

A Predictive Performance Model to Evaluate the Contention Cost in Application Servers[†]

Shiping Chen[‡]

CSIRO Mathematical and Information Sciences
Locked Bag 17, North Ryde, NSW 1670, Australia
Shiping.Chen@csiro.au

Ian Gorton

Pacific Northwest National Laboratory
Richland WA 99352, USA
Ian.Gorton@pnl.gov

ABSTRACT

In multi-tier enterprise systems, application servers are key components to implement business logic and provide services. To support a large number of simultaneous accesses from clients over the Internet and intranet, most application servers use replication and/or multi-thread technologies to handle concurrent requests. While multi-processes and multi-threads enhance the processing bandwidth of servers, they may also increase the contention on application servers. This paper investigates this issue based on our internal middleware benchmark. A cost model is proposed to estimate overall performance of application servers, including the contention overhead. This model can be used to determine the optimal degree of the concurrency of application servers for a specific client load. A case study based on CORBA is presented to validate our model and demonstrate its application.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: performance modeling

General Terms

Measurement and Performance

Keywords

Middleware, Application Server, Performance, Cost Model

1. INTRODUCTION

Multi-tier systems enable enterprises to quickly enhance existing or develop new applications to react rapidly to evolving business needs [5]. In a multi-tier enterprise system, the application servers should be always (24 hours x 365 days) and widely (on the Internet or an intranet) available to provide business services to clients. Typically, these services are implemented in the application servers supported by Enterprise Information Systems (EIS), usually meaning databases.

Most multi-tier enterprise systems are expected to support a large number of clients. For example, it is possible for thousands clients to simultaneously send requests for buying and selling stocks to an online share trading system over the Internet. To efficiently support a large number of simultaneous accesses from clients,

multiple servers or multi-threaded servers are usually used to handle the concurrent requests. In this way, the degree of concurrency of application servers is enhanced. However, multiple servers and multi-threaded servers may raise the contention in the applications.

Most application servers are based on databases to implement business logic. From the viewpoint of databases, an application server is a client, which sends request to the database server to perform transactional operations on data via a database connection. All server threads or server replicates can simultaneously send requests to databases, which may results in contention on database, network and other resources. If too many server threads or server replicates are used in a multi-tier system, the contention overheads can be so high that the benefit from the concurrency can be over-weighted. While more server threads and replicates can increase the processing bandwidth of application servers, a lower degree of the concurrency of application servers is expected to reduce the contention overhead on the EIS tier. Therefore, the tradeoff may exist to achieve the best balance between the concurrency and the contention overhead.

In this paper, we address the issue of determining the optimal degrees of the concurrency for application servers. A cost model is proposed to estimate the overall contention overhead resulting from concurrent clients and multi-thread servers. Since the cost model reveals the interaction between request load and the concurrency of multi-thread servers, it can be used to determine the best numbers of server threads in an application servers for a given request load. A case study is presented to demonstrate how to model and use our cost model. Our case study is based on the CSIRO internal middleware benchmark, *Stockonline*, from CSIRO's Middleware Technology Evaluation (MTE) project www.cmis.csiro.au/adsat/mte.htm [1].

The rest of this paper is organised as follows. Section 2 provides an overview of multi-tier enterprise systems. The contention cost model is defined and described in Section 3. In Section 4, we carry out a case study to build a contention cost model for our internal benchmark, *Stockonline*. Then the cost model is used to predicate the optimal number of server threads in this section. We conclude this paper in Section 5.

[†] To submit to APSEC2002 for review

[‡] Corresponding author

2. CONCURRENCY AND CONTENTION

2.1 Multi-tier Enterprise Systems

A multi-tier enterprise system consists of at least three tiers, i.e. client tier, application server and EIS tier. The client tier can be either a common Internet browser, such as Internet Explorer and Netscape, or a specific application, such as Java and C++ applications. The two kinds of client tier applications are both able to request services from the application over the Internet and an intranet using standard Internet protocols, such as HTTP, IIOP and RMI.

