A New Perspective on Experimental Analysis of N-tier Systems: Evaluating Database Scalability, Multi-bottlenecks, and Economical Operation

(Invited Paper)

Simon Malkowski*, Markus Hedwig†, Deepal Jayasinghe*, Junhee Park*,
Yasuhiko Kanemasa‡, and Calton Pu*
**CERCS, Georgia Institute of Technology, Atlanta, GA 30332-0765, USA*
{simon.malkowski, deepal, jhpark, calton}@cc.gatech.edu
†Chair of Information Systems Research, University of Freiburg, 79098 Freiburg, Germany
markus.hedwig@is.uni-freiburg.de
‡Fujitsu Laboratories Ltd., Kawasaki 211-8588, Japan - kanemasa@jp.fujitsu.com

Abstract

Economical configuration planning, component performance evaluation, and analysis of bottleneck phenomena in N-tier applications are serious challenges due to design requirements such as non-stationary workloads, complex non-modular relationships, and global consistency management when replicating database servers, for instance. We have conducted an extensive experimental evaluation of N-tier applications, which adopts a purely empirical approach the aforementioned challenges, using the RUBBoS benchmark. As part of the analysis of our exceptionally rich dataset, we have experimentally investigated database server scalability, bottleneck phenomena identification, and iterative data refinement for configuration planning. The experiments detailed in this paper are comprised of a full scale-out mesh with up to nine database servers and three application servers. Additionally, the four-tier system was run in a variety of configurations, including two database management systems (MySQL and PostgreSQL), two hardware node types (normal and low-cost), two replication strategies (wait-all and wait-first—which approximates primary/secondary), and two database replication techniques (C-JDBC and MySQL Cluster). Herein, we present an analysis survey of results mainly generated with a read/write mix pattern in the client emulator.

1. Introduction

Scaling N-tier applications in general and multi-tiered e-commerce systems in particular are notoriously challenging. Countless requirements, such as non-stationary workloads and non-modular dependencies, result in an inherently high degree of management complexity. We have adopted a purely empirical approach to this challenge through a large-scale experimental evaluation of scalability using an N-tier application benchmark (RUBBoS [1]). Our experiments cover scale-out scenarios with up to nine database servers and three application servers. The configurations were varied using two relational database management systems (MySQL and PostgreSQL), two database replication techniques (C-JDBC and MySQL Cluster), wait-all (aka write-all) and wait-first (primary/secondary approximation) replication strategies, browsing-only and read/write interaction mixes, workloads ranging from 1,000 to 13,000 concurrent users, and two different hardware node types.

The analysis of our extensive dataset produced several interesting findings. First, we documented detailed node-level bottleneck migration patterns among the database, application, and clustering middleware tiers when workload and number of servers increased gradually. These bottlenecks were correlated with the overall system performance and used to explain the observed characteristics. Second, the identification of multi-bottlenecks [2] showed the performance effects of non-obvious phenomena that can arise in real systems. Third, the initial economic evaluation of our data based on an iterative refinement process revealed their usefulness in deriving economical configuration plans.

The main contribution of this paper is twofold. From a technical perspective, we collected and evaluated a significant amount of data on scaling performance in N-tier applications. We present an evaluation survey and illustrate various interesting findings that are

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8311
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8311

uniquely made possible through our empirical approach. From a high-level perspective, we demonstrate the potential of automated experiment creation and management systems for empirical analysis of large distributed applications. Our results clearly show that important domain insights can be obtained through our approach. In contrast to traditional analytical approaches, our methodology is not founded on rigid assumptions, which significantly reduces the risk of overlooking unexpected performance phenomena.

The remainder of this paper is structured as follows. In Section 2 we establish some background on infrastructure and bottleneck definition. Section 3 outlines experimental setup and methods. Section 4 presents the analysis of our scale-out data. In Section 5 we introduce the use-case of economical configuration planning. Related work is summarized in Section 6, and Section 7 concludes the paper.

2. Background

This section provides previously published background that is of particular importance for the understanding of our approach. Due to the space constraints of this paper each subsection is constrained to a brief overview. Interested readers should refer to the cited references for more comprehensive descriptions.

2.1. Experimental Infrastructure

The empirical dataset that is used in this work is part of an ongoing effort for generation and analysis of N-tier performance data. We have already run a very high number of experiments over a wide range of configurations and workloads in various environments. A typical experimentation cycle (i.e., code generation, system deployment, benchmarking, data analysis, and reconfiguration) requires thousands of lines of code that need to be managed for each experiment. The experimental data output are system metric data points (i.e., network, disk, and CPU utilization) in addition to higher-level metrics (e.g., response times and throughput). The management and execution scripts contain a high degree of similarity, but the differences among them are critical due to the dependencies among the varying parameters. Maintaining these scripts is a notoriously expensive and error-prone process when done by hand.

In order to enable experimentation at this large scale, we used an experimental infrastructure created for the Elba project to automate N-tier system configuration management. The Elba approach [3] divides each automated staging iteration into steps such as converting

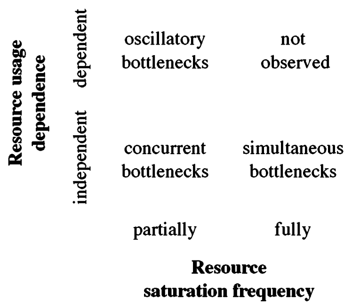


Figure 1. Simple multi-bottleneck classification [2].

policies into resource assignments [4], automated code generation [5], benchmark execution, and analysis of results.

