Profile-driven Component Placement for Cluster-based Online Services*

Christopher Stewart, Kai Shen, Sandhya Dwarkadas, Michael L. Scott
{stewart, kshen, sandhya, scott}@cs.rochester.edu
Department of Computer Science, University of Rochester

Jian Yin
jianyin@us.ibm.com
IBM Research

Abstract

The growth of the Internet and of various intranets has spawned a wealth of online services, most of which are implemented on local-area clusters using remote invocation (*e.g.*, RPC/RMI) among manually placed application components. The placement problem can be a significant challenge for large scale services, particularly when application resource needs are workload dependent. This paper describes our initial work on *automatic* component placement, with the goal of maximizing overall system throughput. The key idea behind our approach is to construct (off-line) a mapping between input workload and individual component resource consumption. Such mappings, called component profiles, are then employed to support high-performance placement. We describe the basic design framework and present preliminary results on a J2EE-based online auction benchmark. Our results suggest that profile-driven tools can indeed identify placements that achieve near-optimal overall throughput.

1 Introduction

Recent years have witnessed significant growth in online services, including Web search engines, digital libraries, and electronic commerce. These services are most often deployed on clusters [8, 14] of commodity machines in order to achieve high availability, incremental scalability, and cost effectiveness in the face of rapid service evolution and increases in user demand. Their software architecture typically comprises many components, some reflecting intentionally modular design, others developed independently and subsequently assembled into a larger application, *e.g.*, to handle data from independent sources. A typical service might contain components responsible for data management, for business logic, and for presentation of results in HTML or XML. The placement of these components on cluster nodes is challenging for several reasons. First, there can be substantial heterogeneity both in component resource needs and in available resources at different nodes. Second, maintaining reasonable quality of service (*e.g.*, response time) is imperative for interactive network clients. Third, the ideal placement may be a function not only of static application characteristics, but of various runtime factors as well, including bursty user demand, machine failures, and system upgrades. Our goal is to develop the software infrastructure needed for efficient dynamic component placement in cluster-based online services.

Our basic approach (shown in Figure 1) is to build per-component resource consumption profiles as a function of input workload characteristics. The currently considered resources are CPU, network bandwidth, and memory usage, each of which we characterize in terms of average and peak resource requirements. Component placement decisions are then made based on component profiles, available system resources, and runtime workload characteristics, using low-complexity optimization techniques when exhaustive search of resource to requirement mappings becomes infeasible. Placement decisions can be made at a centralized executive server or they can be made in a fully distributed fashion. They can also be made dynamically for runtime component migration by monitoring input workload characteristics. Compared with existing component mobility management techniques, our key contribution is a parameterization based on input workload characteristics, which enables inexpensive monitoring and accurate prediction based on the component profiles.

Our work is related to a number of prior studies on distributed component placement. Coign [10] examines the optimization problem of minimizing communication time for two-machine client-server applications. ABACUS [2] focuses on the placement of I/O-specific functions for cluster-based data-intensive applications. Addistant [19] and J-Orchestra [20] support partitioning and distributing execution of "legacy" Java applications through byte code rewriting. DVM [15] further adds security to such support. Other than Coign, these projects focus on mechanisms for transparent remote execution, leaving placement decisions largely to users. Ivan *et al.* examine the automatic deployment of component-based software over the Internet subjected to throughput requirements [11]. In their approach, the resource requirements for each component must be specified by application developers. Urgaonkar

*This work was supported in part by NSF grants EIA-0080124, CCR-0204344, CCR-0219848, ECS-0225413, CCR-0306473, and ITR/IIS-0312925; by the U.S. Dept. of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460; and by equipment or financial grants from IBM and Sun Microsystems Laboratories.

