Dynamic Placement for Clustered Web Applications

A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi
IBM T.J. Watson Research Center
{karve,kimbrel,giovanni,mspreitz,steinder,sviri,tantawi}@us.ibm.com

ABSTRACT

We introduce and evaluate a middleware clustering technology capable of allocating resources to web applications through dynamic application instance placement. We define application instance placement as the problem of placing application instances on a given set of server machines to adjust the amount of resources available to applications in response to varying resource demands of application clusters. The objective is to maximize the amount of demand that may be satisfied using a configured placement. To limit the disturbance to the system caused by starting and stopping application instances, the placement algorithm attempts to minimize the number of placement changes. It also strives to keep resource utilization balanced across all server machines. Two types of resources are managed, one load-dependent and one load-independent. When putting the chosen placement in effect our controller schedules placement changes in a manner that limits the disruption to the system.

Categories and Subject Descriptors

K.6.4 [**Computing Milieux**]: Management of Computing and Information Systems—*System Management*

General Terms

Algorithms, Management, Performance

Keywords

Dynamic application placement, performance management

1. INTRODUCTION

Many organizations rely on web applications to deliver critical services to their customers and partners. These service providers require application middleware platforms capable of dynamically allocating resources to meet the performance goals of these web applications. Today, middleware platforms for web applications already provide functions like monitoring, access control, and interoperability. Some of the middleware platforms also provide clustering functionality. Middleware clustering technology integrates multiple instances of the same application running on multiple server machines so they appear to be a single virtual application. Middleware platforms use clustering technology to provide scalability, high availability and load balancing.

Copyright is held by the International World Wide Web Conference Committee (IWC2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW2006, May 22–26, 2006, Edinburgh, UK.
ACM 1-59593-323-9/06/0005.

To change the amount of computing resources allocated to clustered web applications requires a middleware with the ability of dynamically changing the size of each application cluster as well as deciding which server machine each application instance of each cluster will use. We assume that the middleware hosts multiple clustered applications. Because of resource limitation, a given server machine cannot run instances for all application clusters at the same time. There may also be hardware or other limitations that make a machine unsuitable for running some application clusters.

In this paper, we propose the design of and evaluate a middleware clustering technology capable of dynamically allocating resources to web applications through dynamic *application instance placement*. We define application instance placement as the problem of placing application instances on a given set of server machines to satisfy the resource demand of each application cluster. Our middleware uses a *placement controller* mechanisms that dynamically configures the size and placement of application instances while obeying user-supplied constraints.

The controller places applications based on the available server machine resources and the application demands. Each server machine has two resources, one *load-dependent* and one *load-independent*. The usage of the *load-dependent* resource depends primarily on the intensity of application load. The usage of the *load-independent* resource does not strongly depend on the current intensity of the application workload. Correspondingly, an application has a load-dependent and a load-independent demand. In the case of web applications, typical examples for load-dependent and load-independent resources are CPU and memory, respectively.

The problem of application placement has been studied before [1, 2, 3, 4]. In [5] we have introduced a placement algorithm that improves on the prior art in several aspects. (1) It allows multiple types of resources to be managed, (2) it aims at maximizing satisfied application demand while minimizing placement changes compared to the previous placement, and (3) it is efficient enough to be applicable in online resource allocation. Although some of prior techniques also consider these factors, none of these techniques addresses all factors simultaneously.

In this paper, we extend our prior work in two ways. First, we modify the algorithm such that it produces application placement that allows application load to be better balanced across server machines. To the best of our knowledge this objective has not been investigated in prior solutions to the placement problem. However, placements that allow load to be balanced across servers allow applications to perform better.

Second, we investigate two variants of the algorithm that improve its effectiveness in maximizing the amount of satisfied demand and minimizing the placement churn.

The paper is organized as follows. In Section 2 we introduce

the architecture of a workload management system in which our controller is implemented. In Section 3 we formulate and present the algorithm for the application placement problem. In Section 4 we evaluate placement techniques adopted by the controller. In Section 5 we compare our approach with prior work on this subject.

2. SYSTEM DESCRIPTION

We have implemented the placement controller presented in this paper as a part of middleware technology for managing the performance of web applications [6, 7].

Web applications handle web requests and generate static or dynamic web content. In the case of J2EE, runtime components of web applications include HTTP servers, servlet engines, Enterprise Java Beans (EJB) containers and databases. Web administrators deploy these runtime components on several or more server machines, forming a distributed computing environment. Web application middleware presented in this paper manages applications running on this distributed computing environment and allocates resources to them in real-time to meet their demand.

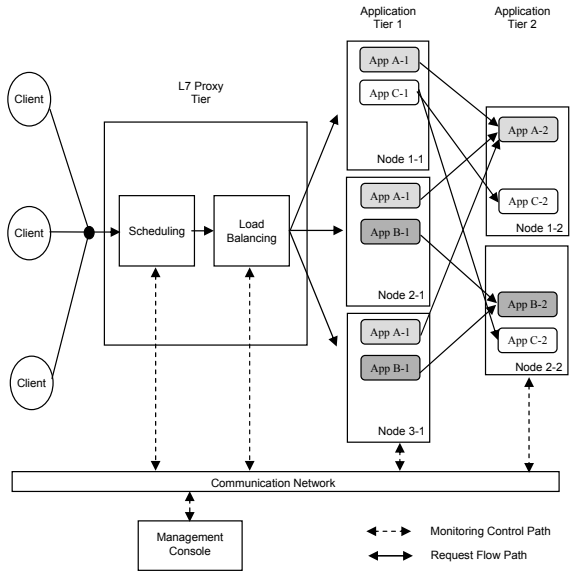

Figure 1: Management system architecture in a sample Web infrastructure.

Figure 1 illustrates the typical architecture of web infrastructure. We show three applications A, B and C deployed across two tiers (Application Tier 1 and Application Tier 2). For example, in the case of a J2EE application the first tier hosts servlets and EJBs while the second tier hosts the database. In this example, the first tier uses three different server machines while the second uses two server machines. In this paper, we use the term *node* when referring to a server machine. In a given tier, a particular application may run on multiple nodes. We refer to the presence of an application on a node as an application *instance* and we use the term *application cluster* to indicate the set of all instances of the same application. By application we mean a set of software entities that form a logical web application unit. Thus, an application may involve multiple servlets, multiple EJB modules, etc.

In our experience, a large enterprise web datacenter hosts many different web applications deployed in clusters with a few tens to several hundred application instances spread across a few to a hundred nodes. In almost every case datacenter nodes do not have enough capacity to have all deployed applications be running on the node at the same time (in the case of J2EE applications memory is the bottleneck, with each application instance requiring on average $1 - 2GB$ of real memory to handle requests). Therefore web administrator partitions applications across the available nodes. In the example of Figure 1 we have the first tier hosting three applications with each node capable of hosting at most two instances of each application. We use a *placement matrix* to show where application instances run. Each element $I_{m,n}$ of this matrix represents the number of instances of application m running on node n.