2.2. Single-bottlenecks vs. Multi-bottlenecks

The abstract definition of a *system bottleneck* (or bottleneck for short) can be derived from its literal meaning as the key limiting factor for achieving higher system throughput. Due to this intuition, similar formulations have often served as foundation for bottleneck analysis in computer systems. But despite their popularity, these formulations are based on assumptions that do not necessarily hold in practice. Because of queuing theory, the term bottleneck is often used synonymously for single-bottleneck. In a single-bottleneck cases, the saturated resource typically exhibits a near-linearly load-dependent average resource utilization that saturates at one hundred percent past a certain workload. However, the characteristics of bottlenecks may change significantly if there is more than one bottleneck resource in the system. This is the case for many real N-tier applications with heterogeneous workloads. Therefore, we explicitly distinguish between single-bottlenecks and the umbrella-term *multi-bottlenecks* [2], [6].

Because system resources may be causally dependent in their usage patterns, multi-bottlenecks introduce the classification dimension of *resource usage dependence*. Additionally, greater care has to be taken in classifying bottleneck resources according to their *resource saturation frequency*. Resources may saturate for the entire observation period (i.e., fully saturated) or for certain parts of the period (i.e., partially saturated). Figure 1 summarizes the classification that forms the basis of the multi-bottleneck definition. It distinguishes between *simultaneous*, *concurrent* and *oscillatory* bottlenecks. In comparison to other bottlenecks, resolving oscillatory bottlenecks may be very challenging. Multiple resources form a combined bottleneck, and it can only be resolved in union. Therefore, the addition of resources does not necessarily improve performance in complex N-tier systems. In fact, determining regions of complex multi-bottlenecks through modeling may be an intractable problem. Consequently, multi-bottlenecks require measurement-based experimental approaches that do not oversimplify system performance in their assumptions.

3. Experimental Setting

3.1. Benchmark Applications

Among N-tier application benchmarks, RUBBoS has been used in numerous research efforts due to its real production system significance. Readers familiar with this benchmark can skip to Table 1(a), which outlines the concrete choices of software components used in our experiments.

RUBBoS [1] is an N-tier e-commerce system modeled on bulletin board news sites similar to Slashdot. The benchmark can be implemented as three-tier (web server, application server, and database server) or four-tier (addition of clustering middleware such as C-JDBC) system. The benchmark places high load on the database tier. The workload consists of 24 different interactions (involving all tiers) such as register user, view story, and post comments. The benchmark includes two kinds of workload modes: browse-only and read/write interaction mixes. In this paper we use both the browse-only and read/write workload for our experiments.

Typically, the performance of benchmark application systems depends on a number of configurable settings (including software and hardware). To facilitate the interpretation of experimental results, we chose configurations close to default values, if possible. Deviations from standard hardware or software settings are spelled out when used. Each experiment trial consists of three periods: ramp-up, run, and ramp-down. In our experiments, the trials consist of an 8-minute ramp-up, a 12-minute run period, and a 30-second ramp-down. Performance measurements (e.g., CPU or network bandwidth utilization) are taken during the run period using Linux accounting log utilities (i.e., Sysstat) with a granularity of one second.

3.2. Database Replication Techniques

In this subsection we briefly introduce the two different database replication techniques that we have used in our experiments. Please refer to the cited sources for a comprehensive introduction.

C-JDBC [7], is an open source database cluster middleware, which provides a Java application access to a cluster of databases transparently through JDBC. The database can be distributed and replicated among several nodes. C-JDBC balances the queries among these nodes. C-JDBC also handles node failures and provides support for check-pointing and hot recovery. The C-JDBC server implements two update propagation strategies for replicated databases: wait-all (write request is completed when all replicas respond) and wait-first (write request is considered completed upon the first replica response). The wait-all strategy is equivalent to the commonly called write-all replication strategy, and the wait-first strategy is similar to the primary-secondary replication strategy.

MySQL Cluster [8] is a real-time open source transactional database designed for fast, always-on access to data under high throughput conditions. MySQL Cluster utilizes a "shared nothing" architecture, which does not require any additional infrastructure and is designed to provide five-nines data availability with no single point of failure. In our experiment we used "in-memory" version of MySQL Cluster, but that can be configured to use disk-based data as well. MySQL Cluster uses the NDBCLUSTER storage engine to enable running several nodes with MySQL servers in parallel.

3.3. Hardware Setup

The experiments used in this paper were run in the Emulab testbed [9] with two types of servers. Table 1(b) contains a summary of the hardware used in our experiments. Normal and low-cost nodes were connected over 1,000 Mbps and 100 Mbps links, respectively. The experiments were carried out by allocating each server to a dedicated physical node. In the initial setting all components were "normal" hardware nodes. As an alternative, database servers were also hosted on low-cost machines. Such hardware typically entails a compromise between cost advantage and performance loss, which may be hard to resolve without actual empirical data.

