Comprehensive Depiction of Configuration-dependent Performance Anomalies in Distributed Server Systems[*]

Christopher Stewart Ming Zhong Kai Shen Thomas O'Neill
Department of Computer Science, University of Rochester
{stewart, zhong, kshen, toneill}@cs.rochester.edu

Abstract

Distributed server systems allow many configuration settings and support various application workloads. Often performance anomalies, situations where actual performance falls below expectations, only manifest under particular runtime conditions. This paper presents a new approach to examine a large space of potential runtime conditions and to comprehensively depict the conditions under which performance anomalies are likely to occur. In our approach, we derive our performance expectations from a hierarchy of sub-models in which each submodel can be independently adjusted to consider new runtime conditions. We then produce a representative set of measured runtime condition samples (both normal and abnormal) with carefully chosen sample size and anomaly error threshold. Finally, we employ decision tree based classification to produce an easy-to-interpret depiction of the entire space of potential runtime conditions. Our depictions can be used to guide the avoidance of anomaly-inducing system configurations and it can also assist root cause diagnosis and performance debugging. We present preliminary experimental results with a real J2EE middleware system.

1 Introduction

Distributed software systems are increasingly complex. It is not uncommon for the actual system performance to fall significantly below what is intended by the high level design [1, 3, 7, 16]. Causes for such anomalies include overly simplified implementations, mis-handling of special cases, and configuration errors. Performance anomalies hurt the system performance predictability, which is important for many system management functions. For instance, quality-of-service (QoS) management relies on predictable system performance behaviors to satisfy QoS constraints [2, 15]. More generally, optimal resource provisioning policies can be easily determined when the system performance under each candidate policy is predictable [4, 9, 17].

The detection, characterization, and debugging of performance anomalies have been investigated by many recent works [1, 5, 6, 7, 8, 11, 13]. In general, these studies focus on anomaly-related issues under the specific runtime conditions encountered during a particular execution. However, complex systems (*esp.*, distributed server systems) often allow many configuration settings (*e.g.*, cache coherence protocol, thread pool size) and they support a variety of workload conditions (*e.g.*, concurrent users, read-to-write ratio). It is desirable to explore performance anomalies over a comprehensive set of runtime conditions. Such exploration is useful for quality assurance or performance debugging without the knowledge of exact runtime conditions.

This paper presents a new approach to examine a large space of potential system runtime conditions and depict the conditions under which performance anomalies are likely to occur. In particular, we investigate techniques to build scalable performance models over a wide range of runtime conditions, to acquire a representative set of sample runtime conditions, to isolate anomalies due to similar causes, and to generate easy-to-interpret anomaly depictions.

The result of our approach is a probabilistic depiction of whether the system performs anomalously under each potential runtime condition. Our depiction can be used to guide the avoidance of anomaly-inducing system configurations under given input workload conditions. Additionally, we can identify the runtime conditions which are most correlated with anomaly manifestations. We can then narrow the search space for performance debugging to the system functions which are affected by the problematic conditions.

In principle, our proposed anomaly depiction approach can be applied to any software system that allows many configuration settings and supports various application workloads. In this paper, we focus on component-based distributed server systems. A typical example of such systems may be built on a J2EE platform and it may support various application components, provide common services (possibly remotely), and manage

[*]This work was supported in part by the National Science Foundation (NSF) grants CCR-0306473, ITR/IIS-0312925, CNS-0615045, CCF-0621472, NSF CAREER Award CCF-0448413, and an IBM Faculty Award.

application-level protocols to access storage and remote services. Additionally, these systems are often deployed in distributed environments with high-concurrency workload conditions.

2 Performance Expectations for Wide Ranges of Configuration Settings

It is challenging to produce accurate performance expectations for distributed server systems due to their inherent complexity. Recent work by Urgaonkar *et al.* models the performance of multi-tier distributed services using a network of queues [20]. Our own past work constructed performance prediction models for component-based distributed systems [17]. However, neither of these models consider the performance effects of wide ranges of configuration settings. Configuration settings can have complex effects on the workload of individual system functions (*e.g.*, component invocation, distributed caching, etc.) which can significantly affect the overall system performance.

