An Observation-Based Approach to Performance Characterization of Distributed n-Tier Applications

Calton Pu, Akhil Sahai†, Jason Parekh, Gueyoung Jung, Ji Bae, You-Kyung Cha, Timothy Garcia, Danesh Irani, Jae Lee, Qifeng Lin
Georgia Institute of Technology, HP Labs†

akhil.sahai@hp.com, {calton, jparekh, gueyoung.jung}@cc.gatech.edu

Abstract- The characterization of distributed n-tier application performance is an important and challenging problem due to their complex structure and the significant variations in their workload. Theoretical models have difficulties with such wide range of environmental and workload settings. Experimental approaches using manual scripts are error-prone, time consuming, and expensive. We use code generation techniques and tools to create and run the scripts for large-scale experimental observation of n-tier benchmarking application performance measurements over a wide range of parameter settings and software/hardware combinations. Our experiments show the feasibility of experimental observations as a sound basis for performance characterization, by studying in detail the performance achieved by (up to 3) database servers and (up to 12) application servers in the RUBiS benchmark with a workload of up to 2700 concurrent users.

I INTRODUCTION

Rapid evolution and growth of web-based applications (e.g., in electronic commerce) have established 3-tier applications with web servers, application servers, and database servers as the main software architecture for current and future enterprise applications. However, 3-tier (and more generally, n-tier with finer-granularity components) applications are notoriously difficult to manage due wide variations in workload. For example, web applications have been reported to have peak loads many times that of sustained load [1]. Consequently, it is almost inevitable in a web environment that any specific hardware configuration will become over-provisioned for the sustained load, but under-provisioned for the occasional peak load. These variations inherent in n-tier applications cause problems for traditional analytical methods such as queuing theory [5], for example, Mean Value Analysis that focuses on the steady state and average behavior.

Complementing theoretical modeling, an alternative approach to performance characterization of complex systems is direct experimentation and observation of system behavior. Observation of real experiments avoids the problem of error propagation or reaching the limitations of assumptions made by analytical models when scaling up or out by a significant number. In addition, experiments provide validation points for model-based characterizations. In this paper, we outline an observation-based approach and show its feasibility as well as promising results in the performance characterization of n-tier applications over a wide range of configuration settings. Measuring and plotting performance of n-tier applications covering a sufficiently large set of parameters and software/hardware combinations can help system analysts make informed decisions at configuration design time. During operation of the system when workload evolves, our observed performance can serve as a guide to system operators and administrators in reconfigurations to obtain reliably the desired service levels.

Although observation is a well-known scientific method to understanding and describing complex system behavior, it has some practical limitations when applied to n-tier applications. It is common in large-scale scientific observations (e.g., high energy physics accelerators and large astronomical telescopes), to build significant experimental infrastructures at a high cost and long construction time. This is acceptable and justified for once-in-a-lifetime discoveries such as Higgs Boson. Unfortunately, computer system configurations have relatively short life span of a few years or months, requiring frequent repetition of experiments. High experimental costs or long construction time would render the observational approach unsuitable for computer systems.

To achieve rapid and low cost experimental performance evaluation, a software infrastructure to generate and manage performance evaluation experiments of n-tier applications has been developed in the Elba project [4][8][10][12]. This infrastructure builds on the Mulini code generator to automatically and systematically generate, deploy, and benchmark n-tier applications. Through automation, Mulini tools lower the costs and improve the reproducibility of computer system performance measurement experiments.

The main contribution of this paper is a practical demonstration of the feasibility of an observation-based approach to performance characterization of n-tier applications. This demonstration is done through a number of actual experiments. Although each benchmark experiment measures the performance of a single combination of settings for a sustained period, the variety and coverage of our experiments give us confidence in using the experimental results for characterization of these applications over a wide range of platforms and parameter

Cluster	Node component		
Warp	Processor	2 x Xeon 3.06Ghz	
	Memory	1GB (a few 2GB)	
	Network	1Gbps Ethernet	
	Disk	5400RPM, 8MB cache	
Rohan	Processor	2 x Xeon 64-bit 3.20Ghz	
	Memory	6 GB	
	Network	1Gbps Ethernet	
	Disk	10000RPM, 8MB cache	
Emulab	Node type	Low-end	High-end
	Processor	P3 600Mhz	Xeon 64-bit 3Ghz
	Memory	256MB	2GB
	Network	5 x 100Mbps	6 x 1Gbps
	Disk	7200RPM	10000RPM

Table 2. Summary of hardware platforms

settings. Our experiments show promising results for two representative benchmarks [3] (RUBiS and RUBBoS) and potentially rapid inclusion of new benchmarks such as TPC-App [18] when a mature implementation is released. Our work shows that Mulini is a very useful tool for creating and managing experiments at this scale of complexity. The feasibility of using other experiment management tools such as LoadRunner and an evaluation of their effectiveness is beyond the scope of this paper.

The rest of this paper is organized as follows. Section II outlines the Mulini code generator that made the experiments feasible and affordable. Section III describes the experimental setup and discusses the management scale of performance experiments. Section IV summarizes the baseline experiments, where we show the benchmark results with known configurations. Section V describes the scalability of these benchmarks with an increasing number of servers for scale-out experiments. Section VI summarizes related work and Section VII concludes the paper.

II GENERATING CODE FOR DEPLOYMENT AND MONITORING

To create the n-tier application deployment code, the input to the Mulini generator [8][10][12] is a CIM/MOF (Common Information Model, Managed Object Format) standard specification format to model and describe resource configurations [14][16]. Mulini translates CIM/MOF into one of several deployment languages, including SmartFrog [11] and typical shell-style scripting languages such as bash shell script for Unix systems. In addition to application deployment code, Mulini also generates the workload parameter settings and the specifications for application performance and system resource consumption monitoring.