We use a four-digit notation #W/#A/#C/#D to denote the number of web servers, application servers, clustering middleware servers, and database servers. The database management system type is either "M" or "P" for MySQL or PostgreSQL, respectively. If the server node type is low-cost, the configuration is marked with an additional "L". The notation is slightly different for MySQL Cluster. If MySQL Cluster is used, the third number (i.e., "C") denotes the number of MySQL

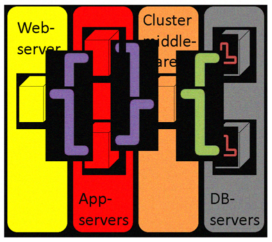
(a) Software setup.

Function		Software
Web server	🟨	Apache 2.0.54
Application server	🟥	Apache Tomcat 5.5.17
Cluster middleware	🟧	C-JDBC 2.0.2
Database server	■	MySQL 5.0.51a PostgreSQL 8.3.1 MySQL Cluster 6.2.15
Operating system		Redhat FC4 Kernel 2.6.12
System monitor		Systat 7.0.2

(b) Hardware node setup.

Type	Components		
Normal	Processor	Xeon 3GHz	64-bit
	Memory	2GB	
	Network	6 x 1Gbps	🟪
	Disk	2 x 146GB	10,000rpm
Low-cost	Processor	PIII 600Mhz	32-bit
	Memory	256MB	
	Network	5 x 100Mbps	🟩
	Disk	13GB	7,200rpm

(c) Sample C-JDBC topology (1/2/1/2L).



(d) Detailed configuration settings.

Apache		**Tomcat**		**MySQL**		**MySQL Cluster**	
KeepAlive	Off	maxThreads	330 (Total of all)	storage engine	MyISAM	storage engine	NDBCLUSTER
StartServers	1	minSpareThreads	5	max_connections	500	max_connections	500
MaxClients	300	maxSpareThreads	50			indexMemory	200MB
MinSpareThreads	5	maxHeapSize	1,300MB	**PostgreSQL**		dataMemory	1,750MB
MaxSpareThreads	50			max_connections	150	numOfReplica	2/4
ThreadsPerChild	150	**C-JDBC**		shared_buffers	24MB		
		minPoolSize	25	max_fsm_pages	153,600		
		maxPoolSize	90	checkpoint_segments	16		
		initPoolSize	30				
		LoadBalancer	RAIDb-1-LPRF				
		RAIDb-1Scheduler					

Table 1. Details of the experimental setup on the Emulab cluster.

servers and the fourth number (i.e., "D") denotes the number of data nodes. A sample topology of a C-JDBC experiment with one web server, two application servers, one clustering middleware server, two low-cost database servers, and non-specified database management system (i.e., 1/2/1/2L) is shown in Table 1(c). Unlike traditional system tuning work, we did not attempt to tune a large number settings to find "the best a product can do", rather to sustain for high workloads we have made minimum changes as outlined in Table 1(d).

4. Scale-out Experiments

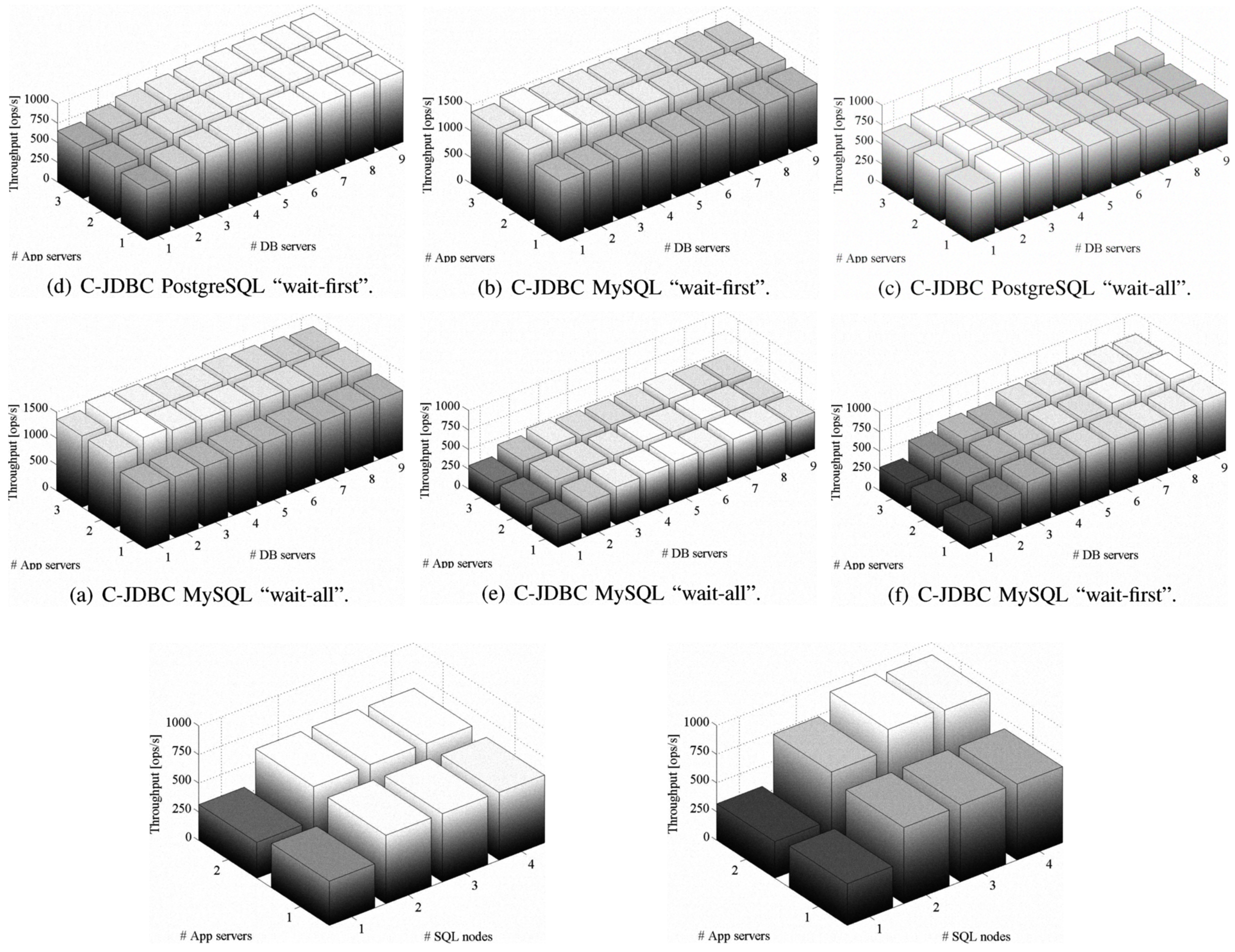
In this section we detail scale-out experiments in which we increase the number of database and application tier nodes gradually to find the bottlenecks at node level (for each configuration). In Subsection 4.1 we show experiments with MySQL running on normal servers. Subsection 4.2 describes experimental results for PostgreSQL on normal servers. These experiments show both commonalities and differences between the two database management systems. In Subsection 4.3 we discuss our experiment results running the same set of experiments with MySQL Cluster on normal nodes. Subsection 4.4 evaluates the change in system characteristics when using low-cost nodes for the deployment of database servers.

