Dynamic Application Placement Under Service and Memory Constraints

Tracy Kimbrel, Malgorzata Steinder, Maxim Sviridenko, and Asser Tantawi

IBM T.J. Watson Research Center,
P.O. Box 218, Yorktown Heights, NY 10598
{kimbrel, steinder, sviri, tantawi}@us.ibm.com

Abstract. In this paper we consider an optimization problem which models the dynamic placement of applications on servers under two simultaneous resource requirements: one that is dependent on the loads placed on the applications and one that is independent. The demand (load) for applications changes over time and the goal is to satisfy all the demand while changing the solution (assignment of applications to servers) as little as possible. We describe the system environment where this problem arises, present a heuristic algorithm to solve it, and provide an experimental analysis comparing the algorithm to previously known algorithms. The experiments indicate that the new algorithm performs much better. Our algorithm is currently deployed in the IBM flagship product Websphere.

1 Introduction

With the proliferation of the Web and outsourcing of data services, computing service centers have increased in both size and complexity. Such centers provide a variety of services, for example web content hosting, e-commerce, web applications, and business applications. Managing such centers is challenging since a service provider must manage the quality of service provided to competing applications in the face of unpredictable load intensity and distribution among the various offered services and applications. Several management software packages which deal with these operational management issues have been introduced. These software systems provide functions including monitoring, demand estimation, load balancing, dynamic provisioning, service differentiation, optimized resource allocation, and dynamic application placement. The last function, namely dynamic application placement, is the subject of this paper.

Service requests are satisfied through the execution of one or more instances of each of a set of applications. Applications include access to static and dynamic web content, enterprise applications, and access to database servers. Applications may be provided by HTTP web servers, servlets, Enterprise Java Beans (EJB), or database queries. When the number of service requests for a particular application increases, the application placement management software deploys additional instances of the application in order to accommodate the increased load. It is imperative to have an on-demand management environment allowing instances of applications to be dynamically deployed and removed. The problem is to dynamically change the number of application instances so as to satisfy the dynamic load while minimizing the overhead of starting and stopping application instances.

We characterize an application by two parameters, a load-independent requirement of resources required to run an application, and a load-dependent requirement which is a function of the external load or demand placed on the application. Examples of load-independent requirements are memory, communication channels, and storage. Examples of load-dependent requirements are current or projected request rate, CPU cycles, disk activity, and number of execution threads.

We also characterize a server by two parameters: a load-independent capacity which represents the amount of resources used to host applications on the server, and a load-dependent capacity which represents the available capacity to process requests for the applications' services.

In this paper we restrict ourselves to a single load-independent resource, say memory requirements, and a single load-dependent resource, say CPU cycles. These are the most constrained resources for most applications, and thus we are able to focus on these and use our solution in practice even though there are other resource requirements and constraints.

The paper is organized as follows. In Section 2 we describe the Websphere environment in which our algorithm is implemented. In Section 3 we describe related work. In Section 4 we present a mathematical formulation of the problem. We describe our heuristic algorithm in Section 5. Experimental results are given in Section 6. Further work is discussed and conclusions are given in Section 7.

2 System Description

Based on the experimental results presented in this paper, our algorithm has been incorporated into the IBM Websphere environment [10]. A Websphere component known as the placement controller receives dynamic information about the load-independent and load-dependent requirements of the various applications, and the load-independent and load-dependent capacities of the various servers. We used memory size and CPU cycles/sec as the representative load-independent and load-dependent parameters, respectively. The placement controller is aware of the configuration, i.e., the mapping of applications onto servers in a given Websphere cell. Upon need, or periodically, the placement controller executes our algorithm in order to determine the change in application placement configuration in response to changes in loads and characteristics of the applications and servers. Then the placement controller realizes the change, automatically or in a supervised mode, through the execution of scripts to start and stop applications servers.

The system includes an application workload predictor and an application profiler. The application workload predictor utilizes historical information regarding the offered load to produce a workload prediction for each application supported by the server farm. For instance, the workload prediction can be characterized by the arrival rate of requests to a given application. Similar to the application workload predictor, the application profiler produces a set of application resource requirements by estimating the amount of server resources required by a single request of each application. The application resource requirements includes, for example, the number of CPU cycles required to process a request.

