Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

USING JAVA METHOD TRACES TO AUTOMATICALLY CHARACTERIZE AND MODEL J2EE SERVER APPLICATIONS

Darpan Dinker
Herb Schwetman

Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025, U.S.A.

ABSTRACT

This paper describes a novel framework used to characterize a J2EE (Java Enterprise Edition) application and develop models of the application by using Java method tracing in a Java-technology based application server. Application servers are critical to large-scale, online servers and serve as middleware to provide secure access to transactional, legacy and web services. The tracing tool in this framework gives a detailed and comprehensive view of the sequences of methods invoked as the application server processes requests. The output of this tool is processed and automatically summarized into a set of transaction profiles which form the input for a simulation model of the application server and its related components. These profiles have proven to be a useful abstraction of the behavior of the transactions processed by the system. After describing the tool and the model, the paper provides results of validation runs and discusses the usefulness of quantitative measurement, analysis and modeling in some areas of system design and system deployment. The models help architects, designers, developers and deployers explore the different facets of performance during all stages of an application's life-cycle, especially during concept development and prototyping.

1 INTRODUCTION

Studies of system performance require accurate descriptions of the system workload. Typically, the goal of a performance study is to gain an understanding of how the system is behaving currently and to make recommendations about corrective steps which can result in improved performance in the future. Each of these steps depend on several ingredients, including an accurate description of the system components and how they interoperate and on an accurate description of the demands being made by the elements of the workload for use of these components.

This paper describes an approach to workload characterization that is based on a trace of Java method invocations created by a special tool, the ByteCode Instrumentation (BCI) tool, developed at Sun Microsystems, Inc. A Java application that is instrumented using BCI can be studied in great detail, and a summary of this trace can be processed to give a detailed and accurate characterization of the activities of the application.

The paper gives a description of a model which uses this characterization to enable the user to make accurate predictions of performance of the system operating in new environments.

2 BACKGROUND

Many modern on-line services are implemented using a multi-tiered approach, placing different services on different tiers of the total system. A typical configuration is shown in Figure 1. In Figure 1, the system is made up of three tiers, as follows:

- The web server tier,
- The application server tier, and
- The database tier.

In more detail, a user of this system is connected to the Internet and is accessing the system using a web browser. The browser sends a request for service to a node in the web server tier. In some cases, the requested information is in the form of a static page stored on the web server nodes; in these cases, the page is returned immediately to the requesting browser. In other cases, the requested information must be developed dynamically for the specific request.

These requests are forwarded to a node in the application server tier. A forwarded request invokes a particular application Java Servlet or program which determines the content for the request and delivers this back to the web server, which in turn delivers this back to the requesting browser. Some of the requests require that the application find and/or update data on a node in the database tier.

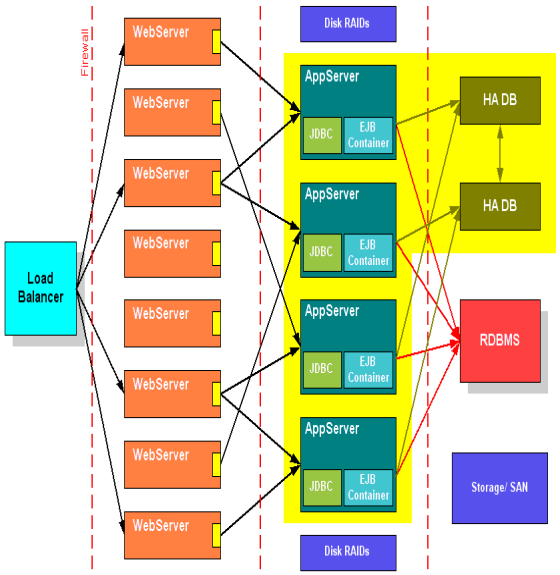

Figure 1: Multi-tiered Online System

The focus of this paper is the performance of the nodes in the J2EE application server tier. In many instances, the J2EE services such as Web services, enterprise beans, message beans, etc. in this tier are implemented using a "container" environment. This container environment provides a multitude of services (e.g. security, transaction, life-cycle management) which are used by the server applications written in Java for the J2EE platform. One such environment is the Sun Java Enterprise System (JES) Application Server (called AppServer) from Sun Microsystems, Inc. With AppServer, application developers can create their applications that are accessed through Web services (HTTP, XML/SOAP, JMS, etc.) or rich-client paths such as RMI/IIOP. An example of a J2EE application is an online retail E-commerce web-site that has serves a product catalog, has a shopping cart, and a check-out module.