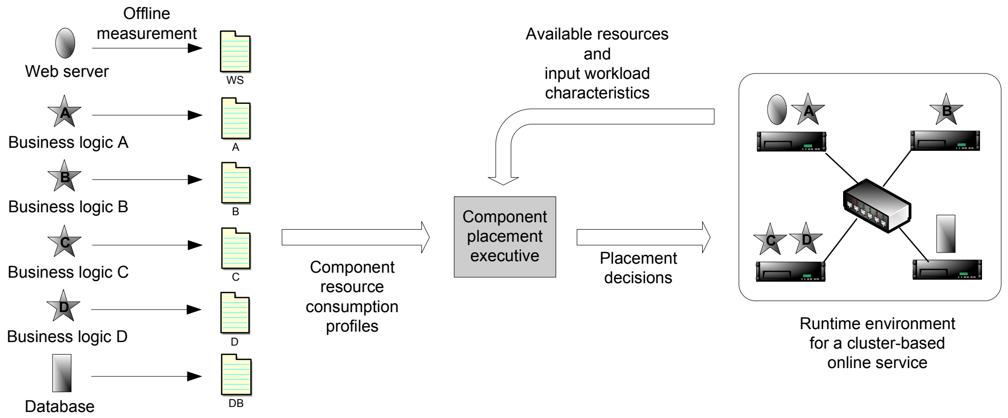

Figure 1: Profile-driven component placement.

et al. study the benefit of allowing applications to overbook CPU and network resources in shared hosting platforms [21]. Their work is limited to the placement of multiple single-component server applications that do not interact with one another. The Aura [16] and Chroma [4] projects propose that system-level mobility information be exported to user-level agents, which can then make strategic decisions (including component placement) based on their model of user activity and intent. User-level involvement in placement systems seems particularly valuable for "intelligent" applications, in which user intent is central and closely tied to mobility. For online services, however, we emphasize transparent system-level management.

The rest of this paper is organized as follows. Section 2 presents a design framework for profile-driven component placement. Section 3 describes a prototype implementation, together with experimental results on a J2EE-based online auction benchmark. Section 4 presents conclusions along with a summary of future directions.

2 Design Framework

This section describes the 3 main features of our component placement system. First, we examine offline measurement and modeling mechanisms for building per-component resource consumption profiles. Second, we describe profile-driven performance projection and an automatic component placement strategy optimized for high system throughput. Third, we consider design issues for runtime workload monitoring and dynamic component migration.

2.1 Building Component Profiles

Our component profile specifies component resource needs as functions of input workload specifications. For resources such as CPU and network I/O, we use an average rate ρ and a peak rate ϕ to capture the resource consumption specification for each resource type. We measure and accumulate resource consumption statistics at periodic intervals. The peak rate is the maximum or a high (*e.g.*, 90) percentile value of such periodic resource consumption statistics. The average and peak resource needs allow us to estimate upper and lower bounds on throughput. In terms of memory usage, variability in available memory size often has a severe impact on application performance. In particular, a memory deficit during one time interval cannot be simply compensated by a memory surplus in the next time interval. Therefore we use the maximum requirement ϕ_{mem} alone for specifying memory requirements in the component profile.

The input workload specifications in component profiles can be parameterized with an average request arrival rate $\lambda_{\text{workload}}$ and other workload characteristics (denoted by δ_{workload}) such as the composition of different request types (or request mix). The request mix is relevant because different request types may not consume the same resources. For example, a workload comprised entirely of Web page retrievals and one comprised entirely of new user registrations consume very different resources. Putting these altogether, the resource consumption profile for a distributed application component specifies the following mapping $f()$:

$$f(\lambda_{\text{workload}}, \delta_{\text{workload}}) \longrightarrow (\rho_{\text{cpu}}, \phi_{\text{cpu}}, \rho_{\text{network}}, \phi_{\text{network}}, \phi_{\text{mem}})$$

Many of the functional relationships in $f()$ are expected to follow simple expressible forms. For instance, ρ_{cpu}, ϕ_{cpu}, ρ_{network}, and ϕ_{network} are likely to be linear in $\lambda_{\text{workload}}$.