Using a domain-specific Testbed Language (TBL) as an input specification, Mulini generates a workload driver (e.g., client web browser emulator), and then parameterizes it with various settings (e.g., the number of concurrent users accessing the application) that stress the application. Typically, the experiments start with a light load that is increased heuristically for scale up and out experiments. When a bottleneck is found (e.g., by the observation of response times longer than the specified by service level objectives – SLOs), we use Mulini to generate new experiments with larger configurations (e.g., increasing the number of bottleneck servers to balance the load). The best heuristics for experimental design is a topic of ongoing research and beyond the scope of this paper. We observe that in order to reconfigure and redeploy application for the next iteration, simply updating input TBL specification is enough, compared to manually changing each benchmark script and configurations of n-tier applications.

For application-level performance monitoring, Mulini parameterizes the workload driver to collect specified metrics, such as response time for each user request and overall throughput, in TBL. It also generates parameterized monitors as separate tools to gather system-level metrics including CPU, memory usages, network I/O, and disk I/O. Mulini accounts for variations across hosts by creating system-level monitoring tools customized to each host. This automation alleviates the clutter of managing data files for each host and avoids any errors from manually launching monitors with different parameters. After each set of experiments, performance data collected from the participating hosts is put into a database for analysis.

III DESCRIPTION OF EXPERIMENTAL CONFIGURATIONS

Our experiments consist of application benchmarks that run on a combination of software and hardware platforms. Many components (e.g., application benchmark, software platform, and hardware platform) can be configured separately. In this section, we outline the main configuration choices explored in this paper.

III.A Experimental Platforms

Table 2 summarizes the three hardware platforms used for our experiments. The first cluster (named Warp) is comprised of Intel Blade Servers with 56 nodes. The second cluster (named Rohan) consists of homogeneous Intel Blade Servers (with a faster CPU) with 53 nodes. The third platform is the Emulab/Netlab distributed testbed [26].

Benchmark	Tier	Components
RUBiS	Database	MySQL Max 5.0.27
	Application	Apache Tomcat 5.5.17
	Web	Apache 2.0.54 JOnAS 3.3.6 Weblogic 8.1
RUBBoS	Database	MySQL Max 5.0.27
	Application	Apache Tomcat 5.5.17
	Web	Apache 2.0.54

Table 1. Summary of software configurations

Experiment set	Figure	Line count of configuration files changes and number of files	Line count of generated scripts	Machine count	Experimental configuration	Collected perf. data size
Baseline RUBiS on JOnAS	Figure 1	1.1 KLOC, 16	99 KLOC	300	50 (1-1-1 with varied write ratio and workload)	696MB
Baseline RUBiS on Weblogic	Figure 3	1 KLOC, 17	189 KLOC	594	99 (1-1-1 with varied write ratio and workload)	1,921MB
Scale-out RUBiS on JOnAS	Figure 5	9 KLOC, 27	815 KLOC	2,935	204 (1-2-1 to 1-12-3 with varied workload)	15,007 MB
Scale-out RUBiS on Weblogic	Figure omitted	0.9 KLOC, 20	152 KLOC	632	72 (1-2-1 to 1-5-1 with varied workload)	2,153 MB

Table 3. Scale of experiments run

Table 1 summarizes the software configurations of the servers used in our experiments. All Warp and Rohan experiments were run on Red Hat Enterprise Linux 4, with Linux kernel version 2.6.9-34-i386 on Warp and kernel 2.6.9-42-x86_64 on Rohan. Emulab experiments were run on Fedora Core 4 with Linux kernel version 2.6.12-1-i386. We used sysstat 7.0.2 to track system resource utilization.

III.B Application Benchmarks

The first application benchmark used in our experiments is RUBiS (Rice University Bidding System), an auction site prototype modeled after eBay, which can be used to evaluate application design pattern and application server's performance scalability. RUBiS stresses the application server and defines 26 interaction types such as browsing by categories or regions, bidding, buying, or selling items, registering users, writing or reading comments. It provides two default transition matrices emulating different workloads: read-only browsing interactions and bidding interactions that cause 15% writes to the database. In our experiments, the write ratio is extended to vary between 0% and 90%.

The second application benchmark used in our experiments is RUBBoS (Rice University Bulletin Board System), modeled after a bulletin board news site similar to Slashdot. RUBBoS is a 2-tier application which places a high load on the database tier. It uses 24 different interaction states in which a user may perform actions such as register, view story, and post comment. The client driver defines two transition matrices as well: read-only user interactions and submission user interactions (with a tunable write ratio). We use the smaller of the two standard RUBBoS data sets.

These benchmarks and the software they depended on had a wide range of complex configurable settings. It is this flexibility, which makes predicting system bottlenecks a non-trivial task and the observations we make more essential. To keep our results as reliable, fair and reproducible as possible, we kept all the configuration settings as close to default as possible. Deviations from a typical hardware or software configuration are mentioned in the following sections.

In the experiments, each trial consists of a warm-up period, a run period, and a cool-down period. The warm-up period brings system resource utilization to a stable state. Then measurements are taken during the run period. This is followed by the cool-down period when measurement halts and no additional requests are made by the clients. For RUBiS, the trials consist of one-minute warm-up and cool-down periods, and a five minutes run period. For RUBBoS, the trials consist of two-and-a-half minutes warm-up and cool-down periods, and a 15 minutes run period.

III.C Scale of Experiments

Table 3 shows the management scale of the experiments run to obtain the data described in this paper. As we can see, the number of script lines required for relatively "simple" figures such as Figure 1, Figure 3, and Figure 5 reach hundreds of thousands of lines. It is impractical to write and maintain scripts manually at such scale. The number of experiments included in this paper has been limited by the number of nodes available in our clusters and Emulab. The experiments continue, since the generation of scripts is largely automated in our environment. In Table 3 we also include the amount of performance data collected, which is typically on the order of gigabytes for each set of experiments run. The format of our collected performance data is the output of the monitoring tools from the sysstat suite.

Both Table 4 and Table 5 list a subset of the files that are generated and modified by Mulini to run RUBiS experiment when two machines are allocated for the application server tier and another 2 machines for the database tier. The experimental configurations shown in Table 3 are indicated by a triple (*w-a-d* in column 5, where *w* is the number of web servers, *a* is the number of application servers, and *d* is the number of database servers). We can see that the number of machines involved (from hundreds to thousands) and number of lines of scripts for the experiments (on the order of hundreds of thousands of lines) show the management complexity of these experiments. Typical experiments such as those reported here require many such graphs, with ten to twenty-fold increases in experiment management complexity.

