Performance Analysis of an Asynchronous Web Server

U. Praphamontripong, S. Gokhale
Dept. of CSE
Univ. of Connecticut
Storrs, CT 06269
ssg@engr.uconn.edu

Aniruddha Gokhale
Dept. of EECS
Vanderbilt Univ.
Nashville, TN 37235
a.gokhale@vanderbilt.edu

Jeff Gray
Dept. of CIS
U. of Alabama at Birmingham
Birmingham, AL 35294
gray@cis.uab.edu

Abstract

Modern Web servers need to process multiple requests concurrently in order to fulfill the workload demands expected of them. Concurrency can be implemented in a Web server using synchronous and asynchronous mechanisms offered by the underlying operating system. Compared to the synchronous mechanisms, the asynchronous mechanisms are attractive because they provide the benefit of concurrency while alleviating much of the overhead and complexity of multi-threading. The Proactor pattern in middleware, which effectively encapsulates the asynchronous mechanisms supported by the operating system, can be used to implement a high performance Web server.

The performance expectations imposed on a Web server make it necessary to analyze its performance prior to deployment. Design-time performance analysis, conducted earlier in the life cycle, can also enable informed configuration and provisioning choices. A model-based approach can be used for such early, design-time performance analysis. In this paper, we describe a performance analysis methodology for an asynchronous Web server implemented using the Proactor pattern. We present a performance model of the Web server and implement the model using CSIM, which is a general purpose language for building simulation models. We demonstrate the use of the model to guide key provisioning and configuration decisions using several examples.

1 Introduction and motivation

Within a relatively short duration since its advent, the World Wide Web (WWW) has become an important source of information and services in our society. In the initial years, users were attracted to the WWW primarily due to the convenience, flexibility, ease of use and low costs associated with the use of these services. However, as the prevalence of WWW in business and critical domains grows, it is becoming evident that WWW services must be offered with superior performance in order to retain existing users and attract new ones [23].

A central component of any WWW service is a Web server. Modern Web servers have to process millions of client requests on a daily basis. In order to fulfill such high workload demands, it is inevitable that modern Web servers be equipped with the capability to process multiple requests concurrently. Concurrency may be implemented in a Web server using the synchronous or asynchronous capabilities provided by the underlying operating system. Although multi-thread and multi-process Web server architectures [13] which rely on the synchronous capabilities are commonly used, the asynchronous mechanisms may be attractive compared to the synchronous mechanisms because they provide the benefit of concurrency while alleviating much of the overhead and complexity of multi-threading. The Proactor pattern in middleware [18], which effectively encapsulates the asynchronous mechanisms supported by the operating system, can be used to implement a high performance Web server.

Due to the high performance expectations associated with a WWW service, it is imperative that service performance be analyzed prior to deployment. Although performance can be measured once the service is implemented, it is often too late and too expensive to take corrective action at this stage if it is discovered that the target performance cannot be met. It is thus cost-effective and advantageous to conduct performance analysis earlier in the life cycle, at design time. Model-based analysis is an attractive approach to conduct such design-time performance analysis.

In this paper we describe a model-based approach for the design-time performance analysis of a Web server which implements concurrent processing capabilities using the asynchronous mechanisms encapsulated in the Proactor pattern. We capture the characteristics of the Proactor pattern that are relevant from a performance perspective into a queuing model. The queuing model is implemented using CSIM [19], which is a general purpose language used to build simulation models. We illustrate how the model implemented in CSIM can be used to guide configuration and provisioning decisions with several examples.

The balance of the paper is organized as follows: Section 2 provides an overview of the Proactor pattern along with a discussion of its advantages. Section 3 describes the performance analysis methodology. Section 4 illustrates the potential of the methodology with examples. Section 5 summarizes the related research. Section 6 offers concluding remarks and directions for future research.

2 Proactor pattern

In this section we provide an overview of the Proactor pattern. We also discuss the advantages of implementing a Web server using the Proactor pattern.

