Online Feedback-based Estimation of Dynamic Page Service Time

Ashwini Kumar, Kaushik Veeraraghavan, Benjamin Wester, Kang Shin
EECS Department, University of Michigan
{ashwinik, kaushikv, bwester, kgshin}@eecs.umich.edu

***Abstract*— We present a framework for estimating the service time of a dynamic HTTP request by using the service times of past requests to the same URL. Our framework tags incoming requests with a timestamp. As the request is processed, server state relevant to the estimation mechanism, such as the number of threads servicing incoming requests, is stored with each request. When a request is handled, it is timestamped again, allowing us to calculate its service time. This service time is added to the history table and used in the computation of future estimates. The estimators we evaluated do not produce accurate predictions when server resources change, leaving room for further work in estimator design.**

I. Introduction

Unlike traditional web pages that can be cached on a web server or in the browser, only certain components (such as standardized text) of dynamic web pages can be served faster by caching or pre-fetching [4]. With the advent of adaptive advertisements, personalized online shopping and similar technologies, dynamic web pages that require extensive server-side processing are becoming increasingly prevalent. One concern when serving dynamic pages is determining the server resources required to provide end-users with good quality of service. A possible way to do this is to analyze incoming service requests and provide greater control over their processing to the web server.

A precise estimate of a request's service time would be useful for providing good quality of service. We propose a general feedback-based framework for estimating this service time. Our framework functions as follows. For each URL, the web server tracks the service time of recently-completed requests in a *history* table. Upon receipt of a new request, we use our history table to compute an estimate for the request. When this request completes, its measured service time updates the history table.

We built our estimation framework, called *Sirocco*, atop the Staged Event Driven Architecture (SEDA) [13] infrastructure for developing event-driven applications. Our estimation mechanism predicts request service times when the server is operating under regular (i.e., non-overloaded) conditions. To preserve the accuracy of our estimates under varying loads, we experiment with different estimate computation schemes. We analyze our results and describe a general set of properties that estimators should possess.

The estimation framework is:

1) *online*: estimation is performed at runtime;
2) *adaptive*: our feedback mechanism tracks varying service times (perhaps due to changing service load); and
3) *cheap*: while increasing the number of tracked URLs is limited by server-side resources such as memory, this does not impact end-user request service times.

Along with its use in web services, we envision the impact of our estimation techniques in other areas:

1) **Load balancing**: Oftentimes, large websites employ load-balancing schemes to distribute resources, and traffic, amongst multiple servers. Sirocco will provide a reasonably accurate service time for every URL, and help construct better load-balancing mechanisms.
2) **Web server administration**: An accurate service time estimation would enable the development of applications that can better manage web servers. For instance, a website that distinguishes between paid and unpaid users can monitor service times for paid users' accesses and guarantee them better service.
3) **Virtual host services**: With the emergence of rentable enterprise-scale data centers [8], dynamic allocation of servers, applications and communication bandwidth is a novel problem. An efficient solution will require an excellent service and load estimation technique to correctly predict resource requirements.

There are some limitations to our framework. To have a meaningful correlation between a URL's history and its service time, the page at that URL must be generated in a constant processing time. We have not fully addressed the behavior of our history/estimation scheme under server overload scenarios.

The rest of this paper describes our framework design, implementation and evaluation. We then describe future work.

II. Related Work

Over the years, web servers have incorporated novel scheduling [2], traffic shaping [12], graceful degradation under overloaded scenarios [1], and other server performance improvements [13], [10]. To the best of our knowledge, there is no work that directly deals with the problem of estimating the service time of dynamic requests to a web server.

III. Design

Figure 1 shows an overview of the framework. The central component of our framework is the *history* table. We chose to denote different dynamic applications as different URLs. For each dynamic URL, we maintain a history table entry

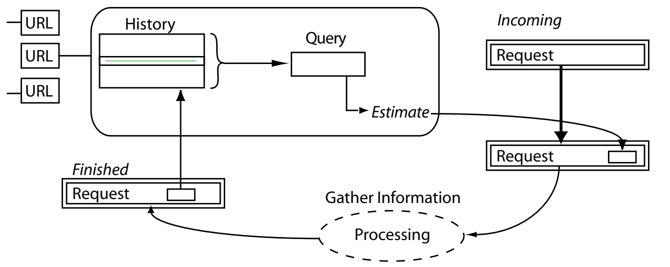

Fig. 1. History framework embedded within a web server. Each request is in a different stage. In the second stage, the history table attaches a tag to a request. In the final stage, the tag in a finished request is used to update the history table, providing feedback.

of information from previous requests for that URL. The information stored depends on the estimator (one of our estimators store the measured service time). When the server receives a new request, its URL entry is located in the history table, and it is given a tag.