The performance of the J2EE applications operating in the application server tier can have a large impact on the performance of the total system (across tiers). Furthermore, the performance of the system can be a major factor in the user's perception of the service delivered by the system, namely response time. Systems maintaining service-level agreements (e.g. 90-th percentile response time < 1.2 sec) can attract and retain customers for the site, while systems exhibiting "poor levels" of performance can discourage customers from visiting the site. From the website administrator and deployer's perspective, throughput is an important metric, in addition to response time. Throughput provides the ability to gage the system's ability to handle multiple user requests concurrently.

The paper begins with a description of a tool (called BCI) that has been developed to give detailed views of Java applications operating on AppServer. The use of BCI is illustrated by using it to study the operation of the Trade2 benchmark suite (An et al. 2002 and Trade2) of transactions. The BCI outputs are used to create a statistical characterization or profile of the Trade2 transactions. Finally, a simulation model that uses these profiles is described. Some outputs from this model are compared to the originally collected summary data, to gage the accuracy of the model. In addition, some "what if" studies show how the model can be used to predict system performance in altered operating environments.

3 THE BCI TOOL

A Java program is compiled into its machine independent representation called Java byte-code that is interpreted and executed by the Java Virtual-Machine (JVM) available on different platforms. Java supports automatic garbage collection (GC), so that the user program does not have to explicitly free objects on the heap. A JVM such as the Java HotSpot^{T} virtual machine maintains different regions in the heap for segregating objects in the heap according to their life span (young and old generations) and runs garbage collector thread(s) to free un-referenced objects when a heap partition is full. Although automatic garbage collection is a boon to programmers, the feature trades off ease-of-use for performance. Another feature of a JVM, called Just-in-time (JIT) compilation, is the ability to compile frequently executed streams of byte-codes to machine representation that result in faster execution. To summarize, features of the JVM can impact the performance of a Java program both positively and negatively.

As indicated above, Java has been widely used to implement server programs operating in the application server tier, and the performance of these programs (as reflected in speed of execution) can be critical to perceived performance of the total system.

Several Java performance tools provide the ability to identify "hot" methods or lines of code. When compared to other methods, the "hot" methods could be using more resources such as processor (CPU), memory (heap), lock (latency), disk and network IO, etc. affecting the response time of a service. Many Java performance tools use the Java virtual machine profiler interface (JVMPI) and also the Java virtual machine tool interface (JVMTI); JVMTI is now included in the Java Developer Kit (JDK) 1.5 release and provides the ability to instrument Java classes statically, at load-time or dynamically. Several profilers, such

as JFluid (Dmitriev 2004) provide developers an intuitive graphical user interface to display "hot" methods.

Many techniques for creating profiles (Harkema et al. 2002 and Putrycz 2004) have evolved; these techniques include profilers and trace routines. Typically profilers use a sampling technique to estimate the time spent in each method, while trace routines actually record each invocation of the specified methods. One advantage of using tracing routines is that counts of the number of method invocations are obtainable; another advantage is that by using the order of invocation, an accurate depiction of the method call-tree is also obtainable. It can be important to determine that one method can appear at multiple points in the execution paths (equivalently in different branches of the call tree) of the application.