4.1. MySQL on normal Servers

For this section we collected data from RUBBoS scale-out experiments with MySQL database servers, C-JDBC, and normal hardware. The experiments start from 1/1/1/1M and go to 1/3/1/9M. For each configuration, we increase the workload (i.e., number of concurrent users) from 1,000 up to 13,000 in steps of 1,000. Due to the amount of collected data, we use two kinds of simplified graphs to highlight the bottlenecks and the shifting of bottlenecks in these experiments. Figure 2(a) shows only the highest achievable throughput (z-axis) for MySQL C-JDBC experiments with normal nodes (read/write workload). Figure 3(a) shows the corresponding bottleneck map. We can see three groups in both figures. The first group consists of configurations 1/2/1/1M and 1/3/1/1M, which have identical maximum throughput. The second group consists of all single application server configurations. The third group has the highest maximum throughput level and consists of all other six configurations with at least two application servers and two database servers.

Full data replication provides the best support for read-only workload, but it requires consistency management with the introduction of updates in a read/write mixed workload. In this subsection we use the C-JDBC clustering middleware to provide consistency management. Figures 2(a) and 2(b) show the

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8311
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8311



(d) C-JDBC PostgreSQL “wait-first”. (b) C-JDBC MySQL “wait-first”. (c) C-JDBC PostgreSQL “wait-all”.

(a) C-JDBC MySQL “wait-all”. (e) C-JDBC MySQL “wait-all”. (f) C-JDBC MySQL “wait-first”.

(g) MySQL Cluster with 2 data nodes. (h) MySQL Cluster with 4 data nodes.

Figure 2. Maximum system throughput for RUBBoS read/write workload with: (a)-(d) normal nodes; (e)-(f) low-cost nodes; and (g)-(h) normal nodes in the database tier.

maximum throughput achieved by the wait-all and wait-first strategies (see Subsection 3.2 for details), respectively. With respect to throughput, Figures 2(a) and 2(b) appear identical. This similarity is due to the delegation of transactional support in MySQL to the storage server, which happens to be MyISAM by default. MyISAM does not provide full transactional support; specifically, it does not use Write Ahead Logging (WAL) to write changes and commit record to disk before returning a response. Since MySQL processes update transactions in memory if the server has sufficient hardware resources, the result is a very small difference between the wait-all and wait-first strategies due the fast response. Both figures show that the three aforementioned groups. Consequently, the analysis summarized in Figure 3(a) applies independently of replication strategies. As indicated in the bottleneck map, all bottlenecks are CPU bottlenecks, and the clustering middleware (CM) eventually becomes the primary bottleneck. Consequently, increasing the number of database and application servers is effective only in cases with relatively small numbers of servers.