Web-application middleware environments typically use layer 7 routers to deliver web requests to applications. The routers map web requests to web applications and send the web requests to a specific instance of the target web application. The layer 7 routers use load balancing techniques and affinity rules to select target instances. The router may also implement admission and flow control. A router equipped with this functionality can control the amount of backend resources used by the different types of web requests. However, its ability to increase the amount of resources used by a given type of web requests is limited by the number of application instances targeted by this type of web requests as well as by the capacity of the nodes where the application instances run. For example, the placement presented in Fig. 1 allows the request router to handle a big amount of web request load for application A but much smaller load for applications B and C. When application C experiences a spike of web requests while the workload of application A decreases, the web requests destined for application C cannot use more capacity in Tier 1 than that of the node on which application C is running. Depending on the control logic of the request router, application C will experience either high service time in Tier 1, long waiting time in the routing layer, or high request rejection rate. At the same time, spare capacity in the system will remain on the remaining nodes in Tier 1.

To adapt to such workload changes, we must modify the application placement dynamically in reaction to the changing load. In the considered scenario, a desired system response to the workload shift would be to start new instances of application C on nodes 2-1 and 3-1 and stop instances of application A on these nodes (to make room for the new instances). When we deal with a large number of applications with multiple resource requirements and a large number of nodes with heterogeneous capacities, deciding on the best application placement for a given workload is non-trivial.

In this paper we formulate the placement problem as an instance of a two-dimensional packing problem and we study several algorithms for solving it. We also describe a placement controller that runs on our middleware and dynamically changes the size of application clusters to respond to dynamic load variations.

2.1 Application placement architecture

An application instance may run on any node that matches application requirements. In J2EE domain, these properties concern features that cannot be changed using J2EE runtime environment, such as network connectivity, bandwidth, the presence of non-J2EE software libraries, etc. In our system, an application is deployed to all nodes that match its properties. Application deployment involves creating persistent configuration of an application server that hosts the application, which includes the definition and configuration of J2EE resources and parameters. We configure application placement by starting and stopping the application servers as needed. When an application instance is stopped on a node it does not consume any of the node's runtime resources.

Using server start and stop operations to control the number and placement of application instances provides a relatively light-weight resource allocation mechanism because it does not require a potentially time-consuming application deployment, server configuration, distribution of configuration files, etc. A possible disadvantage of such an approach is its somewhat limited flexibility as it does not allow arbitrary combinations of applications to be deployed and executed on application servers.

The controller described in this paper requires the managed system to provide certain functionality. When this functionality is present, the controller is also suitable for the management of other than Web applications. Similarly, it could use other control mechanisms than application instance starts and stops. For example, in the presence of virtualization technology [8, 9], we could start and stop virtual partitions as well. The list of the required functions includes the following items.

- Placement sensor – we require that for each application there exist a mechanism that allows us to obtain the up-to-date information about the location of its instances.
- Placement effecter – for each application, we need a mechanism that allows us to start or stop its instance on a given node.
- Request router – the request dispatching mechanism must transparently adjust to placement changes.
- Application demand estimators – the system must provide online estimates of the current application demand for all managed resources.

2.2 Characterizing application requirements

In the management system described in this section, the size and placement of application clusters is determined automatically based on application requirements and node capacities. Application requirements are characterized as load-independent and load-dependent ones.

Load-independent requirements describe resources needed to run an application on an application server, which are used regardless of the offered load. Examples of such requirements are memory, communication channels, and storage. In this paper, we focus on a single resource – memory. Although, J2EE applications also use other resources in a load-independent manner, we have observed that it is memory that is the most frequent bottleneck resource in J2EE applications. We have classified memory as a load-independent resource since a significant amount of it is consumed by a running application instance even if it receives no load. In addition, memory consumption is frequently related to prior application usage rather than to its current load. For example, even in the presence of low load, memory consumption may be high as a result of caching. Thus, although the actual memory consumption also depends on the application load, characterizing memory usage while taking the load-dependent factor into account is extremely difficult. It is more reasonable to estimate the upper limit on the overall memory utilization and treat it as a load-independent memory requirement. Our system includes a component that dynamically computes this upper limit based on a time-series of memory usage by an application. This component is not described in this paper. Consequently, in this paper we assume that an estimate of memory requirements is available.

Load-dependent requirements correspond to resources whose consumption depends on the offered load. Examples of such requirements are current or projected request rate, CPU cycles/sec, disk activity, and number of execution threads. In this paper, we focus on CPU as a resource that is consumed in a load-dependent manner as it is the most frequent bottleneck resource in J2EE applications. CPU demand is expressed as the number of CPU cycles per second that must be available to an application. Computing this estimate is a non-trivial problem, which is not discussed in this paper. However, our system computes such an estimate online. It includes a component, an application profiler [6] that helps us in this process. The application profiler computes the number of CPU cycles consumed by a request of an application. This per-request CPU demand of an application is then multiplied by the required request throughput for the application, whose estimation depends on admission or flow control mechanism used by the request router.

- If no admission or flow control is implemented by the router, then throughput prediction may be based on time-series of prior throughput observed at application servers.
- If either admission or flow control is implemented by the request router, then the prediction is made based on prior workload characteristics observed at the request router while taking into account the control logic of the request router. In fact, a request router provided by our system implements SLA driven flow control [10]. We use workload observations at the router and its optimization algorithm to estimate application throughput that must be delivered by the system to maximize the overall system utility score.

Correspondingly, node capacity is characterized using load-independent and load-dependent values that describe the amount of resources used to host an application on a node and the available capacity to process requests of applications. Similar to application requirements, we focus on the size of memory and the speed of CPU as load-independent and load-dependent capacity measures.

2.3 System architecture

The dynamic control of the size and placement of dynamic clusters is realized by the system depicted in Figure 2.

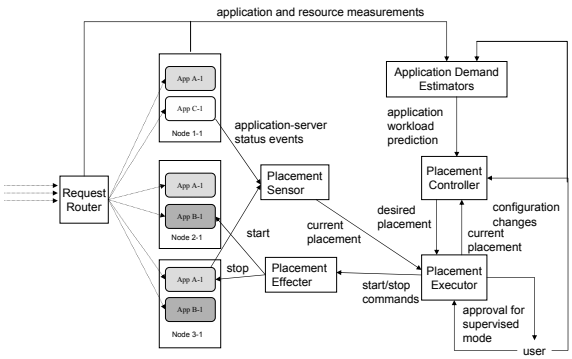

Figure 2: Architecture of the placement management component

At the heart of the system there is *Placement Controller* which computes a new application placement based on load-independent and load-dependent resource demand estimates for all applications. These estimates are obtained online using *Application Demand Estimators*.

