A Scalable Application Placement Controller for Enterprise Data Centers

Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici

IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA
{ctang, steinder, mspeitz, giovanni}@us.ibm.com

ABSTRACT

Given a set of machines and a set of Web applications with dynamically changing demands, an online application placement controller decides how many instances to run for each application and where to put them, while observing all kinds of resource constraints. This NP hard problem has real usage in commercial middleware products. Existing approximation algorithms for this problem can scale to at most a few hundred machines, and may produce placement solutions that are far from optimal when system resources are tight. In this paper, we propose a new algorithm that can produce within 30 seconds high-quality solutions for hard placement problems with thousands of machines and thousands of applications. This scalability is crucial for dynamic resource provisioning in large-scale enterprise data centers. Our algorithm allows multiple applications to share a single machine, and strives to maximize the total satisfied application demand, to minimize the number of application starts and stops, and to balance the load across machines. Compared with existing state-of-the-art algorithms, for systems with 100 machines or less, our algorithm is up to 134 times faster, reduces application starts and stops by up to 97%, and produces placement solutions that satisfy up to 25% more application demands. Our algorithm has been implemented and adopted in a leading commercial middleware product for managing the performance of Web applications.

Categories and Subject Descriptors

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

General Terms

Algorithm, Management, Performance

Keywords

Application Placement, Performance Management

1. INTRODUCTION

With the rapid growth of the Internet, many organizations increasingly rely on Web applications to deliver critical services to their customers and partners. Enterprise data centers may run thousands of machines to host a large number of Web applications that are resource demanding and process client requests at a high rate. Previous studies have shown that the Web request rate is bursty and can fluctuate dramatically in a short period of time [14]. Therefore, it is not cost-effective to over provision data centers to handle the potential peak demands of all the applications.

To utilize system resources more effectively, modern Web applications typically run on top of a middleware system and rely on it to dynamically allocate resources to meet their performance goals. Some middleware systems use clustering technology to improve scalability and availability, by integrating multiple instances of an application, and presenting them to the users as a single virtual application.

Figure 1 is an example of clustered Web applications. The system consists of one front-end request router, three backend machines (A, B, and C), and three applications (x, y, and z). The applications, for example, can be catalog search, order processing, and account management for an online shopping site. The request router receives external requests and forwards them to the application instances.

To meet the performance goals of the applications, the request router may implement functions such as admission control, flow control, and load balancing. These functions decide how to dynamically allocate resources to the running application instances, which are well-studied topics in the literature [4, 14]. This paper studies an equally important problem that receives relatively less attention in the past:

> *Given a set of machines with constrained resources and a set of Web applications with dynamically changing demands, how many instances to run for each application and where to put them?*

We call this problem *dynamic application placement*. We assume that not every machine can run all the applications at the same time due to limited resources such as memory.

Application placement is orthogonal to admission control, flow control, and load balancing, and the quality of a place-

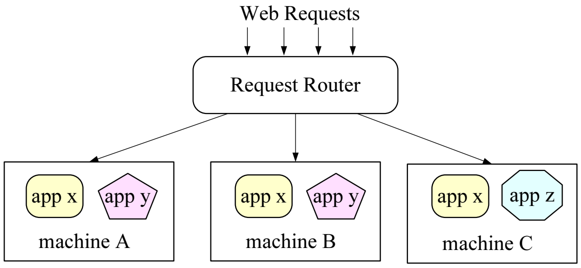

Figure 1: An example of clustered Web applications.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

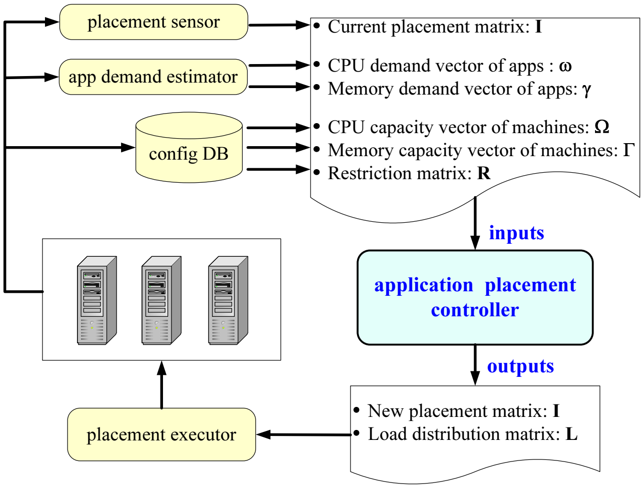
placement sensor
app demand estimator
config DB
• Current placement matrix: I
• CPU demand vector of apps : ω
• Memory demand vector of apps: γ
• CPU capacity vector of machines: Ω
• Memory capacity vector of machines: Γ
• Restriction matrix: R
inputs
application placement controller
outputs
• New placement matrix: I
• Load distribution matrix: L
placement executor

Figure 2: Control loop for application placement.

ment solution can have profound impact on the performance of the entire system. In Figure 1, suppose the request rate for application z suddenly surges. Application z may not meet the demands even if all the resources of machine C are allocated to application z. A smart middleware system then may react by stopping application x on machines A and B, and using the freed resources (e.g., memory) to start an instance of application z on both A and B.

The application placement problem can be formulated as a variant of the Class Constrained Multiple-Knapsack Problem [12, 13]. Under multiple resource constraints (e.g., CPU and memory) and application constraints (e.g., the need for special hardware or software), a placement algorithm strives to produce placement solutions that optimize multiple objectives: (1) maximizing the total satisfied application demand, (2) minimizing the total number of application starts and stops as they disturb the running system, and (3) balancing the load across machines.

The placement problem is NP hard. Existing approximation algorithms [6, 8] can scale to at most a few hundred machines, and may produce placement solutions that are far from optimal when system resources are tight. In this paper, we propose a new approximation algorithm that significantly and consistently outperforms existing state-of-the-art algorithms in terms of both solution quality and scalability. Our algorithm has been implemented and adopted in a leading commercial middleware product [1].

The remainder of the paper is organized as follows. Sections 2, 3, and 4 formulate the application placement problem, describe our algorithm, and present its performance, respectively. Section 5 discusses related work, and Section 6 concludes the paper.

2. PROBLEM FORMULATION

Figure 2 is a simplified diagram of the control loop for application placement. For brevity, we simply refer to "application placement" as "placement" in the rest of this paper. The inputs to the placement controller include the current placement of applications on machines, the resource capacity of each machine, the projected resource demand of each application, and the restrictions that specify whether a given application can run on a given machine, e.g., some application may require machines with special hardware or software. Taking these inputs collected by the auxiliary components, the placement controller computes a placement solution that optimizes certain objective functions, and then passes the solution to the placement executor to start and stop application instances accordingly. Periodically every T minutes, the placement controller produces a new placement solution based on the current inputs (e.g., T=15 minutes).