A *tag* is a data structure that is attached to every request. It encapsulates data relevant to the estimator (in our schemes, these include the timestamps, the queue length, and the number of threads). As a request is processed by the server, this information is stored in its tag. When the request has been serviced, its tag is used to update the URL's history table entry. This completes the feedback loop in our design.

We designed our history table, tagging and estimation mechanism to work in a simple, cohesive manner that can be ported to any web server. The framework only tracks information that is readily available in a web server, and it can be easily extended. The tagging mechanism requires a simple function call when request processing is initiated, and another at completion. On request completion, a history-update function is also invoked.

A. Estimation

For a given URL, we the history table for information from past requests. This information is used to compute a service time estimate for a new request to that URL. This computation requires a two step effort: *i)* we need to incorporate a timestamp mechanism that tracks the request through different stages of processing in the web server; *ii)* we need to develop a suitable estimation scheme capable of generating an accurate prediction.

We use the tag to store a timestamp when the request is first encountered and a second timestamp immediately before the request's response is queued for transmission. Their difference is the *measured service time*.

The design space for an estimation scheme involves several tradeoffs. A high performance web server that is heavily-loaded with requests should not expend all its resources computing a service time estimate. At the same time, a better estimation method may require more resources. Simplicity of design and operation is required to ensure that it does not compound the complexity of the web server. Another important tradeoff is between adaptability and predictability. We would like to discards random 'noise' and extreme swings in service time to keep our estimates predictable. However, we also want the estimates to adapt when the service time does change, perhaps due to the addition of threads.

With the above tradeoffs in mind, we designed a few simple estimators. Our first estimator performed a simple average of past service times and returns this as the estimate. While the averaging scheme is simple, we found it to be unresponsive under a changing load, i.e., we had to flush out a majority of past service times for the estimate to resemble current service times. We realized that we could better adapt to changing loads by accounting for the queue length of requests awaiting processing. For each request, we store STQ, where:

$$STQ = \frac{Service\ Time}{Queue\ Length}$$

For an incoming request, an estimate was generated by retrieving its STQ value and multiplying it by the queue length at that instant. We found this estimator to perform significantly better than the averaging estimator. However, its performance degraded under heavy loads when SEDA detected an overload and spawned additional handler threads. This led us to believe that we might be able to further improve our estimation by incorporating the number of threads servicing incoming requests into our computation. Our third estimator stored the following value.

$$STQT = \frac{Service\ Time}{Queue\ Length\ *\ Number\ of\ Threads}$$

An estimate is generated by retrieving the average $STQT$ from the history table and multiplying it by the queue length and the number of threads servicing incoming requests at that instant.

IV. Implementation

Web server architectures fall into three broad models: process-based, thread-based, and event-driven. Kegel [6] and von Leitner [11] provide a more-detailed look at the differences between these models and explain how the design of event-driven servers, such as Lighttpd [7] and SEDA [13], scale better than process-based and thread-based servers, such as Apache.

We designed our framework so that it can scale from monolithic servers to larger web servers with distributed data processing backends by passing the request tag across different machines. Our prototype implementation deployed the history framework within Haboob, a monolithic event-driven web server implemented on the Sandstorm platform, a reference implementation of SEDA [13]. Haboob is structured as a central execution unit with a set of linked stages that maintain their own request queue and dynamic thread pool. An important design decision is that we have SEDA provide each dynamic URL with its own handler stage. This is later found to greatly impact resource utilization and our ability to measure the framework.

The modified version of Haboob incorporating our history mechanism is called *Sirocco*. Our changes to Haboob are shown in Figure 2. The most important change is the addition

