A Comprehensive Resource Management Solution for Web-based Systems

Filippo Seracini, Massimiliano Menarini, and Ingolf Krüger, ***University of California, San Diego;*** **Luciano Baresi, Sam Guinea, and Giovanni Quattrocchi,** ***Politecnico di Milano***

https://www.usenix.org/conference/icac14/technical-sessions/presentation/seracini

This paper is included in the Proceedings of the 11th International Conference on Autonomic Computing (ICAC '14).

June 18–20, 2014 • Philadelphia, PA

ISBN 978-1-931971-11-9

Open access to the Proceedings of the 11th International Conference on Autonomic Computing (ICAC '14) is sponsored by USENIX.

A Comprehensive Resource Management Solution for Web-based Systems

F. Seracini, M. Menarini, and I. Krüger
Dep. of Computer Science and Engineering
UC San Diego - La Jolla (CA), 92093 USA
fseracini@ucsd.edu

L. Baresi, S. Guinea, and G. Quattrocchi
Dip. di Elettronica, Informazione e Bioingegneria
Politecnico di Milano - Milano, 20148 Italy
luciano.baresi@polimi.it

Abstract

This paper presents an autonomic resource management solution that looks at the non-functional qualities of a Web-based system, as well as at the characteristics of the infrastructural resources it uses. It exploits a detailed performance model of both these aspects to increase the efficiency of resource allocation. The solution is evaluated on an auction and shopping benchmark web site, and compared to a baseline approach and to an existing solution from literature. Results show that, by jointly taking into account the different software and hardware facets of our application, we can reduce the amount of resources allocated by up to 42.5% compared with an existing work from the literature.

1 Introduction

Modern software systems are subject to continuous change, such as changes in the stakeholder requirements, evolutions in the software or resource components, and variations in the execution context [4]. System designers need refined runtime management techniques and tools to cope with these changes, so that the systems can continue to provide the required functional and non-functional qualities.

In this paper we focus on complex Web-based systems. In these cases, change often manifests itself as significant fluctuations in the system's workload. Although they may often follow historical and seasonal patterns, multiple spikes may occur without warning (e.g., flash crowds [2]). These situations can lead to frustrated customers, and loss of online business. As a consequence, service providers tend to over-allocate resources. The result is that the average server utilization, in a typical data center, is around 30-40% [5]; some studies even estimate the utilization level as low as 18% [20]. A better utilization of resources would lead to more efficient systems. Fortunately, the diffusion of dynamic resource allocation techniques provide the flexibility needed to build systems that can adapt their configurations based on the actual workloads and acquire resources on demand.

These systems must embed fine-grained *autonomic* capabilities that cover all their facets, from their hardware resources to their software. By introducing a MAPE (Monitor, Analyze, Plan, and Execute) control loop [11] that can *estimate* the application's load and performance, and help understand how many resources should be provisioned, we can compute and apply anticipatory or corrective actions on the fly.

Many different performance models have been proposed in the past. Unfortunately, in their load estimation they tend to only take into account the volume of requests (e.g. [22, 23]). Regrettably, it has been shown that this is not enough to estimate an application's resource usage effectively [19, 21]. Some systems take a further step and claim to be workload-mix aware, meaning that they distinguish between different types of requests, based on their aggregated response times. Yet, these approaches still do not consider the hardware resource usage that these requests imply, and the resource contention that can arise. Resource contention is one of the main causes of performance degradation in systems with high loads [9, 17, 12, 16]. Knowing the resource usage profile of each of the different types of requests is vital in defining an effective allocation strategy.

Our approach provides a comprehensive, innovative solution for the *autonomic* management of complex Web-based applications. We exploit and suitably extend ECoWare [3], a monitoring and adaptation framework previously developed at Politecnico di Milano for service-based systems. Thanks to these extensions we are now able to perform fine-grained measurements of a Web-based system's behavior, by correlating the runtime data that we retrieve from its hardware and its software. To achieve this we included in ECoWare a completely new performance model that takes into account both the application's workload mix and the hardware infrastruc-

ture. It defines the resource footprint of the different application requests in terms of CPU instructions, cache, and main memory accesses. These metrics are then used to profile the resource usage of the requests, and to increase the accuracy of the performance predictions.