2.1 Proactor description

The Proactor pattern is a software architectural pattern for event handling, which is used to describe how to initiate, receive, demultiplex, dispatch and process events in network systems [18]. It has been primarily developed to support many simultaneous user requests. Its main purpose is to improve the performance of an event-driven application that receives and processes multiple events asynchronously. Conceptually, this pattern simplifies asynchronous operations by integrating the demultiplexing of completion events and the dispatching of the corresponding event handlers. The general idea of the Proactor pattern is to wait for an event to occur and then initiate the appropriate operation. Once the event starts execution, other events may be initiated and processed. When the event finishes execution, the Proactor demultiplexes the completion event and dispatches it to an appropriate event handler for subsequent processing of the results of the operation.

To implement the Proactor pattern (considering the arrivals of events, each of which requires a single operation to complete), when an event arrives, the application's entity called an initiator starts an appropriate asynchronous operation and registers the event with an associated event handler and event dispatcher with the Asynchronous Operation Processor (AOP). Then an initiator invokes the registered asynchronous operation on the AOP. An asynchronous operation is executed without blocking its callers thread of control. As a result, the caller can perform other operations. That is, the operation and the initiator can run independently and the initiator can invoke a new asynchronous operation while others continue executing concurrently. If an operation must wait for the occurrence of an event, such as a connection request generated by a remote application, its execution will be deferred until the event arrives. In this paper, however, we consider only those cases where operations are processed independently and do not wait for the occurrence of other events. Once the operation is complete, AOP retrieves information corresponding to an event handler and a dispatcher, and generates a completion event containing the results of the asynchronous operation. The Proactor then inserts the completion event along with the retrieved information into the completion event queue. It then removes the completion event from the completion event queue and demultiplexes and dispatches the event to the event handler associated with the asynchronous operation. Subsequently, the event handler processes the results of the asynchronous operation and calls back to the application.

2.2 Proactor advantages

There are several advantages to implementing a Web server using the Proactor pattern [18]. These include:

- The Proactor pattern executes each asynchronous operation independently, thus each service that a Web server provides can be processed separately. Accordingly, a particular demultiplexer and a dispatcher used for each completion event associated with an asynchronous operation can be implemented, managed, and treated independently. As a consequence, the implementation of the Web server is decomposed and decoupled, and hence is more manageable.
- Structuring the demultiplexing of completion events and the dispatching of their corresponding completion simplifies the development process of a Web server, which normally requires asynchronous operations.
- Once an asynchronous operation is initiated, the thread that initiated the operation becomes available to service additional requests.
- Since the asynchronous operations are processed concurrently and the completion events associated with the operations are demultiplexed and dispatched asynchronously, the operations are executed without waiting for the completion of the previous ones. Multiple client requests can be processed simultaneously, which may improve server performance.

3 Performance analysis methodology

In this section we discuss the performance analysis methodology for a Web server implemented using the Proactor pattern. We first discuss the characteristics of the Web server, followed by the desired performance metrics. Subsequently, we describe the performance model of the Web server.

3.1 Web server characteristics

A Web server employs the request/reply paradigm, using the HTTP protocol to communicate between itself and the clients (Web browsers). The clients' requests are specified in an HTTP message, which may also include the operation to be performed and the location where the operation should be performed. Though a variety of request operations may be defined, we consider a scenario where the Web server provides only two types of service, namely a read service (service to read a file) and write service (service to write a file). As soon as a read request arrives at the Web server, a read operation is assigned, initiated, and executed, while a write operation is assigned, initiated, and executed when a write request is received. The completion of these operations will be handled in a common queue regardless of the request type. The completion events are then demultiplexed by the Proactor and dispatched to an appropriate completion event handler. The completion events are further processed by the completion event handlers. For instance, if the client requests for a particular file, an HTTP handler can be used as an initiator to initiate a read operation. After the read operation is complete, an HTTP handler, which now acts as a completion event handler, further processes the request by issuing a write operation to transfer a file to the client.

From the point of view of performance analysis, the Web server has the following characteristics:

- The server receives two types of client requests, namely, read and write requests.
- To service these requests, appropriate asynchronous operations are assigned and executed with a pool of handlers for each request type registered with the Proactor. If an incoming request finds that all the event handlers for that request type are busy, the request is rejected. In this paper, to distinguish between handlers that handle asynchronous operations and handlers that deal with completion operations, the former are simply referred to as event handlers, while the latter are called completion event handlers.
- The completion operations of both types of requests, referred to as completion events, are queued in a single completion event queue.
- The completion events are dequeued by the Proactor in a first-in, first-out manner.
- Each request type has a separate queue holding completion event operations which are processed by the completion event handler registered with the Proactor.
- Each request type has a single completion event handler to process the completion events.