In each control cycle, the Placement Controller computes a new placement matrix with the objective to satisfy the entire load-dependent demand of applications without exceeding load-dependent

and load-independent resource capacities of the managed nodes. In addition, the Placement Controller works with the following considerations:

- Each start and stop operation is potentially costly in terms of resource consumption and the impact on the running applications. Depending on the processor speed it may take between tens of seconds to several minutes. The starting or stopping process also consumes a significant fraction of CPU while it occurs. Therefore, the Placement Controller aims at minimizing the number of placement changes compared to the placement currently in effect.

- Oftentimes, more than one placement exists with the same amount of satisfied demand and the number of changes compared to the previous placement. Yet, these placements may differ with respect to how effectively they utilize available resources. Placements that tend to pack a lot of load on a few nodes adversely affect performance and high-availability of applications. Therefore, Placement Controller chooses configurations that allow load to be as evenly distributed across all nodes as possible.

- A user may want to prevent some applications from being managed by the Placement Controller by putting them in *unmanaged mode*. Applications in unmanaged mode cannot be started or stopped, but their load must be taken into account when computing placement for all other applications.

- Nodes may differ with respect to their ability to host a particular application. Placement Controller must observe *allocation restrictions* introduced as a result of this heterogeneity.

Once placement is computed by the Placement Controller, it is effected by the *Placement Executor*, which implements the placement by performing individual start and stop operations on the application servers. As each start and stop operation disrupts the system, the Placement Executor orders the individual placement changes in a manner that minimizes the disruption to the running applications. Placement Executor ensures that:

- No more than one application instance is being started or stopped on any node at a time.

- Load-independent capacity constraint is observed during placement changes. If it is necessary to temporarily violate a node's load-independent capacity constraint, the utilized node-independent capacity exceeds the available capacity by no more than one instance.

- At least one instance of each application is running at all times.

The Placement Executor delays the start and stop actions that would violate the above policies given actions that are already in progress. It monitors the progress of the initiated start and stop actions and reevaluates the actions that have been delayed. As soon as it is allowed by the policies, the Placement Executor initiates the delayed actions.

The Placement Executor implements placement changes using *Placement Effecter*. It monitors the status of the initiated changes and keeps track of placement changes effected by other managers or a user using the *Placement Sensor*.

The system we describe above is highly dynamic allowing various types of configuration changes, such as adding or removing a node, creating and destroying an application cluster, or changing node or cluster properties to be accommodated at runtime. In addition, all components are highly available.

This paper does not attempt to present all components of the system described in this section, although all of them have been implemented as a part of our project. We focus on the functionality of Placement Controller and in the following sections we discuss it in detail.

3. PLACEMENT CONTROLLER

The Placement Controller drives a control loop within which a new placement is computed and implemented. The execution diagram of the control loop is shown in Figure 3. The rest of this section describes the optimization problem solved by the Placement Controller.

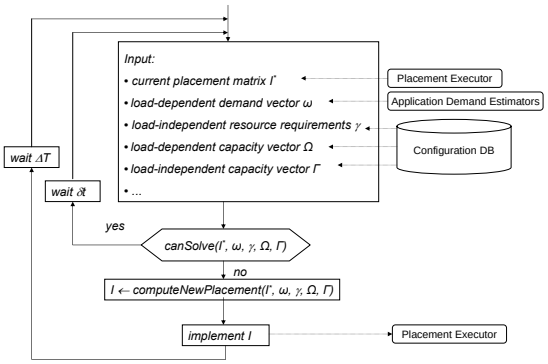

Figure 3: Placement control loop

3.1 Problem Statement

We formulate the placement problem as follows. We are given a set of nodes, $\mathcal{N}$, and write n for a member of that set. We are also given a set of applications $\mathcal{M}$, and write m for a member of that set. With each node n we associate its load-independent and load-dependent capacities, Γ_n and Ω_n, which correspond to the node's memory and CPU power, respectively. With each application, we associate its load-independent and load dependent demand values, γ_m and ω_m, which correspond to the application's memory and CPU requirements, respectively. We assume that γ_m is the amount of memory consumed by an instance of application m, which is independent of offered load. The CPU-speed requirements of the application, ω_m, depend on offered load.

With each application we also associate a boolean flag $\mathrm{managed}_m$ indicating whether instances of the application can be started or stopped by the placement controller. Also, with each application and each node we associate the allocation restriction flag, $\mathcal{R}_{m,n}$ indicating whether an instance of the application m may be started on node n.

The placement problem is to find matrix I, where $I_{m,n}$ is 1 if an instance of application m is running on node n and 0 otherwise, such that (1) allocation restrictions are observed, (2) load-independent capacity limits are observed, and (3) it is possible to allocate load of all applications to instances without exceeding load-dependent capacity limits. If a previous placement matrix $I^*_{m,n}$ exists, we also want to minimize the number of changes between the old and the new matrix. If it is not possible to satisfy the entire demand of applications, we aim at maximizing the amount of satisfied demand. Finally, among all placements that maximize the amount of

satisfied demand, we want to find one that allows the load allocated to nodes to be balanced.

Since we are dealing with multiple optimization objectives, we prioritize them in the formal statement of the problem as follows. Let L be a matrix of real valued *load placement* and $\rho = \frac{\sum_m \sum_n L_{m,n}}{\sum_n \Omega_n}$.

$$\begin{array}{ll} \text{(i)} & \max \sum_m \sum_n L_{m,n} \\ \text{(ii)} & \min \sum_m \sum_n |I^*_{m,n} - I_{m,n}| \\ \text{(iii)} & \min \sum_n (\frac{\sum_m L_{m,n}}{\Omega_n} - \rho)^2 \end{array} \quad (1)$$

s.t:

$$\forall_n \sum_m I_{m,n}\gamma_m \leq \Gamma_n \quad (2)$$

$$\forall_n \sum_m L_{m,n} \leq \Omega_n \quad (3)$$

$$\forall_m \sum_n L_{m,n} \leq \omega_m \quad (4)$$

$$\forall_m \forall_n I_{m,n} = 0 \Rightarrow L_{m,n} = 0 \quad (5)$$

$$\forall_m \forall_n \mathcal{R}_{m,n} = \text{false} \Rightarrow I_{m,n} = 0 \quad (6)$$

$$\forall_m \text{managed}_m = \text{false} \Rightarrow \forall_n I_{m,n} = I^*_{m,n} \quad (7)$$

The optimization problem solved by the Placement Controller is a variant of the Class Constrained Multiple-Knapsack Problem [11] (CCMKP) and differs from prior formulations of this problem mainly in that it also minimizes placement changes. Since this problem is NP-hard, the Placement Controller adopts a heuristic, which is presented in Section 3.2.