1.1 Running Example

To evaluate our approach, we used RUBiS [1], a well-known benchmark that simulates a common auction and shopping web site, and compared our results against a baseline approach and an existing solution from literature [19]. RUBiS was developed using HTML, Java Servlets, and SQL technology. It uses an Apache server for its static content (e.g., the home page, images, etc.), multiple JBoss Application Servers for its dynamic content (e.g., the user's bidding history, the products available in a given geographical region, etc.), and a MySQL Server for its backend data tier. A content-aware load balancer routes the requests accordingly.

In this paper, we focus on how we can use ECoWare to satisfy an SLA of the kind "the X^{th} percentile response time should be within a fixed threshold over a period of time". The percentile response time was calculated taking into account the response times seen by ECoWare over a two and a half minute window.

2 Approach Overview

The overarching goal of our ECoWare project is to empower system designers and maintainers in the development of self-adaptive systems. ECoWare is a general-purpose monitoring and adaptation framework that exploits the common *MAPE (Monitoring – Analysis – Planning – Execution) control loop* approach. System designers can tailor ECoWare to their needs by choosing appropriate technology- and application-specific sensors and actuators, and by choosing appropriate analysis and planning techniques.

Figure 1 shows how ECoWare can be applied to a complex Web-based system. The application uses multiple servers for each tier, and new servers can be added from a *Pool of Free Servers*. The application is on the left-hand side of the figure, and it interacts with ECoWare, which is on the right-hand side of the figure, through an Event Bus. Light gray components represent previous work, while white components are novel contributions of this paper.

For the **Monitoring** step, ECoWare provides two kinds of components: *Key Performance Indicator (KPI) Processors* and *Aggregators*. The former use the low-level events produced by the probes to calculate various non-functional KPIs, such as average response times, arrival rates, and throughputs. The latter correlate multiple

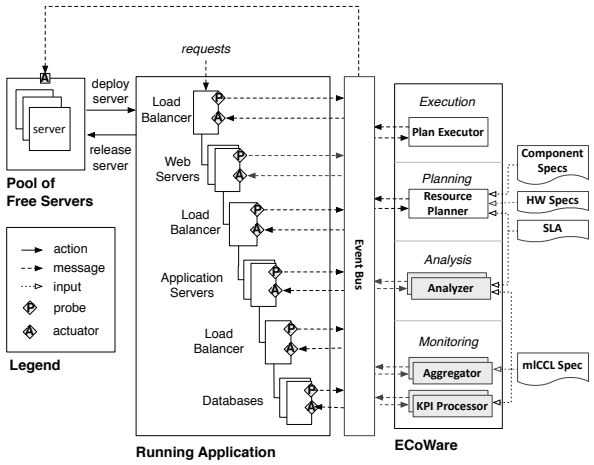

Figure 1: Example of an application with ECoWare.

low-level events and/or KPIs to produce a holistic understanding of the application's behavior. For the **Analysis** step, ECoWare's *Analyzers* listen to the bus for monitoring events (e.g., average response times) and evaluate them against certain constraints. For the **Planning** step, ECoWare provides a *Resource Planner*. This component uses a workload-mix and hardware-aware performance model to understand what changes should be made to the resource provisioning. Finally, for the **Execution** step, ECoWare provides the *Plan Executor*. This component coordinates multiple actuators to accomplish the desired adaptations.

ECoWare is extremely flexible; it can support multiple kinds of management strategies, e.g., *reactive*, *proactive*, and *periodic*. We can even mix different strategies together, to differentiate how we up-scale (e.g., reactively) from how we down-scale (e.g., periodically) our resources. This flexibility allows us to be fast to provision new resources, yet cautious when un-provisioning them; and it allows us to reduce instability.

3 The Resource Planner

The Resource Planner is responsible for identifying the correct amount of resources to add or to remove from the running system. In order to do this, it implements the *Compute Configuration Algorithm*, shown in Algorithm 1. Every decision that the algorithm makes is checked against a hardware- and workload-mix aware *Performance Model*. This allows us to avoid a trial-and-error approach in which we deploy (or remove) hardware, and then wait for the subsequent iteration of the control loop to tell us if it solved the problem.

