Utility-based Placement of Dynamic Web Applications with Fairness Goals

David Carrera
Barcelona Supercomputing Center (BSC)
Technical University of Catalonia (UPC)
Barcelona, Spain
david.carrera@bsc.es

Malgorzata Steinder and Ian Whalley
IBM T.J. Watson Research Center
Hawthorne, NY 10532
steinder@us.ibm.com
inw@us.ibm.com

Jordi Torres and Eduard Ayguadé
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
jordi.torres@bsc.es
eduard.ayguade@bsc.es

***Abstract*—We study the problem of dynamic resource allocation to clustered Web applications. We extend application server middleware with the ability to automatically decide the size of application clusters and their placement on physical machines. Unlike existing solutions, which focus on maximizing resource utilization and may unfairly treat some applications, the approach introduced in this paper considers the satisfaction of each application with a particular resource allocation and attempts to at least equally satisfy all applications. We model satisfaction using utility functions, mapping CPU resource allocation to the performance of an application relative to its objective. The demonstrated online placement technique aims at equalizing the utility value across all applications while also satisfying operational constraints, preventing the over-allocation of memory, and minimizing the number of placement changes. We have implemented our technique in a leading commercial middleware product. Using this real-life testbed and a simulation we demonstrate the benefit of the utility-driven technique as compared to other state-of-the-art techniques.**

I. Introduction

Modern-day data centers can host a large number of clustered web applications and comprise a large number of heterogeneous machines. To manage application performance, the data centers use admission control, flow control, load balancing, and application placement mechanisms, which are controlled using a variety of policies. Due to the increasing size and heterogeneity of datacenters, customers demand that these policies be fine grained, automatic, and dynamic. To fully automate performance management for such complex environments several challenging issues must be solved, which include understanding and modeling workload characteristics with respect to resource usage, workload forecasting, performance modeling, optimizing resource allocation, controlling the overhead of management actions, and observing operational policies that are not easily captured within a single optimization objective.

This paper studies the problem of dynamic application placement to fairly maximize application performance. We study this problem in the context of a larger system that achieves fully automatic management of web application performance by combining dynamic application placement with flow control, and load balancing.

The system leverages properties of modern application server middleware [1] that provides transparent application replication via clustering and request routing, session and transaction state management, and application server quiesce mechanisms. Thanks of the existence of these services and their resilience to dynamic configuration changes, we can concentrate on the problem on dynamic replica placement without having to explicitly address these critical issues. The system also takes advantage of recent advances in fields of resource usage profiling for web applications [2] and performance modeling and overload protection [3].

The application placement component is an online controller that periodically adjusts application placement based on application resource requirements while trying to maximize certain objective functions and observe some operational policies. Prior approaches to the problem [4], [5], [6], [7], including our prior research on this topic [8] express resource requirements directly, in terms of actual capacity requirement, and attempt to maximize the sum of satisfied demand across all applications. As we show in this paper, such problem formulation leads to unfair treatment of some applications, and in some cases to application starvation. The technique proposed in this paper expresses resource requirement in the form of a utility function which encodes application satisfaction from a particular allocation. The utility function is derived based on observed workload intensity, resource usage profile, and performance objective. The optimization objective is to maximize the minimum utility value among all applications. This results in resource allocation that provides fair treatment to applications but requires both more involved interactions among system components and more involved optimization techniques.

The primary contribution of this paper is an online placement technique which achieves fair resource allocation with respect to non-linear performance-based utility functions. We have implemented the proposed approach on top of a leading middleware platform. Using the implemented system as well as a randomized simulation study, we show the benefits of our approach compared to prior techniques.

This paper is structured as follows. In Section II we present the architecture of our resource management system. Sec-

tion III presents the algorithms used in the present evaluation. In Section IV we evaluate our system through experimentation. Related work is discussed in Section V. We conclude the paper in Section VI.

II. System architecture

Figure 1 shows the architecture of a system we consider in this paper—the system being managed consists of a number of heterogeneous server machines, which we refer to as *nodes*. Dynamic web applications are installed into *dynamic clusters*. Each dynamic cluster can have multiple web applications installed in it (Figure 1 combines dynamic clusters with applications for clarity). Each dynamic cluster is configured with a set of eligible nodes — nodes eligible to host *instances* of that dynamic cluster. When an instance of a dynamic cluster is running, all web applications installed in that dynamic cluster can be served by that instance. Each dynamic cluster can run on multiple nodes, and each node can run instances of multiple dynamic clusters. Web applications and dynamic clusters also have various administrator-configured *constraints*.