3.2 Performance metrics

In this section we present the performance metrics for each request type along with their practical relevance.

- *Expected throughput:* It is an estimate of the rate at which the requests are processed by the Web server. The expected throughputs of read and write requests are denoted T_r and T_w respectively.
- *Expected busy handlers:* It is an estimate of the average number of busy event handlers. This can be used to guide provisioning decisions regarding the sizes of the event handler pools for a given load. The expected number of busy handlers for read and write requests are denoted B_r and B_w respectively.
- *Expected queue lengths:* It is an estimate of the average number of completion events in the common and individual completion event queues. The expected queue lengths of the common and the separate completion queues are denoted Q, Q_r and Q_w respectively.
- *Expected probability of request loss:* It is the average probability that an incoming request will be discarded because an event handler is unavailable. The average loss probabilities of the read and write requests are denoted L_r and L_w respectively.

3.3 Performance model

In this section we describe the performance model of a Proactor-based asynchronous Web server. We assume that the read and write requests arrive according to a Poisson distribution with rates λ_r and λ_w. The sizes of the event handler pools are denoted N_r and N_w. The service times of the asynchronous operations follow an exponential distribution with rates μ_r and μ_w. The capacities of the common and the separate completion queues are denoted M, M_r and M_w. The demultiplexing rate of the Proactor is denoted κ. The service times of the completion event handlers are exponentially distributed with rates γ_r and γ_w.

Figure 1 shows the queuing model of a Proactor-based Web server. The event handler pools which handle asynchronous operations for read and write requests are modeled as multi-server processing stations without queuing. These servers feed completion events to the completion event queue and they block if there is no spare capacity in the completion queue. The operation of demultiplexing the completion events is conducted by a server, which accepts the completion events from the completion queue and dispatches them to the read or write completion queues depending on whether the completion event was a result of a read or a write request. Because the scheduling strategy

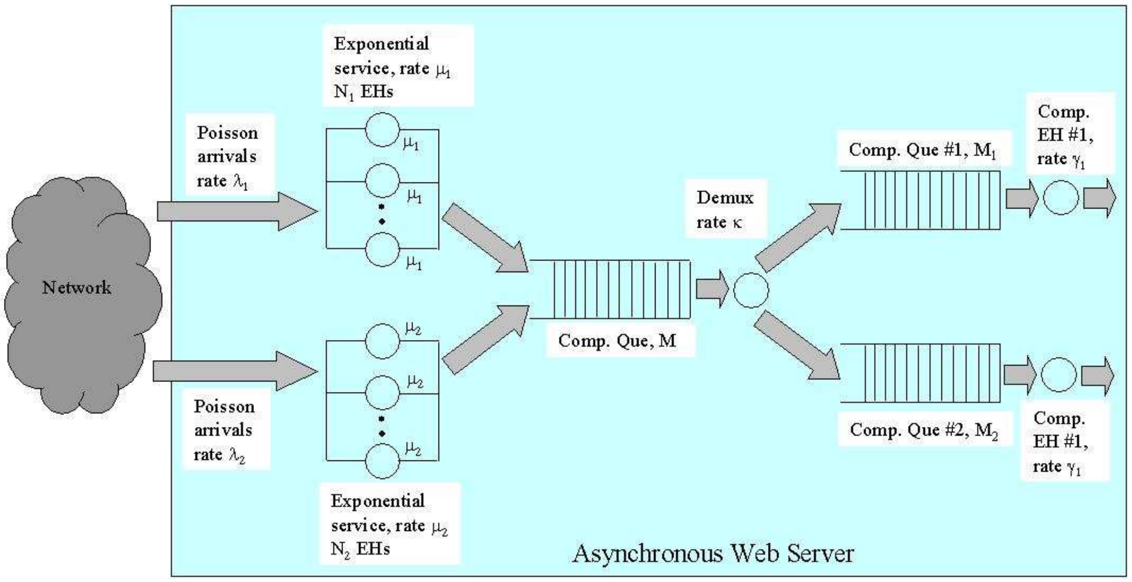
Exponential service, rate μ_1 N_1 EHs
Poisson arrivals rate λ_1
Network
Poisson arrivals rate λ_2
Exponential service, rate μ_2 N_2 EHs
Comp. Que, M
Demux rate κ
Comp. Que #1, M_1
Comp. EH #1, rate γ_1
Comp. Que #2, M_2
Comp. EH #1, rate γ_1
Asynchronous Web Server