The Byte Code Instrumentation (BCI) framework, developed at Sun, contains an instrumentation tool that statically instruments Java classes. The process of instrumentation consists of adding pre-compiled byte-codes (representing measurement calls) to the target Java application. The instrumentation collects a trace of entry and exit points of Java methods and logs this trace to a file at run-time. The objective of the BCI collection framework is to provide a trace representing the user's application executing in the J2EE containers. In order to achieve this objective, the BCI framework intercepts and captures the first points of request processing per unique request, per thread for containers such as:

- Java Server Pages (JSP)/Servlet Web container,
- Enterprise Java Beans (EJB) container,
- Object Request Broker (ORB),
- Message Driven Bean (MDB) container, etc.

Tracing can be dynamically turned on or off by invoking a management service in the BCI run-time. When a request that is being processed by the AppServer arrives at an interceptor, a new record is created for the request using a special buffer in the heap. At subsequent arrivals and departures from instrumented methods, the instrumented code collects some thread or LWP (light-weight process) level measurements such as the wall-clock time and CPU consumption. When the request processing departs the interceptor, the buffer containing traces of methods and their corresponding measurements are flushed to a log file. Post run-time or after tracing is turned off, the log file can be processed, to give a comprehensive view of the methods that executed while the program was being observed.

A standard report gives the hierarchy of the methods called positions (or levels) in the call tree, along with number of calls, and the average CPU time per call (inclusive) and the average elapsed (wall-clock) time per call (inclusive). The term inclusive means that the time reported is for the method and all of the methods below it in the call tree. Table 1 shows an example of the output for a summary of a BCI log file. In Table 1, the term "serv" refers to service or wall-clock time.

Some other tools and papers (Harkema et al. 2002) have used similar techniques as described above. These techniques are simple enough that tracing can be achieved using an existing tool or after a few enhancements to an available Java instrumentation tool.

Table 1: Sample BCI Summary

Level-id	Method	Count	Incl cpu	Incl serv
0-18	TradeScenario..	142235	0.004325	0.051677
1-19	HttpServlet	142235	0.004261	0.051538
2-20	HttpServlet	142235	0.004257	0.051531
3-21	doAccount	14828	0.003372	0.045435
4-22	postInvoke	14828	0.001261	0.012751
5-23	save	14828	0.001042	0.012365
6-24	save	14828	0.001037	0.012355
7-25	doExecuteUpd	14828	0.000730	0.005074
8-26	chl.execute	339	0.000668	0.001636
8-27	chl.execute	14828	0.000672	0.004976
9-28	chl.execute	14828	0.000668	0.004970
7-29	getConnPool	14828	0.000224	0.005298
8-30	chl..execute	14828	0.000123	0.001005
7-31	resourceClosed	14828	0.000010,	0.001813

4 ACCURACY, OVERHEAD AND COVERAGE

The issues of accuracy, overhead and coverage are critical to the usefulness of a data collection tool such as BCI. In particular:

- Accuracy refers to the collection of data which accurately characterizes the operation of the software being analyzed. Accuracy is non-trivial to validate and high-level performance tools depend on tools and support in the platform (operation system and hardware counters) to assess accuracy.
- Overhead refers to the extra-cost (processor cycles, memory, I/O) associated with collecting the data. Data collection, data storage and related run-time processing affect and alter the execution, thus causing variations in the execution time and/or resource utilization. In most cases one observes degraded performance, but in a few interesting cases, a myriad of complexities in today's computer systems (e.g. caching effects in the memory hierarchy) can result in better performance. Most tools available today do not and cannot determine their overhead and thus do not compensate for their overhead. As most performance engineers have observed, there is a trade-off between measurement detail and overhead.
- Coverage refers to the amount of execution time for the application that is actually recorded by the data collection tool. Several Java-based performance tools do not provide insights into code exe-

cuted outside the JVM machine and thus lack coverage outside the JVM machine. Several tools, including BCI, measure using interception points and thus miss measuring code executed in the JVM before the interception points were executed in the call stack. Most CPU's have two distinct modes of operation: system or kernel mode and user mode. Another problem in coverage arises from the inability to collect system-mode CPU time, which is the case with the BCI tool and several other tools. Tools that do not support JVM machine profiling obviously do not produce detailed results for Java applications.