As an indication of module-level complexity of deployment code, 6 or more files in the RUBiS script are modified by Mulini for each experiment. As an illustration, 3 of the files are listed in Table 5. Without an automated code generator such as Mulini, these files need to be modified and maintained by hand. Each of these files is a vendor-specific configuration file associated with a software package such as Apache, Tomcat, and MySQL. Consequently, these files are scattered over different directory locations. In a manual approach, it will be tedious and error prone to visit all these different directory locations to make those changes. In contrast, we modify Mulini's input specification once and the necessary modifications are propagated automatically.

Our experience shows that the automated approach based on Mulini shows much promise in the execution of large-scale experiments. As experiments become increasingly complex with more test variables and variety of configurations, manual maintenance of such scripts become less feasible. Some of these complexities are due to the low abstract level of scripting languages, which may be partially alleviated with a higher level deployment language such as SmartFrog [10]. More concrete experience with increasingly complex experiments will be needed to settle this question. For instance, database table partitioned among multiple database back-end nodes may require different steps to install and configure each database node. One of the advantages of the Mulini code generation approach is its ability to generate deployment code for a variety of platforms, including the higher abstraction deployment languages such as SmartFrog. A detailed evaluation of other tools for generating and management experiments and a detailed evaluation of the experiments themselves are interesting topics of future research.

IV BASELINE EXPERIMENTS & RESULTS

As a starting point to validate our scripts and setup, we ran a set of experiments for both RUBiS and RUBBoS by allocating one machine for each one of: database server, application server, and web server. This section summarizes the results of these baseline experiments (configuration 1-1-1, meaning 1 web server, 1 application server, and 1 database server). Scale out experiments (allocating more than one machine for the bottleneck server) are described in Section V.

Configuration file	Line count	Comment
Workers2.properties	22	Configures Apache to connect to application server tier
Mysqldb-raidb1-elba.xml	16	Configures C-JDBC controller to connect to databases
monitorLocal.properties	6	Configures JimysProbe monitor

Table 5. Examples of configuration files modified

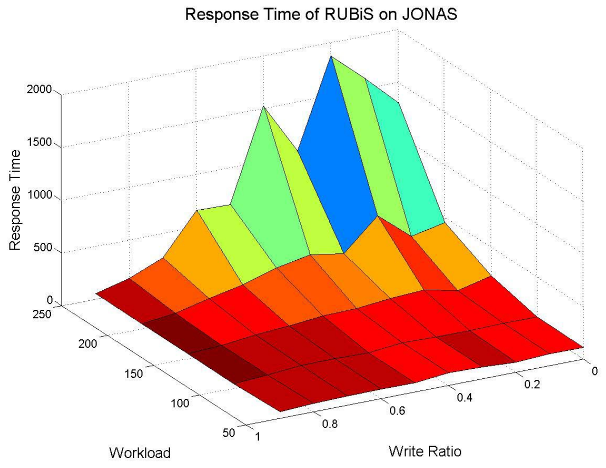

Figure 1. RUBiS on JOnAS response time

Generated script	Line count	Comment
run.sh	898	Calls all the other subscripts to install, configure and execute a RUBiS experiment
TOMCAT1_install.sh	54	Installs Tomcat server #1
TOMCAT1_configure.sh	48	Configures Tomcat server #1
TOMCAT1_ignition.sh	16	Starts Tomcat server #1
TOMCAT1_stop.sh	12	Stops Tomcat server #1
SYS_MON_EJB1_install.sh	11	Installs system monitoring tools on JOnAS server #1
SYS_MON_EJB1_ignition.sh	17	Starts system monitoring tools on JOnAS server #1

Table 4. Examples of generated scripts

IV.A RUBiS Experiments with JOnAS

Although we have conducted the RUBiS experiments on several platforms, we only report in this section the 1-1-1 experiments conducted on the Emulab cluster. Due to the different resources requirements by each server, we allocated different hardware configurations for these servers: database server host's CPU frequency is 600MHz, and the web server and

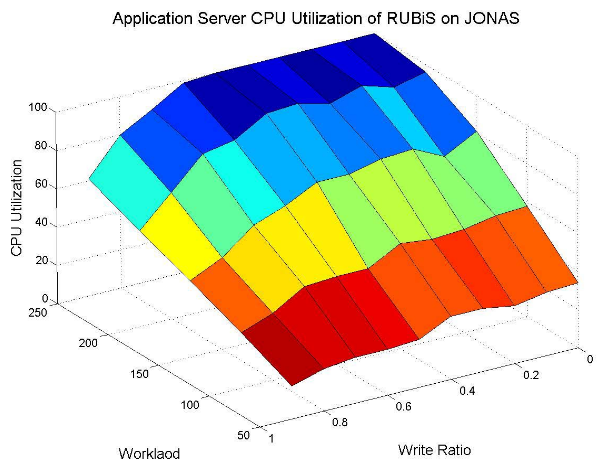

Figure 2. RUBiS on JOnAS application server CPU utilization

application server CPU frequency is 3GHz. The adoption of a slower database server allows for a quicker saturation of the database server, making the experimental results more interesting.