Figure 1. Performance model of an asynchronous Web server

used for demultiplexing is first-in, first-out, the demultiplexing operation blocks if the completion queue to which the present event is to be dispatched is full. For example, if the current completion event to be demultiplexed is of type read and the read completion queue has M_r events, the demultiplexing operation is blocked. The completion event queue of each event type feeds the corresponding completion handler which completes the processing. The throughput of each request type is the effective rate at which the corresponding completion event handlers process the completion events. The performance model was implemented using CSIM [19], which is a general purpose language to implement simulation models.

4 Illustrative examples

In this section we illustrate the potential of the performance model described in Section 3.3 to guide the selection of configuration options using several examples. The parameter settings used here are solely for the sake of illustration. To use the methodology in a practical setting, the parameters specific to the given infrastructure need to be determined. We note that although in this paper exponential distributions are used to model the arrival and service times, the model implementation in CSIM is flexible and powerful and can consider many other distributions including the Pareto which is often used to model these processes [14].

The sizes of the event handler pools and the capacities of the common and the separate completion queues are the configuration parameters of the Proactor-based Web server. We designed three experiments to assess the impact of each one of these configuration parameters on the performance metrics. For these three experiments, the arrival and service rates were set as follows: $\lambda_r = \lambda_w = 1.00$/sec., $\mu_r = \mu_w = 2.00$/sec., $\kappa = 2.0$/sec. and $\gamma_r = \gamma_w = 2.0$/sec. For all the experiments, the confidence intervals are within 5% of the mean and are not shown to avoid visual clutter.

Experiment I:
We study the impact of event handler pool sizes, namely, N_r and N_w on the performance metrics. We vary N_r and N_w from 1 to 5 in steps of 1. The size of the common completion queue is set to 10 and the sizes of the individual completion queues are set to 5. The performance metrics as a function of the sizes of the event handler pools are shown in Figure 2. Since the arrival, service and configuration parameters of both the request types are identical, their performance estimates are also the same and hence metrics for only one request type are shown. The top left plot in Figure 2 indicates that the throughput increases significantly when the event handler pool size increases from 1 to 2. This increase in throughput is also accompanied by a similar significant drop in the loss probability as shown in the bottom left plot in the figure. For each subsequent increment in the event handler pool size after two, the rate of increase in the throughput reduces, and similarly the rate at which the loss probability decreases is reduced. This occurs because as the event handler pool size increases, incoming service requests are processed faster, which increases the rate at which the completion events are fed into the common completion queue. The demultiplexing operation, which is performed by just one downstream