We made several attempts to validate the data provided by commonly available tools and decided to proceed with a custom solution, which we describe briefly here. In order to address the tool overhead problem, we compared several instrumented and un-instrumented runs of micro-benchmarks and also some well-known benchmarks and observed that overhead in the case of BCI was proportional to the number of instrumented methods that were invoked during request processing. To get a global perspective on overhead, we used system measurement tools available with the Solaris operating system, and compared the performance of instrumented and un-instrumented runs of the Trade2 benchmark. In order to validate accuracy of the BCI tool, we compared the collected traces for a method across several sampling tools that had low overhead under low sampling intervals. In order to address the inability of the BCI tool to capture system CPU time and missing measurements before the interception points, we utilized standard UNIX tools such as *mpstat* and *vmstat* to provide transaction metrics that, in turn, were used to deduce the missing time in the BCI traces. As we observed during our experiments and exploration, this technique worked out very well, given that the coverage of BCI was more than 70%. It appears that BCI covers most of the interesting variations in applications that are of interest to J2EE technology architects and developers. Another observation was that the wall-clock coverage with BCI was correct, and these did not require further validation or correction. Thus by using system tools (e.g., *mpstat*) the amount of "lost" CPU time can be estimated, and this lost time is incorporated into the simulation model described below.

A comprehensive description of a Java method instrumentation performance toolkit as applied to a CORBA implementation is given in (Harkema et al. 2002).

5 THE TRADE2 BENCHMARK

Trade2 is a benchmark that is often used to test the performance of application servers. In (An et al. 2004), Trade2 is described as "a collection of Java classes, Java Servlets, Java Server Pages and Enterprise Java Beans integrated into a single application. It is designed to emulate an online brokerage firm". (A database is maintained by Trade2, which holds account information and stock quotes.) The Trade2 transactions are as follows:

- doAccount,
- doAccountUpdate,
- doBuy,
- doHome,
- doLogin,
- doLogout,
- doPortfolio,
- doQuote,
- doSell,
- valveSave (used only for http-session replication).

An important feature of the Trade2 benchmark is the ability to enable *failover* for user application objects such as the stateful session bean (an Enterprise Java Bean with application logic) and the HTTP session (a server-side state representing a web-based client). AppServer provides failover capabilities for these objects using techniques such as storing copies of the objects in shared disk storage or in the backend database tier (persistent storage), or maintaining multiple in-memory copies within the AppServer tier (state replication).

With failover enabled, when an AppServer process or a compute node fails, user requests are diverted to another process or another compute node in the AppServer tier which then uses the objects from persistent storage or the replicas from other AppServer processes to retrieve the most up-to-date user state. In this study, a separate high availability database (HADB) was used to implement failover for the AppServer application.

The benchmark setup for the tests described in this paper used a Sun utility, RequestRunner, to serve as the workload generator. RequestRunner simulates the operation of the Trade2 clients by presenting a stream of Trade2 requests to the AppServer nodes. Key parameters to RequestRunner include the number of clients (users) to be simulated during the test and the average "think time" interval for each client interaction with the AppServer.

6 TRANSACTION PROFILES

The goal of this exploration is to develop system models that can be used to predict the performance of the Sun JES Application Server executing on the application server nodes in a multi-tier system. The initial goal is to apply the analysis and modeling techniques to predict the performance of the Trade2 benchmark with special emphasis on the behavior of the state replication services in the Application Server.

The BCI summaries provide a detailed and comprehensive view of the behavior of the AppServer as reflected in the sequences of method invocations. This level of detail permits a variety of investigations into the behavior of AppServer and its interactions with other components such as the replication service and the database service to be made. For the purposes of parameterizing the system model, a set of service categories was created, and the BCI method costs (counts, CPU times and wall-clock times) were aggregated into these service categories. In addition, the CPU times and wall-clock times were converted from *inclusive* times to *exclusive* times (the method times without the times for the lower level methods). The service categories used in these experiments were as follows:

- Lock: Methods associated with obtaining a lock.
- Bean-management: Methods associated with converting Java object data from internal form to external form.
- Repl-prep: Methods associated with preparing for an interaction with the state-replication service.
- DB-prep: Methods associated with preparing for an interaction with the database service.
- Repl-IO: Methods associated with an interaction with the state-replication service.
- DB-IO: Methods associated with an interaction with the database service.
- Base: All of the methods that are not included in any of the above categories.