The default configuration of the RUBiS benchmark uses MySQL as its database server, JOnAS [19] as application server, and Apache as web server. Figure 1 shows the baseline measurement results with a graph of three dimensions: workload on X-axis, response time on Y-axis, and database write ratio of each interaction on Z-axis. The write ratio does not introduce a database bottleneck in the workload, but it reflects the (inverse of) amount of work done by the interactions and the load on the application server, which is the bottleneck. When the write ratio is high, most operations involve writes to the database which does not stress the application tier much, so the response time is relatively short. On the other hand, when the write ratio is low, most of the operations involve the transformation

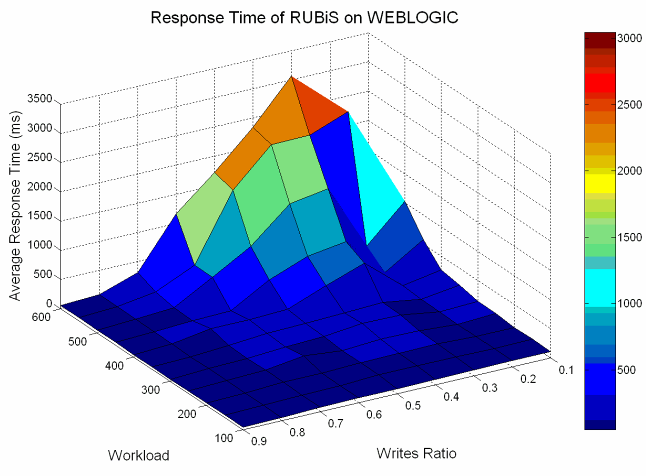

Figure 3. RUBiS on Weblogic response time

of data in the application tier, which make the response time long. This kind of interactions among different servers is typical of n-tier applications. We maintain the term "write ratio" for compatibility with the RUBBoS reference to workload.

The RUBiS (JOnAS, 1-1-1) experiments were conducted on the Emulab cluster with each component of the 3-tier application deployed on a separate node. During our experiments, we collected the system's response time to each emulated client's request. Each experiment's workload varies from 50 to 250 users, increment of 50. Figure 1 shows the increase in response time when the write ratio is low (more work for application server) and number of users increases. A bottleneck is apparent for the region of more than 250 users and write ratio below 30%. Figure 2 shows the application server CPU consumption corroborates the hypothesis of application server being the bottleneck. The two figure show correlated peaks in response time and application server CPU consumption. These baseline graphs also show an expected monotonic growth of response time when a bottleneck is encountered (the application server). At the top of the graph, the high response times reflect the CPU bottleneck, when the measured results show the uncertainties that arise at saturation. Concretely, the response times of 250 users between write ratio of 0% to 40% contain significant random fluctuations due to CPU saturation.

IV.B RUBiS Experiments with Weblogic

As a variation, we ran the RUBiS benchmark with the Weblogic application server replacing the default JOnAS application server. These experiments have similar settings as in Section IV.A and results are shown in Figure 3.

The RUBiS (Weblogic, 1-1-1) experiments were conducted on the Warp cluster with each component of the 3-tier application deployed on a separate node. Each experiment's workload varies from 100 to 600 users, increment of 50. During our experiments, we collected the system's response time to each

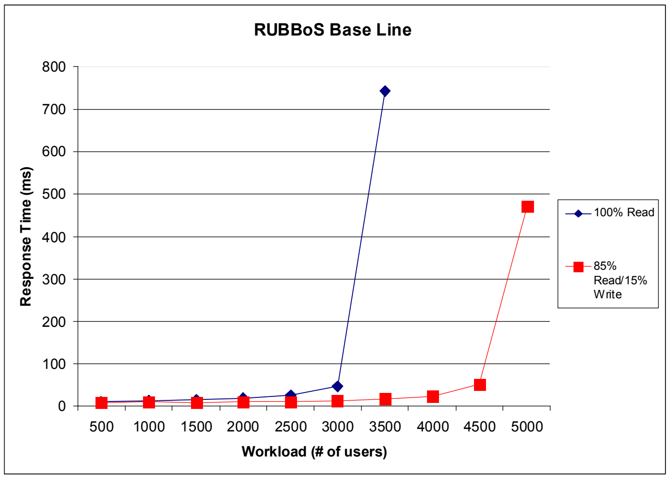

Figure 4. RUBBoS response time

emulated client's request. Figure 3 shows the increase in response time when the write ratio is low (more work for application server) and number of users increases. The system bottleneck is similar to that of Figure 1 for the region of more than 400 users and write ratio below 50%. Weblogic application server also has the same bottleneck (CPU consumption). At the top of the graph, the high response times reflect the CPU bottleneck, when the measured results also show the uncertainties that arise at saturation. Concretely, the response times of 600 users between write ratio of 0% to 50% also contain random fluctuations, but the Weblogic configuration is shown to support a higher number of users than JOnAS (about twice as many users at saturation point).

IV.C RUBBoS Experiments

Similar to RUBiS, the RUBBoS benchmark was run as an application on the Emulab cluster, using the Mulini code generation tools to create the scripts. Figure 4 shows the experimental results for the baseline experiments (1-1-1 configuration). Figure 4 has the workload on X-axis (500 to 5000 concurrent users, increments of 500) and response time on Y-axis (in milliseconds). Each of the 2-tier application components is deployed on a separate node.

Figure 4 compares two read/write ratios provided by the RUBBoS benchmark: 85%(read)/15%(write) shown in the lighter color, and 100% read shown in the darker shade. One can see from the graph that the read-only setting reaches a bottleneck at a much lower workload than the read/write mix. These results are compatible with previous experiments run on the Rohan cluster, which indicate the database server to be the bottleneck.

The baseline experiments described in this section verified the main setup parameters and previous published results on the RUBiS and RUBBoS benchmarks. Concretely, the RUBiS benchmark has its bottleneck in the application server and the database server as the bottleneck of RUBBoS. In the next section, we describe the main new results of this paper, the scale-out experiments that systematically add more resources to alleviate these bottlenecks.