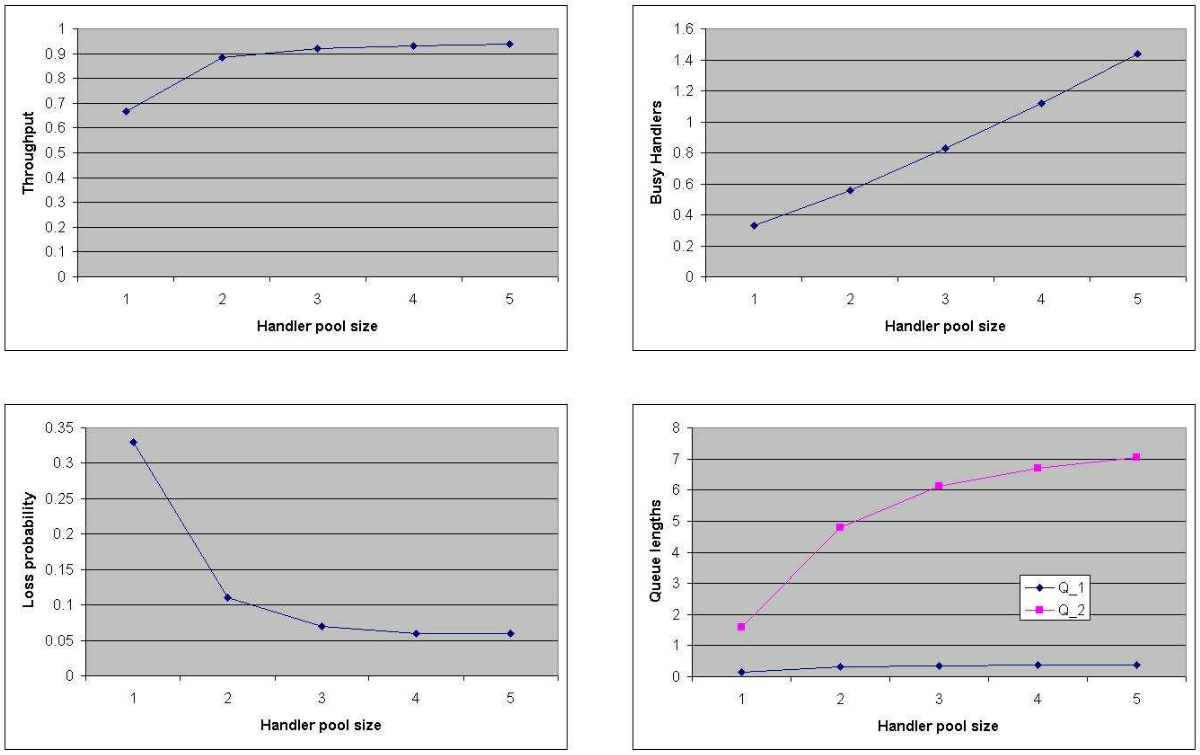

Figure 2. Performance metrics as a function of event handler pool sizes

demultiplexer then becomes the bottleneck which causes an increase in the length of the common completion queue as shown in the bottom right plot in the figure. The top right plot of the figure shows that for the given level of load, the average number of busy handlers is very low. In fact, when the handler pool size is 5, on an average only 1.6 handlers are busy. A larger event handler pool size will invariably require higher hardware resources leading to higher costs. These costs must be offset by a justifiable improvement in the performance. The plots in Figure 2 illustrate that increasing the handler pool size beyond three offers diminishing returns since it provides only a marginal improvement in the performance. Thus, for the load level considered in this experiment, a pool size of three may offer an acceptable tradeoff between cost and performance.

Experiment II:
In this scenario we assess the impact of the size of the common completion queue M on the performance metrics. We vary the completion queue size from 2 to 10 in steps of 2. The sizes of the event handler pools are set to 5 and the sizes of the individual completion queues are also set to 5. The performance metrics as a function of common completion queue size are shown in Figure 3. The metrics for only one request type are shown similar to Experiment I. The top right plot in the figure indicates that the average number of busy event handlers drops with the size of the common completion queue. When the common completion queue is full, the event handlers cannot feed the completion events into the queue, causing them to block, which increases the average time each event handler is busy. This causes a subsequent increase in the average number of busy event handlers. Because the likelihood of the completion queue being full increases as the size decreases, the average number of busy event handlers increases as the queue size decreases. The loss probability in the bottom left plot (throughput shown in the top left plot) shows only a marginal decrease (increase) as a function of the completion queue size. This occurs because the small completion queue size is offset by an increase in the service time of the event handlers. Thus, effectively the event handler pool serves as a queue and compensates for the small completion queue size.

Experiment III:
Through the third experiment, we seek to assess the impact of the sizes of separate completion queues on the performance metrics. We vary the completion queue sizes, namely, M_r and M_w, from 1 to 5 in steps of 1. The sizes of the event handler pools are set to 5 and the size of the common completion queue is set to 10. The performance metrics (for one request type) as a function of the separate