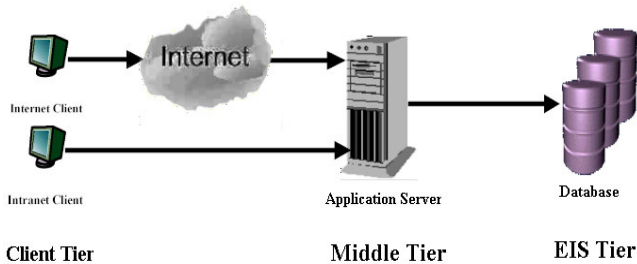


Figure 1. A typical architecture of a multi-tier system

Application servers provided business services to clients. Clients can use these services without knowing their implementation. In this way, the application provides a layer of abstract, which hides the implementation of the application from the client tiers. This feature enables us to continuously adjust enterprise business logic and update the application server using new technologies with no or minimal effects on clients and the whole system. CORBA, COM and EJB are three elegant technologies to build application servers.

Most application servers need supports from the Enterprise Information Systems, i.e. EIS tier as shown in Figure 1. While EIS tier usually means databases, such as Oracle and SQL-Server, it can also include non-database systems, such as file systems and existing legacy applications. As a result, multi-tier technology enables us to develop large-scale distributed enterprise systems to integrate heterogeneous databases and existing applications.

2.2 Concurrency vs. Contention

To efficiently support large numbers of simultaneous accesses from clients, multi-threaded servers and/or multiple servers are used to increase the processing capacity of application systems.

A multi-threaded server can create a set of threads, which handle requests concurrently. Application servers usually use a threads pool to manage these concurrent threads. The size of the thread pool can be dynamically adjusted according to the request load. However, it is a good idea for application servers to be able to control the size of thread pool. For example, IONA's *Orbix2000* [5] and BEA's *WebLogic EJB Server* [6] uses *max_watser_mark* and *weblogic.system.execute.ThreadCountert*, respectively, to set the maximum number of threads in the thread pools.

Similarly, an application server can be replicated on one or more server machines to handle requests on behalf of the same service name. Server replication usually needs supports from naming server [5], which is able to map one service name to a group of objects. An object group is a collection of service objects that run in different replicated servers. In this way, simultaneously coming requests can be efficiently processed concurrently. The two techniques can be used together in an application system.

While multi-threaded and replicated servers enhance the processing bandwidth of the servers, they may also increase the contention on application and databases. Each server thread/replicate sends request to database server to perform some transactional operations on data via a database connection. To maintain the data integrity, a transaction has to issue a lock on a table or a record when updating the data within it. The lock protects the data from being accessed by other transactions until the working transaction is committed or rolled back. The more transactions share data, the higher contention overheads may occur. Hence, the contention on database is application-dependent. In addition, contentions on network, database connections, machine resources (CPU cycles and memory sizes) also contribute to the overall contentions overhead.

Therefore, care must be taken to determine the degree of concurrency of an application system. We believe that given a specific application, the optimal degree of concurrency must exist such that the overall performance is optimal for a specific amount of client.

To determine the optimal degree of concurrency in an application system, a cost model is required to estimate the contention overhead on the application server. Such a cost model must reflect the relationship between concurrency and contention overhead so that the optimal degree concurrency can be achieved with the best tradeoff between the benefit from concurrency and its cost [4].

3. COST MODEL

In this section, we build a cost model to estimate contention overhead in an application server. For simplicity, we consider only one server, which creates a set of threads to serve requests concurrently. Figure 2 illustrates the dynamic process of handling concurrent requests in an application server.

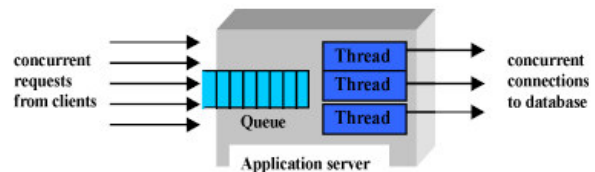


Figure 2: Handling concurrent requests in an app. Server

As shown in Figure 2, the overall contention overhead on an application server basically results from three resources. The first contention happens when all concurrent requests compete for being served by the application server. The contention may include network bandwidth and sockets ports. The second contention is the

waiting time in the queue for being served. The last contention comes from the concurrent access to the database via the concurrent server threads. Letting the number of clients be x and the number of server threads be y , we obtain the overall cost model as follows:

$$C = ax + \frac{bx}{y} + cy \quad (1)$$

where:

- C is the overall contention overhead in time;
- a is the network contention overhead ratio to per concurrent client;
- b is the server time for a single thread server;
- c is the contention overhead ratio to per connection to database;

To make the contention overhead independent from testing execution times and transaction types, we define the contention overhead C as the *Average Response Time* (ART) for all transactions, i.e. $C=T/N$, where T is the wall-time of a test and N is the total number of transactions.

$$y^* = \sqrt{\frac{bx}{c}} \quad (2)$$

where x is the given number of clients; b and c are the model parameters reflecting the characters of a specific application and platform.

This result can be further explained as follows:

- The network contention out of an application server has no direct effect on the degree of the concurrency of an application server. This means that we do not need consider the network contention outside of an application server when determining the degree of concurrency of the application server.
- The more concurrent clients result in the higher contention in an application server. The degree of the concurrency of an application server is proportional to the square root of the number of concurrent clients.
- The higher contention on database restricts an application server to using too many concurrent threads or processes. This case may happen when there are critical data in databases shared and updated by a large number of concurrent transactions.

4. CASE STUDY

First, the benchmark, *Stockonline*, is implemented using IONA's *Orbix2000* and *Oracle 8.1.5*. Then a set of tests is performed to measure the wall-times on the client side with different numbers of clients and server threads. Then the experimental results are used to obtain the parameters in the cost model (1). With these parameters, we can predicate the optimal numbers of threads in the application server for a specific amount of client load. The

predicated optimal numbers of server threads will be compared with the experimental results.

4.1 Specification of CSIRO Middleware Benchmark: Stockonline

4.1.1 Requirements

Stockonline is a simulation of a simple on-line stock-broking system. It enables subscribers to buy and sell stock, inquire about the up-to-date prices of particular stocks, and get a holding statement detailing the stocks they currently own. From a subscriber's perspective, the following services are offered:

Table 1: Complete list of services provided by stockonline

No	Service
1	Create Account
2	Update
3	Query Stock Value
4	Buy Stock
5	Sell Stock
6	Get Holding Statement

4.1.2 Database Design and Population

In *Stockonline*, four database tables are required to support the implementation of the above servers, i.e. *Subaccount* holding holds basic information on a subscriber, *Stockitem* holding information on stock prices, *StockHolding* containing information on the amount of a given stock that a subscriber holds and *Stocktransaction* storing buy and sell transaction records. Prior to each performance test, the database needs to be populated with initial test data, which guarantee that all tests run from the same start point.

Table 2: Database Sepecification

Table	Population
SubAccount	3000
StockItem	3000
StockHolding	3000*10
StockTransaction	0

4.1.3 Client Test Behavior

Each client thread performs a number of iterations of a fixed *transaction mix*. The transaction mix represents the concept of one complete business cycle at the server side. The transaction mix comprises 43 individual transactions listed in Table 1.

Generally, different transaction mixes can be used to model different types of business cycle. The transaction mix that we use in this paper attempts to achieve a balance between read-only transactions (51%) and update transactions (49%).

4.2 Implementation and Configuration

In this case study, we implemented the *Stockonline* based on IONA's *Orbix2000 V1.0* [5]. For simplicity, one multi-threaded server is used so that we can control the degree of concurrency by changing the numbers of threads in the CORBA server. Our EIS tier is a *Oracle 8.1.5* instance running on a separate machine. The *Oracle OCI* interface is used to implement database connections and operations [7]. A connection pool is implemented to support

connection pooling in the CORBA server. All database tables are initialized as described in Section 4.1.2 for each test. A multi-threaded client is implemented running on a separate machine to simulate concurrent clients. The basic test configuration is depicted in Figure 3.