Requests (shown in Figure 1 as dotted lines) arrive first at the *Flow Controller*, where they are classified according to their URI pattern as belonging to one of several configured flows. Then they are placed in a queue corresponding to its flow. Requests are dispatched from queues using a weighted-fair scheduling discipline, whose dispatching weights and concurrency limits are determined automatically as discussed in [3]. A dispatched request passes through the *Load Balancer* which selects the instance where the request should be sent. After a requesed is processed, the response passes back to the client via the same components. Neither the Flow Controller nor the Load Balancer are able to start or stop instances—they take the current state of the system as a given. As requests are served by the instances, *Sensors* on each node are observing the behaviour of the instances. Sensor data is shown in Figure 1 as dashed lines. Information about the throughput and CPU usage of each instance is provided by these Sensors to the *Application CPU Profiler*, is discussed in Section II-A.

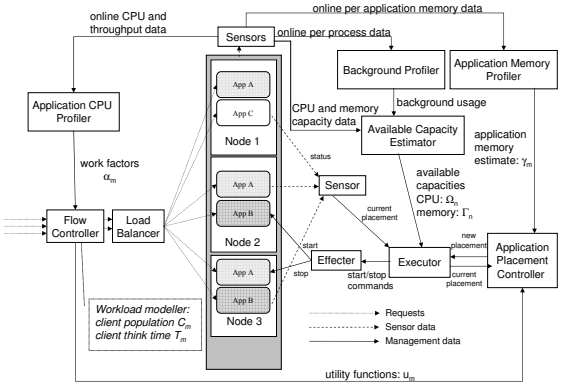

Fig. 1. Architecture of the system

Also monitoring the nodes and the instances are several other components, to which we now turn. The *Executor* is using its own set of *Sensors* to monitor which nodes are available, which instances are running. This information is all provided by the underlying middleware layer, these Sensors are able to simply subscribe to the relevant information. Using these Sensors, the Executor is able to create a 'map' of the current system. This map does not include request flow, it simply includes which instances are running upon which nodes.

The Executor provides this map of the system to the *Application Placement Controller*, which uses it to drive its decision making—see Section III. For the purposes of this discussion, the Application Placement Controller can be regarded as taking an input map (the current system) and producing an output map (the desired system).

The output map is returned to the Executor, which determines if any changes need to be made to the system to make it conformant with the output map. The changes that might be made are limited to starting and stopping instances. If changes are needed, the Executor drives the *Effecters* to make those changes—like the Sensors, the Effecters are provided by the underlying middleware layer.

Other information that the Application Placement Controller requires in order to make its decisions comes from other components also shown on Figure 1. The *Background Profiler* uses the Sensors to observe the amount of CPU and memory on each node that is being used by processes other than the instances. This information is provided to the *Available Capacity Estimator*, which combines it with information obtained from the Sensors concerning the amount of memory and normalized CPU cycles that each node has. Separately, the *Application Memory Profiler* uses the Sensors to observe the memory utilization of instances.

A. Characterising application resource usage

Our system is concerned with managing two kinds of resources: memory and CPU. Even though applications consume other resources, memory and CPU are typical sources of bottleneck for enterprise applications and therefore resources like disk I/O and network bandwidth are of a lesser concern. Nevertheless, the system discussed in this paper can be extended to address these kinds of resources.

Our system derives memory and CPU usage estimates for all managed applications based on online data. We characterize the CPU usage of an application using work factor, α, which represents the normalized number of cycles consumed on average by a request of an application. Work factor is defined using multi-linear regression model discussed in [2].

Characterizing memory usage is a more challenging problem as, due to effects of heap management in Java virtual machine as well as caching, memory usage by an application tends to reveal its historical rather than current requirement. Therefore, we do not attempt to derive a per-request memory consumption. Instead, for each application, we derive a per-instance memory requirement, γ, as a high watermark of physical memory usage for this application over a certain period of time.

B. Characterizing workload intensity

Typical enterprise web applications involve a set of client sessions that involve a sequence of web requests. Workload intensity is determined by the number of open sessions and client think time between consequtive web requests. We model this behavior for each flow using a closed queuing network model with a client population of C_f and think time T_f. Flow Controller estimates model parameters by tuning them to fit recently observed system behavior based on measured application throughput, number in the system, service time, and end-to-end response time. The details of the model are described in [9].

C. Placement algorithms

Previous application placement techniques define application demand in terms of CPU capacity and attempt to maximize the amount of satisfied demand. The approach in [8] combines demand-based placement with utility-based management by having a flow controller compute the desired CPU power distribution across applications that yields the greatest system utility. This computation is agnostic to memory and other constraints and only ensures that the sum of all desired CPU allocations does not exceed the total capacity of machines. Placement controller is responsible for computing placement that delivers the desired CPU power distribution. This necessarily may lead to suboptimal results, when the optimal load distribution cannot be delivered by any correct placement due to constraints. When optimal allocation cannot be delivered, the technique may penalize arbitrary applications, regardless of their performance impact.

This paper introduces a placement technique that is driven by application utility. Thus theoretically it provides a better solution for maximizing application performance. This comes at a cost of higher complexity, as the optimization objective is non-linear, and a more complex design, as more complex information must be exchanged between controllers. This additional complexity would not be justified if the resultant improvement in application performance was not significant or concerned only a small set of scenarios. In Section IV we show the opposite. The new desing leand to a significant improvement in application performance in a wide range of scenarios.

