing link latency. In addition to introducing the link gradient, our work makes two specific contributions:

- An innovative technique using delay injection and spectral analysis that can measure link gradients in a running system with minimal intrusiveness.
- An experimental evaluation that demonstrates how the link gradient can predict end-to-end responsiveness, even in new and unknown configurations.

Our approach has key advantages over the alternatives. First, our techniques can isolate the effects of network latency on individual transactions even if they use the same servers and hardware resources, making it particularly useful in highly shared virtual machine or virtual hosting environments. Second, because it does not require information about system internals, makes no assumptions about system architecture, does not require modification of the target system, and does not need synchronized clocks, measurement of the link gradient is easy for both designers and infrastructure providers alike. Third, the spectral techniques we propose ensure that intrusiveness is kept to a minimum, and in fact, we speculate that the link gradient measurements can be performed continuously even on production systems. That removes the need for an isolated measurement environment and ensures that measurements can be collected using any workload to which the target system might be subjected. Finally, the metric is simple, is compactly expressed, and can easily be used in conjunction with other system metrics. For example, it can be used by designers to ascertain server placements that improve availability while ensuring that responsiveness remains within a tolerable threshold, or by network operators to evaluate the cost versus benefit trade-offs of planned network upgrades.

## 2 Technical Approach

In this section, we define the link gradient of a system, and describe how it can be used to predict the effect of link latency changes on a transaction’s response time. Then, we explain how spectral analysis can allow the

computation of a link gradient with only small perturbations to the system and derive the required formulae.

### 2.1 The Link Gradient

Consider a multitier application consisting of multiple software components and let  $C = \{c_1, c_2, \dots, c_n\}$  be the set of  $n$  logical one-way communication links between them<sup>1</sup>. Each link  $c_i$  is parameterized by a link latency  $l_i$ , which in most cases is the same for the forward and backward links and can be measured by a ping test between the two components. Finally, let the system’s performance metric be quantified using its end-to-end mean response time  $\bar{r}t$ . 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<sup>2</sup>.

Then, the *link gradient*  $\vec{\nabla} \bar{r}t$  quantifies how a change in the link latency for each link affects the response time, and is defined as a vector  $\vec{\nabla} \bar{r}t = \left( \frac{\delta \bar{r}t}{\delta l_1}, \dots, \frac{\delta \bar{r}t}{\delta l_n} \right)$ , where each element  $\nabla \bar{r}t_i = \frac{\delta \bar{r}t}{\delta l_i}$  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  $\Delta l = \{\Delta l_1, \dots, \Delta l_n\}$  represents the amounts by which each link latency changes, the new response time of the system can be approximated by its Taylor expansion:

$$\bar{r}t(\vec{l} + \vec{\Delta}l) \approx \bar{r}t(\vec{l}) + \vec{\nabla} \bar{r}t \cdot \vec{\Delta}l + O(\vec{\Delta}l^2) \quad (1)$$

This equation makes two simplifying assumptions. First, it assumes that the response time (as a function of link latency) does not have any discontinuities. 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  $\Delta l$  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 argue that many systems have large regions of continuous linear relationships in which the higher-order effects can be ignored.

<sup>1</sup>Two-way communications are represented by two independent links, one for each direction.

<sup>2</sup>Although we focus on end-to-end response time, other system metrics whose dependency on link latency is to be quantified can also be used without much modification to the algorithm.

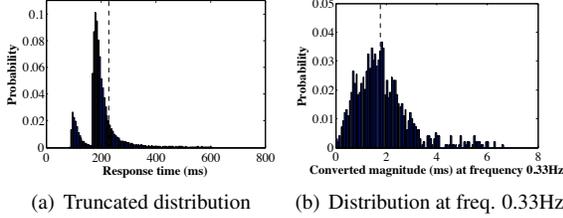


Figure 2: Response Time Distributions

To make the argument, we examine an approximate interpretation of link gradients in terms of message crossings. Consider a node  $a$  that calls another node  $b$  over a link  $c$  that has a latency of  $l$ . The amount by which the mean response time of node  $a$  would change if the link latency changes to  $l + \Delta l$  is dictated to a large extent by the type of communication. For instance, if  $a$  sends a message to  $b$  and waits for a reply before continuing, the response time could be expected to increase by  $\Delta l$ , and the link gradient would be  $\lim_{\Delta l \rightarrow 0} \frac{\bar{r}t(l+\Delta l) - \bar{r}t(l)}{\Delta l} = 1$ . The link gradient would remain the same if  $a$  sent  $m$  messages to  $b$  in a pipelined fashion (e.g., over a TCP link) before waiting for a response. 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  $b$ , 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. However, in reality, factors such as timeouts and nonlinearity due to queuing affect this argument when very large latency changes are considered. We outline ways to tackle nonlinearity in more detail in Section 6, but nevertheless show in Section 5 that the linearity assumption holds quite well in practice.

## 2.2 Link Latency Perturbation

Conceptually, link gradient can be easily computed at run-time by active delay injection. A target communication link can be chosen, and a delay  $\Delta l_i$  can be systematically injected in the packets traversing the link. The end-to-end response time of the system  $\bar{r}t'$  can then be measured while the delay is being injected and compared with the nominal response time  $\bar{r}t$ . The ratio  $\frac{\bar{r}t' - \bar{r}t}{\Delta l}$  is then an approximation to the link gradient  $\vec{\nabla} \bar{r}t_i$ . 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, 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, a standard deviation of 269 ms, and has a long tail (not shown) reaching up to 12s. Even with these many samples, the 95% confidence interval for the mean is  $\pm 3.17$  ms, implying that to detect a change in the mean with 10% error, the mean would have to change by at-least 63.4 ms. Injecting very high delays into a running system is not only disruptive, but can also cause the system dynamics themselves to change, making the measurement inaccurate. Due to the large number of samples, one runs the risk of the system changing behavior during sample collection itself. Moreover, it is not trivial to inject low variance delays into communication links.