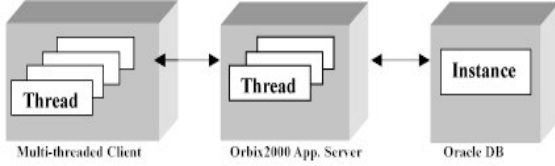


Figure 3: Test Configuration

4.3 Experiment and Modelling

First, we fixed the number of server threads, e.g. $y = 1$. Then, we ran the multi-threaded client for different numbers of clients, i.e. $x = 100, 200, 400, 800$, against the application server. Each client thread executes 10 iterations of the transaction mix. Thus the total number of transaction for each test is 430. The contention overheads were derived from the wall-times measured at the client-side by using $C=T/N$, where T is the wall-time in milliseconds and $N = 430$. In this way, we obtained the first row of contention overheads as shown in Table 4. The experiment was repeated for different numbers of server threads, i.e. $y = 2, 4, 8, 16, 32, 64, 128$. All contention overheads derived from the experimental results are listed in Table 4.

Table 3: Contention overheads in milliseconds

Num. of Threads	Num. of Concurrent Clients			
	100	200	400	800
1	1251	2447	5216	10405
2	841	1605	3096	6385
4	730	1443	2966	5910
8	724	1427	2945	5913
16	737	1435	2894	5773
32	760	1574	2974	5814
64	782	1579	3158	6178
128	813	1629	3389	6849

From Table 4, we can observe that while the contention overheads increase as the number of concurrent clients increase, the contention overheads change in a parabola shape as the number of server threads increase monotonically. This verifies our assumption that too high degree of concurrency in an application server degrades the server overall performance. Therefore, the optimal degree of concurrency indeed exists

$$\begin{aligned}
 a &= 6.65 \text{ msec / client} \\
 b &= 5.42 \text{ msec} \\
 c &= 5.24 \text{ msec / thread}
 \end{aligned}$$

Then the cost model with the three parameters is used to estimate contention overheads, which are compared with the experimental results in Figure 4. As shown in the four figures, basically, our cost model represents the pattern and trend of the contention overhead

and thus can be used to determine degrees of the concurrency in application servers.

4.4 Using the Cost Model to Determine the Optimal Concurrency of Stockonline

Based the above parameters, we can derive the optimal number of server threads by applying (2). A set of theoretical optimal numbers of server threads for different number of concurrent clients is listed in Table 2. As show in Figure 4, basically, the theoretical optimal numbers of server threads fall in the area where the best performance could be achieved.

Table 4: Theoretical optimal numbers of server threads

Num of Clients : x	Optimal Num. of Server Threads : y^*
100	10
200	14
400	20
800	29

Alternative solution to this issue is to dynamically determine the concurrency of middle-tier servers according to the load at real-time. However, this solution requires the support from the low-level load and thread control mechanisms, which are rarely available in current middleware products. We will not deal with that in this paper.

5. RELATED WORK

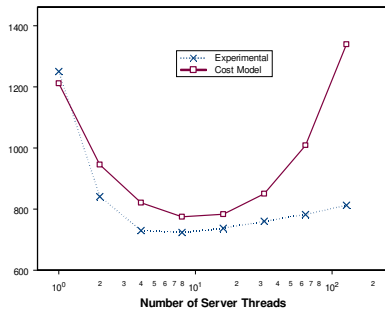
This work is initially inspired by [4] and related to [5][6][10][11].

In [4], the authors developed a communication cost model to estimate the overhead of a network of workstations (NOWs). The cost model is used to drive the optimal of task sizes such that the parallelisms are maximized and communication overheads are minimized. In this paper, we reuse the idea to model contention overhead in middle-tier servers. Similarly, we determine the best degrees of concurrency in middle servers using the cost model.

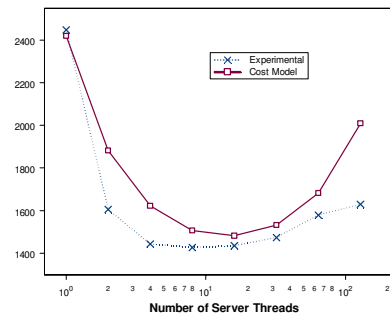
Chiesa addressed a similar issue for tuning Encina transactional Applications In [5]. He suggested that in some cases, increasing thread pool sizes improve throughput by enabling ‘group commit’. He also observed that at a specific point, using additional threads just result in longer transaction response times. However, he did not provide a method or model to find the ‘turning point’.