The Placement Controller avoids solving the optimization problem if the current placement can serve the new demand. This is achieved by first testing whether there exists a load-distribution matrix for placement matrix I^* that satisfies constraints expressed by Eqns. (2)-(5) (function *canSolve* in Figure 3). This test forms a familiar linear-programming problem, for which a number of efficient solutions may be found in literature [12, 13].

If the current placement can serve the new demand, after waiting δt, the Placement Controller proceeds to the next control cycle. Otherwise, the new placement is computed and implemented. After placement is changed, the Placement Controller does not resume the new cycle for $\Delta T > \delta t$. The increased timeout provides sufficient time for the system to adjust to the new placement and reduces the number of placement changes. Both δt and ΔT are configurable parameters with 1 minute and 15 minutes as their default values.

3.2 Placement Algorithm

In this section we present an outline of a heuristic algorithm used by the Placement Controller to solve the placement problem.

The basic algorithm consists of three parts: the residual placement, incremental placement, and rebalancing placement. The residual placement is used at the system start-up, when there is no prior placement, and as a subroutine of the incremental placement. The rebalancing placement is invoked at the end of incremental placement, once the maximizing placement is found, to modify it in a way that allows for a better distribution of load-dependent demand.

3.2.1 Residual placement

Residual placement is based on the following intuitive rule. If we look at allocated memory as the cost of allocating an application's CPU demand, it is wise to first place an application with the highest memory requirement compared to its CPU demand. This way, we can maximize the chances that this application will be placed on the fewest possible number of nodes, and thus the cost of satisfying its CPU demand will be the lowest. When choosing a node on which to place an application, it is reasonable to first search for a node with density similar to the density of the application. It is not wise to load applications with high density on a low density server, since we would be likely to reach the processing capacity constraint and leave a lot of memory unused on that server. Similarly, if low density applications are loaded on high density servers, we would be likely to reach the server's memory constraint without using much of the processing capacity.

In the residual placement, nodes are ordered by the increasing value of their densities, Ω_n/Γ_n, and applications are ordered by increasing densities ω_m/γ_n. The algorithm starts with the lowest density application m and looks for the lowest density node that can fit the application, i.e., a node n such that $\gamma_m \leq \Gamma_n$ and $\mathcal{R}_{m,n}$ is true. If the node satisfies the entire load-dependent capacity of m, i.e., $\Omega_n \geq \omega_m$ then the application is removed from the application list and the node is added back to the ordered node list after its load-dependent capacity is decreased by ω_m. Otherwise, the entire node's available load-dependent capacity is allocated to the application, the node is removed from the node list, and the algorithm searches for another node on which to place the residual demand of the application. The algorithm proceeds until the demand of all applications is assigned to nodes, or no more assignments are possible given nodes' capacities. The computational complexity of residual placement is $\mathcal{O}(|\mathcal{N}||\mathcal{M}|)$.

3.2.2 Incremental placement

The incremental placement combines the residual placement with the maximum flow computation to solve the placement problem while minimizing the number of placement changes. It derives the new placement I from the previous placement I^* incrementally as follows.

In a given iteration of the algorithm, we first check of the current placement, I^* can satisfy the demand. This is done by solving the maximum bipartite flow problem represented by Eqns. (1-i) and (3)-(5). If a solution exists, i.e., $\max \sum_m \sum_n L_{m,n} = \sum_m \omega_m$, the algorithm completes. Otherwise, we are left with some residual demand for each application that could not be satisfied given the current placement, $\omega_m - \sum_n L_{m,n}$. For some applications this residual demand may be zero. We are also left with some residual load-dependent and load-independent capacity on each server, $\Omega_n - \sum_m L_{m,n}$ and $\Gamma_n - \sum_m I_{m,n}\gamma_m$. We apply the algorithm from Section 3.2.1 to find an allocation of the residuals. If the residual placement succeeds in finding a feasible allocation, we set I to the resulting allocation and exit. Otherwise, we remove the assignment with the lowest density, $L_{m,n}/\gamma_m$ from I^* and proceed to the next iteration.

Observe, that the number of iterations performed by the incremental algorithms depends on how hard it is to find the satisfying placement. The problem is hard to solve if the total demand of applications compared to the total available capacity is high, or the total memory requirement of applications approaches total memory available on nodes. The more difficult the problem is to solve, the longer it takes for the algorithm to complete. The upper bound on the number of iterations is the number of assignments in the current placement, i.e., $\sum_n \sum_m I^*_{m,n}$.

When applications in unmanaged mode are present, in the first iteration of the algorithm we adopt a slightly modified procedure. We first find the maximum load allocation for applications in un-

managed mode. Then we subtract the capacity utilized by them from the total available capacity. If any residual demand of these applications remains, it is ignored; the unmanaged applications are removed from further consideration and the algorithm proceeds according to the above procedure.

3.2.3 Rebalancing placement

The last phase of the placement algorithm aims at modifying the solution proposed by the incremental algorithm such that a better load balancing across nodes is achieved. This phase begins with old and new placement matrices I^* and I and load distribution matrix L. We first compute the amount of demand satisfied by L for all applications: $\forall_m \omega_m^+ = \sum_n L_{m,n}$. Then we try to find another load distribution matrix L^+ that satisfies the same demand for all dynamic clusters and that perfectly balances the load assigned to nodes. We find L^+ by solving the following optimization problem:

$$\min \sum_n |\sum_m L_{m,n}^+ - \rho\Omega_n|, \text{ where } \rho = \frac{\sum_m \omega_m^+}{\sum_n \Omega_n} \quad (8)$$

s.t:

$$\forall_n \sum_m L_{m,n}^+ \leq \Omega_n \quad (9)$$

$$\forall_m \sum_n L_{m,n}^+ = \omega_m^+ \quad (10)$$

$$\forall_m \forall_n I_{m,n} = 0 \Rightarrow L_{m,n}^+ = 0 \quad (11)$$

The above problem may be easily transformed into a min-cost flow problem in a bipartite flow network for which a number of polynomial time algorithms exist [13]. Note that if there exists a load-distribution that achieves the perfect balance of node utilization (utilization of all nodes equal to ρ), then the above optimization will find such a load-distribution. When perfect load-distribution cannot be found, the load-distribution found by the optimization problem includes some nodes loaded above ρ as a utilization threshold and some nodes loaded below ρ. The point is to change assignments in such a way so as to allow shifting some load from overloaded nodes to underloaded nodes. Only assignments in $I - I^*$ may be moved. Assignments that overlap with prior placement cannot be moved as this could increase placement churn.