To solve this 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 get precision that is significantly superior to a time domain method. To illustrate, consider Figure 2(b) which shows a distribution of the magnitude of a randomly chosen frequency component (0.33Hz) in the same response time data as Figure 2(a). However, 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), more symmetric, and has a standard deviation of 1.1 ms. The corresponding 95% confidence interval of  $\pm 0.07$  ms allows a change of 1.4 ms to be detected with 10% error.

We perform such spectral analysis using the Fast Fourier Transform (FFT). The basic approach is illustrated in Figure 3. The first graph, Figure 3(a), shows two response time series for one of the transactions of a toy two-tier application that we built for experiments. The dashed time series was snapshot of transaction response time over a window of 128 sec. To examine the time series in the frequency domain, we compute the power spectrum of the waveform using a Fast Fourier Transform (FFT). The power spectrum shows the energy present in the waveform as a function of frequency. From the spectrum shown in Figure 3(b), it can be seen that most of the energy is present in the 0<sup>th</sup> frequency, which represents the non periodic portion of the waveform. The

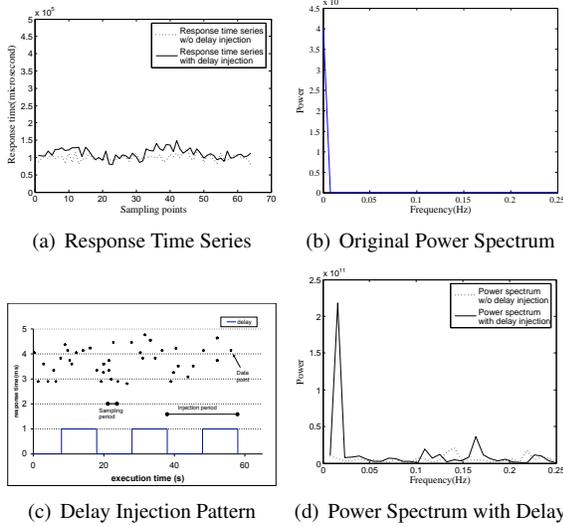


Figure 3: Power Spectra under Delay Injection

energy content at other frequencies is very low. Therefore, if the delay is injected in a periodic manner at any of those frequencies, its magnitude would not have to be very large.

Figure 3(c) shows how a periodic delay might be introduced into the link. The figure shows 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. The (hypothetical) data points above the square wave show how the injected delay might affect the response time of the system. We then actually injected such a delay with a magnitude of 23 millisecond, and frequency of 0.015625 Hz into one of the links, measured response times and convert them into a response time series sampled at frequency 0.5Hz. The resulting response time series is shown by the solid line in Figure 3(a). The corresponding power spectrum is shown in Figure 3(d). Since it is very large, the 0 frequency term of the power spectrum is omitted. As can be seen from the response time series, the amount of delay injected is small compared to the original response time and its standard deviation. Nevertheless, the spike at the injection frequency of 0.015625 Hz in the power spectrum clearly shows that the effect of the injected delay dominates the other frequency components and allows an accurate measurement of the injected delay. Comparing the power spectrum with the original spectrum of Figure 3(b), one can clearly see that the 0 frequency contains two orders of magnitude more power than the injection frequency component.

## 2.3 Spectral Link Gradient Computation

Next we develop formulae to show how the height of the spike in the power spectrum can be used to compute the

link gradient of the link. To do so, we consider some arbitrary response time series  $x_i, i = 0 \dots N - 1$  of length  $N$  measured at uniform intervals of time  $\Delta T_s$ , called the *sampling interval*. Using the values of  $x_i$ , the Discrete Time Fourier Transform can be computed at all frequencies  $f_k = \frac{k}{N \cdot \Delta T_s}, k = 0 \dots N - 1$  (i.e., those frequencies that fit an integral number  $k$  of cycles into the time series duration) using the equation:

$$\text{FFT}(f_k) = \sum_{i=0}^{N-1} x_i e^{-\frac{2\pi j}{N} i k} \quad (2)$$

Now consider a delay added to each response time in a square-wave pattern with magnitude  $A_d$  and frequency  $f_d = \frac{k_d}{N \cdot \Delta T_s}$  for some  $k_d$ . Then, using standard tables, we can show that the Fourier Transform  $\text{FFT}(f_k)$  of the resulting time series is given by

$$\text{FFT}^d(f_k) = \frac{k_d A_d}{\sin(\frac{\pi}{2n})} e^{j(\frac{\pi}{2n} - \delta)} + \text{FFT}^0(f_k) \quad (3)$$

In this equation,  $\text{FFT}^0(f_k)$  is the complex-value Fourier Transform of the original response time series without any delay injection,  $\delta$  is the phase shift of the square wave in comparison with the time series interval, and  $2n = N/k_d$ . In other words,  $2n$  is the total number of data points in each square wave cycle. Because the magnitude of the complex number  $e^{j(\frac{\pi}{2n} - \delta)}$  is one, by manipulating Equation 3, we obtain an expression for the amount of delay introduced into the waveform:

$$A_d = \frac{|\text{FFT}^d(f_k) - \text{FFT}^0(f_k)| \cdot \sin(\frac{\pi}{2n})}{k_d} \quad (4)$$

However, we also know from Equation 1 that if the link latency  $l$  of a link is increased by  $\Delta l$ , then the change in response time is given by  $\bar{r}t(l + \Delta l) - \bar{r}t(l) \approx \frac{\delta \bar{r}t}{\delta l} \cdot \Delta l$ . Equating this change in response time to the delay  $A_d$  measured from the frequency spectrum and Equation 4, we obtain the link gradient  $\frac{\delta \bar{r}t}{\delta l}$  as a function of the system response time series without and with a square wave delay injection of magnitude  $\Delta l$ :

$$\frac{\delta \bar{r}t}{\delta l} = \frac{|\text{FFT}^d(f_k) - \text{FFT}^0(f_k)| \cdot \sin(\frac{\pi}{2n})}{\Delta l \cdot k_d} \quad (5)$$

An important point to note is that the phase shift  $\delta$  of the square wave does not appear in the equation. That means that the square wave injector does not need a clock that is synchronized with the response time measurement.

## 3 Architecture and Implementation

We have implemented a monitoring framework that automatically calculates the link gradients for a distributed

application. This framework can target one or more distributed applications deployed on a set of machines. Figure 4 illustrates the framework with one application. The framework consists of a central coordinator and a set of *sniffer* and *delay* daemons.