Several techniques are available to measure system resource utilization by application components under various workloads. One such technique utilizes OS-provided interfaces to acquire resource consumption statistics. For instance, the SYSSTAT toolkit [18] provides CPU, memory, and I/O resource consumption statistics through access to the Linux `/proc` interface. One drawback of such an approach is that the measurement accuracy is limited by the frequency of statistics reporting from the OS. Another technique, represented by the Linux Trace Toolkit

(LTT) [22], directly instruments the OS kernel to report resource consumption statistics. LTT can provide accurate system statistics, but it requires significant kernel changes.

Many recent studies have addressed application resource consumption profiling. Urgaonkar *et al.* use application resource usage profiling to guide application placement in shared hosting platforms [21]. Amza *et al.* provide bottleneck resource analysis for three dynamic online service benchmarks [3]. Gu and Nahrstedt examine QoS-aware multimedia service partitioning and placement based on service dependency graphs [9]. While their application profiling efforts are similar to our proposed work to a certain extent, our component profiles provide a more detailed characterization of the mapping between input request rate and application resource consumption. Such information is critical to making high-throughput placement decisions when resource needs are workload dependent. A recent study by Doyle *et al.* models the service response time reduction with increased memory cache size for Web servers [7]. However, such modeling is only feasible with intimate knowledge about application memory usage behavior. It does not share our goal of maintaining general applicability on a wide range of applications.

2.2 High Throughput Component Placement

Based on component resource consumption profiles, we seek to automatically place components where they will yield high overall throughput. The key to our optimized component placement is the ability to project system throughput under each placement strategy. Such ability is made possible by a three-step process.

1. The mapping between the input request rate $\lambda_{\text{workload}}$ and component runtime resource demands $(\rho_{\text{cpu}}, \phi_{\text{cpu}}, \rho_{\text{network}}, \phi_{\text{network}}, \phi_{\text{mem}})$ can be learned with the knowledge of the component profile and input workload characteristics.

2. Given a component placement strategy, we can derive the maximum input request rate that can saturate the CPU, network I/O bandwidth, or memory resources at each server. For CPU and network bandwidth, we can use either the average component resource needs or the peak needs in deriving the rate. Let $\tau_{\text{CPU average}}$, $\tau_{\text{CPU peak}}$, $\tau_{\text{network average}}$, $\tau_{\text{network peak}}$, and τ_{memory} denote such saturation rates at a server. When components are collocated on the same server, they must share the host CPU and memory resources while the intra-server component communication can take advantage of high bandwidth IPC mechanisms.

3. Finally, the system throughput can be estimated as the lowest saturation rate for all resource types at all servers. Using the average resource needs, we can derive an *optimistic* throughput estimate:

$$\min_{\text{for all servers}} \{\tau_{\text{CPU average}}, \tau_{\text{network average}}, \tau_{\text{memory}}\}$$

Using the peak resource needs, we can derive a *pessimistic* throughput estimate:

$$\min_{\text{for all servers}} \{\tau_{\text{CPU peak}}, \tau_{\text{network peak}}, \tau_{\text{memory}}\}$$

With the ability to project system throughput under any placement strategy, we can discover a high throughput component placement through exhaustive search. The search space for all possible placement strategies can be very large for applications with a substantial number of components over a large number of servers. Low-complexity optimization algorithms need to be devised when exhaustive search becomes computationally infeasible. Previous studies already proposed such algorithms for certain constrained cases (*e.g.*, Coign [10] and Chroma [4]).

We should also point out that business logic constraints may impact the placement policy. In this case, we simply remove invalid placement candidates from the search space, and select the high throughput choice from the remaining candidates.

2.3 Runtime Component Migration

In addition to supporting static component placement, we examine design issues for dynamic runtime component migration. Equipped with component resource consumption profiles, we also need knowledge of run-time dynamic workload characteristics in order to estimate runtime component resource needs. Such information is fed into the component placement executive to assist dynamic placement decisions. Some component middleware systems can be instrumented to trace inter-component messages [6, 17]. Less intrusive monitoring can be performed through network-level packet sniffing.