The placement controller utilizes the workload prediction and the application resource requirements provided by the application workload predictor and the application profiler to compute predicted load-dependent resource requirements for each application. Considering the predicted resource requirements for each application, the given capacities of each of the server computing nodes in the server farm, and the current application placement, the placement controller uses the algorithm presented here to compute a new placement of applications.

3 Related Work

The problem of optimally placing replicas of objects on servers, constrained by object and server sizes as well as capacity to satisfy a fluctuating demand for objects, has appeared in a number of fields related to distributed computing. In managing video-on-demand systems, replicas of movies are placed on storage devices and streamed by video servers to a dynamic set of clients with a highly skewed movie selection distribution. The goal is to maximize the number of admitted video stream requests. Several movie placement and video stream migration policies have been studied. A disk load balancing criterion which combines a static component and a dynamic component is described in [9]. The static component decides the number of copies needed for each movie by first solving an apportionment problem and then solving the problem of heuristically assigning the copies onto storage groups to limit the number of assignment changes. The dynamic component solves a discrete class-constrained resource allocation problem for optimal load balancing, and then introduces an algorithm for dynamically shifting the load among servers (i.e. migrating existing video streams). A placement algorithm for balancing the load and storage in multimedia systems is described in [5]. The algorithm also minimizes the blocking probability of new requests.

In the area of parallel and grid computing, several object placement strategies (or, meta-scheduling strategies) have been investigated [8, 1]. Communication overhead among objects placed on various machines in a heterogeneous distributed computing environment plays an important role in the object placement strategy. A related problem is that of replica placement in adaptive content distribution networks [**?**, 1]. There the problem is to optimally replicate objects on nodes with finite storage capacities so that clients fetching objects traverse a minimum average number of nodes in such a network. The problem is shown to be NP-complete and several heuristics have been studied, especially distributed algorithms.

Similar problems have been studied in theoretical optimization literature. The special case of our problem with uniform memory requirements was studied in [6, 7] where some approximation algorithms were suggested. Related optimization problems include bin packing, multiple knapsack and multi-dimensional knapsack problems [2].

4 Problem Formulation

We formalize our *dynamic application placement problem* as follows. We are given m servers $1, \ldots, m$ with memory capacities $\Gamma_1, \ldots, \Gamma_m$ and service capacities (number of requests that can be served per unit time) $\Omega_1, \ldots, \Omega_m$. We are also given n applications $1, \ldots, n$ with memory requirements $\gamma_1, \ldots, \gamma_n$. Application j must serve some number of requests ω_{jt} in time interval t.

A feasible solution for the problem at time step t is an assignment of applications' workloads to servers. Each application can be assigned to (replicated on) multiple servers. For every server i that an application j is assigned to, the solution must specify the number ω_{itj} of requests this server processes for this application. $\sum_i \omega_{ijt}$ must equal ω_{jt} for all applications j and time steps t. For every server the memory and processing constraints must be respected. The sum of memory requirements of applications assigned to server i cannot exceed its memory Γ_i and $\sum_j \omega_{ijt}$, i.e. the total number of requests served by this server during the time step t, cannot exceed Ω_i. Note that each assignment (copy) of an application to a server incurs the full memory cost, whereas the processing load is divided among the copies.

The objective is to find a solution at time step t which is not very different from the solution at time step $t-1$. More formally, with every feasible solution we associate a bipartite graph (A, S, E_t) where A represents the set of applications, S represents the set of servers, and E_t is a set of edges (j, i) such that application j is assigned to (or has copy on) server i at time step t. Our objective function is to minimize $|E_t \ominus E_{t-1}|$, i.e., the cardinality of the symmetric difference of the two edge sets. This is the number of application instances that must be shut down or loaded at time t.

4.1 Practical Assumptions

Since finding a feasible solution to the static problem (i.e., that of finding an assignment for a single time step) is NP-hard, we must make certain assumptions about the input. Intuitively, if the input is such that even finding a static solution is hard we cannot expect to find a good solution with respect to the dynamic objective function. Thus the problem instance must be "easy enough" that a relatively straightforward heuristic can find a feasible solution at each time step. In practical terms, this means there must be enough resources to easily satisfy the demand if we ignore the quality of the solution in the sense of the dynamic objective function. In the worst case, we can fall back on the heuristic to find a feasible solution. In fact, our algorithm will degrade gradually to this extreme, but should perform much better in the common case.