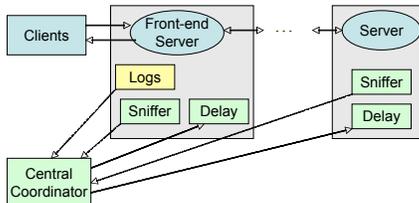
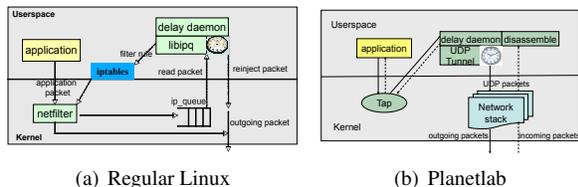


Figure 4: Monitoring architecture

The sniffer daemon collects and reports information about communication links on which at least one message has been exchanged during the reporting interval<sup>3</sup>. Each communication link is defined using a 5-tuple  $\langle \text{Protocol, SourceNode, SourcePort, DestinationNode, DestinationPort} \rangle$ , where the protocol is either TCP or UDP, and each node is represented using a tuple  $\langle \text{HostIP, ProcessID} \rangle$ . When a node uses a temporary port number (from the ephemeral range, 32768 to 61000 in Linux), we replace that port number with port number 0 to avoid spurious creation of multiple records when the same communication link is instantiated multiple times.



(a) Regular Linux (b) Planetlab  
Figure 5: Delay daemon implementations

The delay daemon implements delay injection, and starts and stops injection based on commands from the coordinator. We have implemented two versions of the delay daemons. In Section 4, we present benchmark results for these two injection mechanisms that quantify their accuracy.

The first version is based on the loadable packet filter kernel module in the regular Linux kernel, as illustrated in Figure 5(a). We utilize the `ip_queue` packet filter kernel module and `libipq`<sup>4</sup> to queue packets going through a link, delay them, and reinject them back to the network stack. The delay daemon controls the `ip_queue` module by using `iptables`<sup>5</sup>. It starts (or stops) the injection by installing (or removing) an `iptables` rule that forwards all packets on a designated com-

<sup>3</sup>A threshold higher than 1 can be specified.

<sup>4</sup>`libipq` is a development library for `iptables` userspace packet queuing that provides an API for communicating with `ip_queue`.

<sup>5</sup>`Iptables` is a userspace command line program used to configure the Linux packet-filtering ruleset.

munication link to the in-kernel packet queue. The userspace daemon then uses the `libipq` library to read the metadata of the queued packets from `ip_queue` and reinject the packet back to the network stack after the specified amount of delay. Since the daemon only reads fixed-size metadata (72 bytes long) from the kernel, the efficiency of the mechanism is independent of the size of the packet, and the overhead introduced due to data copying from kernel to userspace is small. The delay daemon uses a timer-wheel implementation [17] with which it maintains a queue of packets scheduled to be sent in order of their send times.

The second version is specific for PlanetLab, since PlanetLab uses an experimental Linux kernel that has the packet filter module disabled. We implement the delay daemon for PlanetLab machines using the `tap` device and UDP tunneling. `Tap` simulates an Ethernet device and operates with Layer 2 packets, such as Ethernet frames. On PlanetLab nodes, `VNET`<sup>6</sup> emulates a single TAP interface, `tap0`. Each slice may access its own packets by reading from and writing to the `tap0` interface via IP or packet sockets. The implementation is illustrated in figure 5(b). We configure applications to send and receive packets via the `tap0` device. The delay daemon listens on `tap0` device and reads from it the applications' packets (including the Layer 2 header). Then the daemon removes the packets' Layer 2 header, modifies the Layer 3 header, delays the packets using the timer-wheel technique and sends the entire Layer 3 packets as payload via a UDP tunnel to the packet's destination. On receiving a packet via the UDP tunnel, the daemon disassembles the packet, removes the UDP header, modifies the Layer 3 header, and writes the payload to the `tap0` device. The user application will then receive the packets from `tap0`. This implementation has some limitations not present in our `ip_queue` module. First, the daemon has to read the entire packet payload to the user space, which introduces variable copy overhead that depends on the packet size. Also, it requires applications to listen on the `tap` device and requires the delay daemon to run on every node in the system, while the `ip_queue` approach does not have this limitation.

To correct for inaccuracies in the timer mechanism caused by the standard clock tick interval of 1ms in the unmodified 2.6 Linux kernel series, the delay daemon measures the amount of delay actually injected into each packet, and reports the mean back to the central coordinator to allow correction.

The central coordinator instructs the delay daemon to start or stop delay injection, and periodically executes the measurement algorithm to measure the link gradients for the target application. The user provides the coordi-

<sup>6</sup>`VNET` is a PlanetLab module that provides virtualized network access on PlanetLab.

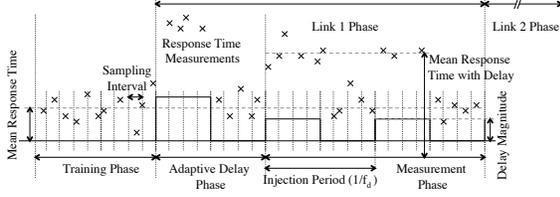


Figure 6: Timeline for the measurement process

nator with a list of nodes (host, process names) that are part of the application, and a target metric. Currently, the framework supports response time as a target metric, and has a module that can parse Apache web server logs to extract response time data for a specified type of URL. Using this module, any application that has a web-based interface can be supported directly, and the link gradients for any individual transaction (URL) can be constructed. The link gradients are measured using only existing application workload response time series extracted from the web logs. No additional workload is required for measurement purposes, and thus, the technique introduces minimal interference to a running system.

## Measurement Algorithm

The process of measuring the link gradients consists of two phases: a) the training phase, and b) a set of measurement phases. In the training phase, the coordinator collects communication links, and passively collects the system response times for the transaction 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 measurement phase is conducted once for every communication link, and is the active phase where a delay is introduced into the target link and the system response is measured. Figure 6 presents a timeline of the entire process.