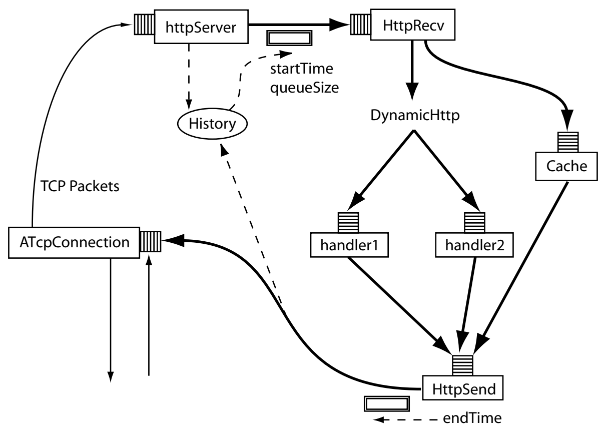

Fig. 2. This figure shows the main internal stages of Sirocco with the STQ estimator. All stages (with their own queues) and data flow between stages are unchanged from Haboob. Sirocco adds the History class. New requests parsed by *httpServer* are forwarded to *History*, which generates the request's tag. Using STQ, the tag contains an initial timestamp and the length of the request handler's queue. In *httpSend*, a final timestamp is added to the request's tag as the request is queued for transmission. The tag is sent to *History* to provide feedback.

of the *httpRequestHistory* class which holds the history table. The history table is stored as a hash table that maps each tracked URL to a data structure containing an array each for the service times, STQ or $STQT$ values, depending on which of the three estimators is in effect. Its public interface consists of two functions: *updateHistory()* and *getTag()*.

As each incoming request is parsed by the Haboob stage *httpServer* (see Figure 2) to generate a request object, *getTag()* generates a new history tag for the request containing a timestamp and a queue size. At this stage, we update the tag with the number of requests pending service in the handler queue (queue length is used in the STQ estimator) and the number of thread handlers (used in $STQT$). Once a request is serviced, it arrives at *HttpSend* where it receives the exit timestamp as it places its response data onto the TCP connection send queue. If the request came from one of the dynamic handler stages, *HttpSend* invokes the *updateHistory()* function with the request's tag as the parameter.

When the history table receives a tag to update, it extracts the embedded URL and looks up its entry in the history table, creating a new one, if necessary. Updating a specific URL's history involves appending the service time (as measured by the start and end timestamps), or the computed STQ or $STQT$ values to the history table, depending on the estimator used. We maintain a fixed-length history array for every URL entry. This array is used as a circular buffer with a Least-Recently-Used (LRU) eviction policy. This history update acts as a feedback mechanism which improves future estimated service times.

V. Evaluation

There are three main areas in Sirocco that must be evaluated:

1) *Overhead* — what is the resource cost of our framework?
2) *Measurement* — how accurately does Sirocco measure the current request service time?
3) *Estimation* — how accurately does Sirocco estimate the current request service time?

A. Methodology

We deployed Sirocco on a Dell GX620 with a Pentium IV 3.4GHz processor and 2GB DDRAM. A ShuttleX PC with Athlon XP 2400+ and 512MB DDRAM was connected to the server over a dedicated 100BaseT Ethernet connection and acted as our client. In this configuration, network delay is negligible as there are no other machines on the network.

To simulate a computational delay on the server, we constructed a dynamic application that accepts an integer as a time parameter encoded on the URL path and calls *Thread.sleep()*. In each set of trials, this application *sleep*s for a constant time, with some small variation due to Java's scheduling. While this does not create any load on the server, it does block the thread depicting similar behavior as a blocking call to a web application server.

We use SURGE [3], a web load generator, to test our estimator under varying load conditions. SURGE generates test pages with a size distribution based on Zipf's law and an invocation model based on a combination of lognormal and Pareto distributions. We configured it to generate 2,000 files, with the most common file accessed 20,000 times. With this configuration, the interesting changes in load occur within the first 25 seconds of the test. Roughly 35% of all requests are mapped to the dynamic handler. A trial in SURGE is deterministic, so we can view identical request patterns under differing server conditions.

We modified Sirocco so that requests to URLs in the history table can be tracked as they are processed by different stages in the web server. These logs serve as our primary data set.

One metric we use is the *SURGE worst-measurement*. Ten trials are run against a configuration, and the worst measured service times for each run are averaged together. All dynamic requests are mapped through one URL which *sleep*s for the same amount of time.

B. Overhead

We have three estimators that use a variable number of stored history values. Each pair of choices we make for a configuration may affect the total overhead differently. To accurately measure the component of the overhead caused by the estimation framework itself, we remove the estimator and history from consideration. We measure the overhead of our framework by comparing *i)* a server that implements a history table which stores no values and has no estimator function and *ii)* the standard Haboob server augmented with logging capabilities.