5 Algorithm

We first describe an algorithm that builds a solution from scratch, i.e. under the assumption that $E_{t-1} = \emptyset$, either because this is the first step ($t = 1$) or because the solution from the previous step $t - 1$ is very bad for serving demands at step t. This heuristic will be also used later as a subroutine when we describe an incremental algorithm which optimizes the objective function as we move from step $t - 1$ to t. At the risk of slight confusion, we will refer to this heuristic as the initial placement heuristic even when it is used as part of the incremental construction.

5.1 Initial Placement

We order all servers by decreasing value of their densities Ω_i/Γ_i, and order applications by decreasing densities ω_{jt}/γ_j. We load the highest density application j to the highest density server i which has enough memory for that application.

If the available service capacity Ω_i of a server i is larger then service requirement ω_{jt} of an application that we assign to the server, then we delete application j from the list of unscheduled applications. We recompute the available memory and service capacities of the server i by subtracting the amounts of resources consumed by application j and insert server i back to the list of servers according to its new density Ω_i/Γ_i with the updated values Ω_i and Γ_i.

If the available service capacity Ω_i of the server i is exceeded by the demand ω_{jt}, we still assign application j to server i, but this application's demand served by this server is limited by the server's (remaining) service capacity. We remove the server from the list.

In the latter case that the service capacity on the server i is exceeded by application j assigned to it, let ω'_{jt} be the amount of demand of application j assigned to this server and let ω''_{jt} be the remaining demand; note $\omega'_{jt} + \omega''_{jt} = \omega_{jt}$. Since the server i cannot serve all demand of application j we will need to load at least one more copy of it on another server, but we do not yet know which server. The density of the remaining demand is ω''_{jt}/γ_j. We place the application back in the list with this value as its density in the sequence of remaining applications (in the appropriate place in the list ordered by densities). Then we move on to the next highest density application, and so on.

The intuition behind the rule is as follows. We should match applications which have many requests per unit of memory with servers which have high processing capacity per unit of memory. It is not wise to assign applications with high density to a low density server, since we would be likely to reach the processing capacity constraint and leave a lot of memory unused on that server. Similarly, if low density applications are loaded on high density servers, we would be likely to reach the server's memory constraint without using much of the processing capacity.

Note that for every server the algorithm splits the demand of at most one application between this server and some other servers. Thus the total number of application-to-server mappings (edges in the bipartite graph) is at most $n+m-1$.

5.2 Incremental Placement

Although the initial placement algorithm is rather conservative in memory allocation, it could be very bad from the viewpoint of the dynamic objective function, which seeks a minimal incremental cost of unloading and loading applications between time steps. We now explain how we can combine the initial placement algorithm with a max flow computation to yield a heuristic for minimizing our objective function.

Given a feasible solution on the previous step (A, S, E_{t-1}) we first would like to check whether we can satisfy the new demands ω_{jt} by simply using the old assignment of applications to servers. We check this by solving a bipartite flow problem. I.e., we use the edge set E_{t-1}. Each node corresponding to application j is a source of ω_{jt} units of flow. We test whether there is a flow satisfying these sources by routing flow to sinks corresponding to the servers, such that the flow into each sink corresponding to a server i is limited by the server's service capacity Ω_i.

If this flow is feasible we are done; the flow values on the edges give the assignments of applications' loads to servers. Otherwise, there is a residual demand for every application (possibly 0 for some) which remains unassigned to servers. Denote the residual demands by ω'_{jt}. For every server there are a residual memory Γ'_i and a service capacity Ω'_i that are not consumed by the assignment given by the flow. Notice that these demands and capacities induce a problem of the same form as the initial placement problem. We apply our greedy initial placement heuristic to this instance. If our heuristic finds a feasible solution to the residual instance, we can construct an overall solution as follows. The residual instance results in a new set of edges, i.e., application-to-server mappings (applications which must be loaded onto servers), which we simply add to the existing edges. The total cost of the new solution is the number of new edges used by the heuristic to route the residual demand. This should not be large since our heuristic is conservative in defining new edges.