Algorithm 1: Compute Configuration Algorithm

Input: $\overline{\lambda} = (\lambda_1, \lambda_2, ..., \lambda_n)$, R_{SLA}, and $SysConf_{Current}$
Output: $SysConf_{New}$

```
if freeServerPool = ∅ then
    return SysConf_Current
else
    select server S ∈ freeServerPool
    SysConf_New = SysConf_Current ∪ S
    R_New = AnalyzePerf(λ, SysConf_New)
    if R_New satisfies R_SLA then
        return SysConf_New
    else
        return ComputeConfiguration(λ, R_SLA, SysConf_New)
    end
end
```

3.1 Compute Configuration Algorithm

Every time the Resource Planner is notified with a SLA violation, it invokes the algorithm illustrated in Algorithm 1. The algorithm takes the vector of arrival rates per request type ($\overline{\lambda}$), the response time value defined in the SLA (R_{SLA}), and the current system configuration ($SysConf_{Current}$), and determines a new system configuration ($SysConf_{New}$) that satisfies the SLA. $\overline{\lambda}$ is provided by ECoWare's monitoring.

The algorithm takes a server S from the pool of free servers ($freeServerPool$) and adds it to the current system configuration to create $SysConf_{New}$. It then verifies the new system configuration by running *AnalyzePerf*, which solves a queuing network (QN) model, i.e., the Performance Model, for $SysConf_{New}$ and the actual workload $\overline{\lambda}$. *AnalyzePerf* returns a vector of response times $\overline{R_{New}}$. If all the values of $\overline{R_{New}}$ are smaller than R_{SLA}, the algorithm terminates returning $SysConf_{New}$; otherwise, the algorithm recursively calls itself with parameters $\overline{\lambda}$, $SysConf_{New}$, and R_{SLA}. In the worst case scenario, i.e., in which it continuously fails to satisfy the SLA, the algorithm continues until all the available resources (i.e. servers) are provisioned.

The autonomic system will also periodically check its provisioning to see whether resources can be safely removed. In this case, the Resource Planner invokes Algorithm 1 passing a base configuration of one server as $SysConf_{Current}$. The algorithm recursively adds servers till the queuing network (QN) solver validates the configuration. By doing so, ECoWare guarantees that the minimal set of servers is always allocated.

3.2 Performance Model

Algorithm 1 leverages a QN model to estimate the application's response times per request type, given a specific workload and a specific system configuration, allowing ECoWare to compare the estimated values with those required by the SLA.

In a QN model (e.g., [7]), each component is represented as a queue, called a service station. A model can be closed, open, or mixed, depending on whether the volume of requests is constant, fluctuating, or mixed. Whenever the number of incoming requests is higher than the computing capacity of a service station, the requests are queued; the time spent in queue is called waiting time. Each type of service request is defined by an arrival rate process (i.e., how the arriving requests are distributed in a unit of time) and a service demand distribution (overall time that a single request spends at each of the service stations during a complete execution). This can be expressed using Kendall's notation in the form $A/S/C - POLICY$ [7]. A describes the arrival process, S describes the distribution of service time of a job, C describes the number of servers at the node, and $POLICY$ describes the queuing discipline used at that node (e.g., first come first served, processor sharing, etc.). In Kendall's notation, M stands for Markov or memoryless, meaning that arrivals occur according to a Poisson process; and G stands for general, meaning an arbitrary probability distribution.

As the arriving traffic of the different types of requests fluctuates over time, so does the mix of requests in execution. As a consequence, resource contention changes over time. Figure 2 shows the performance model of our running RUBiS example. To simplify the presentation, the figure only shows one server (with two cores) for the application tier. The model correctly captures the following three aspects: the fact that the approach is workload mix-aware, the fact that the system is composed of multiple tiers, and the fact that the application tier relies on specific hardware resources.

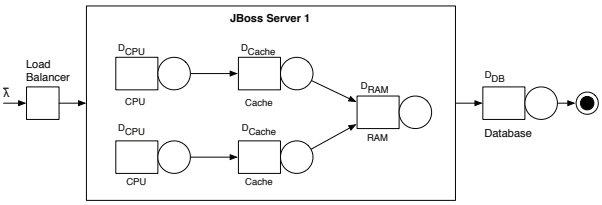

Figure 2: Performance Model in our experiments.