**Training Phase:** The training phase is used to estimate parameters for the active measurement phases. We assume that the transaction request rates and other characteristics remain stable for the duration of the training and measurement phases for a link to ensure parameters learned in the training phase are valid in the measurement phase. The assumptions are reasonable since the time needed to extract the link gradient for a communication link is a few minutes for a practical system, and we do not expect the measurement to be done during periods of rapidly changing traffic. Although not required, we assume that these assumptions hold for the entire duration of the link gradient measurement. That allows us to conduct only a single training phase. The training phase comprises the following steps.

**Step 1: Binning:** Since the application’s own workload is used to perform the link gradient measurement,

the measurement framework has no control over when response time measurements (data points) are collected. To obtain the periodic time series required for the Fourier Transform, the framework creates bins of length equal to a *sampling interval* as shown in Figure 6, and uses the mean of all the data points within a bin as a single sample point in the time series. To choose the length of the sampling interval, the coordinator records the mean  $\bar{m}_t$  and standard deviation  $\sigma(m)_t$  of the time interval for  $k$  consecutive requests. The sampling interval is then chosen as  $\Delta T_s = \bar{m}_t + 3\sigma(m)_t$  to ensure that at least  $k$  points are averaged in each sampling interval with a high probability (0.999 if the time between  $k$  requests has a normal distribution). The parameter  $k$  that governs the number of data points per bin is provided as an input by the user. Once the bin size is computed, the framework can construct a response time series from the web server log files.

**Step 2: Compute FFTs:** To choose an appropriate frequency and magnitude for delay injection, it is important to examine the characteristics of the frequency domain representation of the system’s response time series without perturbation. To do so, the coordinator divides the response time series into  $M$  (we use  $M = 9$ ) different chunks of  $N$  sampling points each. The parameter  $N$  is a user-specified parameter that indicates the number of sampling points used in each link gradient computation. In our implementation,  $N$  is restricted to be a power of 2 so that a fast radix 2 FFT algorithm can be used. The Fourier Transform for each of the  $M$  chunks is then computed, and these transforms are then averaged to obtain the mean  $\overline{\text{FFT}}^0(f)$  and standard deviation  $\sigma(\text{FFT}^0)(f)$  (both complex numbers) of the Fourier Transform at each frequency  $f = \frac{k}{N \cdot \Delta T_s}$ . These can be used to select both an appropriate delay magnitude and a frequency.

**Step 3: Compute Magnitude:** To choose a delay magnitude, imagine that a delay frequency  $f_d = \frac{k_d}{N \cdot \Delta T_s}$ ,  $k_d \in \{0, \dots, N - 1\}$  has been chosen, and a delay of magnitude  $A_d$  has been injected. Then, according to Equation 3, the Fourier Transform of the resulting response time series is equal to  $\text{FFT}^d(f_k) = \frac{k_d A_d}{\sin(\frac{\pi}{2n})} e^{j(\frac{\pi}{2n} - \delta)} + \text{FFT}^0(f_k)$ . However, because they are measured at different times, the value  $\text{FFT}^0(f_k)$  that appears in this equation can be different from the value  $\text{FFT}^0(f)$  computed in step 2 above. To reduce the relative error due to this mismatch, we make it proportional to the standard deviation  $A_d = d * \sigma(\text{FFT}^0)(f)$  computed in the previous step (which is a measure of the natural variability of the Fourier Transform components). The constant of proportionality  $d$  is called the *delay scale factor*, and is set to 30 in all our experiments. We examine the sensitivity of the measurement technique to the delay scale factor in section 4.

**Step 4: Compute Frequency:** Given the method to compute the delay magnitude for a particular frequency, the coordinator simply computes the delay magnitude for all frequencies whose periods are factors of the measurement phase length (i.e.,  $f_d = \frac{k_d}{N \cdot \Delta T_s}$ ,  $k_d \in \{0, \dots, N - 1\}$ ), and picks the frequency that would result in the smallest delay magnitude (to minimize perturbation).

**Measurement Phase:** Given the bin size  $\Delta T_s$ , delay magnitude  $A_d$ , and injection frequency  $f_d$  computed in the training phase, the measurement phases consist of two subphases as shown in Figure 6. In the adaptive delay phase, the coordinator injects a single cycle delay with a small magnitude  $A_d^0$  and frequency  $f_d$  into the target communication link and measures the corresponding response time series from the web server logs. It also obtains the mean value of the delay that was actually injected, and adjusts the value  $A_d^0$  to reflect it. However, if the link gradient  $\frac{\delta r_t}{\delta l}$  is not equal to 1, the actual delay that would appear in the response time as a result of the injection would be  $\frac{\delta r_t}{\delta l} A_d$ , and can become very large if the link is heavily used by the transaction. Therefore, the coordinator computes a rough estimate  $\frac{\delta r_t'}{\delta l}$  of the link gradient using Equation 5, and uses it to scale the magnitude of the injection delay  $A_d' = A_d / \frac{\delta r_t'}{\delta l}$  so that the overall magnitude of the delay in the response time is approximately equal to  $A_d$ . This process ensures that intrusiveness is kept low irrespective of the value of the link gradient.

In the active measurement subphase, 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 5.

## 4 Micro Benchmarks

The algorithm for computing the link gradient requires as parameters the total number of sampling bins  $N$ , the delay scale factor  $d$ , and the minimum number of data points per bin  $k$ . In this section, we examine the sensitivity of the accuracy of the measurement technique to these and other factors. The goal of these experiments is to provide a better understanding of the strengths and limitations of the technique, and to identify ranges of parameter values that can be used during runtime measurement.

**Accuracy of Delay Injection:** The first set of experiments evaluates the amount of actual delay injected as a function of the desired delay. The experiments were conducted under 100% CPU load on regular Linux machines and normal (heavy) loads on PlanetLab machines. We record the time using the *gettimeofday* system call.