Estimating application demands is a non-trivial task. We use online profiling and data regression to dynamically estimate the average CPU cycles needed to process one Web request of a given application [11]. The product of the estimated CPU cycles per request and the projected request rate gives the CPU cycles needed by the application per second. We use a peer-to-peer infrastructure [16] to gather performance metrics from a large number of machines in a scalable and reliable fashion.

In the past, we have developed a middleware system [1, 10, 11, 9] that includes a superset of the control loop in Figure 2. In this paper, we focus on the design and evaluation of the placement controller. The rest of this section presents the formal formulation of the placement problem. We first discuss the system resources and application demands considered in the placement problem.

A running application instance's consumption of CPU cycles and IO bandwidth depends on the request rate. As for memory, our system periodically and conservatively estimates the upper limit of an application's near-term memory usage, and assumes that this upper limit does not change until the next estimate update point, because of several practical reasons. First, a significant amount of memory is consumed by an application instance even if it receives no requests. Second, memory consumption is often related to prior application usage rather than its current load due to data caching and delayed garbage collection. Third, because an accurate projection of memory usage is difficult and many applications cannot run when the system is out of memory, it is more reasonable to use the conservatively estimated upper limit for memory consumption.

Among many resources, we choose CPU and memory as the representative ones to be considered by the placement controller. For brevity, the description of our algorithm only considers CPU and memory, but it can deal with other types of resources as well. For example, if the system is network-bounded, we can use network bandwidth as the bottleneck resource whose consumption depends on the request rate. This introduces no changes to our algorithm.

Next, we present the formulation of the placement problem. Table 1 lists the symbols used in our discussion (see Figure 2 for their roles). The inputs to the placement controller are the current placement matrix I, the placement restriction matrix R, the CPU/memory capacity of each machine (Ω_n and Γ_n), and the CPU/memory demand of each application (ω_m and γ_m). The outputs of placement controller are the updated placement matrix I and the load distribution matrix L.

The placement controller strives to find a placement solution that maximizes the total satisfied application demand. In addition, it also tries to minimize the total number of application starts and stops, because placement changes disturb the running system and waste CPU cycles. In practice, many J2EE applications take a few minutes to start or stop, and take some additional time to warm up their data cache. The last optimization goal is to balance the load across machines. Ideally, the utilization of individual machines should stay close to the utilization ρ of the entire system.

$$\rho = \frac{\sum_{m\in\mathcal{M}} \sum_{n\in\mathcal{N}} L_{m,n}}{\sum_{n\in\mathcal{N}} \Omega_n} \tag{1}$$

As we are dealing with multiple optimization objectives, we prioritize them in the formal problem statement below. Let I^* denote the old placement matrix, and I denote the new placement matrix.

$$(i) \quad \text{maximize} \quad \sum_{m\in\mathcal{M}} \sum_{n\in\mathcal{N}} L_{m,n} \tag{2}$$

$$(ii) \quad \text{minimize} \quad \sum_{m\in\mathcal{M}} \sum_{n\in\mathcal{N}} |I_{m,n} - I^*_{m,n}| \tag{3}$$

$$(iii) \quad \text{minimize} \quad \sum_{n\in\mathcal{N}} \left| \frac{\sum_{m\in\mathcal{M}} L_{m,n}}{\Omega_n} - \rho \right| \tag{4}$$

such that

$$\forall m \in \mathcal{M},\ \forall n \in \mathcal{N} \qquad I_{m,n} = 0 \ \text{ or } \ I_{m,n} = 1 \tag{5}$$

$$\forall m \in \mathcal{M},\ \forall n \in \mathcal{N} \qquad R_{m,n} = 0 \Rightarrow I_{m,n} = 0 \tag{6}$$

$$\forall m \in \mathcal{M},\ \forall n \in \mathcal{N} \qquad I_{m,n} = 0 \Rightarrow L_{m,n} = 0 \tag{7}$$

$$\forall m \in \mathcal{M},\ \forall n \in \mathcal{N} \qquad L_{m,n} \geq 0 \tag{8}$$

$$\forall n \in \mathcal{N} \qquad \sum_{m\in\mathcal{M}} \gamma_m I_{m,n} \leq \Gamma_n \tag{9}$$

$$\forall n \in \mathcal{N} \qquad \sum_{m\in\mathcal{M}} L_{m,n} \leq \Omega_n \tag{10}$$

$$\forall m \in \mathcal{M} \qquad \sum_{n\in\mathcal{N}} L_{m,n} \leq \omega_m \tag{11}$$

This problem is a variant of the Class Constrained Multiple-Knapsack problem [12, 13]. It differs from the prior formulation mainly in that it also minimizes the number of placement starts and stops. This problem is NP hard. We will present an online approximation algorithm for solving it.

3. THE PLACEMENT ALGORITHM

This section describes our placement algorithm. Before presenting its details, we first give a high-level description of the algorithm, a definition of terms, and the key ideas behind the algorithm.

Our algorithm repeatedly and incrementally optimizes the placement solution in multiple rounds. In each round, it first computes the maximum total application demand that can be satisfied by the current placement solution. The algorithm quits if all the application demands are satisfied. Otherwise, it shifts load across machines (without placement changes), and then considers stopping unproductive application instances and starting more useful ones in order to increase the total satisfied application demand. The load-shifting step before the placement-changing step is critical as it dramatically simplifies subsequent placement changes. Note that, in the algorithm description, "placement change", "application start/stop", and "load shifting" are all hypothetical. The real placement changes are executed after the placement algorithm terminates.

3.1 Definition of Terms

A machine is *fully utilized* if its residual (i.e., unused) CPU capacity is zero ($\Omega^*_n = 0$); otherwise, it is *underutilized.* An application instance is fully utilized if it runs on a fully utilized machine. An instance of application m running on an underutilized machine n is *completely idle* if it has no load ($L_{m,n}$=0); otherwise, it is underutilized. The load of an underutilized instance of application m can be increased if application m has a positive residual (i.e., unsatisfied) CPU demand ($\omega^*_m > 0$).