D. Estimating application utility

In our system, a user can associate a response time goal, τ_f and importance level, i_f with each flow. The importance level is an integer value which is greater than or equal to 1 and controls the system behavior when the response time goal cannot be met. Based on the observed response time for an application, t_f we evaluate the system performance with respect to the flow satisfaction using utility function u_f, which is defined as follows [3].

$$u_f(t_f) = \begin{cases} \frac{\tau_f - t_f}{\tau_f} & \text{if } t_f \leq \tau_f \\ \frac{\tau_f - t_f}{i_f \tau_f} & \text{otherwise} \end{cases} \quad (1)$$

The importance level decides the slope of the utility function degradation when the response time exceeds its goal. For less important flows (those with a higher importance level) the value utility function degrades slower with the increasing distance between the response time and its goal.

For the purpose of resource allocation we need to formulate the utility function as a function of allocated CPU capacity, ω_f, that is $u_f(\omega_f) = u_f(t_f(\omega_f))$. Hence, we need to be able to express response time as a function of allocated CPU capacity. It turns out that it is mathematically easier to express the opposite relationship: the amount of CPU power needed to achieve a particular utility. For a value of the utility function u_f^* we will express its corresponding response time as $t_f(u_f^*)$, which is defined by inversing Eq. 1. Given client population C_f and client think time T_f, we can obtain the throughput corresponding to u_f^* as follows.

$$\lambda_f(u_f^*) = \frac{C_f}{t_f(u_f^*) + T_f} \quad (2)$$

Given the average number of CPU cycles consumed by each request of the flow, α_f, we can easily obtain the amount of CPU power needed to achieve u_f^* by multiplying α_f and $\lambda_f(u_f^*)$.

$$\omega_f(u_f^*) = \begin{cases} \frac{\alpha_f C_f}{\tau_f(1-u_f^*)+T_f} & \text{if } u_f^* > 0 \\ \frac{\alpha_f C_f}{\tau_f(1-i_f u_f^*)+T_f} & \text{otherwise} \end{cases} \quad (3)$$

Assuming that the flow controller equalizes the utility among flows belonging to the same application, as it is the case with the controller described in [9], we can now express the CPU requirement of an application (ω_m) as a function of the utility for this application, u_m^*, as a sum over all application flows. To obtain $u_m(\omega_m)$, we sample $\omega_m(u_m^*)$ for various values of u_m^* and from the obtained datapoints, we extrapolate $u_m(\omega_m)$.

III. Algorithm description

In this section we present the algorithm adopted by the placement controller.

We are given a set of machines, $\mathcal{N} = \{1, \ldots, N\}$ and a set of applications $\mathcal{M} = \{1, \ldots, M\}$. We use n and m to index into the sets of machines and applications, respectively. With each machine n we associate its memory and CPU capacities, Γ_n and Ω_n. Both values measure only the capacity available to workload controlled by placement controller. Capacity used by other workloads is subtracted prior to invoking the algorithm. With each application, we associate its load independent demand, γ_m that represents the amount of memory consumed by this application whenever it is started on a machine. CPU requirements of applications are given in the form of utility functions defined in Section II-D.

We use symbol $\mathcal{I}$ to denote a placement matrix of applications on machines. Cell $\mathcal{I}_{m,n}$ represents the number of instances of application m on machine n. We use symbol L to represent a load placement matrix. Cell $L_{m,n}$ denotes the amount of CPU speed consumed by all instances of application m on machine n. Given application utility function from

Section II-D, we can express it also as a function of L, $u_m(L) = u_m(\sum_n L_{m,n})$.

We define load placement utility function $U(L) = (u_{m_1}(L), \ldots, u_{m_M}(L))$, where applications inside the vector are ordered according to increasing $u_{m_k}(L)$. Utility $U(L) = (u_{m_1}(L), \ldots, u_{m_M}(L))$ is greater than utility $U(L') = (u_{m'_1}(L'), \ldots, u_{m'_M}(L'))$ if there exists k such that $u_{m_k}(L) > u_{m'_k}(L')$ and for all $l < k$, $u_{m_l}(L) = u_{m'_l}(L')$. This induces a lexicographic order of utility vectors.

Given an instance placement, all controllers in the system try to find the best possible load distribution. Hence, utility of instance placement is $U(\mathcal{I}) = \max_L U(L)$, where load distribution defined by any considered L does not violate any system constraints. The objective of placement controller is to find $\mathcal{I}$ that maximizes $U(\mathcal{I})$. In addition, the algorithm tries to minimize the number of placement changes, which are time-consuming and CPU-intensive.

Considering the form taken by the utility function, our problem formulation is an extension of $\max\min$ criterion and differs from it by explicitly stating that after $\max\min$ objective can no longer be improved (because the lowest utility application cannot be allocated any more resources), the system should continue improving the utility of other applications. While finding the optimal placement, the controller observes a number of constraints, such as resource constraints, collocation constraints and application pinning, amongst others.