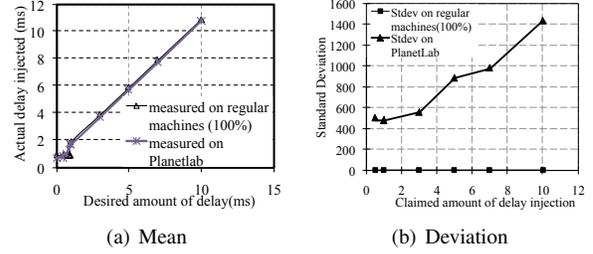


Figure 7: Accuracy of Delay Injection Mechanisms

The mean and standard deviation of the injected delays are shown in Figure 7. 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. The standard deviation of the overhead introduced by the `ip_queue` daemon is less than  $5 \mu s$  even under 100% CPU workload. This 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 overhead introduced by the `tap` delay daemon is as large as 1.5 ms when the desired delay is 10 ms. The variation in the `tap` delay injection mechanism will cause inaccuracy in the link gradient measurement. To reduce this effect, we used larger delays for PlanetLab machines than for regular machines.

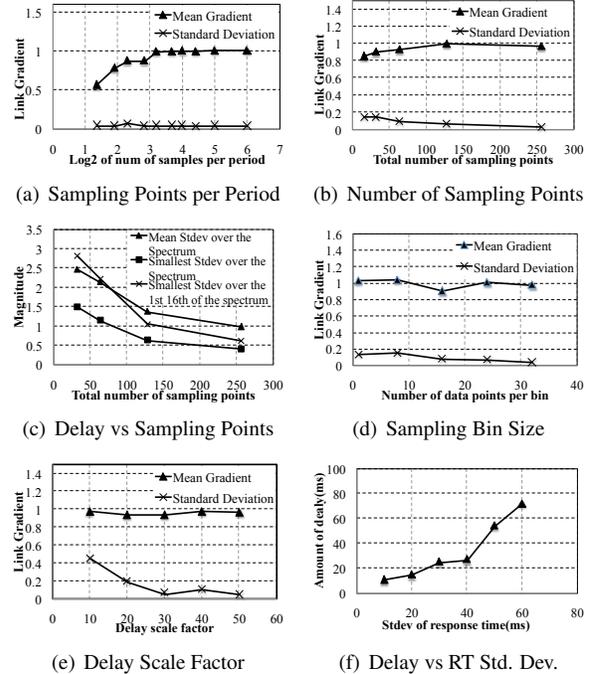


Figure 8: Sensitivity study of Link Gradient

For the remainder of this section, we consider a simple test application in which a single front-end server, on receiving a client request, calls a single back-end server

that processes the request for a normally distributed random amount of time with a mean of 100 ms and a standard deviation of 30 ms. When the back-end server returns, the front-end server generates a reply back to the client. The client node in the experimental setup generates requests one at a time. To measure accuracy, we compare the link gradient measurements with their expected values of 1 for this highly controlled environment. Each experiment is replicated 10 times to compute the mean and standard deviation.

**Sampling Points per Period:** This experiment evaluates the impact of the number of sampling points per injection period on accuracy. We carried out experiments with a fixed minimum number of data points per bin ( $k$ ) set to 8, a fixed amount of delay of 30 ms, and a fixed  $N$  set to 128 (In the rest of this section, we will use these value as the default assignments to these parameters). The number of sampling points per injection period varied between slightly more than 2 (the Nyquist rate) to 64 (corresponding to only 2 cycles during the entire measurement phase). Figure 8(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 decrease) 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). Moreover, further experimentation showed that any increase in the length of measurement phase (and the number of injection periods) does not help compensate for the error introduced by decreasing number of sampling points per period.

**Total Number of Sampling Points:** Next, we investigate how the total number of sampling points affects the accuracy, stability, and intrusiveness of the measurements by varying the total number of sampling points from 16 to 256. Figure 8(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 sampling 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 8(c) shows how the standard deviation of the response time series without any delay injection  $\sigma(\text{FFT}^0)(f_d)$  changes as the number of sampling 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 1/16th part of the spectrum (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. And since the amplitude of the injected delay is set to be a multiple of the frequency with the smallest standard deviation ( $A_d = d * \sigma(\text{FFT}^0)(f)$ ), 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 evaluates the sensitivity of the link gradient measurement to the minimum 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 8(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 sampling 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 sampling points rather than increasing the number of data points per bin, since the former not only leads to an increase in the result stability, but also requires a smaller delay to be injected in the process.

**Delay Scale Factor:** Figure 8(e) shows how the delay scale factor  $d$  affects the accuracy and stability of the result. In practice, 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 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:** Finally, we examine how the amount of delay required for a fixed delay scale factor (set to 30) varies as a function of the vari-

ance of the system’s response time. Figure 8(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 (through changing of the variance of the normally distributed response time of the back-end server). As expected, the amount of delay required also increases almost linearly, and the increase in the standard deviation of response time results in almost the same increase in the required delay injection. We believe that this result indicates that our technique is able to achieve its objective of low intrusiveness (comparable with the normal variance of the system itself) and high accuracy in our micro-benchmarks.

Based on these results, the best way to improve the accuracy and stability of the results while minimizing intrusiveness is to increase the number of sampling 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.

## 5 Evaluation: RUBiS on PlanetLab

In this section, we evaluate the approach and its predictive power in a realistic setting. Predictive power is measured by measuring the link gradient’s ability to predict the response time of each application transaction 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 is accurately able to capture those effects. Finally, we show how the link gradient can be used to optimize application configurations including on a transaction-by-transaction basis.

### 5.1 Experimental Setup

We used a deployment of RUBiS on PlanetLab. RUBiS [8] is a well-known eBay-like auction application that has been extensively used as a benchmark in the literature. 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 network latency. Deployment on PlanetLab nodes distributed across the United States ensured that the measurements were made in a realistic, heavily shared, high-variance, challenging wide-area environment.