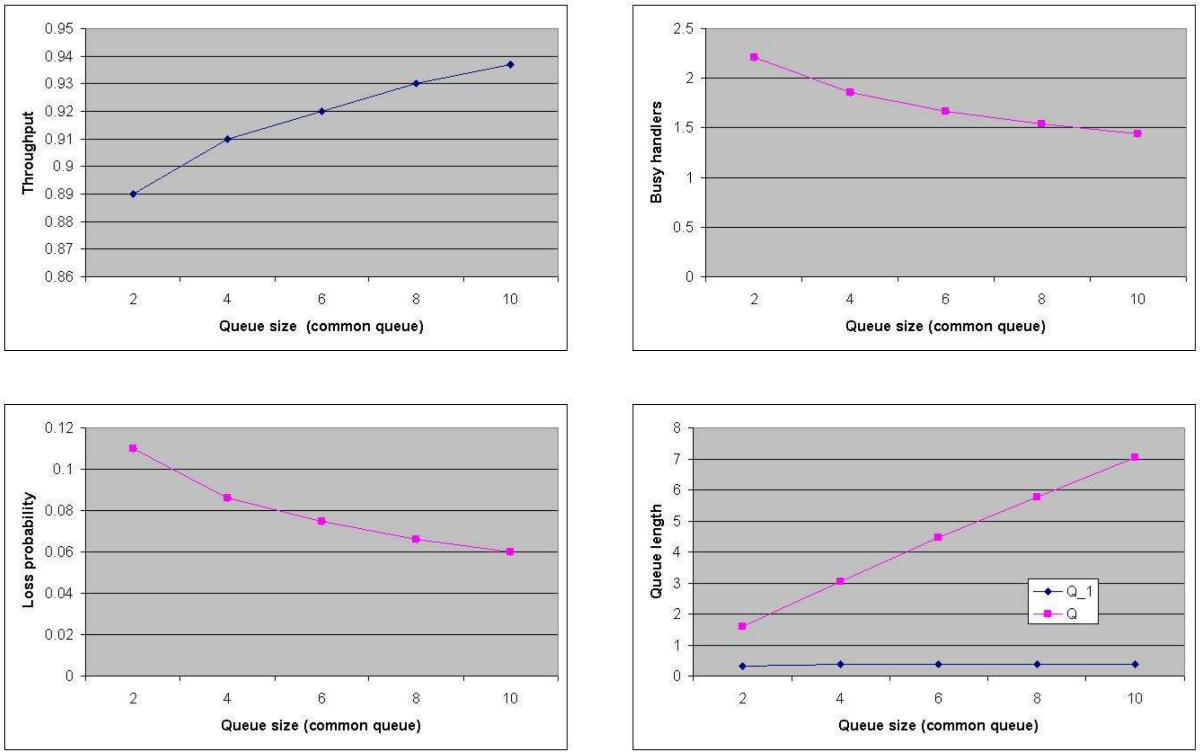

Figure 3. Performance metrics as a function of queue size (common queue)

completion queue sizes are shown in Figure 4. The throughput plot (top left) in Figure 4 indicates a high increase when the sizes of the individual completion queues increase from 1 to 2. When the sizes of the individual queues are low, the demultiplexing blocks due to the lack of buffer space in the downstream queues. This leads to lower throughput, higher loss probability, higher number of busy servers and higher queue length of the common queue, as shown in the plots. Increasing the queue size beyond three, however, offers only marginal performance improvement.

The results of the above three experiments lead us to the following conclusions. First, for a given load, increasing the sizes of the event handler pools and the queues beyond a certain threshold offers a very small performance improvement. Second, for a given set of configuration options, the demultiplexing operation constitutes a performance bottleneck and increasing the demultiplexing rate could thus provide significant performance benefits. The impact of the demultiplexing rate on the performance could be easily analyzed using the model. The above experiments demonstrate the use of the queuing model to guide the selection of configuration options. It is often the case that at design time exact estimates of the arrival and the service rates are not available. The performance model could also be used to analyze the sensitivity of the performance to the variations in the arrival and the service rates.

5 Related research

Research efforts in two areas, namely, performance analysis of middleware services and patterns and performance analysis of Web servers, are relevant to the present work.

Performance analysis of middleware services and patterns can be broadly classified into two, namely, measurement-based and model-based. The measurement-based approach comprises of testing specific implementations with benchmarking suite(s) and then measuring the relevant metrics [4, 10, 16, 22]. The model-based approach consists of building and solving a model using analytical/numerical or simulation methods to obtain performance estimates. Ramani *et al.* [17] present a framework for performability analysis of messaging systems in middleware. Aldred *et al.* [1] develop Colored Petri Net (CPN) models for different types of coupling between the application components and with the underlying middleware. Kahkipuro [11] propose a multi-layer performance modeling framework based on UML and queuing networks for CORBA-based systems. The methodology, however, is for generic CORBA-based client/server systems rather than for systems built using design patterns.