The intensity of the arriving flow of requests in input to the queuing network, that is the workload-mix, is described by $\overline{\lambda}$, the vector of arrival rates per request type. Regarding the fact that we have multiple tiers, the model concentrates on the application tier (JBoss) and on

the database tier (MySQL), and leaves out the web tier (Apache). The reason for this is that, in RUBiS, system performance is dominated by the application tier. The web tier only serves static content and has a negligible influence on the performance of the overall system. The database also shows a very moderate load, but we could not remove it entirely from the model since the JBoss servers query it. However, given its low utilization and resource contention, the role of the database was approximated by a single queue in the performance model.

In the application tier, i.e., in the dominating tier, we used a very fine-grained approach. We used multiple M/G/1-FCFS stations for the main hardware components of the application servers i.e., CPU, cache and main memory. (These were configured using data taken from our hardware probes through ECoWare monitoring.) This means that not only do we consider the traffic of incoming requests and their mix, we also consider the resource usage footprint of each request. On the other hand, we did not need this level of detail neither for the database tier, nor for the incoming load balancer. Therefore, we used a single M/G/1-FCFS station in both cases, and configured them using information collected by the Hibernate and Load Balancer probes, respectively. The results of our experiments, which we will discuss in the following section, show that these simplifications are correct.

By modeling the service demands for the resources, our model can estimate the waiting time of a request at each service station. Whenever a resource becomes congested, the waiting time at the corresponding service station grows. Since each type of request has a different response time and a different resource usage profile, different workload mixes (i.e., requests mixes) will use the hardware resources in a different way. As a result, the overall execution time is both mix- and volume-dependent.

Let us use a simple example to further exemplify our solution. Let us assume we have two types of requests, A and B, and that request type A is CPU-intensive and request type B is memory-intensive. If we mostly receive requests of type B, their total execution time, and the overall system's throughput, will suffer. The reason for this is that the cache will start being used by a high number of threads, and start becoming congested. When this happens, RAM will be accessed more frequently, resulting in longer waiting and execution times. A more balanced load between requests A and B may cause less contention, and the overall system's throughput would benefit.

Table 1 shows the increase in terms of response time, and number of cache and main memory accesses, for a memory intensive workload. Two different hardware architectures are compared: although they have the same CPU frequency and number of cores, HW_1 has twice the cache size per core than HW_2. The table clearly shows that the architecture with the larger cache size experiences much smaller performance degradation. Moreover, at high load levels (about 80% CPU utilization), the system with the smaller cache shows a 49% increase in time spent to access the main memory because of the increased cache conflicts. This demonstrates that reducing resource contention between running tasks is an effective way to improve a system's throughput, without sacrificing response times (this was also shown in [9, 8, 17]), and that explicitly modeling both the hardware and the resource usage profile of the requests can be an effective way to take resource contention into account, and to improve the accuracy of our performance predictions.

Table 1: Increase of response time (RT), cache accesses (CA), and memory accesses (MA) from low to high load levels for different hardware architectures.

HW Conf.	RT	CA	MA
HW_1	6%	$\sim 0\%$	19%
HW_2	26%	3%	49%

4 Experimental Evaluation

For the evaluation of this work, we focused on assessing ECoWare's frugality in allocating resources while satisfying the SLA. We adopted the queuing network model already illustrated in Figure 2. To create and solve the model we used the Java Modeling Tool JMT [6]. For our experiments we used seven servers equipped with Intel Xeon processors running at 2.66 GHz. Each processor was composed of two cores, each addressing a separate 6MB L2 cache; main memory was 32 GB on each machine.

During our initial evaluation of RUBiS, we loaded the system to measure the servlets' resource usage profiles. These were then used to calibrate the queuing network performance model. We focused on two servlets with fixed queries: `ViewUserInfo` and `ViewBidHistory`. Since the servlets' average execution times were very close, and often below 1ms, we decided to increase the load for servlet `ViewBidHistory`, to make it more representative of a real-case scenario. We also added a routine to `ViewUserInfo` that validates the content of the comments that are left about a given user (as typically done in many social web sites).

We evaluated the accuracy of ECoWare's provisioning strategy with multiple non-stationary workloads. Since provisioning decisions are applied independently at each tier, we focused on the application tier and over-provisioned the others to ensure that they did not represent bottlenecks.

To validate our approach, we compared it with a baseline solution that uses an M/M/K-FCFS queue as its performance predictor, and with the approach presented by Singh et al. in [19], where they use a closed formula approximation of a G/G/1 queue to predict server capacity for a given workload mix[1].