$\mathcal{N}$	The set of machines.
n	One machine in the set $\mathcal{N}$.
$\mathcal{M}$	The set of applications.
m	One application in the set $\mathcal{M}$.
R	The placement restriction matrix. $R_{m,n} = 1$ if application m can run on machine n; $R_{m,n} = 0$ otherwise.
I	The placement matrix. $I_{m,n} = 1$ if application m is running on machine n; $I_{m,n} = 0$ otherwise.
L	The load distribution matrix. $L_{m,n}$ is the CPU cycles per second allocated on machine n for application m. L is an output of the placement algorithm; it is not measured from the running system.
Γ_n	The memory capacity of machine n.
Ω_n	The CPU capacity of machine n.
γ_m	The memory demand of application m, i.e., the memory needed to run one instance of application m.
ω_m	The CPU demand of application m, i.e., the total CPU cycles per second needed for application m throughout the entire system.
ω^*_m	The residual CPU demand of application m, i.e., the demand not satisfied by the load distribution matrix L: $\omega^*_m = \omega_m - \sum_{n\in\mathcal{N}} L_{m,n}$.
Ω^*_n	The residual CPU capacity of machine n, i.e., the CPU capacity not consumed by the applications running on machine n: $\Omega^*_n = \Omega_n - \sum_{m\in\mathcal{M}} L_{m,n}$.
Γ^*_n	The residual memory capacity of machine n, i.e., the memory not consumed by the busy applications ($L_{m,n}$>0) running on machine n: $\Gamma^*_n = \Gamma_n - \sum_{m:L_{m,n}>0} \gamma_m$.

Table 1: Symbols used in the placement algorithm.

The *CPU-memory ratio* of a machine n is defined as its CPU capacity divided by its memory capacity, i.e., Ω_n/Γ_n. Intuitively, it is harder to fully utilize the CPU of machines with a high CPU-memory ratio. The *load-memory ratio* of an instance of application m running on machine n is defined as the CPU load of this instance divided by its memory consumption, i.e., $L_{m,n}/\gamma_m$. Intuitively, application instances with a higher load-memory ratio are more "productive".

3.2 Key Ideas in Load Shifting

Figure 4 is the high-level pseudo code of our algorithm. The details will be explained later. The core of the `place()` function is a loop that incrementally optimizes the placement solution. Inside the loop, it first solves the max-flow problem [2] in Figure 3 to compute the maximum total demand $\hat{\omega}$ that can be satisfied by the current placement matrix I. Among many possible load distribution matrices L that can meet this maximum demand $\hat{\omega}$, we employ several load-shifting heuristics to find the one that makes later placement changes easier.

- We classify the running instances of an application into three categories: idle, underutilized, and fully utilized. The idle instances are preferred candidates to be shut down. We opt for leaving the fully utilized instances intact as they already make good contributions.

- Through proper load shifting, we can ensure that every application has at most one underutilized instance in the entire system. Reducing the number of underutilized instances simplifies the placement problem, because the strategy to handle idle instances and fully utilized instances are straightforward.
- We strive to co-locate residual memory and residual CPU on the same machines so that these resources can be used to start new application instances. For example, if one machine has only residual CPU while another machine has only residual memory, neither of them can accept new applications.
- We strive to make idle application instances appear on machines with relatively more residual memory. By shutting down the idle instances, more memory will become available for hosting applications that require a large amount of memory.

3.3 Key Ideas in Placement Changing

The load-shifting subroutine in Figure 4 prepares the load distribution in a way that makes later placement changes easier. The placement-changing subroutine further employs several heuristics to increase the total satisfied application demand, to reduce placement changes, and to reduce computation time.

- The algorithm walks through the underutilized machines sequentially and makes placement changes to them one by one in an isolated fashion. When working on a machine n, the algorithm is only concerned with the sate of machine n and the residual application demands. This isolation drastically reduces the computation time.
- The isolation of machines, however, may lead to inferior placement solutions. We address this problem by alternately executing the load-shifting subroutine and the placement-changing subroutine for multiple rounds. As a result, the residual application demands released from the application instances stopped in the previous round now have the chance of being allocated to other machines in the later rounds.
- When sequentially walking through the underutilized machines, the algorithm first considers machines with a relatively high CPU-memory ratio (see the definition in Section 3.1). As it is harder to fully utilize the CPU of these machines, we prefer to process them first when we still have abundant choices.
- When choosing applications to run on a machine, the algorithm tries to find a combination of applications that lead to the highest CPU utilization of this machine. It prefers to stop "unproductive" running application instances with a relatively low load-memory ratio to accommodate new application instances.
- To reduce placement changes, the algorithm does not allow stopping application instances that already deliver a "sufficiently high" load. We refer to these instances as *pinned instances*. The intuition is that, even if we stop these instances on their current hosting machines, it is likely that we will start instances of the same applications on other machines. Our algorithm dynamically computes the pinning threshold for each application.

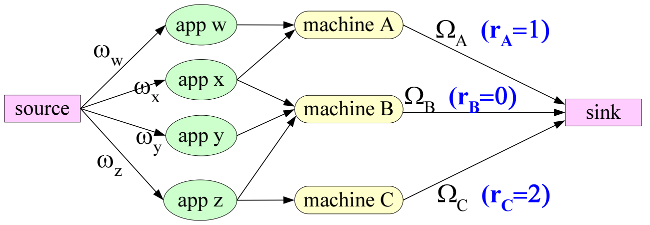

Figure 3: This figure shows two network flow problems. (1) When the link costs (i.e., r_A=1, r_B=0, and r_C=2) are not used, this figure is an example of the max-flow problem whose solution gives the maximum total demand that can be satisfied by the current placement matrix I. (2) When the link costs are used, it is an example of the min-cost max-flow problem solved by the load-shifting subroutine to compute a load distribution that makes later placement changes easier.

Below, we present in detail the load-shifting subroutine, the placement-changing subroutine, and the full placement algorithm that utilizes these two subroutines.

3.4 The Load-Shifting Subroutine

Given the current application demands, the placement algorithm solves a max-flow problem [2] to derive the maximum total demand that can be satisfied by the current placement matrix I. Figure 3 is an example of this max-flow problem, in which we consider four applications (w, x, y, and z) and three machines (A, B, and C). Each application is represented as a node in the graph. Each machine is also represented as a node. In addition, there are a source node and a sink node. The source node has an outgoing link to each application m, and the capacity of this link is the CPU demand of the application (ω_m). Each machine n has an outgoing link to the sink node, and the capacity of this link is the CPU capacity of the machine (Ω_n). The last set of links are between the applications and the machines that currently run those applications. The capacity of these links is unlimited. In Figure 3, application x currently runs on machines A and B. Therefore, x has two outgoing links: $x \rightarrow A$ and $x \rightarrow B$.