To consider wide ranges of configuration settings, we derive performance expectations from a hierarchy of sub-models. Each sub-model predicts a workload property from lower-level properties according to the effects of system functions under given configuration settings. At the highest level of our hierarchy, the predicted workload properties are the desired performance expectations. The workload properties at the lowest-levels in our model are canonical workload properties that can be independently measured with no concern of the system functions or their configuration settings. These properties include component CPU usage and inter-component communication in terms of bandwidth usage and blocking send/receive count.

The architecture of hierarchical sub-models allows us to manage increasing system complexity with multiple independent smaller modules. More specifically, sub-models can be independently adjusted to accommodate new system configuration settings. Further, the intermediate workload predictions can also be examined for performance anomalies. In particular, an anomalous workload detected in one sub-model but not in its input sub-models can narrow the scope of potential problematic system functions during debugging.

Figure 1 illustrates our model that considers several system functions and configuration settings that have significant impact upon performance. The system functions we consider include distributed caching, remote component invocation, distributed component placement, and server concurrency management. Implementation and configuration errors related to these functions are most likely to cause large performance degradation. While our

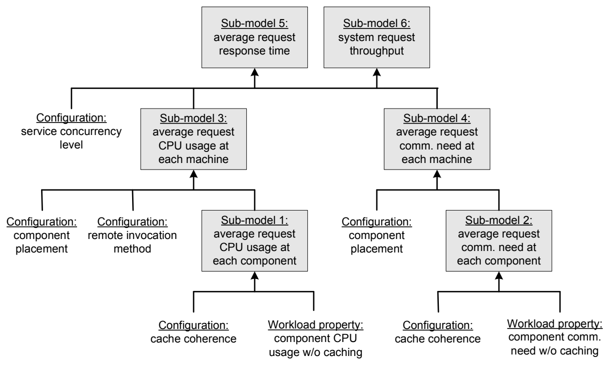
Sub-model 5: average request response time
Sub-model 6: system request throughput
Configuration: service concurrency level
Sub-model 3: average request CPU usage at each machine
Sub-model 4: average request comm. need at each machine
Configuration: component placement
Configuration: remote invocation method
Sub-model 1: average request CPU usage at each component
Configuration: component placement
Sub-model 2: average request comm. need at each component
Configuration: cache coherence
Workload property: component CPU usage w/o caching
Configuration: cache coherence
Workload property: component comm. need w/o caching

Figure 1: A hierarchy of sub-models. Low-level workload properties are assembled into higher-level properties according to the performance impact of configuration settings.

high level model should be generally applicable to most component-based distributed server systems, our specific model construction targets J2EE-based distributed systems and our effort builds on our earlier work in [17]. We omit modeling details in this paper due to space limitation.

3 Anomaly Sample Collection

For complex distributed server systems, there is often a large space of possible runtime conditions in terms of system configurations and workload condition. Conceptually, the runtime condition space has multiple dimensions where each system configuration and workload condition that can be independently adjusted is a dimension. We choose sample conditions from the space of potential runtime conditions in a uniformly random fashion. We then compare measured system performance with our model-based expectation under these conditions. Anomalous conditions are those at which measured performance trails the expectation by at least a certain threshold. Below we present techniques to determine the size of our random sample and to select the anomaly error threshold.

#1. Sample size. Anomaly checking at too many sample conditions may consume an excessive amount of time and resource (due to performance measurement and expectation model computation at all sampled conditions). At the same time, we must have a sufficient sample size in order to acquire a representative view of performance anomalies over the complete runtime condition space. Our approach is to examine enough sample conditions so that key cumulative sample statistics (*e.g.*, average and standard deviation of sample performance expectation errors) converge to stable levels.