The placement algorithm proceeds in three phases: demand capping, placement calculation, and maximizing load distribution. Demand capping constraints the amount of CPU capacity that may be allocated to an application, which is used by placement calculation. The phase of maximizing load distribution takes placement obtained by placement calculation phase and calculates the best corresponding load distribution.

The basic algorithm, as described above, is surrounded by the *Placement control loop*, which resides within the Executor in Figure 1. This is designed to have the Application Placement Controller periodically inspect the system to determine if placement changes are now required to better satisfy the changing extant load. The period of this loop is configurable and can be interrupted when the configuration of the system is changed.

The placement change problem is known to be NP-hard and heuristics must be used to solve it. Based on our prior study focusing on the placement problem with a linear optimization objective [6], we identified several heuristics that are applicable also in the placement problem with non-linear optimization objective.

The outline of the placement change phase is based on the algorithm in [6]. The placement change phase is executed several times, each time being referred to as a 'round'. Each round invokes the placement change method, which makes a single new placement suggestion starting from the placement suggestion provided by the previous round execution. We perform up to 10 rounds or break out before if no improvement in placement utility is observed at the end of a round.

The placement change method first iterates over nodes. For each node, it iterates over all instances placed on this node and attempts to remove them one by one, thus generating a set of configurations whose cardinality is linear in the number of instances placed on the node. For each such configuration it iterates over all applications whose satisfied demand is less than the limit calculated in the capping phase, attempting to place new instances on the node as permitted by the constraints (see Section III). The placement change method has been modified in serveral aspects with respect to the version described in [6]. The most important change is the fact that using utility-based heuristics to decide the order in which nodes and applications instances are visited helps us introducing a number of optimizations and shortcuts in the algorithm that reduce the complexity of our calculations.

IV. Experiments and results

In this section, we experimentally evaluate our approach using both real system measurements and a simulation.

A. Testbed results

In this section we describe an experiment we carry out to illustrate the benefit of placement technique introduced in this paper, using the real system implementation described in section II, and integrating it with WebSphere Extended Deployment [1] application server middleware. We deploy three applications, A1, A2, and A3 in a system composed of four homogeneous nodes. Applications are identical with respect to their per-request CPU requirements and each request involves the same amount of computation interleaved with sleep time that simulates applications backend activity. We configure only one flow in each application, thereby making an application the smallest unit of management for the purpose of this experiment. Neither allocation restrictions nor collocation restrictions are defined, but placements are still subject to resource constraints, such as available memory of the nodes.

Property	**Node 1**	**Node 2**	**Node 3**
Effective total CPU capacity[MHz]	3800	3800	3800
Effective Memory capacity[MB]	2500	2500	2500

Property	**Application 1 and 2**	**Application 3**
Memory demand[MB]	1200	1800
Response time goal [ms]	1200	350
Importance	50	50

TABLE I
TESTBED CHARACTERISTICS

We run the experiments on a cluster of IBM xSeries 335 servers, each containing 2 2.4GHz Intel Xeon processors with hyperthreading enabled. All the servers are connected through a switched gigagbit Ethernet network and run Linux 2.6.

The properties for the nodes and applications used in our experiments are shown in table I. Each node is able to support roughly 38 concurrent sessions of either application at a time, before overload protection becomes necessary. The base service time of each application is about 240 ms, which makes the response time goal for A3 rather aggressive. Also, notice that A1 and A2 use 40% of the memory capacity of a node

each, while A3 uses 75%. Hence, A1 and A2 can both fit on a node, but neither of them can be placed together with A3. We configure such memory requirements by configuring a corresponding *maxHeapSize* value on application server JVM.

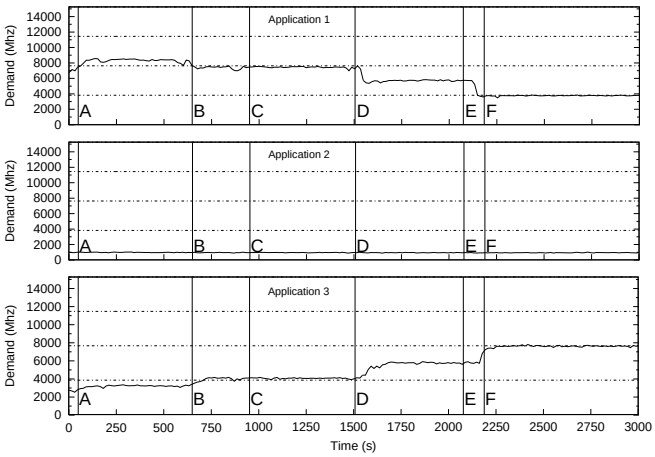

Fig. 2. Aggregated CPU demand

1) Baseline experiment: Before experimenting with placement algorithm we baseline the system to observe the amount of CPU demand imposed by each application. We set all applications in manual mode, thus preventing any placement changes. We also configure memory requirements of applications such that all applications can be placed together on a node. Then we start an instance of each application on every node.