When load distribution is formulated as this max-flow problem, the maximum volume of flows going from the source node to the sink node is the maximum total demand $\hat{\omega}$ that can be satisfied by the current placement matrix I. (Recall that $I_{m,n} = 1$ if application m is running on machine n. See Table 1 for the notations.) If all application demands are satisfied, no placement changes are needed. Otherwise, we make placement changes in order to satisfy more application demands. Before doing so, we first adjust the load distribution matrix L produced by solving the max-flow problem in Figure 3. (Recall that $L_{m,n}$ is the CPU cycles per second allocated on machine n for application m.) The goal of load shifting is to achieve the effects described in Section 3.2, e.g., co-locating residual CPU and residual memory on the same set of machines, and ensuring that each application has at most one underutilized instance in the entire system.

The task of load shifting is accomplished by solving the min-cost max-flow problem [2] in Figure 3. We sort all the machines in increasing order of residual memory capacity Γ_n^*, and associate each machine n with a rank r_n that reflects its position in this sorted list. The machine with rank 0 has the least amount of residual memory. In Figure 3, the link

```
function place ()
{
  for (i = 0; i < K; i++) {   // K=10 by default.
      calc_max_demand_satisfied_by_current_placement ();
      if (all_demands_satisfied) break_out_of_the_loop;
      load_shifting ();     // No placement changes here.
      placement_changing (pin_app=false);
      placement_changing (pin_app=true);
      choose_the_better_one_as_the_solution; // Pin or not.
      if (no_improvement) break_out_of_the_loop;
  }
  balance_load_across_machines ();
}
```

```
function placement_changing (boolean pin_app)
{
 //-----------------------------------------------outermost loop----
 // Change the placement on one machine at a time.
 for (all_underutilized_machines_n) {
     if (pin_app==true) identify_pinned_app_instances();
     // Suppose machine n currently runs c not-pinned
     // app instances (M1, M2, ..., Mc) sorted in
     // increasing order of load-memory ratio.

     //----------------------------------------intermediate loop----
     for (j=0; j < c; j++) {
         if (j > 0) stop_j_apps_on_machine_n (M1,M2,...,Mj);

         //---------------------------------------innermost loop----
         // Find apps to consume n's residual resources
         // that become available after stopping the j apps.
         for (all apps x with a positive residual demand) {
             if (app_x_fits_on_machine_n) start_x_on_n ();
         }
         if (is_the_best_solution_for_machine_n_so_far)
             record_it();
}}}
```

Figure 4: High-level pseudo code of our algorithm.

between a machine n and the sink node is associated with the cost r_n. The cost of all the other links is zero, which is not shown in the figure for brevity. In this example, machine C has more residual memory than machine A, and machine A has more residual memory that machine B. Therefore, the links between the machines and the sink node have costs $r_B = 0$, $r_A = 1$, and $r_C = 2$, respectively.

The load distribution matrix L produced by solving the min-cost max-flow problem in Figure 3 possesses the following good properties that make later placement changes easier: (1) An application has at most one underutilized instance in the entire system. (2) Residual memory and residual CPU are likely to co-locate on the same set of machines. (3) The idle application instances appear on the machines with relatively more residual memory.

Theorem 1. *In the load distribution matrix L produced by solving the min-cost max-flow problem in Figure 3, each application has at most one underutilized instance in the entire system.*

Proof: We prove this by contradiction. Suppose there are two underutilized instances of the same application running on two underutilized machines A and B, respectively. Without loss of generality, we assume that machine A has less residual memory than machine B, i.e., $r_A < r_B$. Because machine A still has residual CPU capacity and the cost of using machine A is lower than that of using machine B ($r_A < r_B$), the min-cost max-flow algorithm can further reduce the total cost of the maximum flow by moving load from machine B to machine A, which contradicts with the fact that the current solution given by the min-cost max-flow algorithm already has the lowest cost. □

Theorem 2. *In the load distribution matrix L produced by solving the min-cost max-flow problem in Figure 3, if application m has one underutilized instance running on machine n, then (1) application m's idle instances must run on machines whose residual memory is larger than or equal to that of machine n; and (2) application m's fully utilized instances must run on machines whose residual memory is smaller than or equal to that of machine n.*

Proof: The proof is similar to that for Theorem 1. □

3.5 The Placement-Changing Subroutine

The placement-changing subroutine takes as input the current placement matrix I, the load distribution matrix L generated by the load-shifting subroutine, and the residual application demands not satisfied by L. It tries to increase the total satisfied application demand by making placement changes, for instance, stopping "unproductive" application instances and starting useful ones.

The main structure of the placement-changing subroutine consists of three nested loops (see Figure 4). The *outermost loop* iterates over the machines and asks the *intermediate loop* to generate a placement solution for one machine n at a time. Suppose machine n currently runs c not-pinned application instances $(M_1, M_2, \cdots, M_c)$ sorted in increasing order of load-memory ratio (see the definition in Section 3.1). The intermediate loop iterates over a variable j $(0 \le j \le c)$. In iteration j, it stops on machine n the j applications $(M_1, M_2, \cdots, M_j)$ while keeping the other running applications intact, and then asks the *innermost loop* to find appropriate applications to consume machine n's residual resources. The innermost loop walks through the residual applications, and identifies those that can fit on machine n. As the intermediate loop varies the number of stopped applications from 0 to c, it collects $c + 1$ different placement solutions for machine n, among which it picks the best one as the final solution.

Below, we describe the three nested loops in detail.

The Outermost Loop. Before entering the outermost loop, the algorithm first computes the residual CPU demand of each application. We refer to the applications with a positive residual CPU demand (i.e., $\omega_m^* > 0$) as *residual applications.* The algorithm inserts all the residual applications into a right-threaded AVL tree called *residual_app_tree*. The applications in the tree are sorted in decreasing order of residual demand. As the algorithm progresses, the residual demand of applications may change, and the tree is updated accordingly. The algorithm also keeps track of the minimum memory requirement γ_{min} of applications in the tree,

$$\gamma_{min} = \min_{m \,\in\, residual\_app\_tree} \gamma_m, \tag{12}$$

where γ_m is the memory needed to run one instance of application m. The algorithm uses γ_{min} to speed up the computation in the innermost loop. If a machine n's residual memory Γ_n^* is smaller than γ_{min}, the algorithm can immediately infer that this machine cannot accept any applications in the *residual_app_tree.*

The algorithm excludes fully utilized machines from the consideration of placement changes, and sorts the underutilized machines in decreasing order of CPU-memory ratio. Starting from the machine with the highest CPU-memory ratio, it enumerates each underutilized machine, and asks the intermediate loop to compute a placement solution for each machine. Because it is harder to fully utilize the CPU of machines with a high CPU-memory ratio, we prefer to process them first when we still have abundant choices.