V SCALE-OUT EXPERIMENTS

V.A Design of Scale-Out Experiments

Starting from the baseline experiments in the previous section, we ran scale-out experiments by following a simple strategy. As the workload (number of users) increases for a given configuration (1-1-1 in the baseline experiments), the system response time also increases. If we are able to see a system component bottleneck (e.g., application server in RUBiS), we increase the number of the bottleneck resource to alleviate the bottleneck. This is a simple strategy chosen for the experiments described here. In order to reconfigure and redeploy

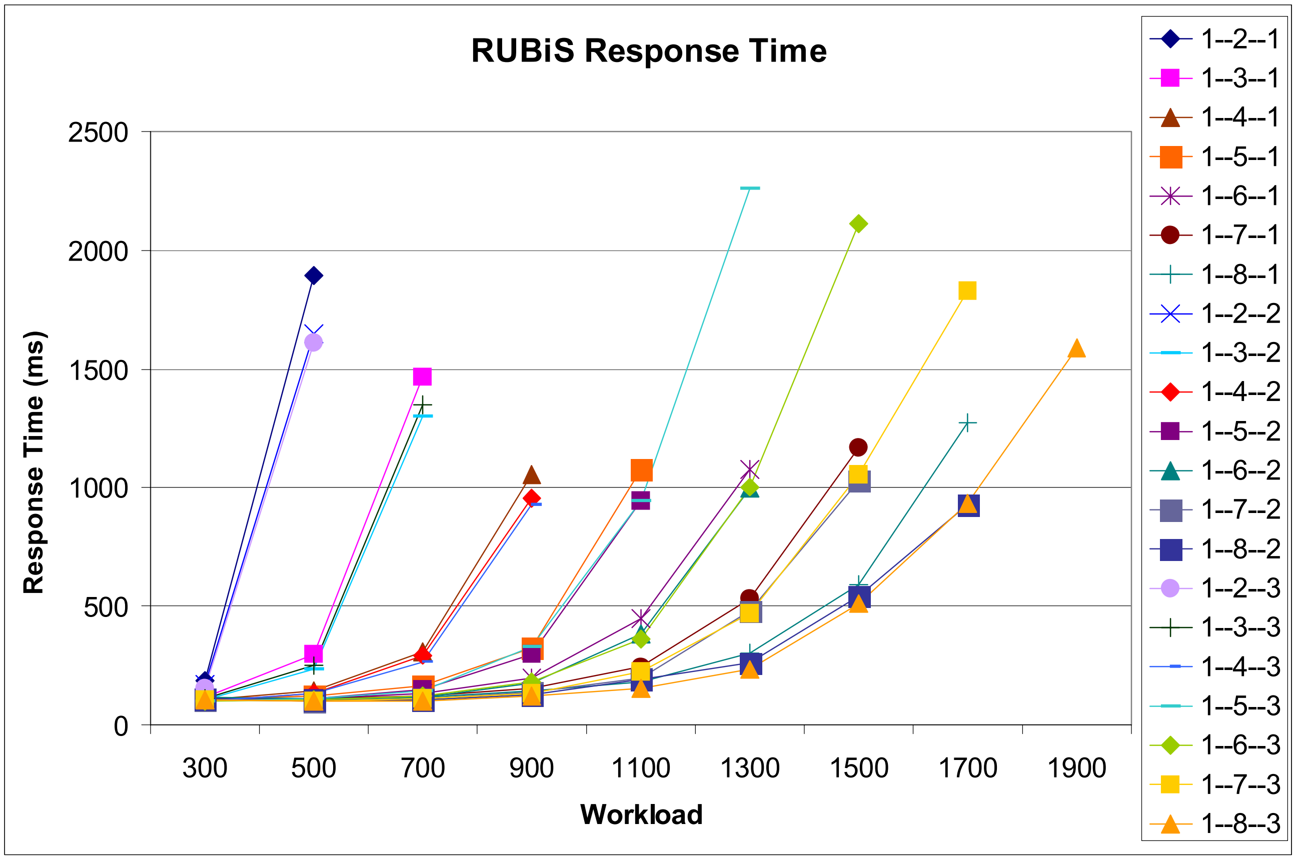

Figure 5. RUBiS on JOnAS scale-out response time for 2 to 8 application servers

application for the next iteration, simply updating input TBL specification is enough. The best heuristics for experimental design is a topic of ongoing research and beyond the scope of this paper.

This scale-out of the initial bottleneck in RUBiS continues by the gradual increase of workload, leading to system saturation. The experiments continue with the addition of another application server to alleviate the bottleneck. This loop continues until the system response time is not improved by the addition of another application server. This is an indication of a different bottleneck in the system. Then we add other system resources, in this case more database servers to identify the system bottleneck and to improve system performance.

In addition to this simple strategy, we also have run experiments that increase the number of servers that may be bottlenecks, to improve our understanding of the interactions among the components. The different strategies to explore the system configuration space is a challenging research topic in itself and beyond the scope of this paper. In the following sections, we show the concrete experiments and the insights they provide in the performance evaluation of different system configurations.

V.B RUBiS Scale-Out Experiments with JOnAS

In the case of RUBiS, we know the application server was the bottleneck for the 1-1-1 baseline configuration. Consequently, we increase the number of application servers from 1 in the baseline experiments to 12 (from 1-1-1 to 1-12-1). Similarly, we also increase the number of database servers from 1 in the baseline experiments to 3 as bottleneck in database server is detected. In RUBiS, web server performs as the workload distributor and does very little work. In all experiments we have a single web server and it has not been shown to be a bottleneck. To simplify the experiments (see Section III.C) and their analysis, we focus on the application performance with different system configurations for a fixed write ratio of 15% which is a typical write ratio in auction websites. The experiments to evaluate the sensitivity of our results with respect to variations in write ratio is the subject of future research.

Figure 5 and Figure 6 shows all the RUBiS (JOnAS) experiments on Emulab, with response times measured for all combinations of 1 web server, 1 to 12 application servers, and 1 to 3 database servers. Although the details of the lines can be seen better at higher resolution, some trends can be seen in the graph.

First, the leftmost overlapping lines show that 1-2-1, 1-2-2 and 1-2-3 corroborate the observation that the database server is not the bottleneck. Adding more database servers makes very little difference in the system response time. In contrast, adding an application server (e.g., from 1-2-1 to 1-3-1) improves the system response time significantly. While the saturation of the 1-2-1 configuration occurs at about 500 users, the 1-3-1 configuration saturates at about 750 users. This trend is repeated for the 1-3-X, 1-4-X, 1-5-X, 1-6-X, and 1-7-X (where X varies from 1 to 3) configurations. For these configurations, adding database server helps very little, but adding an application server improves the system performance by supporting (roughly) 250 additional users.