We choose total workload intensity so as not to overload the system. Then we vary the number of client sessions for applications within this limit.

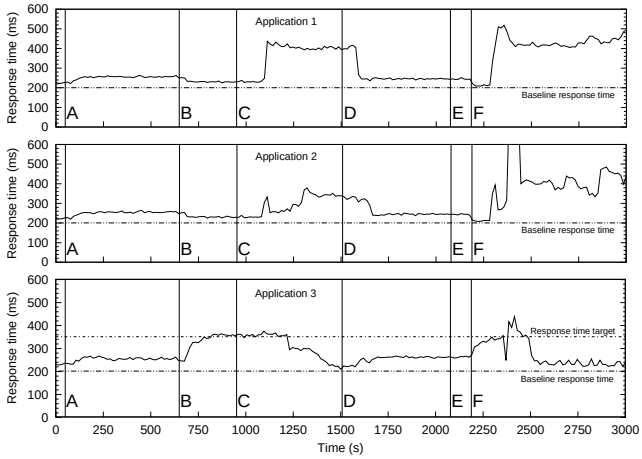

Fig. 3. Response time

Figure 2 shows the amount of CPU demand imposed on the system throughout the experiment. After a warm-up period, we start 95, 10, and 35 client sessions for A1, A2, A3, respectively (point A in Figure 2). This setting gives us a total number of client sessions of 140, which is slightly below the total that may be satisfied by our four-node system, 152. At point B, we increase the number of client sessions for A3 by 10 and correspondingly decrease the number of session for A1 by 10. At point D, we further increase the load for A3 by 20 clients and decrease load for A1 by the same amount. Finally, at point E, we further increase load for A3 by 19 clients, and correspondingly decrease the load for A1. Since throughout the experiment the system is never overloaded, the CPU usage observed across all nodes for each application gives us the CPU demand of this application.

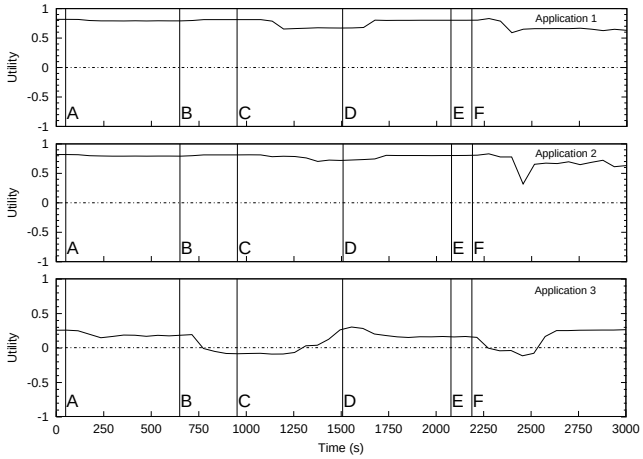

Fig. 4. Utility

2) Utility-based placement experiment: In the second experiment, we enable dynamic placement of applications and configure the applications as specified in Table I. We configure initial placement such that A1 and A2 are both placed on three nodes, and A3 occupies one node. We run the same workload scenario as in the baseline experiment. Figures 3 and 4 show observed response time for applications and their corresponding utility value. Figure 5 shows CPU capacity allocated to each application. In phase A-B of the experiment, workload distribution is such that with the existing placement, all application CPU demands are satisfied and response time goals are met. Utility values of applications are different as a result of them having different goals. When workload changes at point B, A3 cannot be satisfied by a single node on which it is running. Its response time increases and it starts to violate the goal. At the same time, A1 experiences very good performance relative to its goal. At this point, it is reasonable to consider making a placement change that would remove A1 and A2 from one node and give this node to A1. The decision is made at point C. There is a rather long delay between the time workload has changed to the time placement changes. This time is used by the controllers to accumulate enough statistics to build performance models for the new workload conditions. After the change is executed, within 2-3 minutes, response time for A1 returns to normal, while response time for A1 and A2 increases. Given the high goal for the latter applications, their utility is very moderately affected by this change. At point E, we further increase load for A3 and decrease it for A1. We experience a similar

change in performance as at point B. This time placement decision is done earlier, at point F. However, due to imprecise models, the decision is quickly reversed back, and remade again in the consecutive cycles of placement algorithm. This is clearly the evidence of instability, which in real deployments of our controller is avoided by introducing a stabilization delay after each placement change. In this experiment, we have not exercised this stabilization interval. In the final placement, A3 is allocated three nodes, and A1 and A2 share a single node. After the last placement change is completed, the response time for all applications is within the configured goal.