Recent work by Aguilera *et al.* [1] proposed performance debugging for multi-component online applications based on message traces. Their main technique is to analyze inter-component causal message paths and then derive the corresponding component response time. However, their study used pre-collected message traces and they did not explicitly address the acquisition of runtime workload characteristics. Odyssey [12] trades application resource demands for service quality, or *fidelity*, using a history-based predication of resource demands. In comparison, the availability of component profiles allows us to only monitor the input workload characteristics at runtime, which can be performed more cheaply and accurately than directly predicting runtime resource demands.

In addition to performance optimization, runtime component migration must consider several additional issues:

- In order to achieve a high level of **scalability** and **fault-tolerance**, the component placement executive can be constructed with a decentralized or even peer-to-peer architecture. For instance, servers can share

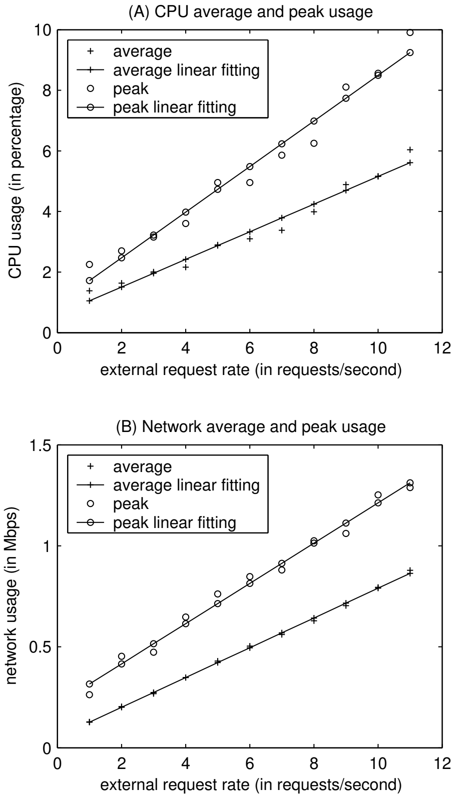

Figure 2: Linear fitting for the `Bid` component.

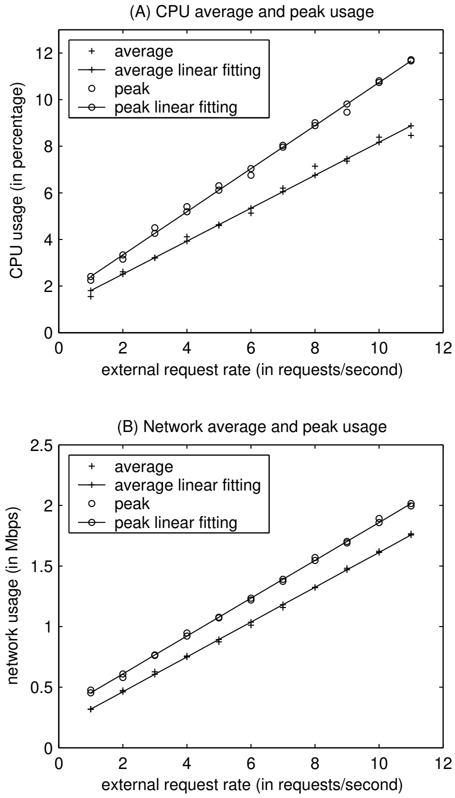

Figure 3: Linear fitting for the Web server.

runtime workload characteristics and resource consumption profiles of components with their peers. Component migration decisions could then be made independently when they are considered beneficial by participating servers.

- System **stability** is important for runtime component migration, especially when migration decisions are made in a decentralized fashion. A certain level of system stability can be achieved by employing a component migration threshold such that prospective migrations that do not produce over-the-threshold benefit are not performed. A careful balance must be maintained between responsiveness and system stability.