The rebalancing placement proceeds from the most overloaded node and attempts to move some of its instances to one or more underloaded nodes. We always choose an instance whose removal brings the node utilization closest to ρ. This procedure continues until all nodes are balanced or no more useful changes can be done.

3.3 Algorithm variants

When the algorithm described in the previous section is applied to the system in which load-dependent demand of applications is high compared to the total available load-dependent capacity, its execution time increases, and the ability to find the optimal placement decreases. We have observed that the same algorithm applied to a demand that is reduced to a certain percentage of the total available capacity not only executes faster, but is also more likely to produce placement that satisfies the entire real demand.

Another factor that impacts the effectiveness of the algorithm is the content of the previous placement matrix; the same algorithm applied to different prior placement matrices produces different results.

Taking these observations into account, we have implemented three simple variants of the placement algorithm.

- Basic algorithm (BA) - the algorithm as described in the previous section
- Load-reduction algorithm (LRA) - basic algorithm executed with modified input; whenever total application demand exceeds 90% of the total capacity we proportionately reduce the demand of each application so as to bring the total demand down to 90% of the total available capacity
- Multiple-runs algorithm (MRA) - if placement matrix produced by the basic algorithm does not satisfy the entire demand, we execute the basic algorithm one more time using the output of the first execution as prior-placement matrix. We repeat this process until no more improvement in the amount of satisfied demand is observed but no more that 10 times.

4. SIMULATION RESULTS

In this section we study the proposed placement algorithms via simulation. We investigate the algorithms in three dimentions: the effectiveness in satisfying the demand, ability to achieve balanced load distribution, and computational complexity.

4.1 Effectiveness of the algorithms

In the simulation study, we vary the size of the problem (by varying the number of nodes and dynamic clusters) and the hardness of the placement problem that is being solved.

4.1.1 Defining problem hardness

The hardness of the placement problem is affected by three factors: memory load factor, CPU load factor, and workload variability factor. When we ignore minimum/maximum instances policies, vertical stacking, and allocation restrictions, we can define the hardness factors as follows.

- Let $\bar{\gamma}$ and $\bar{\Gamma}$ be the average load-independent demand and capacity values, respectively. The expected maximum number of applications that may be hosted on a group of N nodes is $\frac{\bar{\Gamma}}{\bar{\gamma}}N$. The memory load factor is defined as MLF $= \frac{M\bar{\gamma}}{N\bar{\Gamma}}$, where M is the number of applications.
- The CPU load factor is defined as CLF $= \frac{\sum_m \omega_m}{\Omega}$, where $\Omega = \sum_n \Omega_n$.
- The placement problem is harder to solve if the current load-dependent demand distribution differs significantly from the distribution at the time of last placement computation. Let $\omega(t)$ be a random variable of load-dependent demand of an application at time t. Suppose that we know the maximum value of CLF. The maximum change of load-dependent demand distribution occurs, for example, when in each control cycle the entire demand in the system, CLF Ω, is contributed by just one, each time different application. The average maximum load-dependent demand change is then $\frac{\text{CLF } \Omega}{M}$. Given the actual probability distribution function of $\omega(t)$, we measure workload variability by dividing the expected per-application absolute difference between load-dependent demand values in the current and previous cycles by the average maximum difference, i.e., using workload variability factor WVF $= E(|\omega(t+1) - \omega(t)|)/\frac{\text{CLF } \Omega}{M} \leq 1$. When WVF increases, the placement algorithm results in more placement changes. When WVF $= 0$, after the initial placement is found no placement changes are ever needed.

4.1.2 Varying the problem hardness

In the experimental design, we vary the number of nodes and dynamic clusters. Load independent capacity and demand values are uniformly distributed over sets $\{1, 2, 3, 4\}$ and $\{0.4, 0.8, 1.2, 1.6\}$, respectively with average values of $\bar{\Gamma} = 2.5$ and $\bar{\gamma} = 1$, respectively. Load-dependent capacity is the same for all nodes and is equal to 100.

- To achieve the memory load factor of MLF given the number of nodes N, for each trial we vary MLF between 40% and 100%, and set M to $2.5 \times \text{MLF} \times N$.
- To control the CPU load factor CLF we generate load-dependent demand randomly and normalize it such that its total amount is equal to CLFΩ. We vary CLF between 90% and 99%.
- We focus on only one way to generate load-dependent demand. In our experiments, $\omega(t)$ is uniformly distributed over the entire range independently of $\omega(t-1)$. This technique yields the workload variability factor WVF $= \frac{2}{3}$ and is harsh on the algorithm. In reality, $\omega(t)$ is expected not to differ very much from $\omega(t-1)$ over short time intervals.

4.1.3 Evaluation criteria

We compare the algorithm variants based on the following criteria: (1) the percentage of satisfied demand, (2) the number of placement changes, and (3) execution time. These measurements are collected using 100 experiments. In each experiment we randomly generate system configuration (i.e., load-independent demand and capacity values), and then perform a sequence of 11 placement computations including one initial placement. In each computation, a new load-dependent demand vector is generated.

For each computed placement, the percentage of satisfied demand is calculated by first finding the load-distribution matrix L that maximizes the amount of satisfied demand given the computed placement, as defined through Eqns. (3)-(5). Then we calculate $\frac{\sum_m \sum_n L_{m,n}}{\sum_m \omega_m}$.

4.1.4 Results

In Figs. 4, 5, and 6, we present experimental results for the fraction of demand satisfied, number of placement changes, and execution time, respectively. The results are presented with 95% confidence intervals. When memory load factor is high, e.g., MLF=100%, the algorithm is memory-constrained and regardless of the CPU load factor it usually fails to place all applications and consequently does not meet CPU demand. As shown in Fig. 4, in a very memory- and CPU constrained scenario of MLF=100% and CPU=99%, the Multiple-runs Algorithm (MRA) and Load-reduction Algorithm (LA) appear to perform somewhat better than the Basic Algorithm (BA), however, all algorithms fail to meet the entire load-dependent demand. Also, they generate roughly the same (huge) number of placement changes (Fig. 5). From the perspective of execution time, extremely resource-constrained problems are particularly harsh on the Multiple-runs Algorithm because they trigger multiple algorithm iterations thereby elevating the placement computation time (Fig. 6).

However, when the memory constraint is relaxed to MLF=60%, the placement problem becomes easier to solve and the differences between algorithms more obvious. As shown in Fig. 4, MRA satisfies more demand than LRA and BA. This result is achieved with the same number of placement changes as with BA. LRA performs worse than MRA in terms of satisfying the demand but it proves better in reducing the placement churn (Figs. 4 and 5). The improved effectiveness of MRA comes at the price of multiplied execution time compared to LRA and BA. Our experiments have revealed that it not necessary to reduce MLF to as low as 60%. We have observed similar results, though not as prominent, at MLF=80% as well.