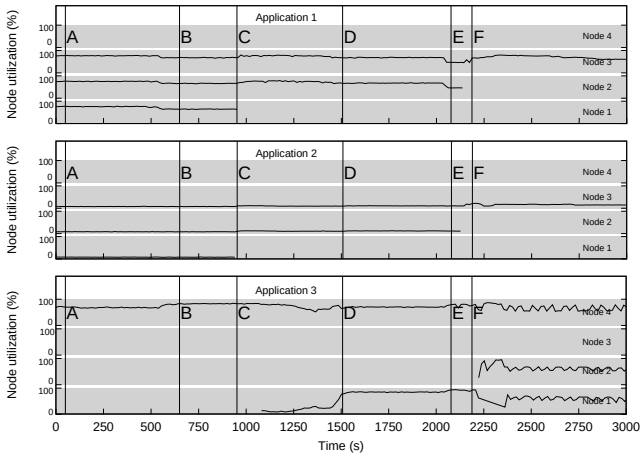

Fig. 5. Per node allocation

3) Comparison of utility-driven and demand-based techniques: Let us examine the demand values obtained in the baseline experiment (Figure 2) and consider what demand-based algorithm would do when presented with these inputs. In phase A-B, the demand of all applications can clearly be satisfied using the initial placement, hence no changes would happen. In phase B-C, the demand of A3 cannot be satisfied by a single node. The demand-based algorithm should now look at the offered demand of the applications, which is about 1.05 nodes (4000MHz) for A3 and 2.23 nodes (8500MHz) for A1 and A2 combined. Clearly, when 4 nodes are available, satisfied demand is maximized (at 3.23 nodes) with the current placement, even though in the current placement A3 is missing the goal. In phase D-E, the demand of A3 is 1.58 nodes (6000MHz) and the total for A1 and A2 is 1.82 nodes (7000MHz). At this time, the demand-based algorithm transfers a node from A1 and A2 to A3. In the last phase of the test, the load for A3 is 2.1 nodes (8000MHz) and for A1 and A2 it is 1.31 nodes (5000MHz). Again, to maximize satisfied demand, it is better to leave placement unchanged, as this will result in satisfied demand of 3.31 nodes as opposed to 3.1 nodes if a change happened. Clearly, from the performance perspective, this is not the right decision.

B. Simulation results

In this section, we evaluate the effectiveness of our placement algorithm when subject to a number of different workload conditions and when managing a large number of nodes and applications. For each application we generate a realistic and randomized workload across 300 placement control cycles, which results in a varying CPU demand. The memory demand is uniform across applications for each simulation run.

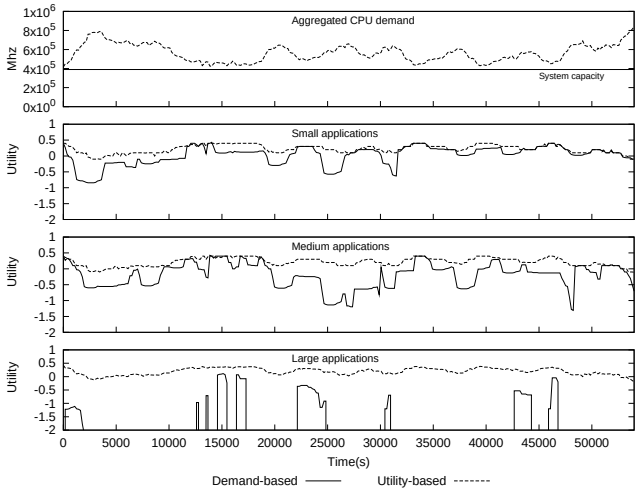

Fig. 6. Maximizing minimum utility across applications

We focus our study on three different evaluation criteria for the algorithm: evaluation of the algorithm's ability to achieve its objectives (maximization of the minimum utility across applications and minimization of placement changes), evaluation of the sensitivity of the algorithm to the different parameters present in the placement problem, and evaluation of the quality of the placement decisions made by the algorithm. At each step we compare our algorithm with a state-of-the-art dynamic application placement algorithm, described in [8], that differs with respect to our approach in that it tries to maximize the satisfied demand in the system.

To provide utility functions for the simulation, we have implemented a utility function generator that produces a realistic curve (directly comparable with those generated by the Flow Controller in the real system) whose shape is controlled by maximum utility value and CPU allocation required to reach this maximum point. In our simulations, a new maximum CPU allocation is generated for each application at each control cycle and the corresponding utility function updated. These utility functions are directly fed into the utility-based algorithm.