The Intermediate Loop. Taking as input the *residual_app_tree* and a machine n given by the outermost loop, the intermediate loop computes a placement solution for machine n. Suppose machine n currently runs c not-pinned application instances. (Application instance pinning will be discussed later.) We can stop a subset of the c applications, and use the residual resources to run other applications. In total, there are 2^c cases to consider. We use a heuristic to reduce this number to $c + 1$. Intuitively, we prefer to stop the less "productive" application instances, i.e., those with a low load-memory ratio ($L_{m,n}/\gamma_m$).

The algorithm sorts the not-pinned application instances on machine n in increasing order of load-memory ratio. Let $(M_1, M_2, \cdots, M_c)$ denote this sorted list. The intermediate loop iterates over a variable j $(0 \leq j \leq c)$. In iteration j, it stops on machine n the j applications $(M_1, M_2, \cdots, M_j)$ while keeping the other running applications intact, and then asks the innermost loop to find appropriate applications to consume machine n's residual resources that become available after stopping the j applications. As the intermediate loop varies the number of stopped applications from 0 to c, it collects $c + 1$ placement solutions, among which it picks as the final solution the one that leads to the highest CPU utilization of machine n.

The Innermost Loop. The intermediate loop changes the number of applications to stop. The innermost loop uses machine n's residual resources to run some residual applications. Recall that the *residual_app_tree* is sorted in decreasing order of residual CPU demand. The innermost loop iterates over the residual applications, starting from the one with the largest amount of residual demand. When an application m is under consideration, the algorithm checks two conditions: (1) whether the restriction matrix R allows application m to run on machine n, and (2) whether machine n has sufficient residual memory to host application m (i.e., $\gamma_m \leq \Gamma^*_n$). If both conditions are satisfied, it places application m on machine n, and assigns as much load as possible to this instance until either machine n's CPU is fully utilized or application m has no residual demand. After this allocation, m's residual demand changes, and the *residual_app_tree* is updated accordingly.

The innermost loop iterates over the residual applications until either (1) all the residual applications have been considered once; or (2) machine n's CPU becomes fully utilized; or (3) machine n's residual memory is insufficient to host any residual application (i.e., $\Gamma^*_n < \gamma_{min}$, see Equation 12).

3.6 The Full Placement Algorithm

The full placement algorithm is outlined in Figure 4. It incrementally optimizes the placement solution in multiple rounds. In each round, it first invokes the load-shifting subroutine and then invokes the placement-changing subroutine. It repeats for up to K rounds, but quits earlier it sees no improvement in the total satisfied application demand after one round of execution. The last step of the algorithm balances the load across machines. We reuse the load-balancing component from an exiting algorithm [6], but omit its detail here. Intuitively, it moves the new application instances between machines to balance the load, while keeping the total satisfied demand and the number of placement changes the same.

The placement algorithm deals with multiple optimization objectives. In addition to maximizing the total satisfied demand, it also strives to minimize placement changes, because they disturb the running system and waste CPU cycles. Our heuristic for reducing unnecessary placement changes is not to stop application instances whose load (in the load distribution matrix L) is above certain threshold. We refer to them as *pinned instances*. The intuition is that, even if we stop these "productive" instances on their current hosting machines, it is likely that we will start instances of the same applications on other machines.

Each application m has its own pinning threshold ω^{pin}_m. The value of this threshold is crucial. If it is too low, the algorithm may introduce many unnecessary placement changes. If it is too high, the total satisfied demand may be low due to insufficient placement changes. The algorithm dynamically computes the pinning threshold for each application using information gathered in a dry-run invocation to the placement-changing subroutine. The dry run pins no application instances. After the dry run, the algorithm makes a second invocation to the placement-changing subroutine, and requires pinning the application instances whose load is higher than or equal to the pinning threshold of the corresponding application, i.e., $L_{m,n} \geq \omega^{pin}_m$. The dry run and the second invocation use exactly the same inputs: the matrices I and L produced by the load-shifting subroutine. Between the two placement solutions produced by the dry run and the second invocation, the algorithm picks as the final solution the one that has a higher total satisfied demand. If the total satisfied demands are equal, it picks the one that has less placement changes.

Next, we describe how to compute the pinning threshold ω^{pin}_m using information gathered in the dry run. Intuitively, if the dry run starts a new application instance, then we should not stop any instance of the same application whose load is higher than or equal to that of the new instance. This is because the new instance's load is considered sufficiently high by the dry run so that it is even worthwhile to start a new instance. Let ω^{new}_m denote the minimum load assigned to a new instance of application m started in the dry run.

$$\omega^{new}_m = \min_{I_{m,n} \in \{\text{new instances of app m started in the dry run}\}} \{L_{m,n} \text{ after the dry run}\} \tag{13}$$

Here $I_{m,n}$ represents a new instance of application m started on machine n in the dry run. $L_{m,n}$ is the load of this instance. In addition, the pinning threshold also depends the largest residual application demand ω^*_{max} not satisfied in the dry run.

$$\omega^*_{max} = \max_{m \in \{\text{residual\_app\_tree\_after\_the\_dry\_run}\}} \omega^*_m \tag{14}$$

Here ω^*_m is the residual demand of application m after the dry run. We should not stop the application instances whose load is higher than or equal to ω^*_{max}. If we stop these instances, they would immediately become the applications that we try to find a place to run. The pinning threshold for application m is computed as follows.

$$\omega^{pin}_m = \max\left(1,\ \min\left(\omega^*_{max},\ \omega^{new}_m\right)\right) \tag{15}$$

Because we do not want to pin completely idle application instances, Equation 15 stipulates that the pinning threshold ω_m^{pin} should be at least one CPU cycle per second.

3.7 Complexity and Practical Issues

The computation time of our placement algorithm is dominated by the time spent on solving the max-flow problem and the min-cost max-flow problem in Figure 3. One efficient algorithm for solving the max-flow problem is the highest-label preflow-push algorithm [2], whose complexity is $O(s^2\sqrt{t})$, where s is the number of nodes in the graph, and t is the number of edges in the graph. One efficient algorithm for solving the min-cost flow problem is the enhanced capacity scaling algorithm [2], whose complexity is $O((s \log t)(s + t \log t))$. Let N denote the number machines. Due to various resource constraints, the number of applications that a machine can run is bounded by a constant. Therefore, in the network flow graph, both the number s of nodes and the number t of edges are bounded by $O(N)$. The total number of application instances in the entire system is also bounded by $O(N)$.

Under these assumptions, the complexity of our placement algorithm is $O(N^{2.5})$. In contrast, under the same assumptions, the complexity of the state-of-the-art placement algorithm proposed by Kimbrel et al. [6, 8] is $O(N^{3.5})$. This difference in complexity is the reason why our algorithm can do online computation for systems with thousands of machines, while their algorithm can scale to at most a few hundred machines (see the results in Section 4).