When memory and CPU constraints are further reduced (to say MLF=60% and CLF=90%), the placement problem becomes easy to solve and all algorithms perform equally well. Figs. 4-6 show the results obtained with MLF=40% and CLF=90%.

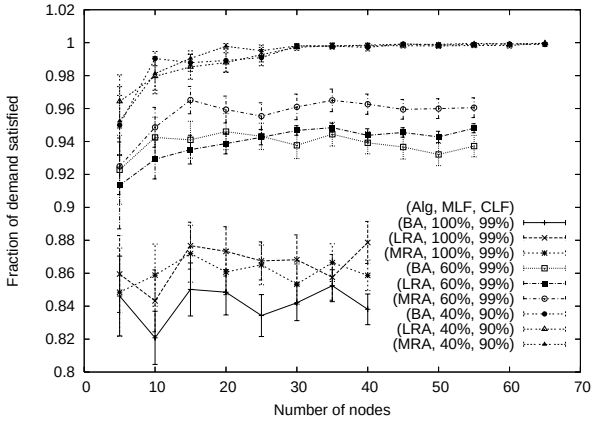

Figure 4: Fraction of demand satisfied by algorithms

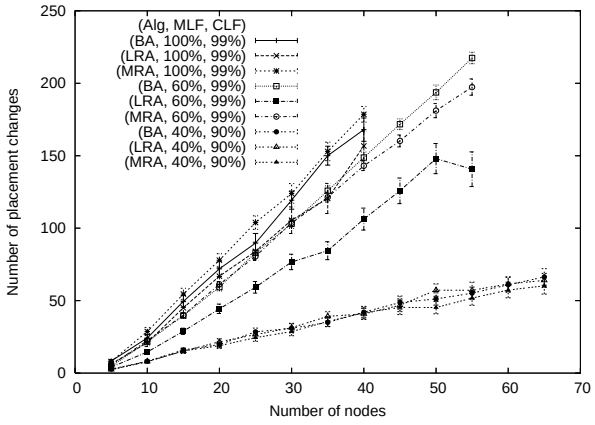

Figure 5: Number of placement changes

4.2 Effectiveness of load distribution

In this section we evaluate the effectiveness of rebalancing placement applied to MRA. For this purpose, we compare two versions of this algorithm: with and without the rebalancing placement phase. We compare the algorithms while looking at load distribution in the initial placement, which starts with the empty set of placement assignments, and at the average of ten subsequent placements, which start with a non-empty set of prior placement assignments. Distinguishing these two cases is important as in the initial placement we

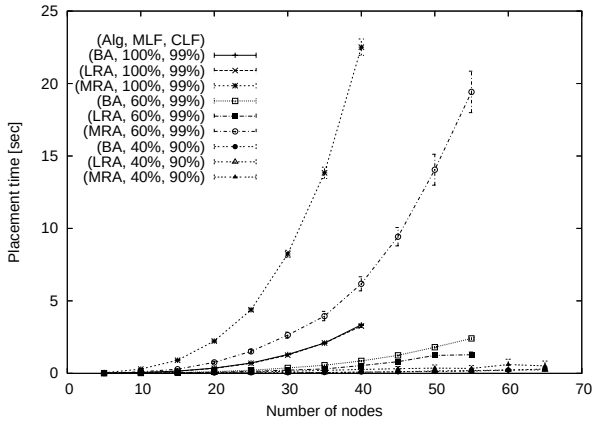

Figure 6: Placement computation time

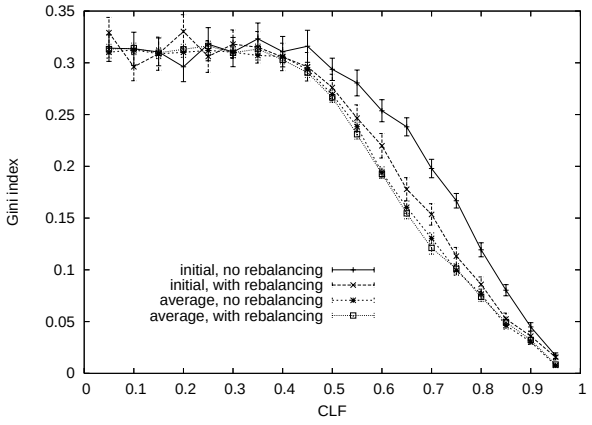

Figure 7: Gini index of initial placement and the average Gini index for MLF=16%

typically have greater freedom in replacing instances than in subsequent placements, where only a subset of all instances is eligible for being moved.

4.2.1 Evaluation criterion

To assess the effectiveness of rebalancing placement we must consider that load distribution is performed by a router. It is the effectiveness of the router that ultimately affects the evenness of the load distribution. In this section, we assume a router that can perfectly balance load within limits imposed by the placement in effect. To obtain load distribution that is achieved by such an ideal router, we need to solve the optimization problem expressed by Eqns. (1-iii) and (3)-(5). This non-linear problem may be quite easily solved as a sequence of at most N problems defined by Eqns. (8)-(11), which are straightforward to linearize.

Once the most balanced distribution is known, we must measure the amount of inequality in the distribution. As a good candidate of such a measure, we consider the Gini index [14] (Gini), which is defined as a ratio of the area between the line of perfect (uniform) distribution (45 degree line) and the Lorenz curve of the actual distribution to the total area below the 45-degree line. This coefficient is most frequently used as a measure of income inequality. The Gini index of 0 indicates perfect equality. The Gini index approaches one to represent extreme imbalance.

4.2.2 Experiment design

We evaluate the effectiveness of rebalancing placement in a network composed of 10 nodes with load-independent demand uniformly distributed over a set $\{1, 2, 3, 4\}$.. We compute placement for a system with 10, 30, and 50 applications with identical memory requirements of 0.4. This gives us MLF values of 16%, 48%, and 80%, respectively. We vary CLF from 5% to 95%.

4.2.3 Results

Figs. 7-9 show initial and average Gini indices for MLF values of 16%, 48%, and 80%, respectively.

Our study reveals that the potential improvement in load balance depends on the number of degrees of freedom the rebalancing placement has while moving instances. For low values of MLF, we deal with only a small number of instances that can be moved with relatively high fraction of total demand allocated to each of them. For high values of MLF, we deal with many instances with potentially small fraction of total load allocated to them but we have little extra space to move these instances to. In addition, high MLF implies that load allocated to each instance is also high. Oftentimes, we can only improve balance by creating an additional instance for an application and distributing the application load across the increased number of instances. However, doing so would result in increasing the disruption to the system and is not considered by the rebalancing algorithm even when applied to initial placement.