4.2. PostgreSQL on normal Servers

In this subsection we ran similar experiments as in Subsection 4.1 on a different database management system, the PostgreSQL. The comparison of the read/write workloads in Figures 2(a), 2(b), 2(c), and 2(d) reveals some differences between MySQL and PostgreSQL. While both MySQL configurations have similar throughput that peaks at around 1,500 operations per second, the PostgreSQL configurations only

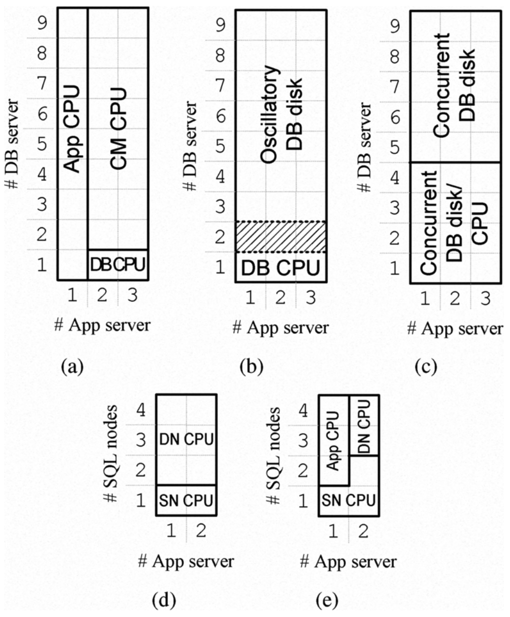


Figure 3. MySQL Bottleneckmaps: (a) C-JDBC, normal nodes; (b) C-JDBC, low-cost nodes, wait-all; (c) C-JDBC, low-cost nodes, wait-first; (d) MySQL Cluster, normal nodes, 2 data nodes; (e) MySQL Cluster, normal nodes, 4 data nodes.

achieve two thirds of this throughput. Additionally, the wait-all strategy (Figure 2(c)) becomes a bottleneck for PostgreSQL. However, it is not easy to identify the underlying system bottleneck. Figure 4 shows potential resources that could be saturated in the 1/2/1/2P configuration. All three CPU utilization densities show normal-shaped densities that do not reach the critical high percentile sector. From the throughput graphs, it is evident that the bottleneck cannot be resolved through application or database server replication.

As we increase the number of servers, hardware resources eventually becomes abundant. Through a detailed analysis, we found two explanations for the bottlenecks in the database and the clustering middleware tiers. First, PostgreSQL uses WAL. Its log manager waits for the completion of the commit record written to disk. Because the log manager does not use CPU cycles during the waiting time, the CPU becomes idle. At the same time, the C-JDBC server serializes all write requests by default to avoid occurrence of database deadlocks. Consequently, PostgreSQL receives serialized write queries, which limits the amount of parallel processing in PostgreSQL and idle CPU during I/O waiting times. This explanation has been tested by a separate experiment in which C-JDBC is removed, with a single PostgreSQL server, which showed saturated database CPU consumption.

4.3. MySQL Cluster on Normal Servers

In order to explicitly investigate the effects of different database replication technologies, we have conducted a similar setup of experiments as discussed in Subsection 4.1 with MySQL Cluster. First, we have run the experiments without partitioning the database, which implies that we can have only two data nodes (DN) and one management node, while increasing the number of MySQL servers (SN) from one to four and varying application servers between one or two. The summary of the results is contained in Figures 2(g) and 3(d). As illustrated in Figure 2(g), the maximum throughput that can be achieved is seemingly low compared to the same number of nodes with C-JDBC MySQL. The main reason is that that the MySQL data nodes become the bottleneck when increasing the workload (see Figure3(d)). Therefore, we have partitioned the database into two parts using the MySQL Cluster default partitioning mechanism and repeated the set of experiments with four data nodes. The performance summary is shown in Figures 2(h) and 3(e). The comparison of Figures 2(g) and 2(h) reveals that increasing the number of data nodes help us to increase the maximum throughput. However, as Figures 3(d) and 3(e) illustrate splitting the database causes the bottlenecks to shift from the data nodes to the application servers initially. When the workload increases even further, the bottlenecks shifted back to the data nodes. Consequently, our empirical analysis reveals that the system is not bottlenecked in the MySQL servers in the case of read/write workload.

4.4. MySQL on Low-cost Servers

In the following we analyze data from RUBBoS scale-out experiments with C-JDBC and MySQL on low-cost hardware. Because of the limited memory on low-cost machines, the throughput graphs in Figures 2(e) and 2(f) are dominated by disk and CPU utilization in the database tier. Initially, in all configurations with a single database server, the maximum throughput is limited by a shifting bottleneck between database disk and CPU. Figure 5 shows the corresponding database server CPU and disk utilization densities in the 1/2/1/1ML configuration. Note that no replication strategy is necessary in the single database server case. Both the CPU (Figure 5(a)) and the disk (Figure 5(b)) densities have two characteristic modes for higher workloads, which correspond to a constant shifting between resource saturation and underutilization. If

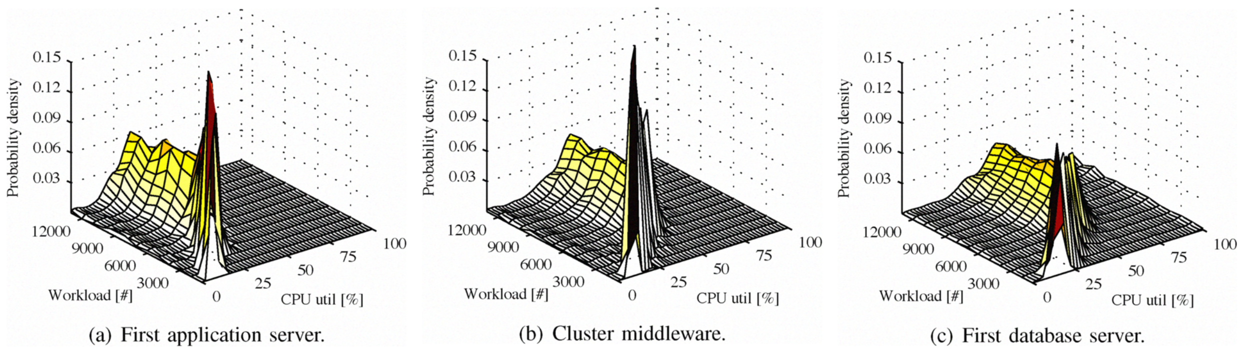


Figure 4. PostgreSQL with C-JDBC on normal servers densities with read/write workload, 1/2/1/2 configuration, and wait-all replication strategy.

combined into a single maximum utilization value in Figure 5(c), the density profile shows a stable saturation behavior. Both Figure 2(e) and Figure 2(f) show that this bottleneck can be resolved by replicating the database node, which increases the maximum throughput.