As an example of a profile for the doAccount transaction, consider the data in Table 2. In this profile, the *Count* field in the first line refers to the number of times this transaction was executed during an experiment. In the remaining lines, the *Count* field specifies the number of times the category was invoked per-call to the transaction. The *CPU* field designates the average CPU time (in seconds) used by the methods in the category per call to the service category and the *serv* field designates the average wall-clock time experienced by the methods in the category (in seconds) per call to the transaction.

Table 2: Profile of DoAccount Transaction

Method/Cat.	Count	Cpu time	Serv time
doAccount:	29583		
base:	7.00	0.0002281	0.0004531
lock:	1.00	0.0001019	0.0042386
bean-mgmt:	3.00	0.0001223	0.0062204
repl-prep:	4.00	0.0000320	0.0005392
Db-prep:	42.00	0.0000075	0.0000171
repl-IO:	2.02	0.0004067	0.0030005
Db-IO:	10.00	0.0000062	0.0009666

Some of the transactions invoke another transaction; the profiles for transactions that do this have a call-transaction line, showing the transaction being called. For example, the doAccountUpdate transaction calls the doAccount transaction. These profiles shown in Table 3, which are automatically constructed from the BCI summary report, give a comprehensive view of the average use of the CPU and the average wall-clock time experienced by a call to the Trade2 transactions. The transaction profiles used by the model consist of the nine or ten profiles, one for each transaction type in the Trade2 benchmark.

Table 3: Profile of the doAccountUpdate Transaction

Method/Cat.	Count	Cpu time	Serv time
doAccountUpdate:	10871		
base:	4.00	0.0001789	0.0002655
lock:	0.00	0.0000000	0.0000000
bean-mgmt :	2.00	0.0001395	0.0025863
repl-prep :	0.00	0.0000000	0.0000000
db-prep:	50.00	0.0000083	0.0000150
repl-IO :	0.00	0.0000000	0.0000000
db-IO:	11.00	0.0000071	0.0014276
called trans :	10871	doAccount	

7 MODELING TRANSACTION PROCESSING

The predictive capabilities that were required for the project are provided by a simulation model; this model has simulation objects that represent the hardware and software components shown in Figure 2: the system (the nodes or CPU's, and disk drives), the clients, the application server(s) executing Trade2 transactions, the replication server(s), and the database server).

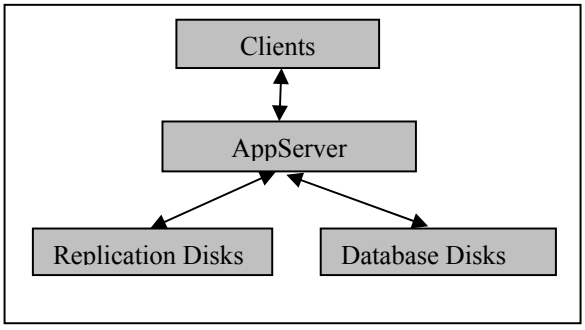

Figure 2: Diagram of System Model

The model is written using CSIM (Mesquite 2002). The component that mimics the AppServer processing a particular Trade2 transaction uses the counts and average CPU times from the profile for that transaction to mimic usage of the simulated CPU in the model. The average wall-clock (serv) times are used to inject delays for the lock category and the bean-management category. The

delays can be derived from a component model representing the lock and bean management services.