#2. Anomaly error threshold. Another important ques-

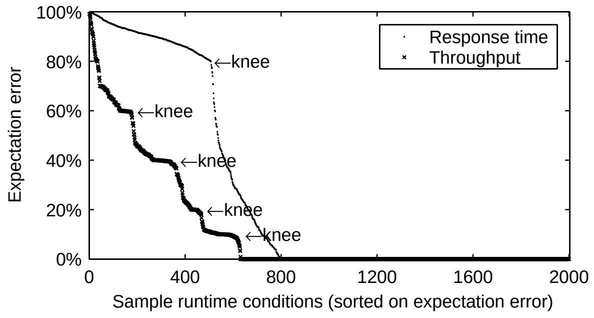

Figure 2: Response time and throughput expectation errors of 2000 sampled runtime conditions sorted in decreasing order. We mark knee points on the two curves for possible identification of the anomaly error threshold.

tion is ”How much deviation between the measured performance and the expectation counts for an anomaly?” The determination of anomaly error threshold depends on the eventual purpose of anomaly depiction. When the depiction result is used for the avoidance of anomaly-inducing conditions, the anomaly error threshold is usually specified at a tolerance level of expectation error by the system administrator. When the depiction result is used for performance debugging, it is important for the anomaly sampling to contain minimal noises that are not related to targeted bugs.

To minimize the noises, we utilize an observation that performance anomaly manifestations due to the same cause (*e.g.*, an implementation bug or a misconfiguration) are more likely to share similar error magnitude than unrelated anomaly manifestations do. In our approach, we sort all sampled conditions according to their performance expectation errors. We then look for knee points on the performance expectation error curve, which serve as likely thresholds attributing anomaly manifestations to different hidden causes. One-dimensional clustering algorithms (*e.g.*, k-means [10]) can be used to determine knee points in a given set of expectation errors — the knee points are those thresholds that separate different error groups generated by the clustering algorithm.

Figure 2 illustrates an example of knee points discovered on the error curves of sample conditions sorted in the order of expectation errors. When multiple knee points are available, we can choose a low error threshold to capture multiple bugs in one anomaly depiction. Alternatively, we can choose a high error threshold in order to focus on a few severe bugs in one depiction, fix them, and then characterize more bugs in later rounds of anomaly depiction. Conceivably, we can choose a maximum and minimum error threshold to target a specific bug, however we have not yet investigated noise reduction using this approach.

4 Performance Anomaly Depiction

From a representative set of sampled runtime conditions (each labeled “normal” or “anomalous”), we study the problem of deriving a comprehensive performance anomaly depiction. The depiction is essentially a classifier of system and workload configurations that partitions the runtime condition space into normal and anomalous regions of hyper-rectangular shapes. There has been a great number of well-proven classification techniques, such as naive Bayes classifiers, perceptrons, decision trees, neural networks, Bayesian networks, support vector machines, and hidden Markov models. We use decision trees to build our performance depiction because they have the following desirable properties.

- *Easy interpretability*. Compared with other “black-box” classification techniques (*e.g.*, neural networks, support vector machines) where the explanation for the results is too complex to be comprehended, decision trees can be easily understood as IF-THEN rules or graphs, which provide great assistance to further debugging efforts.
- *Prior knowledge free*. Decision trees do not require any prior knowledge on underlying models, data distributions, or casual relationships, which are often not available in our studied systems. In comparison, hidden Markov models and Bayesian networks typically need to have a preliminary model before they learn to parameterize the model.
- *Efficiency and robustness*. Decision trees can quickly handle a large amount of noisy data with both numerical and categorical elements, requiring little preparation work such as normalization, missing value prediction, *etc*. Such ease of implementation is especially desirable when we consider a large number of factors on the system performance.