We initially used *Httperf* [9] to compute the overhead of Sirocco. We configured httperf to send requests to our server at a constant rate and to measure the connection time for each request while we varied the server conditions. This test

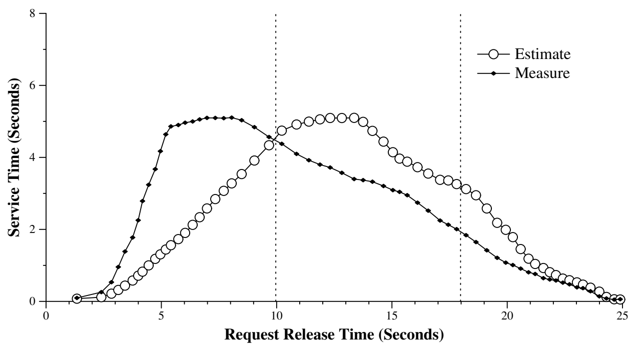

Fig. 3. This graph shows the Average Service Time estimator on a SURGE trial. The estimated service time given to a request is compared against the measured service time. The points at time x show a request's measured and estimated times. Each dotted line indicate the instant when a new thread began servicing requests. This estimator trails the measured service time.

indicated that our framework added no measurable overhead under any server variation.

However, by comparing the graphs of SURGE trials, we found that there was in fact a difference. Using the SURGE worst-measurement metric, the history gathering framework produced a worst measurement that was 13% greater than Haboob. It was necessary to stress our system before any overhead became evident, and SURGE made the best attempt.

We wanted to measure how the overhead changed as the system tracked more URLs. Because of our design choice to use one SEDA stage per URL, adding more URLs resulted in having more threads in our system. With many threads we cannot provide enough load to the server to generate interesting behavior. The large number of stages also consumes server memory resources. After accessing 3,500 URLs (creating 3,500 stages), our server ran out of memory. For these reasons, we could not measure the overhead associated with a varying number of URLs.

Neither SURGE nor httperf showed any significant performance overhead from changing the number of history values stored.

C. Measuring Service Time

As mentioned before, we measure the service time of a request by attaching two timestamps to it. We call their difference the *measured* service time. The request does pass through queues outside of this boundary, so the measured time is inexact. We call the time span including these components the *actual* service time.

The difference between these two time values can be obtained by comparing the SURGE logs to the Sirocco logs. The error value per request is the absolute difference recorded by these two sources. Our tests produced a mean error of 8.86ms (standard deviation 35.69). On closer investigation, we found that half the values are under 2ms, with 95% under 35ms. There are very few values that have an abnormally large error. Error does not appear to follow a normal distribution, so it cannot be easily characterized by its mean value.

D. Estimating Service Time

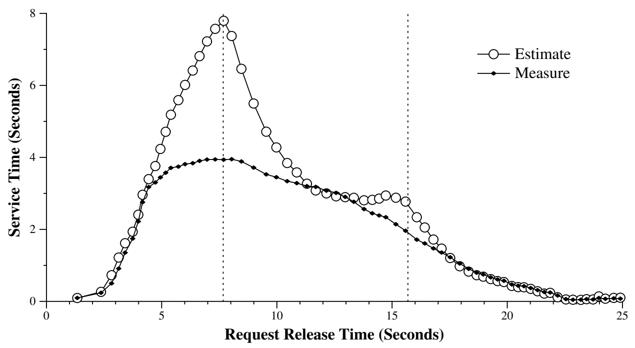

Fig. 4. This graph shows the STQ estimator. The estimate deviates from the measurements in the vicinity of thread additions.

1) Average Service Time: Figure 3 shows a plot of a request's estimated service time compared to the measured service time. Clearly, our estimation is not accurate. This is due to the reactive nature of the request function. Since we base our estimate off of the average of the response times in the history, when the server is changing, the values in the history table will always trail the correct value. This simple estimator only works well in the trivial case where the server load for the specific URL is relatively constant.

2) STQ: Figure 4 shows a trial using STQ estimation. There are instances where the estimated time closely follows the measured service time. Two deviations (where the service time is not accurately predicted) occur for the following reasons. At time 7.6, a new thread is created. After this event, the rate at which requests are consumed from the handler's queue doubles. Every request in the queue at that time, already had its service time predicted. Those predictions will not change when the thread is added; they will continue to reflect an estimation made when the number of threads was different. The deviation before such resource changes is unavoidable unless the estimator can predict future changes in number of threads.

What about the deviation immediately after the thread addition? All STQ values implicitly encode the rate at which requests are processed. When the underlying resources change, the processing rate changes, and old history entries are no longer accurate. The STQ estimator assumes that the rate at which a request is serviced remains constant throughout the life of the server. When this assumption fails, STQ will not give an accurate estimate.