The first workload was designed to follow the structure of the workload used in [19]: it has a varying mix, yet a constant volume of requests. Our results showed that ECoWare used 13.75% fewer server-minutes (i.e., number of servers per allocation time) than the baseline and 42.5% less than [19] [2]. The second workload had both a varying workload mix and a varying volume of requests. The workload was executed several times and showed consistent results. For the sake of simplicity, we show here a randomly selected execution.

The Workload Figure 3 shows the workload mixes and the different time lengths of the experiment's steps. The maximum total number of requests per second at each step was limited to 22.5 in order to generate a workload that could be handled by the amount of servers at our disposal. The same randomly generated workload was used for all three solutions.

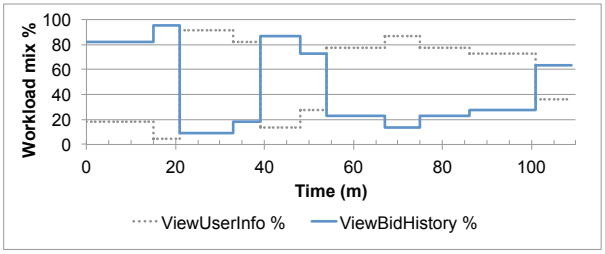

Figure 3: Request mix.

Server Provisioning and SLA. The baseline approach shows a strong instability throughout the entire workload. Figure 4 describes the provisioning decisions for the workload. The baseline approach allocates two or more servers every time the system experiences a transitory phase. These phases typically happen right after a steep variation in the workload and last for 1 or 2 minutes. After the transient phase is over, the baseline starts reducing the number of allocated servers, often incurring in SLA violations —see Figure 5. So the baseline approach has a tendency to under-allocate, and consequently experiences lots of SLA violations.

The solution from [19] exceeds the SLA limit twice at the beginning of the workload. The reason for that is the very high load that was selected by the random generator at the very beginning. Since we always begin with one allocated server, which obviously cannot satisfy the demand at the workload's first step, the two SLA violations are unavoidable and cannot be attributed to an erroneous provisioning. Indeed, the provisioning strategy completes the workload with no further violations, confirming its soundness. In terms of allocated servers, the solution from [19] appears to be more generous, requesting up to four servers for extended periods of time. Even though only three servers are actually allocated even when four are requested, the 95% percentile of the response time remains steadily well below the limit. An almost flat line for the response time throughout the workload shows that the servers are running at a very low utilization.

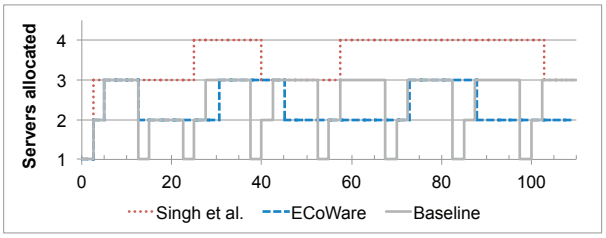

Figure 4: Number of servers allocated.

ECoWare experiences the same SLA violations at the beginning of the experiment as the previous two approaches. Similarly to [19], it does not incur in any further violation for the remaining of the workload. ECoWare correctly adapts its provisioning decisions throughout the experiment, allocating up to three servers in three different moments. ECoWare's diagram in Figure 5 shows wide fluctuations in the response time, proving that the servers have a much higher utilization, hence a longer execution and waiting time for the requests. However, ECoWare is able to maintain those fluctuations under the SLA limit, thus improving the overall utilization of the allocated servers.

Result. For this experiment, the baseline approach scored 302.5 server-minutes, but incurred in a significant amount of violations because of its tendency to under-allocate. The provisioning strategy from [19] used 385 server-minutes, but with a perfect response time, except for the two initial violations that could not be avoided. Finally, ECoWare only allocated 255 server-minutes, with no SLA violations except for the usual two at the beginning. ECoWare also showed a higher fluctuation in the response time, testifying a higher utilization of the servers. Overall, ECoWare used 16% fewer server-minutes than the baseline approach with less violations, and 33% less than [19], with the same amount of violations.

[1] In order to perform our comparisons we implemented Singh et al.'s approach according to the formulae presented in their paper.