We use the Iterative Dichotomiser 3 (ID3) algorithm [12] to generate our performance anomaly depiction. Our depiction is a decision tree that classifies a vector of workload conditions and system configurations into one of two target classes — “normal” or “anomalous”. In ID3, the decision tree is generated in a top-down way by iteratively selecting an attribute that best classifies current training samples, partitioning samples based on their corresponding values of the attribute, and constructing a subtree for each partition until current samples all fall into the same category. Specifically, the tree-generation algorithm uses *information gain* to select an attribute that is most effective in classifying training samples. The information gain of an attribute a over

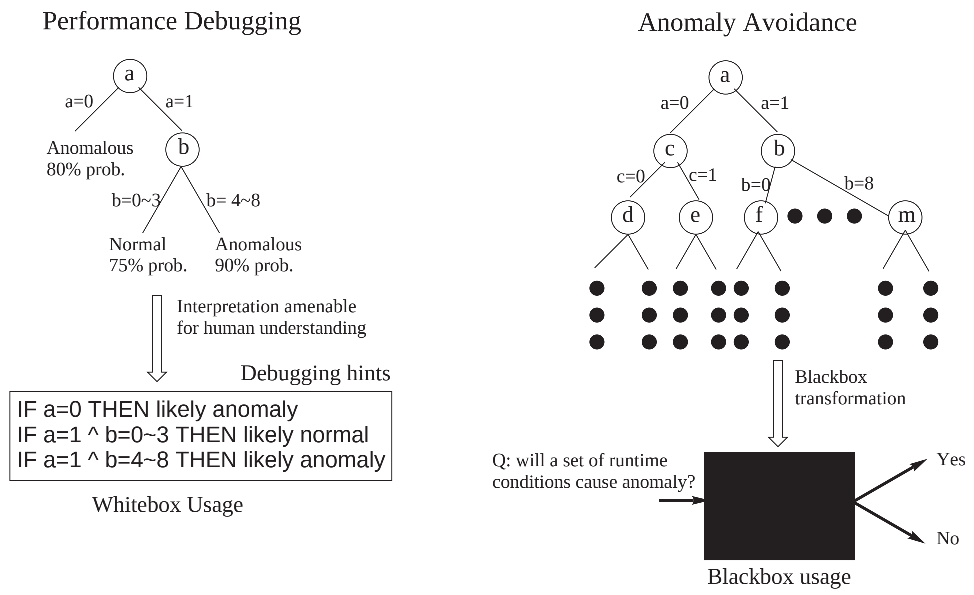

Figure 3: Decision tree-based performance anomaly depiction.

dataset S is defined as follows.

$$Gain(S,a) = H(S) - \sum_{v \in values(a)} \frac{|S_v|}{|S|} \cdot H(S_v)$$

where S_v is the subset of S whose values of attribute a are v. $H(S)$ is the information entropy of S. Intuitively, information gain measures how much data impurity is reduced by partitioning S based on attribute a.

As illustrated by Figure 3, the primary usage of our performance anomaly depiction falls into the following two categories.

- *White-box depiction usage — performance debugging*. Performance anomaly depiction is used to make correlations between performance problems and system functions. For this usage, human interpretability of the depiction is most important. As a result, we stop growing the decision tree at a certain depth so that the result can be easily understood.
- *Black-box depiction usage — anomaly avoidance*. Performance anomaly depiction is used as an oracle, which answers whether a given system configuration (with current input workload) leads to anomaly or not. For this purpose, the depiction accuracy is paramount.

5 Preliminary Experimental Results

We experiment with JBoss — a commercial-grade open-source J2EE application server. We consider 8 runtime conditions: cache coherence protocols, component invocation method, component placement strategy, thread pool size, application type, concurrent users, read-/write mix, and database access mix. These conditions combine to form over 7 million potential runtime conditions. In our experiments, JBoss may service the application logic of RUBiS [14], StockOnline [18], or TPC-W [19] depending upon runtime settings. Our tests run on a small cluster of four machines each with a 2.66 GHz P4 processor and 512 MB of memory.