1) Evaluation criterion: minimum utility: First, we evaluate the capability of the algorithm to maximize the minimum utility across applications. We simulate a slighlty overloaded system, composed of 100 nodes and 20 applications. A system is overloaded if the total amount of demand needed to maximize the utility of all applications is greater than the total CPU capacity of the system. Each application, on average, requires an allocation equivalent to 5.5 nodes to be fully satisfied. Given this scenario, we run three simulations, each producing the same per-application CPU demand, but using different application memory demands each time. In the first simulation, we use applications that require little memory, resulting in a configuration where up to 6 application instances can be placed

on the same node. For the second simulation, we use medium applications, resulting in a configuration where 2 instances at maximum can be placed on the same node. Finally, in our third simulation we simulate applications large enough to ensure that only one instance can be placed on each node. Increasing the memory demand of the application instances also increases the hardness of the problem. The summary of the results obtained in this experiment is shown in Figure 6, where our algorithm is referred to as 'Utility-based' and the algorithm described in [8] is referred to as 'Demand-based'. The utility values shown in the figure correspond to the lowest utility observed across applications at each control cycle.

This experiment shows that our utility-driven algorithm consistently achieves an overall minimum utility higher than the value obtained by the demand-based algorithm. In particular, the harder the problem input becomes, the bigger the difference between the two algorithms is. The results obtained for this experiment also indicate that our algorithm has low sensitivity to the problem hardness, shown by the fact that the minimum utility achieved across applications is very similar given three different memory fragmentation scenarios.

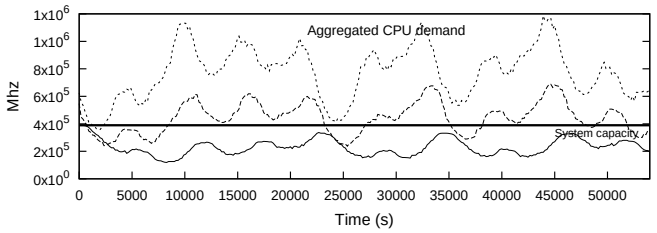

Fig. 7. Minimizing placement changes: generated workload

2) Evaluation criterion: number of placement changes: In our second experiment we evaluate the capability of our algorithm to minimize the number of placement changes over time. A placement change incurs a significant cost in terms of resources and time and it is thus desirable that placement algorithm keep the number of placement changes to a minimum. Such changes should occur only when the benefit that they introduce in terms of utility improvement and fairness is significant. For this experiment, we compare our utility-driven algorithm with the demand-based placement algorithm described in [8], when subjected to a particularly hard scenario. We simulate a system composed of 100 nodes and 200 applications. Each application instance requires half of the memory capacity of a node to be placed, so we can place a only 200 instances. With respect to the CPU demands, we consider three different scenarios: first, when no overload is present in the system; second, when overload is only present in some stages of the test; and third, when the system is always overloaded.

Figure 8 shows that the minimum utility achieved by the utility-driven algorithm is slightly worse than the obtained by the demand-driven algorithm when the system is not overloaded or only partially overloaded, but clearly better when the system is completely overloaded. This is because our algorithm is driven by utilities, while the demand-based algorithm tries to maximize the satisfied demand for all applications. This tight scenario forces the demand-based algorithm to make many placement changes because the severe memory constraints make the problem challenging. Our algorithm, instead, decides that because the utility improvement from making any changes is so low, no changes should be made after the initial placement. Notice that, at some points, the demand-based algorithm makes up to 400 placement changes, which means that it is effectively stopping all instances and starting them in different places, chasing a better one-to-one combination of applications sharing nodes that helps it to improve the overall satisfied demand. In addition, the average utility charts demonstrate that although the minimum utility achieved by our algorithm is lower than the result obtained for the demand-based algorithm, the average utility value achieved across applications is very close for the two algorithms. These results confirm that our utility-driven algorithm is achieving its second objective of minimizing placement changes even in hard placement problems.

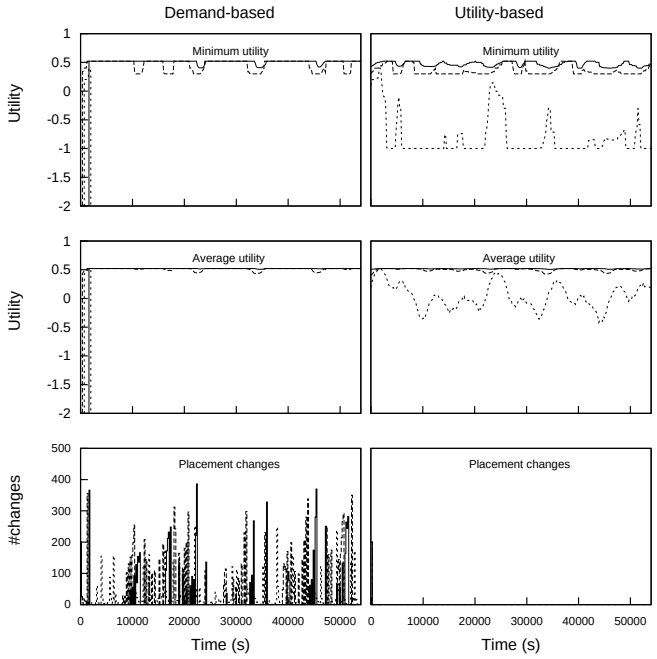

Fig. 8. Minimizing placement changes: The solid, dashed, and dotted lines represent non-overloaded, partially overloaded, and overloaded scenarios, respectively. In the overloaded system test, the minimum and average utility achieved by the demand-based algorithm is below the range of the chart and the corresponding curve is not shown in the figure.