We used the 3-tier Java-servlet-based version of RUBiS with a front-end Apache server (WS), middle-tier Tomcat application servers (AS), and a back-end

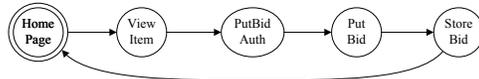


Figure 9: Client Transition Diagram

Users/ $\lambda$	Link	View Item	Store Bid	PutBid Auth	Put Bid	$\lambda_{lg}$
20/ 2.23	AS-WS	1.92	1.05	1.78	2.04	2.21
	DB-AS	15.14	18.76	0.63	18.03	2.20
30/ 3.40	AS-WS	1.47	0.68	1.46	1.75	3.31
	DB-AS	16.99	18.90	0.51	19.13	3.32
40/ 4.41	AS-WS	1.86	1.02	1.78	1.89	4.41
	DB-AS	16.28	18.19	0.83	18.77	4.39
50/ 5.56	AS-WS	1.70	0.82	2.08	1.58	5.54
	DB-AS	18.24	20.37	0.46	20.40	5.48
60/ 6.75	AS-WS	1.48	0.87	2.05	1.97	6.62
	DB-AS	21.74	22.80	0.87	23.15	6.53

Table 2: Link Gradients

MySQL database server (DB). We used the standard RUBiS workload generator with randomly generated TPC-W client think times [16], but for space reasons, restricted each client to only 5 out of 26 possible transactions, chosen because of their potential to stress different parts of the system. Each client uses the bidding oriented transition diagram shown in Figure 9 - *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 web and application server centric, while the other transactions are application and database server oriented.

### 5.2 Link Gradient Computation

We used a setup of RUBiS identical to the one shown in Figure 1 of Section 1 with a single server of each type and the default configuration settings. The workload generator and web server were located in San Diego (UCSD), while the app and db servers 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 2 shows the measured link gradients for all the transactions as the workload is varied from 20 to 60 concurrent clients. It also shows the normal throughput of the system for each workload ( $\lambda$ ) and the modified throughput during link gradient measurement ( $\lambda_{lg}$ ) 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 only a single direction because the results were very similar. However, as we show later, that may not necessarily be the case, and the ability of our approach to measure link gradient in both directions independently can be useful in asymmetric network setups

such as ADSL lines and satellite links.

Comparing the link gradients for the application server-database 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 (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 increases with workload for all links, possibly due to queuing effects. This observation strengthens our claim that an unintrusive runtime technique is desirable for link gradient measurement. 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 3: Response Time Perturbation

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 3. 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, we believe that using the `ip_queue` injector leads to significantly better results.

### 5.3 Predictive Power

Next, we use Table 2 to evaluate the link gradient’s predictive power. We use the link gradient equation  $rt^2 = rt^1 + \sum_{i \in \text{Links}} (l_i^2 - l_i^1) \cdot \vec{\nabla} rt(l_i)$  to predict a transaction’s response time  $rt^2$  in a new configuration based on its current response time  $rt^1$  and link latencies  $l_i^1$ , the link gradient, and the latencies  $l_i^2$  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

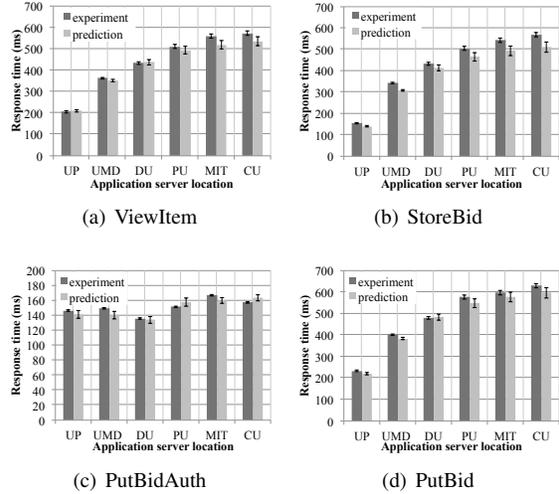


Figure 10: Predicting the Effects of Component Placement. Server location key: UP:U.Pitt, DU:Duke, PU:Princeton, CU:Cornell

RTTs, and divide it by two.

In the first set of experiments, we kept the workload constant at 30 clients and the locations of the web and database server fixed at UCSD and CMU respectively, but moved the application server to 6 different sites across the US. Figure 10 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  $rt^2$  takes into account the errors introduced due to both the original response time  $rt^1$  and the latency measurements, and is calculated using the equation  $\rho(rt^2) = \rho(rt^1) + \sum_{l_i \in \text{Links}} \rho(l_i) \cdot \vec{\nabla} rt(l_i)^2$ , where  $\rho(rt^2)$ ,  $\rho(rt^1)$  and  $\rho(l_i)$  are the variances in the original response time and latency measurements, respectively.

The results show good agreement between the predicted and measured response times across all transactions, which suggests that the link gradient is able to accurately capture the effect of link latency changes on application response time. Although most results are within the margin of error, there is a small but systematic increase in error as the difference between old and new link latencies increases<sup>7</sup>. The reason is that increasing latency may cause an increase in non-linear queuing effects, especially if a system is heavily loaded, thus making the link gradient, which is a linear metric, increasingly optimistic. We discuss the practical implications in detail in Section 6, but note that up to the transcontinental scales we tested, the metric shows excellent accuracy.

In the next set of experiments, we kept the web and database server unchanged at UCSD and CMU respectively, changed the location of the application server to

<sup>7</sup>The configurations are arranged in increasing order of latency between the application server and database sites.

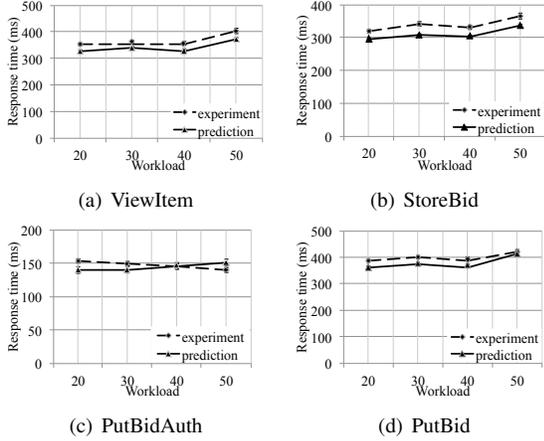


Figure 11: Prediction results under different workload

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 4: Link Gradients with Connection Pooling

UMD (University of Maryland), and measured the predictive ability of the link gradient as the workload was changed from 20 to 50 concurrent clients. The results presented in Figure 11 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 the 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 system component placements under varying environmental conditions.