If our heuristic fails to find a feasible solution, we delete an edge in the graph (A, S, E_{t-1}) and repeat the procedure. We continue in this fashion until a feasible solution is found. The total cost is the number of deleted edges in addition to the number of new edges. In the worst case, we eventually delete all edges in the graph and build the solution from scratch using our initial placement heuristic, which is possible by our assumption that the instance is "not too hard."

It remains to define which edge should be deleted. A good heuristic choice should be the edge which minimizes the ratio of the total demand routed through this edge (i.e., the flow on this edge) divided by the memory requirement of the corresponding application. The intuition for this is that we would like to delete an edge which uses memory in the most inefficient way. Experimental analysis could find a better function to define a candidate for deletion.

6 Experimental Evaluation

6.1 Uniform Memory Requirements

In this section we evaluate our proposed algorithm for uniform memory requirements $\gamma_j = const$. We concentrate on its performance in terms of the number of placement changes it suggests and in terms of the execution time. The algorithm is compared against two other placement algorithms we considered for implementation.

The first algorithm, known as the *Noah's Bagels* algorithm, is a solution to a class-constrained Multiple Knapsack problem as defined in [6, 7]. This algorithm is applicable to the application placement problem only after making the assumption that all applications have the same memory requirement d, i.e., for every application j, $\gamma_j = d$. In consequence, the memory of each server can be expressed as a multiple d; normalizing by setting $d = 1$, server i can host Γ_i applications. The algorithm fills nodes in increasing order of Ω_i. To fill a node i, it considers applications in increasing order of ω_j, and finds the minimum j such that $\sum_{k=j}^{j+\Gamma_i-1} \omega_k \geq \Omega_i$. Applications $j, \ldots, j + \Gamma_i - 1$ are loaded on i, possibly splitting the demand of $j+\Gamma_i-1$; in this case, $j+\Gamma_i-1$ is put back in the list of applications, sorted according to its remaining unmet demand. This algorithm is known to solve a maximization version of our problem, maximizing $\sum_i \sum_j \omega_{ijt}$, when for every i, $\frac{\Omega_i}{\Gamma_i} = \text{const}$ and $\sum_i \Gamma_i \geq n+m-1$. If the above conditions are not satisfied, the algorithm is sub-optimal, but if we allow resource augmentation, i.e., if we increase each Γ_i by 1, then this algorithm finds a solution with value lower bounded by the optimal value for the original instance.

Under our assumption on practical inputs, we can use this maximization algorithm to find a feasible solution to our problem (i.e., optimal under the multiple knapsack objective). Our implementation of the *Noah's Bagels* algorithm computes a new placement in each iteration from scratch, not taking the previous placement into account. It may produce a different placement, even if no placement changes are needed to satisfy the new demand. To avoid this unnecessary placement churn, in our implementation, a new placement is computed only if the previous one is unable to satisfy the current demand as witnessed by the existence or non-existence of a solution to the satisfying flow problem.

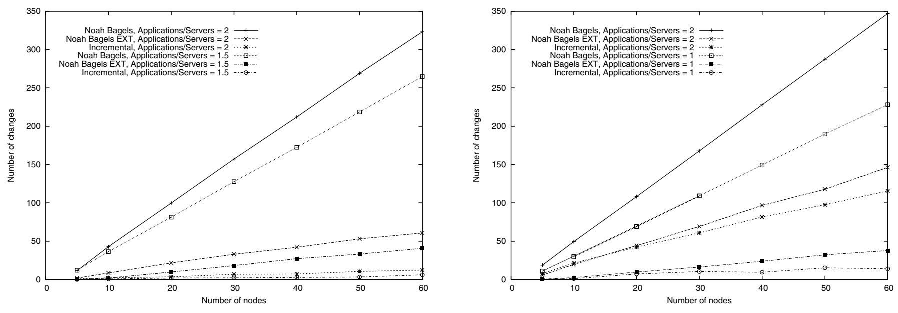
The second algorithm is a modification of the *Noah's Bagels* algorithm that we developed to make it take the previous placement into account. The algorithm adopts several heuristics to modify the previous placement to satisfy the new demand. It first accepts the old placement on nodes that are well utilized by the new demand. We say a node is well utilized if applications placed on it use the total CPU capacity and no more than the node's available memory, and the CPU load of no more than one application needs to be transfered to another node. When no more well utilized nodes exist, the algorithm fills unused memory on underutilized nodes using the original *Noah's Bagels* algorithm. If this step does not result in a well utilized node, an application with the smallest CPU demand is removed from the node and the process is repeated until the node is well utilized. When no more underutilized nodes remain, overutilized nodes are processed by removing applications from them, starting from ones with the highest CPU demand, until a node becomes well utilized. We call this new algorithm *Noah's Bagels Ext*. Like the original algorithm, the *Noah's Bagels Ext* algorithm assumes that all applications have the same memory requirements and it guarantees a placement satisfying the demand of all applications if for every i, $\frac{\Omega_i}{\Gamma_i} = \text{const}$ and $\sum_i \Gamma_i \geq n + m - 1$. The algorithm *Noah's Bagels Ext* was the previous candidate to be implemented in the placement controller and this why we compare our algorithm with it.