In more detail, a transaction process originates at the client node; it travels to the AppServer node, where the type of transaction is determined by a routine that accurately models the Trade2 servlet. With a specific transaction type (equivalently with a specific transaction profile) attached, the transaction process "executes" the simulated routines that model the behavior specified by the profile. Using the "doAccount" profile (see Table 2) as an example, the transaction first visits the simulated CPU seven times using an average of 0.0002881 seconds per visit (the *base* service category). Next, to model the activity of the *lock* category, it visits the simulated CPU once, using an average of 0.0001019 seconds and then occupies a "lock" resource for an amount of time so that the total elapsed time for the lock visit (CPU response time plus lock service time) is the lock category serv time (0.0042386 seconds). Simulating the activity of the *bean-management* service category is similar to the activity of the *lock* category. The activities of the *repl-prep* and *dp-prep* categories are CPU usage intervals (as with the *base* category); the delays arise from the queueing at the CPU.

Modeling the behavior of the *repl-io* and *db-io* categories is more complicated. It would be possible to use the average serv (wall-clock or elapsed time) values from the profiles, but this would minimize the predictive uses of the model. Instead, a IO category call is modeled by having the transaction visit a separate (simulated) server - a replication server for a *repl-io* category call and a database server for a *db-io* category call. All of the visits to a remote server are modeled as an average visit (i.e., the behavior of a visit is not specified by the type of transaction that caused the visit). The parameters that control a visit to a remote server are the mean CPU time per visit, the probability of a disk operation per visit, the mean disk operation time, and the mean server delay.

The values of the parameters are determined by combining the counts from the BCI reports with data from system data that is collected for an experiment; on UNIX, the *iostat* and *mpstat* utilities give the necessary information: CPU utilization, disk operations per second and average time per operation.

Estimating the mean server delay is less straightforward. This delay is intended to model that part of a visit that is not covered by waiting for the CPU and the disk drive plus the time spent using the CPU and the disk drive. This delay could include network times, waiting for locks on the server, waiting for connections, and waiting for other types of activities to complete. Currently this delay is estimated by computing an estimated active time (CPU service plus the probability of a disk operation times the disk service time) and then inflating this by a factor. While not entirely satisfactory, this approach does yield fairly accurate results. An alternative to this procedure of estimating a delay interval would be to implement a more accurate model of the each type of server. Adding models of the node interconnection network components could also enhance the fidelity of the results.

One issue that is handled in the model is the "lost CPU time" on the AppServer - the CPU time used by the AppServer that is not reported by the BCI tool (mentioned above). The amount of "lost time" can be estimated by comparing the total CPU time reported in the BCI summary and CPU utilization reported by a system utility such as *mpstat*. The model uses this estimate of lost time to parameterize extra processes on the simulated AppServer node. These extra process use the simulated CPU(s), in short slices of time, so that the "lost time" is in fact accounted for in the model. These extra processes are necessary to achieve two results:

- The CPU utilization for the AppServer nodes are correct, and
- The waiting time for the transactions at the CPU is approximately correct.

Without these extra processes, the CPU utilization would be lower than that reported in the data and the response times (serv times) for the transactions would be lower than the times reported in the BCI summary.

8 VALIDATION OF THE MODEL

A number of experiments on a variety of hardware and software configurations were run. In one experiment, Trade2 was run with 40 simulated clients sending requests to a pair of Sun V20Z nodes; each V20Z node has two AMD Opteron CPUs (with clock frequencies of 1.6 GHz) plus two gigabytes of main memory and a single disk drive. In addition to the two AppServer nodes, there was another pair of V20Z nodes hosting the replication servers and a single V40Z node (with four AMD Opteron CPUs) hosting the database server.

In the initial experiments, variations in the number of accesses to the database server per transaction were observed. Some investigation revealed that in the doQuote transaction, stock quotes are retrieved from a stock-quote cache and that quotes in the cache are expired after a time-out interval. To guarantee consistent results for varying workloads and different architectures, the stock-quote cache was disabled and all stock quotes were retrieved from the database server. The alternative would have been to implement a more detailed model of the cache with the time-out feature.