The single disk bottleneck becomes an oscillatory bottleneck for more than one database server with the wait-all replication strategy because write queries have to await the response of all MySQL servers (Figure 3(b)). An oscillatory bottleneck means that at any arbitrary point in time (usually) only one disk is saturated, and that the bottleneck location changes constantly. Once a third database server has been added, the CPU bottleneck is resolved completely, and the maximum throughput remains invariant to any further hardware additions. Although the saturated resources (i.e. database disks) are located in the database tier, the C-JDBC configuration prohibits the increase of maximum throughput through additional hardware in the backend. Figure 6(b) shows that in the case of nine database nodes, the bottleneck has been further distributed among the database disks. Solely a slight mode at the high percentile values indicates an infrequent saturation. In order to visualize the overall resource saturation, Figure 6(c) shows the density for the maximum utilization of all nine database servers. This density has a large peak at the resource saturation sector (similar to Figure 5(c)), which is interpretable as a high probability of at least one saturated database disk at any arbitrary point during the experiment. Such bottlenecks are particularly hard to detect with average utilization values. Their cause is twofold—on the one hand, all write queries are dependent on all database nodes, and on the other, the load-balancer successfully distributes the load in case one database server is executing a particularly long query.

In contrast to the wait-all case, the wait-first repli-

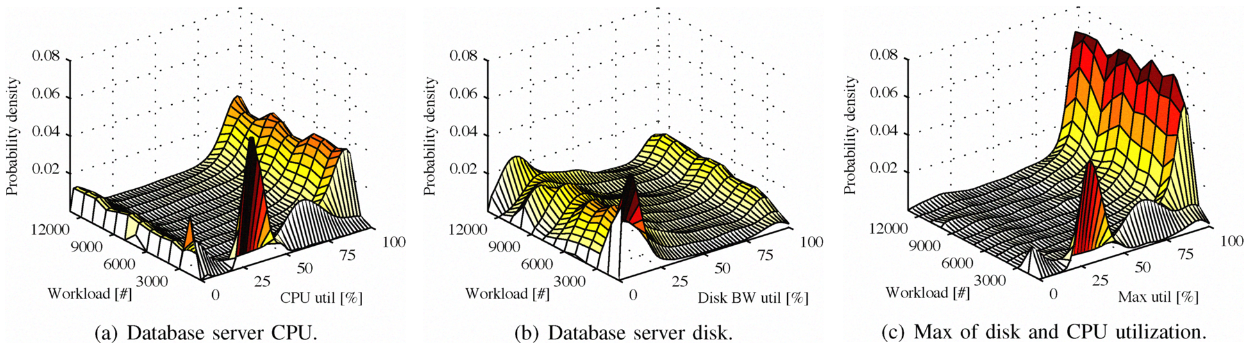
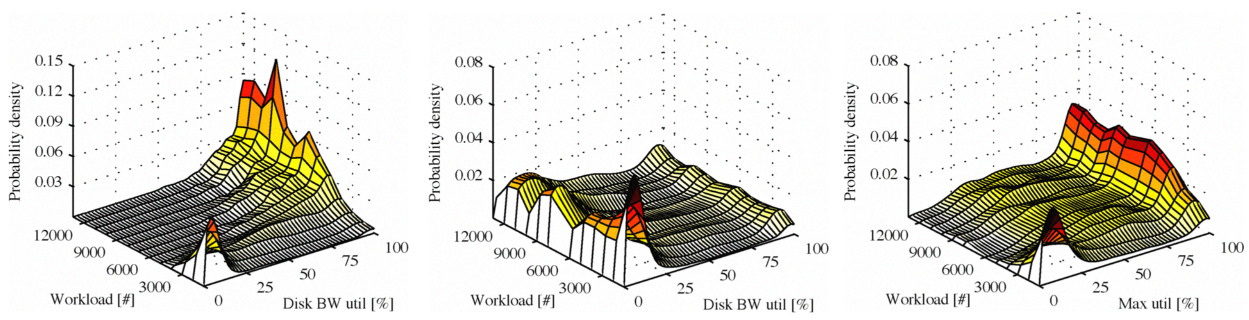


Figure 5. Database server densities for read/write workload, C-JDBC, wait-all replication strategy, and 1/2/1/1ML.



(a) First database server disk with wait-first replication strategy.

(b) First database server disk with wait-all replication strategy.