## 5.4 Communication Pattern Variations

So far we have shown that the link gradient works for a conventional request response type application. However, real enterprise systems typically have many different types of communication patterns such as synchronous and asynchronous calls, load balancing, and connection pooling. Next, we examine if the link gradients accurately capture the effect of link latency on response time across such variations.

**Connection Pooling:** While debugging the large link gradients between the app server and database in the default RUBiS setup, we discovered that the Tomcat server could be configured to use “connection pooling,” whereby it would optimize by recycling database connections, thus altering the application’s communication patterns. When we computed the link gradients for a

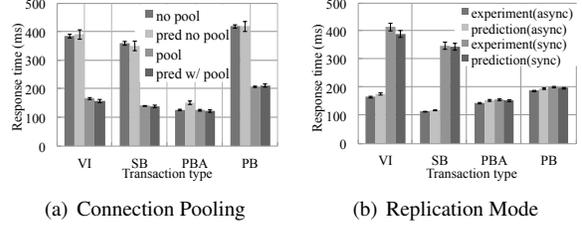


Figure 12: Communication Pattern Effects

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 5: Link Gradients for Replication Modes

workload of 60 clients after enabling connection pooling, we obtained the link gradients shown in Table 4. Comparing the gradients with those for the default setup in Table 2, 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 12(a) shows the predicted vs. measured response time results in a new configuration (with the app server 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 is forwarded all the requests by the web server. 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 store dummy session state, but left the other transactions untouched. Then, we computed link gradients using both synchronous and asynchronous replication modes with the web server in UCSD and the database in CMU as before, but with both application servers placed at the same site (Cornell) because of the replication modes’ multicast requirements.

Table 5 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.

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 6: Link Gradients with Load Balancing

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 reason for the slight dependency of the response time on the AS1-AS2 link is because of interference effects, possibly due to locking, between the threads handling the request and the communications thread responsible for sending the session state to the backup replica.

Figure 12(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.

**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 the web server in UCSD, and the two application servers and the database located at CMU. The results are shown in Table 6. Comparing the results with the link gradients for the unreplicated service shown in Table 2, it can be seen that the link gradients are roughly halved (compared to the gradients without load balancing) for *all* the links, not just the ones between the web server and the application servers. 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. That behavior is very different from probabilistic causality graphs such as those constructed in [14]. In any case, Figure 13(a) shows that the predicted vs. measured response times show excellent agreement for all transactions when one of the application servers is moved from CMU to UMD.

The results validate our hypothesis that even though the link gradient is a simple metric that captures linear effects, it provides an accurate predictor of the effects of link latency on response time across various workloads,

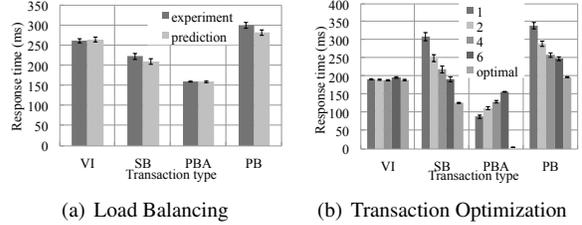


Figure 13: Load Balancing Effects

configurations, and communication patterns without requiring prior knowledge about the system architecture.

## 5.5 Per-Transaction Optimization

Although we believe that link gradient predictors are useful for a variety of purposes, such as system understanding, debugging, optimization, and management, we show a simple example that shows how they can be used to optimize a system for responsiveness at a fine granularity without detailed application information. To do so, we utilize the fact that although different transactions might have behaviors that suggest conflicting optimal locations for the application server, i.e., local to the clients vs. remote (local to the database), the web server can route transactions independent of each other. Therefore, we can obtain an optimized configuration by having *both* local and remote application server 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 web server and one application server at UCSD, and the database and the other application server at CMU. To compare, 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 13 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.

Although the opportunities for optimization in a simple application such as RUBiS are limited, our results should provide the reader with an understanding of how the link gradients can be used for response time optimization in more complicated settings, especially when combined with search algorithms that allow a systematic exploration of the configuration space. The ability

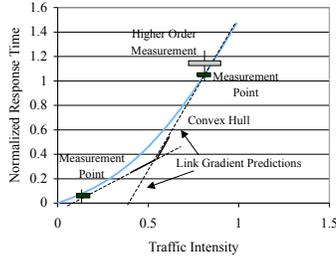


Figure 14: Effect of nonlinearity on Link Gradients

to provide per-transaction response time predictions not only allows optimization based on location, but also allows an exploration of how systems behave under different types of user behavior patterns. Testing the prediction with large production systems, especially those running at AT&T, is part of our future work.

## 6 Limitations

Although the link gradient metric and our measurement algorithm have several strengths, there are some limitations. The first and most obvious limitation is the need for injection of perturbations in the target system—a requirement that limits its appeal for some critical applications. However, it is this very characteristic that provides the technique with some of its strengths such as the ability to isolate effects at the individual transaction level and the ability to perform direct measurement without intermediate models. Therefore, this limitation cannot be eliminated, but could be alleviated. Although the spectral approach provides a good start, we believe there is ample scope for further algorithmic improvements in minimizing the perturbations required for accurate measurements.

A more subtle limitation is related to nonlinearity. Since the link gradient only captures the *linear* relationship between response time and latency, nonlinearities caused by factors such as timeouts and queuing effects (increased link latency may increase the service time of upstream nodes) are ignored. While our results demonstrate that linearity holds over a wide range of scenarios, that may not always be the case. For example, Figure 14 shows how response time varies with increasing service time (captured traffic intensity) in an  $M/M/1/K$  finite-buffer queuing station with a fixed workload. Two link gradient measurements are shown, along with a representation of the perturbations they introduce in the system (the intervals represented by black bars) are shown. The link gradient measures effectively compute the tangents at each measurement point, and thus, over large latency changes, fail to capture the nonlinearity in the response time curve.