3) Evaluation criterion: optimality: Ideally, we would like to compare our technique to an optimal, even if very complex, algorithm. Unfortunately, implementing such an algorithm is extremely difficult, and the execution is extremely slow, preventing us from running any useful experiments. Therefore, we implemented a heuristic algorithm which ignores all but CPU and memory capacity constraints, and does not aim to minimize the number of placement changes. Consequently, we can design a heuristic that achieves a better result in terms of maximizing the minumum utility.

We compared this heuristic with our algorithm in 13 different representative scenarios. The results obtained after these tests is that the minimum utility achieved by our algorithm is, in average, in the range of the 95% to the 110% of the minimum utility achieved by the heuristic discussed above.

V. Related work

The problem of dynamically allocating server resources to applications has been extensively studied. The algorithm proposed by Kimbrel et al. [8], [4] is the closest to our work. With our experiments we demonstrate that a utility-driven system can overcome a demand-based approach in terms of application satisfaction fairness.

A popular approach to dynamic server provisioning is to allocate full machines to applications as needed [10], which does not allow applications to share machines. In contrast, our placement controller allows this sharing and is optimized for it. The algorithm proposed in [7] allows applications to share machines, but it does not change the number of instances of an application, does not minimize placement changes, and only considers one bottleneck resource. In contrast to our work, none of them is directed by high-level objectives.

Placement problems have also been studied in the optimization literature, including bin packing, multiple knapsack, and multi-dimensional knapsack problems [11]. The special case of our problem with uniform memory requirements was studied in [12], and some approximation algorithms were proposed. The optimization problem that we consider presents a non-linear optimization objective while previous approaches [8], [4] to the same problem use linear optimization objectives.

Meta-scheduling algorithms for grid and parallel computing also deal with the placement problem [13], but in our case we are not concerned about communication overheads between application servers because this is not a key point in the management of dynamic web applications.

Using utility functions to represent high-level application objectives is a practical and effective way to manage such objectives. It has been described as useful way of designing and implementing self-managed autonomic systems [14]. This way users and system administrators can be isolated from the IT low-level metrics and allowing them to express satisfaction of users and service level based on their high-level criteria [15], [16]. The utility functions used in our system are based on the work described in [9].

VI. Conclusions and future work

In this paper we present a system that automatically allocated resource to clustered web applications. It is based on a utility-driven application placement algorithm to achieve equalized satisfaction across applications. Additionally it minimizes the number of placement changes necessary to achieve its goals. The system has been implemented and integrated with a commercial application server middleware, what provides the support for executing placement decisions. Our system is driven by high-level application goals and takes into account the application satisfaction with how well the goals are met. We are not aware of any comparable implementation. We have demonstrated, both using a real-system experiment and a simulation, that this approach improves satisfaction fairness across applications compared to existing state-of-the-art solutions. We have also demonstrated that the system consistently achieves its goals independently of the workload conditions and the system configuration. As described in [17] we have begun to implement a system that applies our technique to manage virtual machines according to the SLA requirements of workloads hosted by them.

Acknowledgments

This work is partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625 and by the BSC-IBM collaboration agreement SoW Adaptive Systems

References

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

[2] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "Dynamic estimation of cpu demand of web traffic," in *VALUETOOLS*, Pisa, Italy, Oct. 2006.

[3] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance management for cluster-based web services," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 12, Dec. 2005.

[4] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic application placement under service and memory constraints," in *International Workshop on Efficient and Experimental Algorithms*, Santorini Island, Greece, May 2005.

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

[6] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *WWW Conference*, Banff, Alberta, Canada, 2007.

[7] 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.

[8] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, "Dynamic placement for clustered web applications," in *WWW Conference*, Edinburgh, Scotland, May 2006.

[9] 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.

[10] 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 *Int'l Symposium on Integrated Management*, Seattle, WA, May 2001, pp. 14–18.

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

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

[13] A. Turgeon, Q. Snell, and M. Clement, "Application placement using performance surfaces," in *Proceedings. The Ninth International Symposium on High-Performance Distributed Computing*, Pittsburgh, PA, August 2000, pp. 229–236.

[14] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *IEEE Internet Computing*, vol. 11, no. 1, pp. 40–48, 2007.

[15] M. N. Bennani and D. A. Menasce, "Resource allocation for autonomic data centers using analytic performance models," in *Int'l Conf. Automatic Computing*, Washington, DC, 2005.

[16] D. Gilat, A. Landau, and A. Sela, "Autonomic self-optimization according to business objectives," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 206–213.

[17] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess, "Server virtualization in autonomic management of heterogeneous workloads," in *Int'l Symposium on Integrated Management*, Munich, Germany, 2007.