Table 7 shows the average throughput (successful client requests per second) for the configurations of 1-2-1 to 1-8-1. It shows some interesting trends more clearly than the figures. First, the throughput at low workloads is the same across the multiple servers (on the same row), validating the software scale out capability. This happens for loads of 300 through 700. The 2 application server configuration fails to complete the experiment for loads higher than 700. More generally, the missing squares in Table 7 reflect the experiments that could not complete. We also note that the last number of each column show an anomalously low number. We speculate this is caused by the same technical issues that make the experiments fail at high loads.

Performance improvement by adding servers	+1	+2	+3	+4	+5	+6
App server (%)	84.3	92.4	93.6	94.1	94.1	94.1
DB server (%)	13	14.9				

Table 6. Comparison of performance improvement (percentage of response time decrease) from 1-2-1 to 1-8-3 with 500 users in the RUBiS (JOnAS) experiments

	Configuration (w-a-d)						
Load	1-2-1	1-3-1	1-4-1	1-5-1	1-6-1	1-7-1	1-8-1
300	42.0	42.3	42.5	42.2	42.3	42.3	42.3
500	56.4	69.1	70.5	70.6	70.8	70.8	70.9
700	57.4	83.7	96.4	98.1	98.3	98.6	98.5
900	12.7	62.8	111	123	125	126	126.0
1,100		39.3	97.3	137	148	152	149.8
1,300				132	162	173	178.0
1,500				69.7	166	185	199.2
1,700					138	191	206.5
1,900						196	215.8
2,100						77.1	215.4
2,300							156.6
2,500							90.4

Table 7. RUBiS measured average throughput for configu-

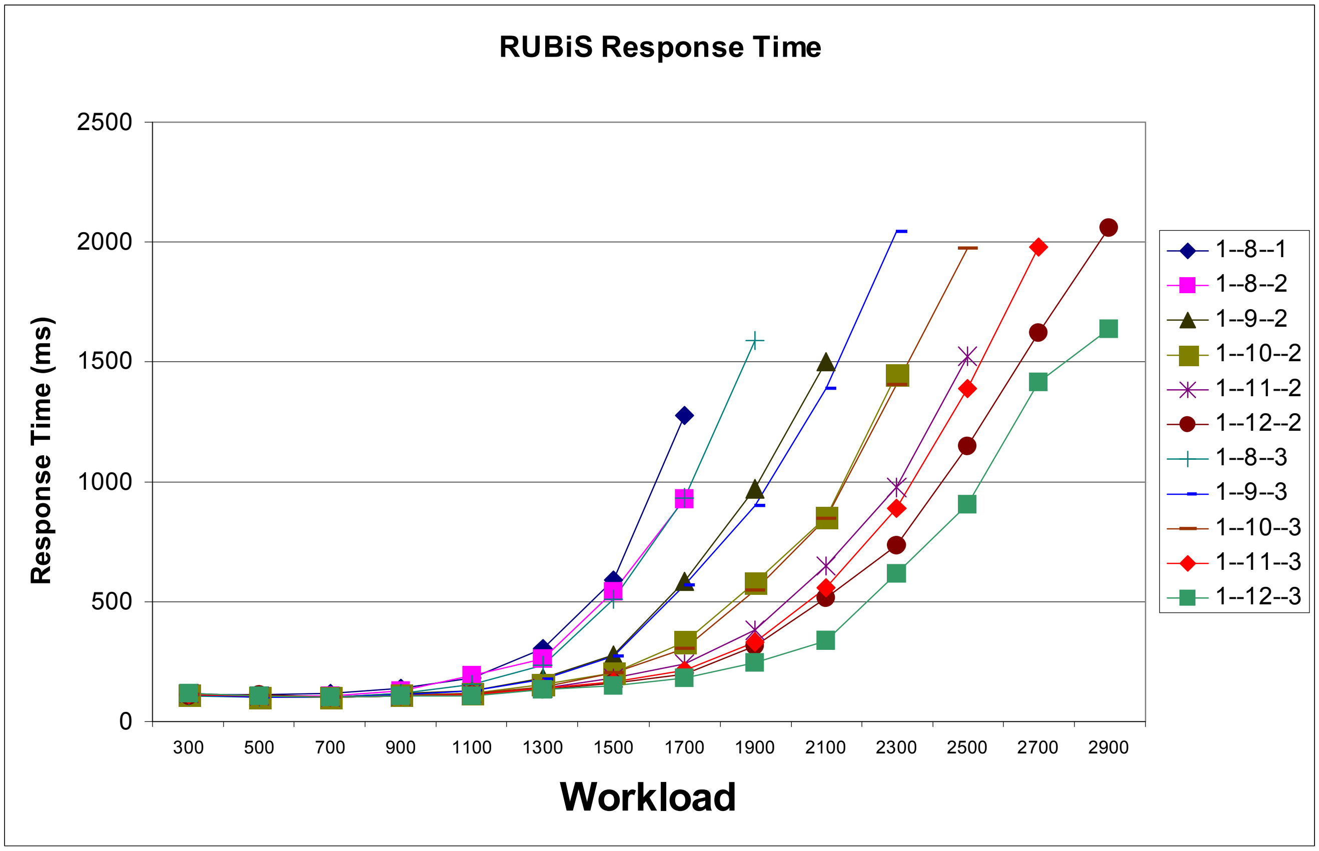

Figure 6. RUBiS on JOnAS scale-out response time for 8 to 12 application servers

From the system management point of view, this trend is shown in Table 6, where the response time improvements are compared for the 1-2-1 configuration as base, with 500 users (near the saturation point). Table 6 shows the percentage of response time improvement gained by adding more servers into the system. Adding one application server to yield the 1-3-1 configuration can get 84.3% improvement, compared with 13% improvement by adding one database server to yield the 1-2-2 configuration. Consider the case of capacity planning for 500 concurrent users. Table 6 shows that 3 or 4 application servers would match well with 1 database server in terms of minimizing response time and avoiding over-provisioning.