In order to make meaningful comparisons of different system architectures, the transaction injection rate was fixed to be as close to 400 requests per second as was possible. This was accomplished by modifying the load generator (RequestRunner). Specifying a fixed request rate

means that using transactions per second (TPS) as the primary performance metric is not meaningful. In particular, using TPS to assess the validity of the simulation models was not reasonable (almost any model should be able to produce the specified TPS). Instead of TPS, the average simulated CPU time per transaction and the average simulated wall-clock time (sometimes called the average response time) per transaction are used as bases for comparison. In addition, the CPU time by service category and the wall-clock time by service category are compared. The graphs in Figures 3 and 4 show the measured versus modeled comparisons for two experiments: one using the replication service described above and one that does not use bean and state replication.

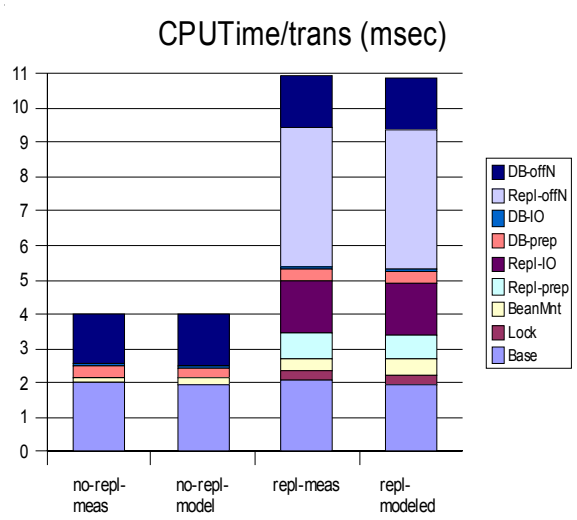

Figure 3: Comparison of CPU Time/Transaction with Service Categories

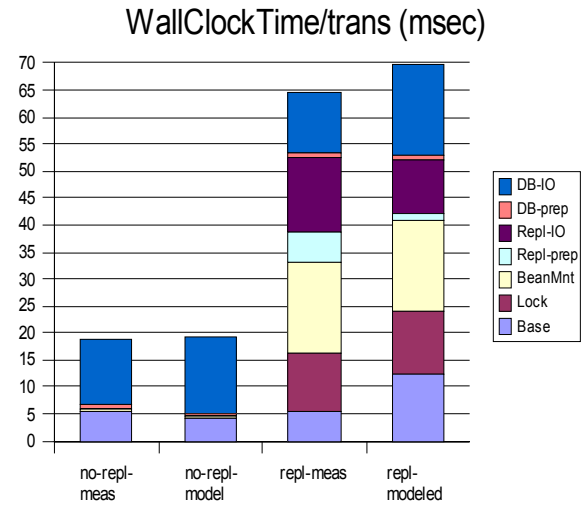

Figure 4: Comparison of Wall-clock Times/transaction with Service Categories

9 USES OF QUANTITATIVE MEASUREMENT, ANALYSIS AND MODELING

The framework described in the preceding sections can be used in a number of ways to analyze and improve system operation. In particular:

- Capacity planning: An important aspect to successfully deploying servers in production environments is capacity planning. Analytical or simulation models based on the measurement and analysis of systems can be used by a system administrator to scale important metrics. e.g., expected peak load for holiday shopping.
- Architecture and design analysis: Workload characterizations (profiles) produced by the methodology described in this paper have the ability to produce high-level service abstractions for each transaction type in the workload. This ability to characterize and model a service rather than using individual method calls in a trace provide a powerful capability for analyzing system architectures and designs. For example a call-stack trace of methods representing several file IO operations can be coalesced into a "File-IO" category or profile entry.
- Trade-off analysis: A critical facet of system architecture and design is trade-off analysis, where the important metrics are either chosen and added as model parameters or are the results of the execution of the model. Variations in the model parameters, the system architecture or the design can be compared using modeled results. For example, a model for a website may study the use of two, four and eight processor systems and evaluate system cost for the targeted transaction rate as a metric.
- Root cause analysis: Performance bottlenecks can be analyzed when a system has been modeled in sufficient detail. The system model can be used to analyze resource constraints, locking delays, etc. For example a model of a middleware system can predict delays caused by reaching an upper limit for a connection pool to the database. Another example is exhausting disk bandwidth while the processors are under utilized.