Avoidance Figure 4 provides an illustration of full depiction result for the purpose of anomaly avoidance. We evaluate the accuracy of our decision tree by comparing the prediction result with measured result on some additional randomly chosen system runtime conditions. We examine two metrics: *precision* (the proportion of real depicted anomalies in all depicted anomalies) and *recall* (the proportion of real depicted anomalies in all real anomalies). There is a fundamental trade-off between precision and recall in that attempts to increase recall tend to enlarge the depicted anomaly region, which may mistakenly include normal settings and hence reduce precision. Attempts to increase precision typically exclude some real anomalous points from the depicted anomaly region and thus decrease recall. Such a trade-off can be exploited by setting different anomaly error threshold. At the anomaly error threshold of 30% (*i.e.*, performance deviation of 30% or more from the expectation is considered an anomaly), our depiction precision is 89% and recall is 83%. This result indicates a high accuracy of our depiction result.

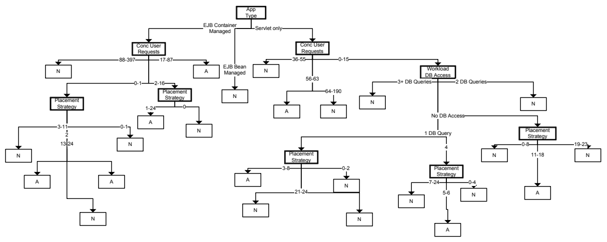

Figure 4: The full depiction tree for anomaly avoidance.

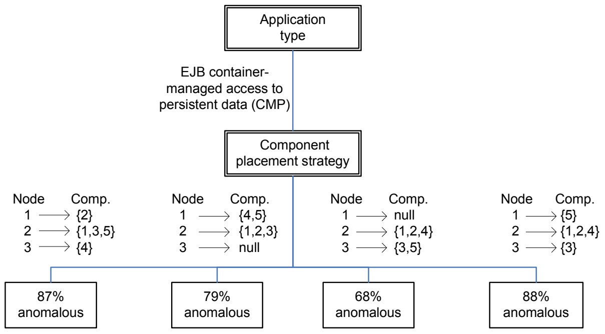

Figure 5: A three-level decision tree produced for debugging. Non-leaf nodes represent system configurations or workload conditions while leaf nodes represent decision points. The value inside each decision point indicates the likelihood of performance anomalies within that decision point's region. For brevity, we only show the decision points in the tree that are substantially anomalous.

Performance Debugging We also explore the debugging use of our depiction result. For each anomalous leaf node, the path leading back to the decision tree root represents a series of anomaly-correlated conditions (system configurations and workload conditions). Such knowledge may narrow down the debugging scope and provide hints to human debuggers. As noted in Section 4, the decision trees we generated for debugging have artificially low depths. This is to make the depiction results amenable for human understanding. Even with the aid of anomaly depiction, it is important to note that the actual discovery of each anomaly root cause is still a substantially manual process. In our investigation, the human debugger has at his disposal a set of system tracing and debugging tools at both the EJB application server level and inside the operating system kernel.

Our depiction-aided debugging has uncovered three performance problems in the system: a component placement configuration error due to ambiguous J2EE specification, a mis-handled exception in the JBoss thread management, and a potential deadlock in distributed concurrency management. Due to space limitation, we only show the debugging of the first problem in this paper.

Figure 5 shows the three-level decision tree produced for debugging. We observe a strong correlation between performance anomaly and four specific component placement strategies. A closer look revealed that in all of these correlated placement strategies, a majority of the components were placed on node #2 in our testbed. We inserted trace points into the system to monitor the network traffic and component lookup behaviors concerning node #2. We noticed that component lookup procedures executed correctly, but that subsequent invocations were never performed on node #2 and were instead routed to node #1. We traced the abnormal behavior of the lookup function to a mis-understood J2EE specification by the programmer. The component lookup procedure returns a pointer to a component instance in the system, but not necessarily at the node upon which the lookup is performed. In particular, our node #2 was mis-configured to route component invocations to node #1 despite the programmer intention for components to execute on node #2. The increased queuing delay at node #1 caused response time and throughput degradation beyond what was expected by our performance models.

6 Conclusion and Discussions

This paper presents a new approach to comprehensively depict performance anomalies over a large space of potential runtime conditions. We also demonstrate the practical benefits of such comprehensive anomaly depiction, in terms of avoiding anomaly-inducing runtime conditions and of assisting performance debugging. Below we discuss several additional issues.