(c) Maximum disk utilization among all databases with wait-all replication strategy

Figure 6. Database server densities for read/write workload for 1/2/1/9ML, C-JDBC with wait-first replication strategy and wait-all replication strategy [2].

cation strategy enables growing maximum throughput when scaling out the database server (Figure 2(f)). Instead of oscillatory disk bottlenecks, the configurations with more than one database server have concurrent bottlenecks in the database disks (Figure 3(c)). After the database server has been replicated five times the database disks are the only remaining bottleneck. The difference between oscillatory and concurrent bottlenecks is clearly visible in the comparison of Figures 6(a) and 6(b). Both show the 1/2/1/9ML configuration for read/write workload. While the wait-first replication strategy (Figure 6(a)) results in a clearly saturated resource, the wait-all replication strategy (Figure 6(b)) results in an infrequently utilized database disk. The actual system bottleneck only becomes apparent in conjunction with all other database disks (Figure 6(c)). Due to overhead, the maximum throughput growth diminishes once very high database replication states are reached. In comparison to the normal hardware results, the low-cost configurations reach an overall lower throughput level for read/write workloads. Please refer to our previous work [2] for a detailed bottleneck analysis of this particular interesting scenario.

5. Economical Configuration Planning

While the preceding section emphasized the potential and complexity of empirical analysis of N-tier systems, this section presents a specific use-case. We show how these data may be employed in economically motivated configuration planning. In this scenario the empirical observed performance is transformed into economic key figures. This economic view on the problem is of particular interest given the new emerging trend of cloud computing that enables flexible resource management and offers transparent cost models. The cloud paradigm allows the adaption of infrastructure according to demand [10] and determining cost optimal infrastructure sizes for enterprise systems [11]. However, taking full advantage of cloud computing requires an even deeper understanding of the performance behavior of enterprise systems and its economical implications.

Any economic assessment requires at least a basic cost model and a simple revenue model. Our sample cost model assumes a constant charged for each server hour. The revenue model is a simple implementation of an SLA. For every successfully processed request, the provider receives a constant revenue. For every unsuccessful request, the provider pays a penalty, whereby a successful request is defined as a request with a response time lower than a certain threshold. The total profit is defined as the total revenue minus the cost of operation.

Figure 7 shows the transformation of the empirical data to aggregated economic values. Instead of the number of machines in the infrastructure, the graph shows the cost incurred by operating a particular infrastructure. The z-axis shows the expected revenue when operating a particular infrastructure size at any given workload (y-axis). The data are taken from the previously introduced RUBBoS with MySQL Cluster setup with browse-only workload. The graph clearly shows the complex relationship between infrastructure size, system performance, and expected revenue. Based on a similar data aggregation, system operators are able to derive the optimal infrastructure size constrained by their system and profit model.

Figure 8 shows a higher aggregation of the previous

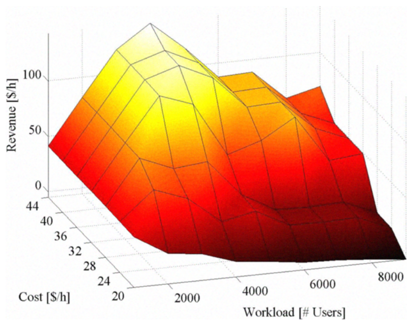


Figure 7. Revenue and cost analysis of a RUBBoS system under read/write workload.

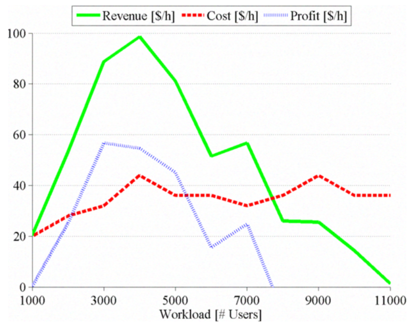


Figure 8. Profit and cost analysis of a RUBBoS system under read/write workload.

graph. The dashed and the dotted line represent the revenue and the cost for the profit optimal configuration, respectively. The solid line shows the resulting maximal profit for any configuration. The profit maximum lays at a workload of 3,000 users, whereas the maximum revenue is located at 4,000 users. However, the optimal points of operation strongly depend on the definition of the SLA model as well as on the cost of the infrastructure. Therefore, depending on these factors, the economical size of the infrastructure may vary strongly.

6. Related Work

Cloud computing, data center provisioning, server consolidation, and virtualization have become ubiquitous terminology in the times of ever-growing complexity in large-scale computer systems. However, many hard problems remain to be solved on the way to an optimal balance between performance and availability. Specifically, scaling systems along these two axis is particularly difficult with regard to databases where replication has been established as the common mechanism of choice. In fact, replication consistently falls short of real world user needs, which leads to continuously emerging new solutions and real world database clusters to be small (i.e., less than 5 replicas) [12]. In this paper we address this shortcoming explicitly. A central evaluation problem is that scalability studies traditionally only address scaled load scenarios to determine best achievable performance (e.g., [13]). Concretely, experimentation methodologies with parallel scaling of load and resources mask system overhead in realistic production system conditions, which are typically over-provisioned. Consequently, a reliable body of knowledge on the aspects of management, capacity planning, and availability is one of the limiting factors for solution impact in the real world [12]. Moreover, it is well understood that transactional replication can exhibit unstable scale-up behavior [14] and it is not sufficient to characterize replication performance in terms of peak throughput [12].