We outline two approaches for this problem. First, the link gradient definition can be expanded to include

higher-order terms. During link gradient measurement, perturbations with multiple magnitudes can then be introduced (as shown by the gray bar in the figure), and the additional measurements can be used in a regression to compute the coefficients for the higher order terms. Another solution is to recompute link gradients whenever a prediction turns out to be inaccurate, and maintain a set of link gradients whose convex (or concave in the case of gradients that decrease with increasing latency) hull as shown by the heavy lines in the figure can be used to improve predictive capability. Based on our experience, we believe that a very small number of link gradients would be sufficient to provide acceptable accuracy over a system’s entire range of operating conditions.

## 7 Related Work

While we are not aware of any work directly comparable to ours, since the early days of distributed computing, there has been work on measuring and profiling different aspects of distributed systems for debugging, optimization, modeling, and failure diagnosis.

*Critical path analysis* for parallel program execution was introduced in [18] and extended to an on-line version in [10]. The critical path is the longest path in the program activity graph (PAG) that represents program activities (computation, communication), their durations, and their precedence relationships during a single execution of the parallel program. 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 [12]. A number of techniques for determining causal paths have been proposed [12, 9, 5, 2, 15, 1], each with its own advantages and disadvantages in terms of assumptions on communication patterns, accuracy, and execution cost. Although by no means trivial, it is conceivable that by putting some restrictions on communication patterns, causal paths might be used to compute the “mean number of message crossings in the path of the system response” described in Section 2, and thus approximate link gradient. 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 gradient exactly. Of the literature on determining causal paths, only Magpie [5] collects enough information about resource usage along paths that detailed response time modeling might be attempted. However, the need for extensive (albeit lightweight) instrumentation required by that approach

precludes its use by hosting providers such as AT&T who often do not have the required access to the applications they host.

Methods for identifying failure dependencies between system components have been developed to help in failure diagnosis and localization in complex distributed systems. The ADD (Active Dependency Discovery) technique determines failure dependencies by active perturbation of system components and observation of their effects [6]. The ADD approach is generic and does not specify the perturbation and effect measurement methods. In [3], 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 [7]. Finally, multilevel dependencies in large enterprise networks are identified by use of passive observation [4]. 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. E.g., a server's response time is typically not affected by its DHCP server even though it may have a failure dependency on it. Conversely, non-zero link gradients do not necessarily imply failure dependencies. For example, a distributed application may use a primary-backup replication approach, and, if the primary fails, simply switch to backup with no impact on the service (and thus, no failure dependency).

Signal-injection-based techniques have been used by others, mostly for determining failure dependencies [6, 3, 7]. While our delay injection 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. Finally, while Fourier analysis has been used by others to detect periodic behavior in network routing updates [11], we are not aware of any other work in software systems research that uses the specific combination of signal injection and Fourier analysis to improve measurement accuracy.

## 8 Conclusion and Future Work

In this paper, we introduced a new metric, the *link gradient*, that can be used to approximate how the response time of a system changes with link latencies. We proposed a novel technique to compute the link gradients by using square wave signal injection and Fast Fourier Transforms with low perturbation. We implemented an automatic framework for computing the link gradients of a running multitier enterprise application. We demonstrated the efficiency and accuracy of our technique by using micro-benchmarks as well as distributed deployments of RUBiS on PlanetLab. We showed that link gradients can be used to predict service response times for

new configurations. Our future work will consist of evaluating alternative techniques for injecting signals in distributed systems and identifying new system metrics.

## References

- [1] AGARWALA, S., ALEGRE, F., SCHWAN, K., AND MEHALINGHAM, J. E2EProf: Automated end-to-end performance management for enterprise systems. In *Proc. 37th IEEE Int. Conf. on Dependable Systems and Networks* (June 2007).
- [2] AGUILERA, M., MOGUL, J., WEINER, J., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proc. SOSP* (Oct 2003).
- [3] BAGCHI, S., KAR, G., AND HELLERSTEIN, J. 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).
- [4] BAHL, P., CHANDRA, R., GREENBERG, A., S.KANDULA, MALTZ, D., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. SIGCOMM'07* (Aug 2007).
- [5] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *Proc. of OSDI'04* (Dec 2004), pp. 259–272.
- [6] BROWN, A., KAR, G., AND KELLER, A. 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.
- [7] CANDEA, G., DELGADO, M., CHEN, M., AND FOX, A. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proc. 3rd IEEE Workshop on Internet Applications (WIAPP)* (Jun 2003).
- [8] CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. Performance and scalability of EJB applications. In *Proc. OOPSLA'02* (2002), pp. 246–261.
- [9] CHEN, M., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. of DSN'02* (2002), pp. 595–604.
- [10] HOLLINGSWORTH, J. Critical path profiling of message passing and shared-memory programs. *IEEE Trans. Parallel Distrib. Syst.* 9, 10 (1998), 1029–1040.
- [11] LABOVITZ, C., MALAN, R., AND JAHANIAN, F. Internet routing instability. In *Proc. SIGCOMM '97* (1997), pp. 115–126.
- [12] MILLER, B. DPM: A measurement system for distributed programs. *IEEE Trans. on Computers* 37, 2 (1988), 243–248.
- [13] NAHUM, E., ROSU, M.-C., SESHAN, S., AND ALMEIDA, J. The effects of wide-area conditions on WWW server performance. In *Proc. SIGMETRICS'01* (2001), pp. 257–267.
- [14] P.REYNOLDS, WIENER, J., MOGUL, J., AGUILERA, M., AND VAHDAT, A. WAP5: black-box performance debugging for wide-area systems. In *Proc. Int. WWW Conf.* (May 2006).
- [15] REYNOLDS, P., WIENER, J., MOGUL, J., AGUILERA, M., AND VAHDAT, A. WAP5: black-box performance debugging for wide-area systems. In *Proc. Int. WWW Conf.* (May 2006).
- [16] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC benchmark W (web commerce) specification, v.1.7. [www.tpc.org/tpcw/spec/tpcw.17.pdf](http://www.tpc.org/tpcw/spec/tpcw.17.pdf), Oct 2001.
- [17] VARGHESE, G., AND LAUCK, A. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM. Trans on Networking* 5, 6 (1997), 824–834.
- [18] YANG, C.-Q., AND MILLER, B. Critical path analysis for the execution of parallel and distributed programs. In *Proc. 8th Int. Conf. on Distributed Computing Systems* (1988), pp. 366–373.