3) STQT: An $STQT$ estimation trial is given in Figure 5. Immediately after the thread addition, it recovers to a point near the measured value. The estimation then further diverges from the correct path before rejoining the measured values.

An individual request can accurately measure the consumption rate per thread, as long as the number of threads is constant during that request's lifetime. In the period after a thread addition, currently-completing requests contain value which combine two different rates. As these values replace older, correct values, the estimation degrades. So while removing all history might have been an improvement for STQ,

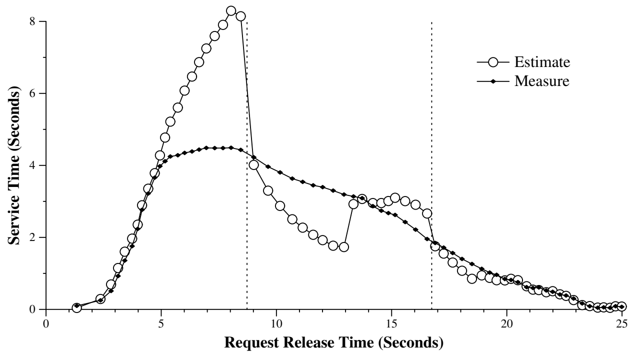

Fig. 5. This graph shows the *STQT* estimator, tracking 16 times. While its estimations are closer to the measurements than *STQ*'s, estimates after a thread addition are still not accurate because the addition is still affecting completing requests.

an improved *STQT* would adjust or ignore new values.

E. Estimator Construction Guidelines

To summarize our evaluation, we offer the following considerations for constructing future estimators.

1) The estimator cannot use only the history table. It takes time for a request to be processed, so the view captured in history does not show the current state.
2) Some values seen by the estimator will not follow the general trend and should not be used in the history table. Perhaps some sort of adaptive filter could be used to eliminate extraneous input.
3) When the server's resources change, old history values might give the wrong view of a system. Removing these old values keeps them from interfering with new data.

VI. Conclusion

We have developed a method for estimating request service times for dynamic web applications. A URL's request history can be used to extrapolate an estimated service time for an incoming request to that URL. We evaluated three estimation functions and found them to behave well for steady loads, but collapse under changing server conditions. We analyzed our results to derive a set of properties for constructing better estimators.

The work presented here could be extended in the following areas:

- *History*: Rather than weighing each stored value in the history table equally, we could adopt a decay mechanism that weighs the values based on age.
- *Estimation*: We used several overly-simple estimators in our current design. We would like to further explore this design space and experiment with more complex/multi-dimensional estimators.
- *Web load generators*: We struggled to stress our web server and to find a load generator that plays well with dynamic requests. SURGE may work well for stressing a server with static requests, but we are unsure of its applicability to dynamic content. We would like to try evaluating our server with Eve [5] since it includes improved modular programmable-client approach that generates dynamic requests.
- *Web server*: Our history framework can be implemented on different types of web servers, which may change its observed behavior or allow us to test different properties.

The primary observation of our study is that a history module which tracks useful information for dynamic requests may be an accurate method to estimate request service time. This information can be processed at run-time without significant overhead.

References

[1] Tarek F. Abdelzaher and Nina Bhatti. Web content adaptation to improve server overload behavior. In *WWW '99: Proceeding of the eighth international conference on World Wide Web*, pages 1563–1577, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[2] Nikhil Bansal and Mor Harchol-Balter. Analysis of srpt scheduling: investigating unfairness. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 279–290, New York, NY, USA, 2001. ACM Press.

[3] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 151–160, New York, NY, USA, 1998. ACM Press.

[4] Azer Bestavros. Using speculation to reduce server load and service time on the www. Technical report, Boston, MA, USA, 1995.

[5] Hani Jamjoom. *Network Oriented Controls of Internet Services*. PhD thesis, University of Michigan, Electrical Engineering & Computer Science, 2004.

[6] Dan Kegel. The c10k problem. http://www.kegel.com/c10k.html, 1999.

[7] Jan Kneschke. Lighttpd. http://www.lighttpd.net, 2005.

[8] Hewlett-Packard Labs. Data center architecture. http://www.hpl.hp.com/research/dac/index.html, 2006.

[9] David Mosberger and Tai Jin. httperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[10] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[11] Felix von Leitner. Scalable network programming or: The quest for a good web server (that survives slashdot). http://bulk.fefe.de/scalable-networking.pdf. 2003.

[12] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. 2003.

[13] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.