There exist many popular solutions to database replication, and the most common commercial technique is master-slave replication. Some replication product examples are IBM DB2 DataPropagator, replication in Microsoft SQL Server, Sybase Replication Server, Oracle Streams, replication in MySQL. Academic prototypes that use master-slave replication are Ganymed [15] and Slony-I [16]. The latter is a popular replication system supporting cascading and failover. However, in this work we focus on multi-master architectures. Some commercial products that use this technique are MySQL Cluster and DB2 Integrated Cluster. Academic multi-master prototypes are C-JDBC [7], which is used in this work, Tashkent [17], and Middle-R [18]. Further efforts focus on adaptive middleware to achieve performance adaptation in case of workload variations [19].

The evaluation of database replication techniques often falls short of real representability. There are two common approaches to evaluation with benchmarks. Performance is either tested through microbenchmarks [18] or through web-based distributed benchmark applications such as TPC-W, RUBiS, and RUBBoS [20]. These benchmark applications are modeled on real applications (e.g., Slashdot or Ebay.com), offering real production system significance. Therefore this approach has found a growing group of advocates (e.g., [3], [7], [13], [15], [17]).

Most recent literature on the topic of database replication deals with specific implementation aspects such as transparent techniques for scaling dynamic content web sites [21], and the experimental evaluation in such works remains narrow in focus and is solely used to confirm expected properties.

7. Conclusion

In this paper we used an empirically motivated approach to automated experiment management. Which features automatically generated scripts for running experiments and collecting measurement data, to study the RUBBoS N-tier application benchmark in an extensive configuration space. Our analysis produced various interesting findings such as detailed node-level bottleneck migration maps and insights on generation of economical configuration plans. Furthermore, our analysis showed cases with non-obvious multi-bottleneck phenomena that entail very characteristic performance implications.

This paper corroborates the increasing recognition of empirical analysis work in the domain of N-tier systems. Findings such as multi-bottlenecks demonstrate how actual experimentation is able to complement analytical system evaluation and provide higher levels of confidence in analysis results. More generally, our results suggest that further work on both the infrastructural as well as the analysis part of our approach may lead to a new perspective on analysis of N-tier systems with previously unseen domain insight.

Acknowledgment

This research has been partially funded by National Science Foundation grants ENG/EEC-0335622, CISE/CNS-0646430, CISE/CNS-0716484, AFOSR grant FA9550-06-1-0201, NIH grant U54 RR 024380-01, IBM, Hewlett-Packard, Wipro Technologies, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

References

[1] "RUBBoS: Bulletin board benchmark," http://jmob.objectweb.org/rubbos.html.

[2] S. Malkowski, M. Hedwig, and C. Pu, "Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks," in *IISWC '09*.

[3] C. Pu, A. Sahai, J. Parekh, G. Jung, J. Bae, Y.-K. Cha, T. Garcia, D. Irani, J. Lee, and Q. Lin, "An observation-based approach to performance characterization of distributed n-tier applications," in *IISWC '07*, September 2007.

[4] A. Sahai, S. Singhal, V. Machiraju, and R. Joshi, "Automated generation of resource configurations through policies," in *in Proceedings of the IEEE 5 th International Workshop on Policies for Distributed Systems and Networks*, 2004, pp. 7–9.

[5] G. Jung, C. Pu, and G. Swint, "Mulini: an automated staging framework for qos of distributed multi-tier applications," in *WRASQ '07*.

[6] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai, "Bottleneck detection using statistical intervention analysis," in *DSOM '07*.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-jdbc: Flexible database clustering middleware," in *USENIX '04*.

[8] "Mysql Cluster," http://www.mysql.com/products/database/cluster/.

[9] "Emulab - Network Emulation Testbed," http://www.emulab.net.

[10] M. Hedwig, S. Malkowski, and D. Neumann, "Taming energy costs of large enterprise systems through adaptive provisioning," in *ICIS '09*.

[11] M. Hedwig, S. Malkowski, C. Bodenstein, and D. Neumann, "Datacenter investment support system (DAISY)," in *HICSS '10*.

[12] E. Cecchet, G. Candea, and A. Ailamaki, "Middleware-based Database Replication: The Gaps Between Theory and Practice," in *SIGMOD '08*.

[13] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Partial replication: Achieving scalability in redundant arrays of inexpensive databases," *Lecture Notes in Computer Science*, vol. 3144, pp. 58–70, July 2004.

[14] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *SIGMOD '96*.

[15] C. Plattner and G. Alonso, "Ganymed: scalable replication for transactional web applications," in *Middleware '04*.

[16] "Slony-I," http://www.slony.info.

[17] S. Elnikety, S. Dropsho, and F. Pedone, "Tashkent: uniting durability with transaction ordering for high-performance scalable database replication," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 117–130, 2006.

[18] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso, "Middle-r: Consistent database replication at the middleware level," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 375–423, 2005.

[19] J. M. Milan-Franco, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme, "Adaptive middleware for data replication," in *Middleware '04*.

[20] C. Amza, A. Ch, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, and W. Zwaenepoel, "Specification and implementation of dynamic web site benchmarks," in *5th IEEE Workshop on Workload Characterization*, 2002.

[21] C. Amza, A. L. Cox, and W. Zwaenepoel, "A comparative evaluation of transparent scaling techniques for dynamic content servers," in *ICDE '05*.