The consistent trend between 1 and 7 application servers is changed when we reach the 1-8-1 and 1-8-2 configurations. When the number of application servers reaches 8 and the number of users 1700, the response time difference between 1-8-1 (around 1.3sec) and 1-8-2 (about 0.9sec) is about 40%. This difference is much bigger than the numbers in Table 6 (third row). However, the response time of 1-8-2 and 1-8-3 configurations are quite similar. This observation is consistent with the hypothesis that the single database server has become the bottleneck at 1-8-1 configuration and 1700 users. Once a second database server has been added (1-8-2), the bottleneck shifts away and the database server(s) is no longer the bottleneck.

Figure 6 shows a trend similar to Figure 5, after the database bottleneck detected at 1700 users. The curves of 1-8-Y, 1-9-Y, and 1-10-Y overlap (where Y varies from 2 to 3).

To study the variation in response time in more detail, Figure 7 shows the response time difference between different numbers of database servers, where X-axis contains the number of users and Y-axis shows the difference of response time in milliseconds. The first curve shows the difference between the 1-8-1 and 1-8-2 configurations (from 1 to 2 database servers),

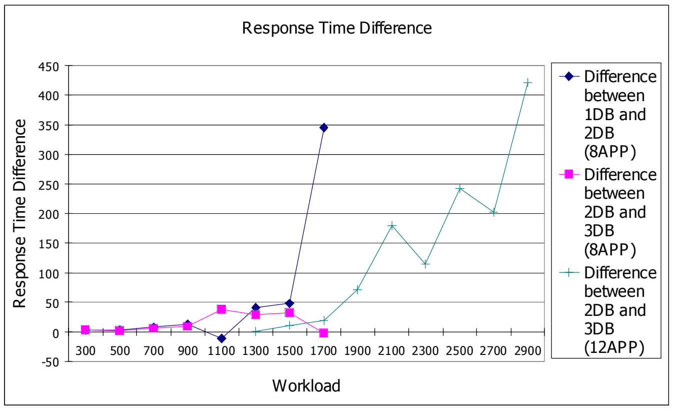

Figure 7. RUBiS on JOnAS scale-out response time difference between configurations

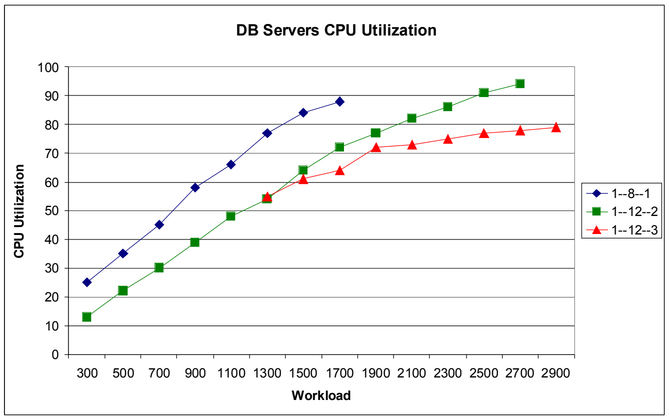

Figure 8. RUBiS on JOnAS scale-out CPU utilization between configurations

which is flat on the left side, but has a sudden jump at 1700 users. The second curve shows there is little difference between 1-8-2 and 1-8-3 configurations (a third additional database server) up to 1700 users.

The third curve in Figure 7 shows the difference in response time between the 1-12-2 and 1-12-3 configurations, between 1700 users and 2900 users. At this point, we observe the jump in response time difference between 2 database servers and 3 database servers. This is consistent with the hypothesis that the 2 database servers have become a bottleneck at 2900 users. From the capacity planning point of view, 2 database servers appear to suffice for a workload smaller than 2900 users.

We also plotted the CPU utilization of database servers in Figure 8, showing only 3 “critical” cases (1-8-1, 1-12-2, and 1-12-3). The first two cases show the gradual saturation of the database servers’ CPU utilization at 1700 users (1 server) and 2700 users (2 servers), which corroborates our previous analysis. The third curve shows the non-saturation of 1-12-3 configuration, at least with respect to the database servers.

V.C Discussion

The experiments in the previous section (V.B) show that we can perform the measurements of large scale benchmarks by scaling the number of bottleneck servers. This way, we can verify the actual properties of each benchmark on representative system configurations. These experimental results can be used to confirm or disprove analytical models within the system parameter ranges covered by the experiments. The design of an analytical model that fits our measurements is beyond the scope of this paper.

Another practical use of the experimental results in Section V.B is in the design of system configurations for n-tier applications similar to RUBiS. Given a concrete set of service level objectives and workload levels, one can use the numbers in **Figure 5** through **Figure 8** to choose the appropriate system resource level that will achieve the specified quality of service for the workload levels of interest.

VI RELATED WORK

Traditional performance analysis [1][5] [6] based on analytical methods such as queuing theory have limitations when handling the variations and evolution of n-tier applications, often due to assumptions made in the underlying model. Assumptions (such as mean inter-arrival time), can vary widely in real-world situations due to troughs and peaks in workloads. For n-tier applications, especially in the case of multiple servers at any tier, the average errors in prediction in any one tier get magnified by the average errors in the next. Consequently, direct application of modeling techniques on n-tier applications has been limited.

A second area of related research consists of experimental studies of complex system performance, staging environments to validate specific configurations. IBM's performance optimization tool, an autonomic computing tool, identifies the root cause of poor performance observed during a test by round-trip response time decomposition [20]. The response time is broken down into the times spent within each component in the system enabled by the Tivoli ARM data collector. However, since ARM needs changes to the application code, this approach is intrusive in nature.

A third area of related work consists of the dynamically adaptive systems in the autonomic computing area. Our observation-based approach complements well the recent efforts in the autonomic computing area [21][22][23][24]. We can see autonomic approaches to adjust system configuration in response to workload variations [6] as a dynamic discovery process that searches for an optimized mapping of workload to system resources. The observation-based performance characterization work reduces the assumptions made and uncertainties inherent in unknown configurations in the search space, by observing the real system behavior beforehand.