For the sake of brevity, our formulation of the placement problem and our algorithm description omit several practical issues. For instance, an administrator may impose restrictions on the minimum/maximum number of instances of a given application allowed in the entire system. It is also possible that multiple instances of the same application need to run on a single machine because, for example, one instance of the application cannot utilize all the CPU power of the machine due to internal bottlenecks in the application. Moreover, the optimization objective can be maximizing certain utility function instead of maximizing the total satisfied application demand. Finally, the actual start and stop of application instances should be carefully coordinated to implement a fast transition and avoid stopping all instances of an application at the same time. One version of our algorithm adopted in a leading commercial middleware product [1] addresses these practical issues, but we omit a detailed discussion here due to space limitations.

4. EXPERIMENTAL RESULTS

This section studies the performance of our placement algorithm, and compares it with the state-of-the-art algorithm [6, 8] proposed by Kimbrel et al. We use this algorithm as the baseline because the evaluation [8] shows that it outperforms two variants of another state-of-the-art algorithm [12, 13]. For brevity, we simply refer to our algorithm and the algorithm proposed by Kimbrel et al. as the "new" and "old" algorithms, respectively. Because the two algorithms use the same technique for load balancing, we omit its results here, and refer interested readers to [6].

Both algorithms have been implemented in a leading commercial middleware product [1]. In this paper, we evaluate only the placement controller component (as opposed to the entire middleware), by feeding a wide variety of workloads directly to the placement algorithms. Some of the workloads are representative of real-world traffic (e.g., application demands that follow a power-law distribution), while others are extreme configurations for stress test. In all the experiments, we assume no placement restriction for applications, i.e., $\forall m \in \mathcal{M}\ \forall n \in \mathcal{N}\ R_{m,n} = 1$.

The placement controller works in *cycles*. At the beginning of a cycle, the placement algorithm is given a set of machines, a set of applications, the current demands of the applications, and the placement matrix I left from the previous cycle. The placement algorithm then produces a new placement matrix I and a load distribution matrix L, which are used for performance evaluation. The evaluation metrics include the *execution time*, the number of *placement changes* (i.e., application starts/stops), and the *demand satisfaction* (i.e., the fraction of the total application demands satisfied by the placement solution: $\frac{\sum_{m\in\mathcal{M}}\sum_{n\in\mathcal{N}} L_{m,n}}{\sum_{m\in\mathcal{M}} \omega_m}$).

In the experiments, the configuration of machines is uniformly distributed over the set {1GB:1GHz, 2GB:1.6GHz, 3GB:2.4GHz, 4GB:3GHz}, where the first number is memory capacity and the second number is CPU speed. The memory requirement of applications is uniformly distributed over the set {0.4GB, 0.8GB, 1.2GB, 1.6GB}. A *system configuration* includes a fixed set of machines and applications. All the reported data are averaged over the results on 100 randomly generated system configurations. For each configuration, the placement algorithm executes for 11 cycles (including an initial placement) under changing application demands. Hence, each reported data point is averaged over 1,000 placement results (excluding the initial placement).

4.1 Problem Hardness

We compare the algorithms while varying the size of the placement problem (i.e., the number of machines and applications) and the hardness of the problem. The hardness is defined from four dimensions: CPU load, memory load, application CPU demand distribution, and demand variability.

CPU Load Factor L_{cpu}. It is defined as the ratio between the total CPU demand and the total CPU capacity, $L_{cpu} = \frac{\sum_{m\in\mathcal{M}} \omega_m}{\sum_{n\in\mathcal{N}} \Omega_n}$, where ω_m is the CPU demand for application m, and Ω_n is the CPU capacity of machine n.

Memory Load Factor L_{mem}. Let $\overline{\gamma}$ denote the average memory requirement of applications, and $\overline{\Gamma}$ denote the average memory capacity of machines. The average number of application instances that can be hosted on N machines is $\frac{\overline{\Gamma}}{\overline{\gamma}}N$. The memory load factor is defined as $L_{mem} = M/\frac{\overline{\Gamma}}{\overline{\gamma}}N = \frac{M\overline{\gamma}}{N\overline{\Gamma}}$, where M is the number of applications. Note that $0 \le L_{mem} \le 1$ and the problem is most difficult when $L_{mem} = 1$.

In the experiments, we vary the CPU load factor L_{cpu}, the memory load factor L_{mem}, and the number N of machines. The number M of applications is configured according to N and L_{mem}: $M = \frac{\overline{\Gamma}}{\overline{\gamma}}NL_{mem} = 2.5NL_{mem}$.

Application Demand Distribution. Once the CPU load factor L_{cpu} and the total CPU capacity $\Omega = \sum_{n\in\mathcal{N}} \Omega_n$ are determined, the total application CPU demand is set to $\Omega \cdot L_{cpu}$. We experiment with two different ways of partitioning this total demand among applications. With the *uniform distribution*, each application's initial demand is generated uniformly at random from the range $[0, 1]$. With the *power-law distribution*, application m's initial demand is set to $j^{-\alpha}$, where $\alpha = 2.16$ and j is application m's rank in a random permutation of all the applications. For both uniform and power-law distributions, the demand of each application is normalized proportionally to the total application demand.

Application Demand Variability. Given a system configuration, the placement algorithm executes for 11 cycles. The application demands in the first cycle follow either a *uniform* distribution or a *power-law* distribution. Starting from the second cycle, the application demands change from cycle to cycle. The placement problem is harder to solve if this change is drastic. We experiment with four different demand changing patterns. With the *vary-all-apps* pattern, each application's demand changes randomly and independently within a ±20% range of its initial demand. With the *vary-two-apps* pattern, we keep the demands of all the applications constant except for the two applications with the largest demands. The sum of these two applications' demands is kept constant, but the allocation between them randomly changes 10% from cycle to cycle. With the *reset-all-apps* pattern, the demands in two consecutive cycles are independent of each other. This "unrealistic" pattern represents the most extreme demand change. With the *add-apps* pattern, the placement algorithm executes for M placement cycles, where M is the number of applications. Starting with an empty, idle system, the demand for one new application is introduced into the system in every cycle.

Below, we concisely represent the hardness of a placement problem as (L_{cpu}, L_{mem}, "demand-distribution", "demand-variability"), e.g., (L_{cpu}=0.9, L_{mem}=0.4, power-law-apps, vary-all-apps).

4.2 Performance Results

To choose the best algorithm for the commercial product [1], we compared more than a dozen different variants of the placement algorithms (including some variants not described in this paper), and generated thousands of performance graphs. Due to space limitations, we present in this paper only some representative results. Overall, when the placement problem is easy to solve (i.e., the system has abundant resources), both algorithms can satisfy almost all the application demands. However, when the placement problem is hard, the new algorithm significantly and consistently outperforms the old algorithm.