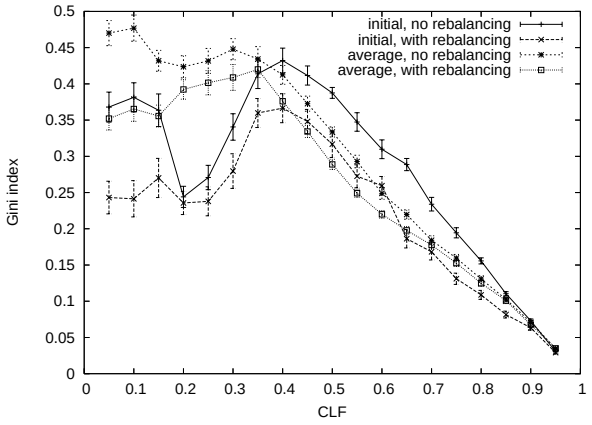

Figure 8: Gini index of initial placement and the average Gini index for MLF=48%

Our study also shows that initial placement typically has a much better potential for distributing load than subsequent placements. This observation concerns both the algorithm with rebalancing and the one without it. This difference results from the way we prioritized the objectives: we effect the fewest number of changes to the previous placement that are necessary to satisfy the entire demand even though the resultant load distribution may be unbalanced.

Our study also concerns the analysis of rebalancing cost, which may be quantified as an increase in placement time. However, our study shows that this cost is negligible. In a system of 10 nodes and 30 applications, the rebalancing phase takes less than 10 msec and is invoked only when placement changes are suggested by the

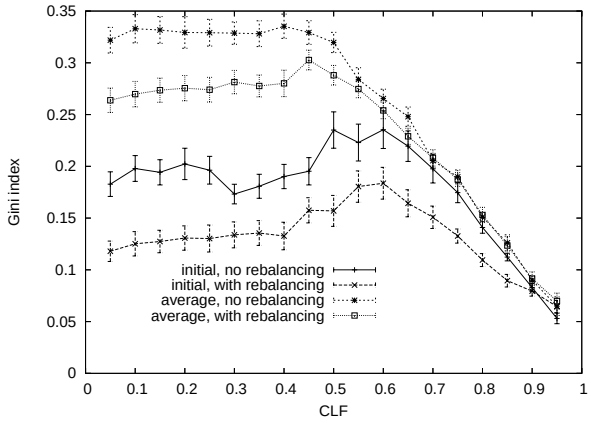

Figure 9: Gini index of initial placement and the average Gini index for MLF=80%

algorithm. In configurations where placement changes are frequent (for example with high CLF), the rebalancing time is only a small fraction of the overall execution time.

Placement rebalancing affects not only the placement time, but also the number of placement changes. Even though our rebalancing algorithm does not directly introduce any additional placement changes, it modifies the input to the subsequent control cycle. This modified input may lead to a placement with a higher or lower number of placement changes than if no rebalancing was performed. However, in our study we could not find any statistically significant difference in the average number of placement changes between the two algorithms.

5. RELATED WORK

The problem of dynamically allocating server resources to applications has been extensively studied. A popular approach to this problem relies on dynamic server provisioning in which full server machines are allocated to applications (and provisioned for them) as needed [15]. This approach does not allow applications to overlap on nodes (unless several applications are grouped and managed as one). The solution proposed in this paper allows multiple applications to share the same server thereby achieving finer granularity of allocation and better utilization of resources compared to dedicated-server approaches. Chandra et al. [16] have argued that such fine-grained resource allocation at small time-scales can lead to substantial multiplexing gains.

The problem of application placement in a shared cluster of servers has been investigated by Urgaonkar at el. [1]. In their formulation, instances of an application that execute on different hardware servers, which are called capsules, are managed independently of one another. Capsule resource requirements are estimated independently based on prior resource usage by a capsule. Then capsules are packed on servers in such a way so as not to exceed each server's capacity. Independent packing of capsules is a rather inflexible mechanism of application placement as it does not allow the number of capsules serving a given application to be changed or the load to be shifted from one capsule to another. Another issue with the approach presented in [1] is that capsule requirements are characterized using a single value that can represent either CPU or network bandwidth requirements. The approach introduced in this paper allows applications to be placed based also on their memory requirements. Finally, the technique in [1] ignores previous capsule placement while finding the new placement. It is therefore unsuitable for frequent online placement modifications, which is a goal of the technique described in this paper.

In the work of Ardagna et al. [4], the placement problem aims at minimizing the costs of running the server machines: each machine has a cost, and it is paid if any tier of any application is placed there. This objective is different from ours. Unlike Ardagna et al., we must be concerned with the possibility that some demands may not be met and therefore try to maximize the amount of demand that can be satisfied. On the other hand, our approach does not consider the cost of running the server machines. There are more differences between our problem statements: they accept co-residency prohibitions of a certain form; our formulation does not have this feature; our formulation includes a desire to minimize changes from an existing placement, while their formulation has no such concern; they are concerned with a larger resource allocation problem that also includes routing and scheduling, while we solve placement problem independently.

Wolf and Yu in [2] propose an algorithm that maps Web sites to cluster servers allowing an overlap between Web-site clusters. The algorithm proposed in [2] attempts to minimize the number of placement changes by exploring new configurations starting from those that differ no more than one Web-site-to-server assignment from the current placement and limiting the total number of placement changes to a configured number. Unlike the technique proposed in this paper, Wolf et al. [2] focus on a single-resource capacity constraints. Moreover, their optimization technique is intended to be used infrequently (about once a week) and it may be executed on a dedicated server. Hence, there is no concern about its computational overhead. In the problem discussed in this paper, the application placement is re-evaluated frequently (in the order of minutes) and the computation is co-located with applications. Therefore, its computational overhead must be very small.

Our work can be also compared to the work on Cluster Reserves by Aron et al. [3], where an application and application instance considered in our paper correspond to a service class and a cluster reserve in [3]. The resource allocation problem solved in [3] is to compute the desired percentage of each node's capacity to be allocated to each service class based on overall cluster utilization ratios desired for each service class. This problem differs from the problem solved in our paper in several aspects. First, we aim at finding the best placement of application instances on nodes rather than an optimal allocation of each node's capacity. Our technique does compute such an allocation as a side effect. However, this allocation is not enforced, meaning that we only compute it to determine if it is feasible to allocate applications' demand to nodes with a given placement. The actual resource allocation is determined by the request router at a much finer time scale, which improves the system responsiveness to workload changes. Second, we aim at satisfying the most of the offered demand rather than keeping the relative resource allocation as close to the goal as possible. Third, our problem takes into account the limited memory capacity of nodes, which constraints the number of application instances that may be placed on each of them. Finally, the objectives of our optimization problem include minimizing changes from the previous allocation, which is not the case in [3].