3 Preliminary Results

We present some preliminary results to illustrate the effectiveness of our profile-driven component placement. We only provide results for static placement in this section. Dynamic component migration will be investigated in the future. Our experiments are based on the RUBiS benchmark [5, 13], an auction site prototype modeled after `eBay.com`. RUBiS implements the core functionality of an auction site: selling, browsing, and bidding. It follows the three-tier Web service model containing a front-end Web server, nine movable business logic components, and a back-end database. Various versions of RUBiS have been implemented [5], we use the Enterprise Java Beans version with bean-managed persistence (BMP).

Profiling and experiments were conducted on a Linux cluster connected by a 100Mbps Ethernet switch. Each server is equipped with dual 1.26GHz Pentium III processors and 2GB memory. The RUBiS EJB components are hosted on a JBoss 3.2.3 Application server with an embedded Tomcat 5.0 servlet container. The database server runs MySQL 4.0. The dataset is sized according to database dumps published on RUBiS's Web site [13]. The total dataset size is around 1GB.

3.1 Component Profiling

RUBiS contains 11 components: a Web server, a database, and nine EJB components implementing the auction service logic (`Bid`, `BuyNow`, `Category`, `Comment`, `Item`, `Query`, `Region`, `User`, and `UserTransaction`). During our profiling runs, each component runs on a dedicated server and we measure the component resource consumption at a number of input request rates (1 req/sec, 2 reqs/sec, . . ., 11 reqs/sec). We use a request mix similar to one in [5] (15% read-write requests and 85% read-only requests). We use a modified SYSSTAT toolkit [18] in the measurement. Peak CPU and network usage are determined as 90 percentile values for measured rates at 1-

Component	CPU usage (in percentage)		Network usage (in Mbps)	
	average	peak	average	peak
Web server	$0.707 \cdot \lambda_{\text{workload}} + 1.101$	$0.925 \cdot \lambda_{\text{workload}} + 1.488$	$0.144 \cdot \lambda_{\text{workload}} + 0.175$	$0.156 \cdot \lambda_{\text{workload}} + 0.297$
Database	$0.012 \cdot \lambda_{\text{workload}} + 0.875$	$0.061 \cdot \lambda_{\text{workload}} + 0.778$	$0.008 \cdot \lambda_{\text{workload}} + 0.063$	$0.009 \cdot \lambda_{\text{workload}} + 0.218$
`Bid`	$0.456 \cdot \lambda_{\text{workload}} + 0.594$	$0.752 \cdot \lambda_{\text{workload}} + 0.967$	$0.074 \cdot \lambda_{\text{workload}} + 0.053$	$0.100 \cdot \lambda_{\text{workload}} + 0.216$
`BuyNow`	$0.006 \cdot \lambda_{\text{workload}} + 0.494$	$0.033 \cdot \lambda_{\text{workload}} + 0.775$	$0.000 \cdot \lambda_{\text{workload}} + 0.049$	$0.000 \cdot \lambda_{\text{workload}} + 0.213$
`Category`	$0.000 \cdot \lambda_{\text{workload}} + 0.912$	$0.000 \cdot \lambda_{\text{workload}} + 1.093$	$0.000 \cdot \lambda_{\text{workload}} + 0.072$	$0.000 \cdot \lambda_{\text{workload}} + 0.217$
`Comment`	$0.004 \cdot \lambda_{\text{workload}} + 0.817$	$0.000 \cdot \lambda_{\text{workload}} + 0.898$	$0.000 \cdot \lambda_{\text{workload}} + 0.049$	$0.000 \cdot \lambda_{\text{workload}} + 0.214$
`Item`	$0.306 \cdot \lambda_{\text{workload}} + 1.605$	$0.528 \cdot \lambda_{\text{workload}} + 1.949$	$0.074 \cdot \lambda_{\text{workload}} + 0.151$	$0.088 \cdot \lambda_{\text{workload}} + 0.259$
`Query`	$0.000 \cdot \lambda_{\text{workload}} + 0.898$	$0.000 \cdot \lambda_{\text{workload}} + 0.987$	$0.000 \cdot \lambda_{\text{workload}} + 0.055$	$0.000 \cdot \lambda_{\text{workload}} + 0.214$
`Region`	$0.104 \cdot \lambda_{\text{workload}} + 1.023$	$0.266 \cdot \lambda_{\text{workload}} + 1.065$	$0.041 \cdot \lambda_{\text{workload}} + 0.053$	$0.052 \cdot \lambda_{\text{workload}} + 0.219$
`User`	$0.091 \cdot \lambda_{\text{workload}} + 1.040$	$0.221 \cdot \lambda_{\text{workload}} + 1.130$	$0.036 \cdot \lambda_{\text{workload}} + 0.045$	$0.046 \cdot \lambda_{\text{workload}} + 0.206$
`UserTransaction`	$0.000 \cdot \lambda_{\text{workload}} + 0.803$	$0.000 \cdot \lambda_{\text{workload}} + 0.903$	$0.000 \cdot \lambda_{\text{workload}} + 0.048$	$0.000 \cdot \lambda_{\text{workload}} + 0.213$