From the infrastructure point of view, the Mulini generator builds on code generation concepts, techniques, and software tools developed in the Elba project [10][18][10]. Although the main application area of Elba tools has been in system management, specifically in the validation of staging deployment scripts, the Elba tools have been very useful in the generation and management of our experiments.

VII CONCLUSION

N-tier applications have grown in economic and social importance in recent years. While they offer unprecedented flexibility to grow and evolve, they also introduce significant management problems due to the changing nature of their workload and evolution of their functionality. However, traditional methods of performance evaluation, based on models such as queuing theory, have difficulties with systems that may or may not have a steady state. Unfortunately, experimental methods that measure directly system performance also have difficulties, this time with the cost and errors in manually written scripts for many configuration settings.

We have developed the Mulini code generator to support automated deployment and evaluation of n-tier applications in a distributed environment. By generating the deployment code/script automatically, we are able to lower the development costs and reduce errors at the same time. The first major contribution of this paper is the application of code generation tools to support large-scale experimental measurements of n-tier application benchmarks such as RUBiS and RUBBoS. Using our tools, we generated several hundred thousands of lines of scripting code and ran hundreds of experiments to observe the actual performance of the benchmark applications on a variety of software and hardware combinations.

Our experiments show both expected and unexpected results. Section IV describes the baseline experiments (one machine for each of database server, application server, and web server) for RUBiS (with two different application servers) and RUBBoS. Section V describes the scale-out experiments for RUBiS (for JOnAS and Weblogic) on the system bottleneck – the application server, and for RUBBoS also on the bottleneck – the database server. The RUBiS experiments show quantitatively how many application and database servers are required to provide good response time for a given number of users. For the configuration studied (Emulab), 1 web server and 1 database server can serve up to 1700 users, but 7 application servers are needed for the 1700-user workload. For higher workloads, 2 database servers and 12 application servers are able to serve up to 2700 users.

Our experiments show both the feasibility of the observation-based performance characterization approach. Further, the non-trivial interactions among the scaled out severs as well as interactions among components shows the need for more research (and perhaps experiments) to study and characterize these phenomena using theoretical or analytical models.

Acknowledgements. This research has been partially funded by National Science Foundation grants CISE/IIS-0242397, ENG/EEC-0335622, CISE/CNS-0646430, AFOSR grant FA9550-06-1-0201, IBM SUR grant, Hewlett-Packard, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment.

REFERENCES

[1] G. Bolch, S. Greiner, H. de Meer, K. S. Trivedi. Queueing networks and Markov chains: modeling and perform-ance evaluation with computer science applications, 2nd Edition, New York: Wiley, 2006, chapter 7 and 13.

[2] T. Brecht, D. Pariag, and L. Gammo, accept()able Strategies for Improving Web Server Performance, *USENIX* 2004, pp 227-240.

[3] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic Web content," *Middleware* 2003.

[4] G. Jung, G. S. Swint, J. Parekh, C. Pu, and A. Sahai. Detecting Bottleneck in n-Tier IT Applications through Analysis, *DSOM* 2006.

[5] L. Kleinrock. Queuing Systems, Vol. 2: Computer Applications, NewYork: Wiley, 1976, chapter 4 and 6.

[6] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. Capacity planning and performance modeling: from mainframes to client-server systems. Prentice-Hall, Inc., 1994, chapter 3.

[7] D. A. Menascé, M. N. Bennani, and H. Ruan. On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems. In the book Self-Star Properties in Complex Information Systems, LNCS, Vol. 3460, Springer Verlag, 2005.

[8] Jason Parekh, Gueyoung Jung, Galen Swint, Calton Pu, Akhil Sahai. Issues in Bottleneck Detection in Multi-Tier Enterprise Applications. IWQoS 2006.

[9] A. Sahai, S. Singhal, R. Joshi, V. Machiraju. Automated Policy-Based Resource Construction in Utility Computing Environments, NOMS 2004, pp 381-393.

[10] A. Sahai, C. Pu, G. Jung, Q. Wu, W. Yan, and G. S. Swint. Towards Automated Deployment of Built-to-Order Systems, In Proceeding of DSOM, 2005, pages 109-120.

[11] V. Shepelev, and S. Director. Automatic Workflow Generation, European Design Automation Conference with EURO-VHDL, 1996, pages 104-109.

[12] G. S. Swint, G. Jung, C. Pu, and A. Sahai. Automated Staging for Built-to-Order Application Systems, In Proceedings of NOMS, 2006, pages 361-372.

[13] V. Talwar, Q. Wu, C. Pu, W. Yan, G. Jung, D. Milojicic. Approaches to Service Deployment. In *IEEE Internet Computing,* 9(2): 70-80 (March 2005).

[14] Common Information Model (CIM) Standards, http://www.dmtf.org/standards/standard_cim.php.

[15] SNIA CIM Object Manager (CIMOM), http://www.opengroup.org/snia-cimom/.

[16] DMTF-CIM Policy, http://www.dmtf.org/standards/documents/CIM/CIM_Schema26/CIM_Policy26.pdf.

[17] SmartFrog, Smart Framework for object groups, http://www-uk.hpl.hp.com/smartfrog/.

[18] TPC BENCHMARK App. Specification version 1.1.1, Aug. 2005, http://www.tpc.org/tpc_app/spec/TPC-App_V1.1.1.pdf.

[19] JOnAS application server. http://jonas.objectweb.org/

[20] IBM Performance Center, Rational Performance Tester, http://www-306.ibm.com/software/awdtools/tester/performance/.

[21] IBM Autonomic Computing, http://www.ibm.com/autonomic.

[22] SUN N1 Software, http://www.sun.com/software/n1gridsystem/.

[23] Microsoft The Drive to Self-Managing Dynamic Systems, http://www.microsoft.com/systemcenter/default.mspx.

[24] Platform LSF, http://www.platform.com/products/LSF.

[25] Automated Design Evaluation and Tuning, http://www.cc.gatech.edu/systems/projects/Elba.

[26] Emulab distributed testbed. http://www.emulab.org.