[2] Due to lack of space, we cannot focus on this workload; a detailed presentation can be found online at `http://home.deib.polimi.it/guinea/ICAC2014/experiments.pdf`.

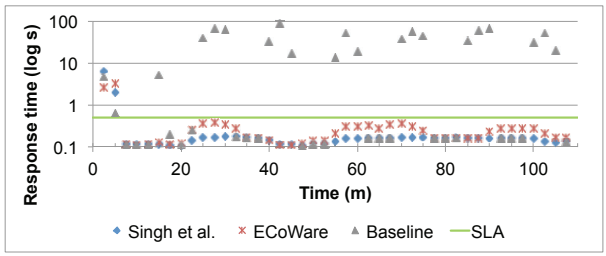

Figure 5: 95% Percentile response time.

4.1 Threats to Validity

A possible objection to our evaluation is that we only focus on two servlets. We consider this to not be an over-simplification (e.g. [19] uses three servlets). It is quite common, when there are many different kinds of requests, to use K-means clustering to group them into fewer clusters with similar service demands [19]. Clustering also allows us to mitigate a limit of queuing networks, i.e., that they do not scale well when the number of classes of jobs significantly increases.

Table 2: Average solving time for the QN model for different numbers of classes.

# of classes	2	3	5	7	10
Solving time (s)	3	5	9	15	18

To evaluate the scalability of our approach we benchmarked the QN solver with up to 10 classes (see Table 2), a realistic top limit for the number of clustered request types. The table shows that the solver still terminates within a reasonable time frame. We also bound the computational time to 20 seconds, at the cost of a possible loss of accuracy. This does not impact our results, since it takes the solver longer to terminate when the infrastructure is close to saturation. When this happens the response time will violate any given SLA, since it will grow unboundedly. Therefore, ECoWare's strategy saves time without losing any accuracy, by adding one server and re-running the solver.

5 Related Work

Most existing approaches for achieving dynamic resource provisioning are mix un-aware; they only consider the volume of arriving requests when determining their provisioning. In their QN model Almeida et al. [2] use the peak session arrival rate. Villela et al. [23] leverage QNs to model application servers and to solve an optimization problem for profit maximization; however, they only consider the aggregate number of requests generated by the clients. Kusic et al. [13] present an optimization framework for resource allocation, expressed as a sequential decision making problem under uncertainty, and solved using a limited look ahead control scheme. The approach distinguishes between different classes of clients with different SLA limits, but does not consider the resource usage profile. He et al. [10] focus on dynamic allocation for interactive systems (e.g. web search engines) where the quality of the results depends also on the allocated time; they do distinguish between interactive and batch jobs. In [25] Zhou et al. present a two-tier resource management framework that focuses on optimizing resource allocation while provisioning proportionality fairness to clients. They consider different classes of clients, associated with different SLA agreements. SPIRE [15] is an autonomous system for service provisioning driven by a utility function: optimize the average earned revenue over time, while satisfying the SLA. SPIRE also assumes a uniform service time for all the requests. As additional formulation of the dynamic allocation problem we also mention the decentralized control theory approach of [24] and the machine learning technique proposed in [14].

As we have seen in our study, different types of requests can have very different service times and resource usage profiles. Among mix-aware approaches, we mention Sharma et al. [18]. They use a network of M/G/1-PS queues and an approximate model to compute response time distribution. Their solution takes hardware configuration into consideration. However, their approach differs from ours as they focus on dealing with a multitude of hardware configurations in terms of service rates. Our approach instead focuses on modeling the hardware to achieve a higher accuracy in the performance predictions. We plan to extend our approach to consider different hardware configurations as well. Works in [17, 16] also focus on mitigating performance degradation caused by shared resource contention. Finally, Krebs et al. [12] provides metrics to quantify performance isolation in the context of shared resources.

6 Conclusions and Future Work

Allocation strategies that take into account the resource usage profiles of the different requests help increases the accuracy of our performance predictions, which in turn allows us to improve the utilization of our infrastructure. In the future we will continue to evaluate ECoWare in the context of cloud technology, and evaluate the advantages that could derive from the use of layered queuing networks for our models.

7 Acknowledgments

We thank Giuseppe Serazzi, Marco Gribaudo, and William Griswold for their excellent feedback.

References

[1] RUBiS – Rice University Bidding System – `http://rubis.ow2.org/`.