Figure 5 shows the execution time of the two algorithms for the setting (L_{cpu}=0.99, L_{mem}=1, uniform-apps, reset-all-apps). For hard problems, the execution time of the new algorithm is almost negligible compared with that of the old algorithm. As an online controller, the old algorithm can only scale to at most a few hundred machines and applications, while the new algorithm can scale to thousands of machines and applications.

Figure 6 reports results on the scalability of the new algorithm. We vary the number of machines from 100 to 7,000 and the number of applications from 250 to 17,500. The new algorithm takes less than 30 seconds to solve the difficult 7,000-machine, 17,500-application placement problem under the tight resource constraints and the extreme demand changes from cycle to cycle. This execution time is measured on a 1.8GHz Pentium IV machine. The demand satisfaction in Figure 6 stays around 0.946 as the system size increases, showing that the new algorithm can produce high-quality solutions regardless of the problem size. The number of placement changes is high because this "reset-all-apps" configuration for stress test unrealistically changes application demands between cycles in an extreme manner.

The experiment in Figure 7 introduces the demand for one new application into a 100-machine system in every placement cycle. This figure reports the number of placement changes occurred when adding the i-th application rather than the aggregated number of placement changes occurred before adding the (i+1)-th application. Because the resources are not very tight (L_{cpu}=0.9 and L_{mem}=0.4), both algorithms can satisfy all the demands, but the old algorithm introduces a much larger number of placement changes. For example, to handle the added demand for the last application, the old algorithm makes 51.3 placement changes on average, while the new algorithm makes only 1.6 placement changes on average. In a real system, this difference can have dramatic impact on the whole system performance.

The experiments in Figures 8, 9, and 10 use different combinations of CPU load factor (H_{cpu}=0.99 or 0.6), memory load factor (H_{mem}=1 or 0.6), demand distribution (uniform or power-law), and demand variability (vary-two-apps, vary-all-apps, or reset-all-apps). Under all these settings, the new algorithm consistently outperforms the old algorithm: it improves demand satisfaction by up to 25%, and reduces placement changes by up to 90%.

The experiment in Figure 11 varies the memory load factor L_{mem} from 0.1 to 1 for a 100-machine system, while fixing the CPU load factor L_{cpu}=0.9. As the hardness of the problem increases, the demand satisfaction of the old algorithm drops faster than that of the new algorithm. More importantly, the number of placement changes in the old algorithm increases dramatically. The experiment in Figure 12 varies the CPU load factor L_{cpu} from 0.1 to 1 for a 100-machine system, while fixing the memory load factor L_{mem}=0.4. The old algorithm and the new algorithm have similar performance when the CPU load factor is below 0.8. However, when the CPU load factor is between 0.8 and 0.9, the number of placement changes in the old algorithm increases almost exponentially. The situation could get even worse as the CPU load factor further approaches 1. To deal with this pathological case, the improved version [6] of the old algorithm (the version used in the comparison) added a 90% load reduction heuristic—whenever the CPU load factor is above 0.9, the old algorithm first (artificially) reduces it to 0.9 by scaling all the application demands proportionally, and then executes the core of the algorithm on the reduced demands. This heuristic helps the old algorithm to reduce placement changes, but it also decreases the demand satisfaction (see the dip in Figure 12). In contrast, the new algorithm achieves a better performance even without using such hard-coded rules to handle the corner cases.

In summary, the new algorithm significantly and consistently outperforms the old algorithm in all three aspects: execution time, demand satisfaction, and placement changes. The new algorithm's ability to achieve a higher demand satisfaction is mainly owing to its load-shifting heuristics and the strategy that first does placement changes to the machines with a high CPU-memory ratio. The new algorithm's fast speed is mainly owing to the strategy that does placement changes to machines one by one in an isolated fashion. Two heuristics in the new algorithm help reduce placement changes: application instance pinning and machine isolation. For hard placement problems, the old algorithm may simultaneously free a large number of application instances on different machines, and then try to place them, which may produce solutions that simply shuffle application instances across machines (see Figure 7). In contrast, due to the new algorithm's machine isolation strategy, it never simultaneously frees application instances running on different machines and hence avoids unnecessary shuffling.

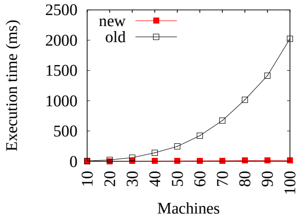

Figure 5: Execution time with configuration (L_{cpu}=0.99, L_{mem}=1, uniform-apps, reset-all-apps).

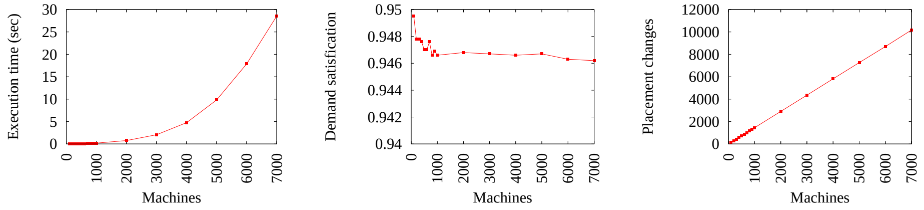

Figure 6: Scalability of the new algorithm with configuration (L_{cpu}=0.99, L_{mem}=1, uniform-app, reset-all-apps).

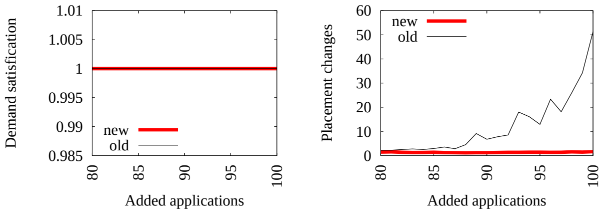

Figure 7: Demand satisfaction and placement changes with configuration: (L_{cpu}=0.9, L_{mem}=0.4, uniform-app, add-apps), 100 machines, and 100 applications.

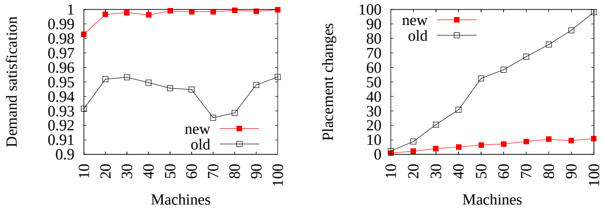

Figure 8: Demand satisfaction and placement changes with configuration (L_{cpu}=0.99, L_{mem}=1, uniform-app, vary-two-apps).