10 SUMMARY

A new tool, called BCI, for instrumenting Java server applications has been described. This tool gives an accurate, detailed and comprehensive trace of the methods used by the application, along with the inclusive CPU times and inclusive wall-clock times for each method. This tool was used to study the behavior of the Sun JES Application

Server processing runs of the Trade2 benchmark. Summaries from these runs were then used to create statistical profiles of each Trade2 transaction.

A simulation model used the automatically created transaction profiles to simulate the behavior of these transactions on simulated systems. The validation runs suggest that a model combined with these profiles can yield accurate results.

In the future, the project will use these profiles together with the simulation model to study a variety of techniques for improving replication services for application servers. One approach to improving replication service is to use new components, including new communications hardware. Another approach could be based on changes to the implementation of replication services, both on the replication services nodes and within the application server. It is possible to alter the call counts and service times so as to reflect these kinds of changes.

The use of the BCI tool, coupled with the summary programs and the simulation model described above are proving to be a flexible tool for analyzing performance of the Application Server and for predicting the impacts of changes to both the hardware and software components that make up these multi-tier systems.

TRADEMARKS

REFERENCES

An, Y., T. K. T. Lau,, and P. Shum. 2002. A Scalability Study for WebSphere Application Server and DB2 Universal Database. DB2 Universal Database Performance & Advanced Technology, IBM Toronto Lab, IBM Canada.

Dmitriev, M. 2004. Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation. In *Proceedings of the 4th International Workshop on Software and Performance*, 139 – 150, January 14-16. Redwood City, CA, ACM.

Harkema, M., D. Quartel, B. M. M. Gijsen,, and R.D. van der Mei. 2002. Performance Monitoring of Java Applications. In *Proceedings of the Workshop on Software and Performance*. Rome, Italy, 114 – 127, ACM.

J2EE 1.3 Specification. <http://java.sun.com/j2ee/>, Sun Microsystems, Inc.

Java Enterprise System Application Server. <http://docs.sun.com/app/docs/coll/ApplicationServer8_ee_04q4>, Sun Microsystems, Inc.

JVMPI - Java Virtual Machine Profiler Interface. <http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html>, Sun Microsystems, Inc.

JVMTI – JVM Tool Interface. <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html> Sun Microsystems, Inc.

Mesquite Software, Inc. 2002. *CSIM19/C++ User's Guide*, Mesquite Software, Inc.

Putrycz, Eric. 2004. Using Trace Analysis for Improving Performance of COTS Systems. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research,* 68 – 80, IBM.

Trade2 Benchmark. Available at <http://www.ibm.com/developerworks/db2/library/techarticle/0205an.html>, IBM.

AUTHOR BIOGRAPHIES

DARPAN DINKER is a Senior Staff Engineer at Sun Microsystems Laboratories (Menlo Park, California) in the Computer Architecture and Performance group. He leads an exploration into breakaway multi-tier clustered systems where the research includes workload characterization, analysis, modeling and prototyping. His current research interests include high performance integrated hardware-software architectures, transaction processing and fault tolerance/recovery in clustered systems. His prior experience is in the architecture, engineering, performance, sustaining, and support/service of commercial software system products and applications. He holds a Bachelors degree in Computer Engineering from University of Pune, India and has studied advanced topics in computer architecture and databases at Stanford University. His e-mail address is <darpan@sun.com>.

HERB SCHWETMAN is a Senior Staff Engineer at Sun Microsystems Laboratories in the Computer Architecture and Performance Group; he has been at Sun Microsystems, Inc. since 2001. From 1994 to 2001, he was President and CEO of Mesquite Software, Inc., a startup company in Austin, TX, focused on simulation software. He was a Senior Member of the Technical Staff at MCC in Austin from 1972 to 1984 and was a Professor of Computer Sciences at Purdue University from 1972 to 1984. He received a Ph.D. in Computer Sciences from The University of Texas in Austin in 1970, a M.S. in Mathematics from Brown University, and a B.S. in Mathematics from Baylor University. His e-mail address is <herb.schwetman@sun.com>.