First, our configuration-dependent anomaly depiction approach targets performance anomalies that are correlated with static system configurations and workload conditions. For instance, our approach may find a RMI (remote method invocation) performance anomaly that is triggered under a particular component placement policy. However, a performance bug that only manifests when several different types of RMIs are made in a particular order may not be discovered by our methods. Although this is an important limitation, we believe the proposed approach is still effective for discovering many existing performance anomalies, as demonstrated in our experimental results.

Second, constructing performance expectations over many system configurations for a target system is still a substantially manual effort in our approach. The performance expectation model itself may not be accurate at all considered runtime conditions and such inaccuracy may also be a source for "performance anomalies". While these "anomalies" may overshadow or even conceal real problems in system implementation and configuration, we argue that our anomaly depiction can similarly assist the discovery and correction of the expectation model inaccuracies. Since the performance model is typically much less complex than the system itself, its analysis and debugging should also be much easier.

Our work is ongoing. In the short-term, we plan to extend the range of runtime conditions (*esp.* system configurations) supported by our performance model. We also hope to expand our empirical study to comprehensively depict anomalies for other systems. Further anomaly depictions may shed more light into common areas of performance problems in distributed server systems. In the long-term, we will investigate systematic methods to depict the correlation between performance anomalies and source code level system components or parameter settings. This capability could further narrow the investigation scope in depiction-driven performance debugging.

Acknowledgment

We would like to thank Arun Iyengar, Arrvindh Shriraman, Jian Yin, and the anonymous referees for helpful discussions and valuable comments.

References

[1] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, October 2003.

[2] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-Scale Storage Systems. *ACM Trans. on Computer Systems*, 19(4):483–418, November 2001.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modeling. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[4] J.S. Chase, D.C. Anderson, P.N. Thakar, A.M. Vahdat, and R.P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 103–116, Banff, Canada, October 2001.

[5] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proc. of the First USENIX Symp. on Networked Systems Design and Implementation*, pages 309–322, San Francisco, CA, March 2004.

[6] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Systems. In *Proc. of Int'l Conf. on Dependable Systems and Networks*, pages 595–604, Washington, DC, June 2002.

[7] I. Cohen, J.S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 231–244, San Francisco, CA, December 2004.

[8] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *Proc. of the 20th ACM Symp. on Operating Systems Principles*, pages 105–118, Brighton, United Kingdom, October 2005.

[9] R.P. Doyle, J.S. Chase, O.M. Asad, W. Jin, and A.M. Vahdat. Model-based Resource Provisioning in a Web Service Utility. In *Proc. of the 4th USENIX Symp. on Internet Technologies and Systems*, Seattle, WA, March 2003.

[10] J. B. McQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281 – 297, 1967.

[11] T. Kelly. Detecting Performance Anomalies in Global Applications. In *Proc. of the Second Workshop on Real, Large Distributed Systems*, pages 43–48, San Francisco, CA, December 2005.

[12] J.R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[13] P. Reynolds, C. Killian, J.L. Wiener, J.C. Mogul, M.A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proc. of the Third USENIX Symp. on Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[14] RUBiS: Rice University Bidding System. http://rubis .objectweb.org.

[15] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, pages 225–238, Boston, MA, December 2002.

[16] K. Shen, M. Zhong, and C. Li. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *Proc. of the 4th USENIX Conf. on File and Storage Technologies*, pages 309–322, San Francisco, CA, December 2005.

[17] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proc. of the Second USENIX Symp. on Networked Systems Design and Implementation*, pages 71–84, Boston, MA, May 2005.

[18] The StockOnline Benchmark. http://forge.objectweb.org /projects/stock-online.

[19] Transaction Processing Performance Council. TPC-W Web E-Commerce Bencmark. http://www.tpc.org/tpcw.

[20] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proc. of the ACM SIGMETRICS*, pages 291–302, Banff, Canada, June 2005.