[2] ALMEIDA, V., AND MENASCE, D. Capacity Planning an Essential Tool for Managing Web Services. *IT Professional 4* (Aug. 2002), 33–38.

[3] BARESI, L., AND GUINEA, S. Event-based Multi-level Service Monitoring. In *Proceedings of the 20th IEEE International Conference on Web Services (ICWS)* (2013), pp. 83–90.

[4] BARESI, L., NITTO, E. D., AND GHEZZI, C. Toward Open-World Software: Issue and Challenges. *Computer 39*, 10 (2006), 36–43.

[5] BARROSO, L. A., AND HÖLZLE, U. The Case for Energy-Proportional Computing. *Computer 40*, 12 (Dec. 2007), 33–37.

[6] BERTOLI, M., CASALE, G., AND SERAZZI, G. JMT: Performance Engineering Tools for System Modeling. *SIGMETRICS Performance Evaluation Review 36*, 4 (2009), 10–15.

[7] BOLCH, G., GREINER, S., MEER, H. D., AND TRIVEDI, K. S. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, 2nd edition ed. Wiley, 2006.

[8] CASALE, G., AND SERAZZI, G. Bottlenecks Identification in Multiclass Queueing Networks using Convex Polytopes. In *Proceedingds of the 12th IEEE Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)* (2004), pp. 223–230.

[9] DHIMAN, G., MARCHETTI, G., AND ROSING, T. vGreen: A System for Energy-Efficient Management of Virtual Machines. *ACM Transactions on Design Automation of Electronic Systems 16*, 1 (2010), 1–27.

[10] HE, Y., YE, Z., FU, Q., AND ELNIKETY, S. Budget-based Control for Interactive Services with Adaptive Execution. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)* (2012), pp. 105–114.

[11] KEPHART, J., AND CHESS, D. The Vision of Autonomic Computing. *Computer 36*, 1 (2003), 41–50.

[12] KREBS, R., MOMM, C., AND KOUNEV, S. Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)* (2012), pp. 91–100.

[13] KUSIC, D., AND KANDASAMY, N. Risk-Aware Limited Lookahead Control for Dynamic Resource Provisioning in Enterprise Computing Systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC)* (2006), pp. 74–83.

[14] LAMA, P., AND ZHOU, X. AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)* (2012), pp. 63–72.

[15] MAZZUCCO, M. Towards Autonomic Service Provisioning Systems. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)* (2010), pp. 273–282.

[16] MENASCE, D. Two-level Iterative Queuing Modeling of Software Contention. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)* (2002), pp. 267–276.

[17] ROYTMAN, A., KANSAL, A., GOVINDAN, S., LIU, J., AND NATH, S. PACMan: Performance Aware Virtual Machine Consolidation. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)* (2013), pp. 83–94.

[18] SHARMA, U., SHENOY, P., AND TOWSLEY, D. F. Provisioning Multi-tier Cloud Applications using Statistical Bounds on Sojourn Time. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)* (2012), pp. 43–52.

[19] SINGH, R., SHARMA, U., CECCHET, E., AND SHENOY, P. Autonomic Mix-aware Provisioning for Non-stationary Data Center Workloads. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)* (2010), pp. 21–30.

[20] SNYDER, B. Server Virtualization has Stalled, Despite the Hype. *InfoWorld* (2010).

[21] SPICUGLIA, S., BJÖERKQVIST, M., CHEN, L. Y., SERAZZI, G., BINDER, W., AND SMIRNI, E. On Load Balancing: a Mix-aware Algorithm for Heterogeneous Systems. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)* (2013), pp. 71–76.

[22] URGAONKAR, B., AND CHANDRA, A. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC)* (2005), pp. 217–228.

[23] VILLELA, D., PRADHAN, P., AND RUBENSTEIN, D. Provisioning Servers in the Application Tier for e-commerce Systems. *ACM Transactions on Internet Technology 7*, 1 (2007).

[24] WANG, R., AND KANDASAMY, N. On the Design of Decentralized Control Architectures for Workload Consolidation in Large-scale Server Clusters. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)* (2012), pp. 115–124.

[25] ZHOU, X., AND IPPOLITI, D. Resource Allocation Optimization for Quantitative Service Differentiation on Server Clusters. *Journal of Parallel and Distributed Computing 68*, 9 (2008), 1250–1262.