The problem of optimally placing replicas of objects on servers, constrained by object and server sizes as well as capacity to satisfy a fluctuating demand for objects, has appeared in a number of fields related to distributed computing. In managing video-on-demand systems, replicas of movies are placed on storage devices and streamed by video servers to a dynamic set of clients with a

highly skewed movie selection distribution. The goal is to maximize the number of admitted video stream requests. Several movie placement and video stream migration policies have been studied. A disk load balancing criterion which combines a static component and a dynamic component is described in [17]. The static component decides the number of copies needed for each movie by first solving an apportionment problem and then solving the problem of heuristically assigning the copies onto storage groups to limit the number of assignment changes. The dynamic component solves a discrete class-constrained resource allocation problem for optimal load balancing, and then introduces an algorithm for dynamically shifting the load among servers (i.e. migrating existing video streams). A placement algorithm for balancing the load and storage in multimedia systems is described in [18]. The algorithm also minimizes the blocking probability of new requests.

In the area of parallel and grid computing, several object placement strategies (or, meta-scheduling strategies) have been investigated [19, 20]. Communication overhead among objects placed on various machines in a heterogeneous distributed computing environment plays an important role in the object placement strategy. A related problem is that of replica placement in adaptive content distribution networks [21, 20]. There the problem is to optimally replicate objects on nodes with finite storage capacities so that clients fetching objects traverse a minimum average number of nodes in such a network. The problem is shown to be NP-complete and several heuristics have been studied, especially distributed algorithms.

Similar problems have been studied in theoretical optimization literature. The special case of our problem with uniform memory requirements was studied in [22, 23] where some approximation algorithms were suggested. Related optimization problems include bin packing, multiple knapsack and multi-dimensional knapsack problems [24].

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose the design of and evaluate a middleware clustering technology capable of dynamically allocating resources to web applications through dynamic *application instance placement*. We define application instance placement as the problem of placing application instances on a given set of server machines to satisfy the resource demand of each application cluster. We introduce a *placement controller* mechanism that dynamically configures the size and placement of application instances while obeying to user-defined policies. We introduce a placement algorithm that can compute placement obeying load-dependent and load-independent capacity constraints. The algorithm produces a placement that limits the number of placement changes and allows load to be better balanced across nodes. In the experimental study we evaluate the proposed algorithm and its variants with respect to their ability to satisfy the application demand, reduce the number of placement changes, and evenly distribute load.

Our work can be improved in a number of ways. First, our algorithm does not explicitly model a cost of a placement change. We only assume that this cost is high and therefore worth minimizing. A possibly better approach could model the impact of a placement change on the performance of applications. Instead of minimizing cost, it would attempt to maximize the overall system utility. This new approach would be particularly suitable when application start or stop mechanism is a light-weight one, e.g., with the usage of OS-level virtualization technology.

Second, we do not prioritize applications. If the entire demand cannot be satisfied, some applications will be affected either by their increased execution time, or increased waiting time, or increased rejection rate. We do not attempt to control which applications are affected this way.

Third, our rebalancing technique aims at equalizing CPU utilization across nodes. However, with heterogeneous nodes, better application performance may be achieved if response times of an application on different nodes are equalized instead. Our technique does not model or manage response times as a function of node utilization.

7. REFERENCES

[1] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," in *Proc. Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[2] J. Wolf and P. Yu, "On balancing load in a clustered web farm," *ACM Transactions on Internet Technology*, vol. 1, no. 2, pp. 231–261, 2001.

[3] M. Aron, P. Druschel, and W. Zwaenpoel, "Cluster reserves: A mechanism for resource management in cluster-based network servers," in *Proc. ACM SIGMETRICS*, Santa Clara, CA, Jun. 2000.

[4] D. Ardagna, M. Trubian, and L. Zhang, "SLA based profit optimization in multi-tier web application systems," in *Int'l Conference On Service Oriented Computing*, New York, NY, 2004, pp. 173–182.

[5] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic application placement under service and memory constraints," in *Int'l Workshop on Efficient and Experimental Algorithms*, Santorini Island, Greece, May 2005.

[6] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, and A. Youssef, "Managing the response time for multi-tiered web applications," IBM, Tech. Rep. RC 23651, 2005.

[7] "WebSphere eXtended Deployment (XD)," http://www-306.ibm.com/software/webservers/appserv/extend/.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, 2003.

[9] "VMware," http://www.vmware.com/.

[10] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef, "Performance management for cluster based web services," in *Proc. Int'l Symposium on Integrated Network Management*, Colorado Springs, USA, March 2003, pp. 247–261.

[11] H. Shachnai and T. Tamir, "Noah's bagels – some combinatorial aspects," in *Proc. Int'l Conf. Fun with Algorithms*, 1998, pp. 65–78.

[12] G. Dantzig, *Linear Programming and Extensions*. Princeton University Press, 1998.

[13] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.

[14] C. Gini, "Variabilità e mutabilità," in *Memorie di metodologica statistica*, E. Pizetti and T. Salvemini, Eds. Libreria Eredi Virgilio Veschi, 1955.

[15] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano SLA based management of a computing utility," in *Proc. Int'l Symposium on Integrated Network Management*, Seattle, WA, May 2001, pp. 14–18.

[16] A. Chandra, P. Goyal, and P. Shenoy, "Quantifying the benefits of resource multiplexing in on-demand data centers," in *First ACM Workshop on Algorithms and Architectures for Self-Managing Systems*, San Diego, CA, Jun. 2003.

[17] J. L. Wolf, P. S. Yu, and H. Shachinai, "Disk load balancing for video-on-demand systems," *ACM Multimedia Systems Journal*, vol. 5, no. 6, 1997.

[18] D. N. Serpanos, L. Georgiadis, and T. Bouloutas, "Mmpacking: A load and storage balancing algorithm for distributed multimedia servers," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 1, February 1998.

[19] A. Turgeon, Q. Snell, and M. Clement, "Application placement using performance surfaces," in *Proc. Ninth Int'l Symposium on High-Performance Distributed Computing*, Pittsburgh, PA, August 2000, pp. 229–236.

[20] J. Kangasharju, J. Roberts, and K. W. Ross, "Object replication strategies in content distribution networks," in *Proc. Sixth Int'l Workshop on Web Content Caching and Distribution*, Boston, MA, 2001.

[21] S. Buchholz and T. Buchholz, "Replica placement in adaptive content distribution networks," in *Proc. 2004 ACM symposium on Applied Computing*, March 2004.

[22] H. Shachnai and T. Tamir, "On two class-constrained versions of the multiple knapsack problem," *Algorithmica*, vol. 29, 2001.

[23] ——, "Noah bagels - some combinatorial aspects," in *Int'l Conference on FUN with Algorithms*, Isola d'Elba, June 1998.

[24] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Springer–Verlag, 2004.