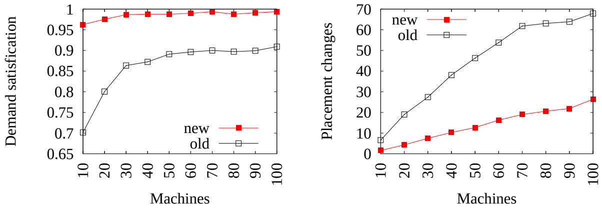

Figure 9: Demand satisfaction and placement changes with configuration (L_{cpu}=0.99, L_{mem}=1, power-law-app, vary-all-apps).

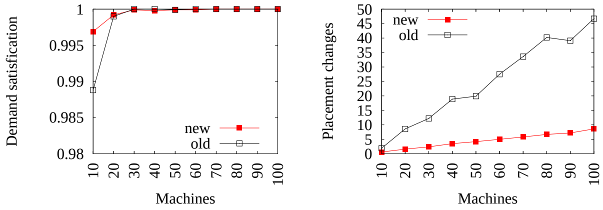

Figure 10: Demand satisfaction and placement changes with configuration (L_{cpu}=0.9, L_{mem}=0.6, uniform-app, vary-all-apps).

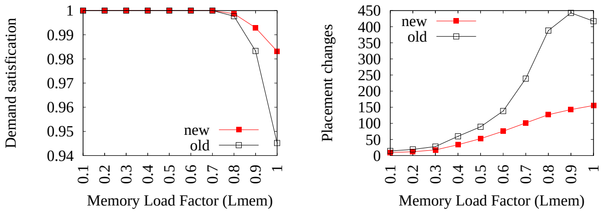

Figure 11: Vary L_{mem} from 0.1 to 1. Configuration: (L_{cpu}=0.9, L_{mem}=X, uniform-apps, reset-all-apps), 100 machines.

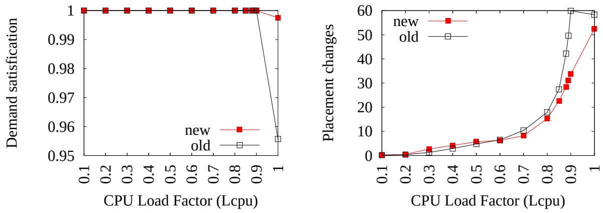

Figure 12: Vary L_{cpu} from 0.1 to 1. Configuration: (L_{cpu}=X, L_{mem}=0.4, uniform-apps, reset-all-apps), 100 machines.

5. RELATED WORK

The problem of dynamic application placement in response to changes in application demands have been studied before. The algorithm proposed by Kimbrel et al. [6, 8] is the closest to our work. The comparison in Section 4 shows that our algorithm significantly and consistently outperforms this algorithm. The biggest difference between the two algorithms is that, in the previous algorithm, the placement decisions for individual machines are not isolated—stopping one application instance on one machine may lead to reconsideration of the placement decisions for all the other machines.

A popular approach to dynamic server provisioning is to allocate full machines to applications as needed [3], which does not allow applications to share machines. In contrast, our placement controller allows this sharing and is optimized for it. The algorithm proposed by Urgaonkar et al. [17] allows applications to share machines, but it does not dynamically change the number of instances of an application, does not try to minimize placement changes, and only considers a single bottleneck resource.

Placement problems have also been studied in the optimization literature, including bin packing, multiple knapsack, and multi-dimensional knapsack problems [7]. The special case of our problem with uniform memory requirements was studied in [12, 13], and some approximation algorithms were proposed. These algorithms have been shown to be inferior to the algorithm proposed by Kimbrel et al. [8]. Our algorithm further significantly outperforms an improved version [6] of the algorithm proposed by Kimbrel et al.

One limitation of our algorithm is that it makes no attempt to co-locate on the same machine the set of applications that have high-volume internal communication. This issue has been studied before [5, 15], but it still remains a challenge to design for commercial product a fully automated algorithm that does not rely on manual offline profling.

6. CONCLUSION

In this paper, we proposed an application placement controller that dynamically starts and stops application instances in response to changes in application demands. It allows multiple applications to share a single machine. Under multiple resource constraints, it strives to maximize the total satisfied application demand, to minimize the number of application starts and stops, and to balance the load across machines. It significantly and consistently outperforms the existing state-of-the-art algorithm [6, 8]. Compared with [6, 8], for systems with 100 machines or less, our algorithm is up to 134 times faster, reduces unnecessary application starts and stops by up to 97%, and produces solutions that satisfy up to 25% more application demands.

We believe that our algorithm is the first online algorithm that, under multiple tight resource constraints, can efficiently produce high-quality solutions for hard placement problems with thousands of machines and thousands of applications. This scalability is crucial for dynamic resource provisioning in large-scale enterprise data centers. The outstanding performance of our algorithm stems from our novel optimization techniques such as application pinning, load shifting, and machine isolation. Our algorithm has been implemented and adopted in a leading commercial product [1].

7. REFERENCES

[1] WebSphere Extended Deployment, http://www-306.ibm.com/software/webservers/appserv/extend/.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, editors. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, New Jersey, 1993. ISBN 1000499012.

[3] 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 *Proceedings of the International Symposium on Integrated Network Management*, pages 14–18, Seattle, WA, May 2001.

[4] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Symposium on Operating Systems Principles (SOSP)*, 1997.

[5] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *OSDI*, 1999.

[6] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic Application Placement for Clustered Web Applications. In *the International World Wide Web Conference (WWW)*, May 2006.

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

[8] T. Kimbrel, M. Steinder, M. Sviridenko, and A. N. Tantawi. Dynamic Application Placement Under Service and Memory Constraints. In *International Workshop on Efficient and Experimental Algorithms*, 2005.

[9] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef. Performance management for cluster based web services. In *Proceedings of the International Symposium on Integrated Network Management*, 2003.

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

[11] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi. Dynamic Estimation of CPU Demand of Web Traffic. In *Proceedings of the First International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*, 2006.

[12] H. Shachnai and T. Tamir. Noah's bagels - some combinatorial aspects. In *Proc. 1st Int. Conf. on Fun with Algorithms*, 1998.

[13] H. Shachnai and T. Tamir. On two class-constrained versions of the multiple knapsack problem. *Algorithmica*, 29(3):442–467, 2001.

[14] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of OSDI*, 2002.

[15] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proc. of the Second USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[16] C. Tang, R. N. Chang, and E. So. A Distributed Service Management Infrastructure for Enterprise Data Centers Based on Peer-to-Peer Technology. In *Proc. the International Conference on Services Computing*, 2006. Winner of the Best Paper Award.

[17] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proc. of OSDI*, 2002.