Douglas C. Schmidt’s CORBA term presented a study of thread pool for RT-CORBA in [6]. In [6], they evaluated various thread pool strategies and pattern including Half-sync/ Half-async and Leader/Followers. They observed that in some cases, using a small number of threads yields better throughput than a higher number of threads. They attributed the performance degradation to overheads of memory allocation, locking and data movement. Again, no cost mode is developed to model these overheads. Thus, they cannot determine the optimal thread pool sizes.

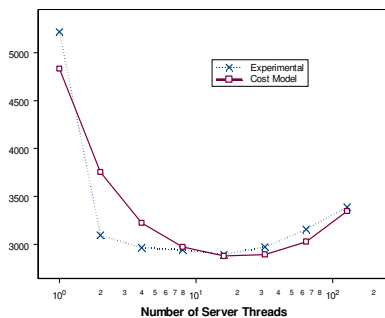
While MVA [10] and LQNS [11] can be used to model sophisticated multi-tier systems more preciously, their models are algorithm-oriented and thus no closed-form analytic expressions are provided for either the overall performance or the concurrency configuration of middleware application servers.



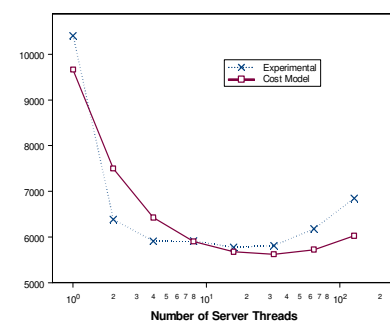
(a) 100 Clients



(b) 200 Clients



(c) 400 Clients



(d) 800 Clients

Figure 3: Experimental Results vs. Cost Model

6. CONCLUSIONS

In this paper, the problem of determining degrees of the concurrency in application servers has been explored. The relationship between concurrency and contention was discussed and the contention overhead was defined. Based on the discussion and definition, we proposed a cost model to estimate overall performance for middleware application servers. We demonstrated modeling and application of our cost model by using the CSIRO internal middleware benchmark implemented in Orbix2000 and Oracle.

The insight behind determining the optimal degrees of the concurrency in application servers is to achieve the best balance between the benefit from concurrency and the contention overhead. This insight can direct us to configure application servers to achieve the best performance.

7. ACKNOWLEDGMENTS

The author of this paper wishes to thank IONA for providing Orbix2000 evaluation version and CMIS's Splus group for their assistant in using Splus in this study.

8. REFERENCES

- [1] Ian Gorton. *Middleware technology evaluation series—OrbixOTM3.0*. CSIRO Australia, 2000. www.cutter.com/itgroup/reports/orbix.html
- [2] Phong Tran. *Middleware technology evaluation series – evaluating Forte 4GL release 3*. CSIRO Australia, 2000. www.cutter.com/itgroup/reports/forte.html
- [3] Longhow Lam. *An introduction to SPLUS for windows*. CANdiensten, 1999, ISBN 90-804652-2-4
- [4] Shiping Chen and Jingling Xue. *Partitioning and scheduling loopson NWS*. J of Computer Communications, 22(11):1017-1033, 1999
- [5] D.P. Chiesa. *T Tuning Encina Applications: A Practical Guide*. IBM white paper, www.transarc.ibm.com
- [6] Marina Spivak, Douglas C. Schmidt, and Ron Cytron. *Optimizing Thread-Pool Strategies for Real-Time CORBA*, Proc. of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001), Snowbird, Utah, June 18, 2001. www.cs.wustl.edu/~schmidt/new.html

- [7] *Orbix 2000 Programmer's Guide*. IONA Technologies PLC, March 2000
- [8] *BEA WebLogic Server Administration Guide*. BEA Systems, 2000
- [9] *Oracle Call Interface Programmer's Guide* Release 8.1.5. Oracle Corporation, Feb. 1999
- [10] M. Reiser and S. Lavenberg, *Mena-value Analysis of Closed Multi-Chain Queuing Network*. J of ACM, vol. 27, no. 2, 1980
- [11] J. A. Rolia and K.C. Sevik, The Method of Layers, IEEE Transaction. Software Eng. vol. 21, no. 8. 1995, pp. 689-700