Table 1: Component resource profiles for RUBiS (based on linear fitting of measured resource usage at 11 input request rates). $\lambda_{\text{workload}}$ is the average request arrival rate (in requests/second).

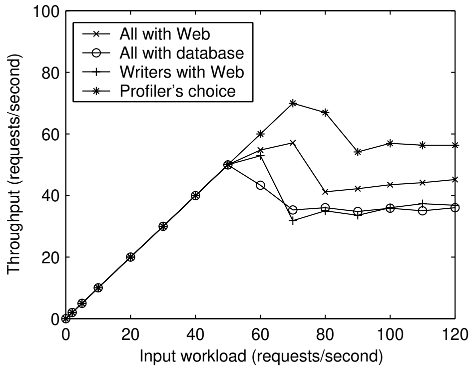

Figure 4: Performance of different component placement strategies for RUBiS.

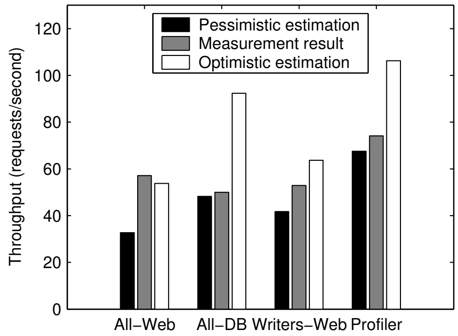

Figure 5: Accuracy of throughput estimations.

second intervals.

After acquiring the component resource consumption at the measured input rates, we derive general functional mappings using linear fitting. Figures 2 and 3 show such a derivation for the `Bid` component and the Web server respectively. The complete profiling results for all 11 RUBiS components are listed in Table 1. We do not show the memory profiling results because we are not able to precisely measure the component memory consumption using SYSSTAT. This does not affect the component placement decisions for this experiment since the server memory is not the bottleneck resource under any placement.

3.2 High Throughput Component Placement

We examine the component placement problem of RUBiS on two servers. We place a restriction on collocation of the Web server and the database — they are never collocated on one server. Since each of the remaining nine components can be placed either on the Web server or on the database server, there are 512 possible component configurations in this setup. Following the scheme described in Section 2.2, we project the optimistic and pessimistic system throughput for each of the 512 configurations. We choose the placement with the best pessimistic performance (called *profiler's choice*). In this placement, `Query`, `Region`, and `User` are collocated with the Web server while the other EJB components are collocated with the database. We compare this placement with three other strategies based on common heuristics. Specifically, the first two strategies place all EJB components with the Web server or with the database, respectively. In the third strategy, we place all read-write components (*i.e.*, `Bid`, `BuyNow`, `Comment`, `Item`, and `UserTransaction`) with the Web server and the read-only components with the database. We call this placement *writers with Web*. The intuition behind this strategy is that read-only components tend to interact more with the database.