With the growing complexity of software systems and increasing pressure to reduce the time to market, there is a significant push towards composing large systems us-

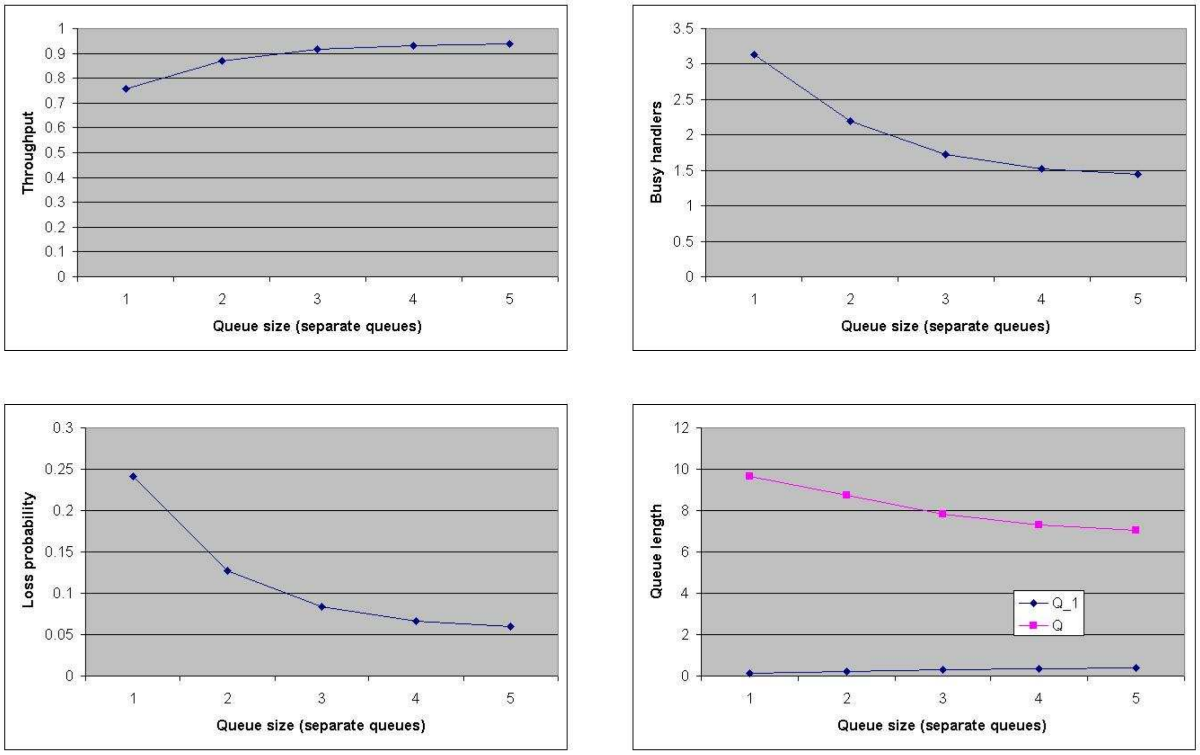

Figure 4. Performance metrics as a function of queue size (separate queues)

ing reusable building blocks or patterns [3, 18]. Performance analysis of such a composed system requires models of the individual building blocks and their composition. Our previous work has developed an analytical model of the Reactor pattern [5]. In this paper we have developed a performance model of the Proactor pattern. Although the model was developed to analyze the performance of an asynchronous Web server, it is generic and could be used for the performance analysis of any Proactor-based system.

Web server performance analysis can also be broadly classified into measurement-based and model-based approaches. The former approach measures the server performance using benchmarks [8, 7, 9]. Many model-based approaches use queuing networks to analyze performance [20, 6, 2, 15, 21, 12]. Most of these techniques however, are for Web servers which use synchronous mechanisms for concurrency, whereas the model presented in this paper is applicable for the analysis of an asynchronous Web server.

6 Conclusions and future research

In this paper we presented a model-based approach for the design-time performance analysis of a Web server which implements concurrent processing capabilities using the asynchronous mechanisms encapsulated in the Proactor pattern. We represented the characteristics of the Proactor pattern that are relevant from a performance perspective in the form of a queuing model, which was then implemented using CSIM [19]. We illustrated how the simulation implemented in CSIM can be used to guide provisioning decisions with several examples. Our future research consists of developing an analytical/numerical approach for the performance analysis of the Proactor. Developing model decomposition strategies to alleviate the issue of large models is also a topic of future research.