Figure 4 shows the RUBiS throughput under different levels of input workload. In our experiments, a request is counted as successful only if it returns within 8 seconds. We define the throughput of a placement as the highest throughput achieved at any input request rate. The results show that *profiler's choice* outperforms all other placement strategies by over 30%. Performance degrades slightly after the maximum throughput is attained. This is because some requests exceed the timeout limit of 8 seconds despite being partially completed.

Figure 5 illustrates the accuracy of pessimistic and optimistic throughput estimations for the four placement strategies. On average, the pessimistic estimation is 39% smaller than the optimistic estimation. We observe that the measurement results almost always fall between the two estimations. We believe component profiles constructed using more fine-grain resource usage measurements (*e.g.*, LTT [22]) should improve the accuracy of the throughput estimations. Taking consideration of other factors, such as component context switch and remote invocation overhead, may also improve accuracy. We plan to investigate these issues in the future.

4 Conclusion

This paper describes our work on profile-driven component placement for cluster-based online services. With the assistance of offline derived component profiles parameterized using the input request rate, our placement framework can identify high throughput component placements on commodity clusters. Our preliminary experimental studies with RUBiS, a J2EE-based online auction benchmark, suggest that profile-driven component placement can indeed achieve improved throughput. Future work will proceed on two principal fronts: 1) heuristic placement for heterogeneous systems with large numbers of components and nodes, and 2) runtime workload monitoring and dynamic online component migration to handle changing input workload characteristics.

Acknowledgment: We would like to thank members of the URCS systems group and the anonymous referees for their 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 (Lake George), NY, October 2003.

[2] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic Function Placement for Data-Intensive Cluster Computing. In *Proc. of the USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[3] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and Implementation of Dynamic Web Site Benchmarks. In *Proc. of the 5th IEEE Workshop on Workload Characterization*, Austin, TX, November 2002.

[4] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-Based Remote Execution for Mobile Computing. In *Proc. of the 1st USENIX Intl. Conf. on Mobile Systems, Applications, and Services*, San Francisco, CA, May 2003.

[5] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *Proc. of the 17th ACM Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 246–261, Seattle, WA, November 2002.

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

[7] 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.

[8] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th ACM Symp. on Operating System Principles*, pages 78–91, Saint Malo, France, October 1997.

[9] X. Gu and K. Nahrstedt. Dynamic QoS-Aware Multimedia Service Configuration in Ubiquitous Computing Environments. In *Proc. of the 22nd Intl. Conf. on Distributed Computing Systems*, pages 311–318, Vienna, Austria, July 2002.

[10] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[11] A.-A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments. In *Proc. of the 11th IEEE Symp. on High Performance Distributed Computing*, pages 103–112, Edinburgh, Scotland, July 2002.

[12] D. Narayanan and M. Satyanarayanan. Predictive Resource Management for Wearable Computing. In *Proc. of the 1st USENIX Intl. Conf. on Mobile Systems, Applications, and Services*, San Francisco, CA, May 2003.

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

[14] 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.

[15] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, pages 202–216, Charleston, SC, December 1999.

[16] J. P. Sousa and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Proc. of the 3rd Working IEEE/IFIP Conf. on Software Architecture*, pages 29–43, Montreal, Canada, August 2002.

[17] R. J. Stets, Galen C. Hunt, and Michael L. Scott. Component-Based APIs for Versioning and Distributed Applications. *IEEE Computer*, 32(7):54–61, July 1999.

[18] SYSSTAT Utilities. http://perso.wanadoo.fr/sebastien.godard.

[19] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *Proc. of the 12nd European Conf. on Object-Oriented Programming*, Budapest, Hungary, June 2001.

[20] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proc. of the 13th European Conf. on Object-Oriented Programming*, Malaga, Spain, June 2002.

[21] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, pages 239–254, Boston, MA, December 2002.

[22] K. Yaghmour and M. R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proc. of the USENIX Annual Technical Conf.*, San Diego, CA, June 2000.