Acknowledgments
This research was supported by the following grants from the National Science Foundation (NSF): Univ. of Connecticut (CNS-0406376 and CNS-SMA-0509271), Vanderbilt Univ. (CNS-SMA-0509296) and Univ. of Alabama at Birmingham (CNS-SMA-0509342).

References

[1] L. Aldred, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. “On the notion of coupling in communication middleware”. In *Proc. of Intl. Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, 2005.

[2] J. Cao, M. Andersson, C. Nyberg, and M. Kihl. Web server performance modeling using an M/G/1/K*PS queue. In *10th International Conference on Telecommunications (ICT'03)*, pages 1501–1506, 2003.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[4] A. Gokhale and D. C. Schmidt. "Measuring and optimizing CORBA latency and scalability over high-speed networks". *IEEE Trans. on Computers*, 47(4), April 1998.

[5] S. Gokhale, A. Gokhale, and J. Gray. "Response time analysis of an event demultiplexing pattern in middleware for network services". In *Proc. of IEEE Global Telecommunications Conference (GLOBECOM), Symposium on Advances for Networks and Internet*, St. Louis, MO, November 2005.

[6] J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Transactions on Networking*, 5(5):616–630, 1997.

[7] J. C. Hu, I. Pyarali, and D. C. Schmidt. "Measuring the impact of event dispatching and concurrency models on Web server performance over high-speed networks". In *Proc. of GLOBECOM*, pages 1024–1031, 1997.

[8] Y. Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the Apache Web server. In *IEEE International Performance, Computing and Communications Conference (IPCCC'99)*, pages 261–267, 1999.

[9] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-performance Web site design techniques. *IEEE Internet Computing*, 4(2):17–26, March 2000.

[10] M. Juric, I. Rozman, M. Hericko, and T. Domajnko. "CORBA, RMI and RMI-IIOP performance analysis and optimization". In *Proc. of SCI 2000*, pages 582–587, Orlando, FL, July 2000.

[11] P. Kahkipuro. "*Performance modeling framework for CORBA based distrbuted systems*". PhD thesis, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland, May 2000.

[12] K. Kant and C. R. M. Sundaram. A server performance model for static Web workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, pages 201–206, 2000.

[13] D. Menascé. Web server software architecture. *IEEE Internet Computing*, 7(6):78–81, 2003.

[14] R. Nossenson and H. Attiya. Evaluating self-similar processes for modeling Web servers. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'04)*, pages 805–812, 2004.

[15] R. Nossenson and H. Attiya. The N-burst/G/1 model with heavy-tailed service-times distribution. In *Proceedings of 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*, pages 131–138, 2004.

[16] O. Othman, J. Balasubramanian, and D. C. Schmidt. "Performance evaluation of an adaptive middleware load balancing and monitoring service". In *Proc. of the 24th IEEE Intl. Conference on Distributed Computing Systems*, pages 135–146, Tokyo, Japan, May 2004.

[17] S. Ramani, K. Goseva-Popstojanova, and K. S. Trivedi. "A framework for performability modeling of messaging services in distributed systems". In *Proc. of 8th IEEE Intl. Conference on Engineering of Complex Computer Systems (ICECCS 02)*, Greenbelt, MD, December 2002.

[18] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[19] H. Schwetman. "CSIM reference manual (revision 16)". Technical Report ACA-ST-252-87, Microelectronics and Computer Technology Corp., Austin, TX.

[20] L. Slothouber. A model of Web server performance. In *Proceedings of the Fifth International World Wide Web Conference*, 1996.

[21] M. S. Squillante, D. D. Yao, and L. Zhang. Web traffic modeling and Web server performance analysis. In *Proceedings of the 38th Conference on Decision and Control*, pages 4432–4439, 1999.

[22] P. Tuma and A. Buble. "Overview of the CORBA performance". In *Proc. of the 2002 EurOpen CZ Conference*, September 2002.

[23] R. D. van der Mei, R. Hariharan, and P. Reeser. Web server performance modeling. *Telecommunication Systems*, 16(3-4):361–378, 2001.