Experiment Design. In the experimental study we vary the following variables:

1. The number of servers. We start from 5 servers and increase their number to 60. This range of values covers the server-cluster sizes that are used in practice.
2. The number of applications that may be hosted on each server, which corresponds to their memory sizes. We set this value to 3 in all experiments, i.e. $\Gamma_i = 3$, $i = 1, \ldots, m$.
3. The ratio of the number of applications to the number of servers. We use values of 1, 1.5, and 2.
4. Memory requirements of applications. For this section, memory requirements are the same for all applications, i.e., $\gamma_j = 1$, $j = 1, \ldots, n$. In the next section we will consider the case in which different applications have different memory requirements.
5. CPU speeds of servers. In our study, the CPU speeds of all servers are the same.
6. Utilization factor, which is the ratio of the sum of CPU processing requirements of all applications to the sum of the CPU speeds of all servers. We use utilization factors of 0.8, 0.9, and 0.99.

Given the above parameters, we generate the test scenarios iteratively as follows. We set the initial placement to be empty. We repeatedly produce CPU demand for applications by independently generating a set of n random numbers in the interval $[0, 1]$, $p_1, \ldots, p_n$, one for each application. Then we normalize these values by setting $p_j^* = \alpha p_j$, where $\alpha = \frac{1}{\sum_j p_j}$. Then we set the load-dependent demand of application j, $\omega_j = p_j^* \rho \sum_i \Omega_i$, where ρ is the utilization factor. Given the new demand, we compute the new placement, taking the placement from the previous iteration as the current placement.

It should be mentioned that, in practice, demand in a given cycle is correlated with the demand in the previous cycle, whereas in our experiments the new demand is independent of the old demand. However, this correlation is very



(a) Utilization factor of 0.8 (b) Utilization factor of 0.99

Fig. 1. The number of placement changes

difficult to quantify. Our random method provides a pessimistic case on which to test the algorithms.

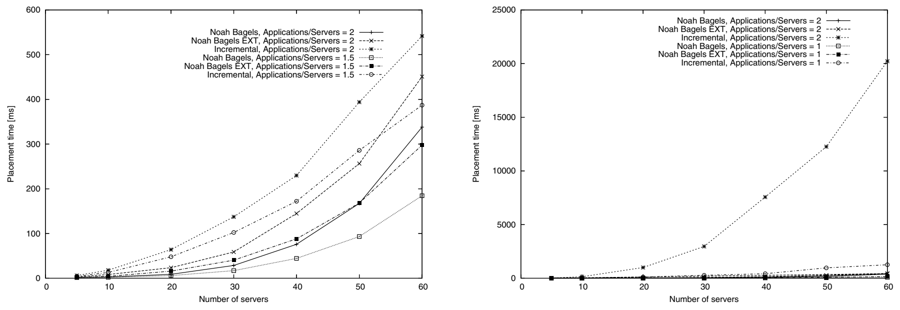
Each data point presented in the following sections was obtained as an average of 100 consecutive placement recomputations.

Number of Placement Changes. In this section we compare the three algorithms with respect to the number of changes to the previous placement required to satisfy the new demand. Figures 1(a) and 1(b) show this difference for various utilization factors and application-to-server ratios.

We can conclude that our incremental algorithm causes less perturbation of the placement than the previous algorithms. For reasonable utilization factors (e.g., 0.8 in Figure 1(a)), this difference is very significant.

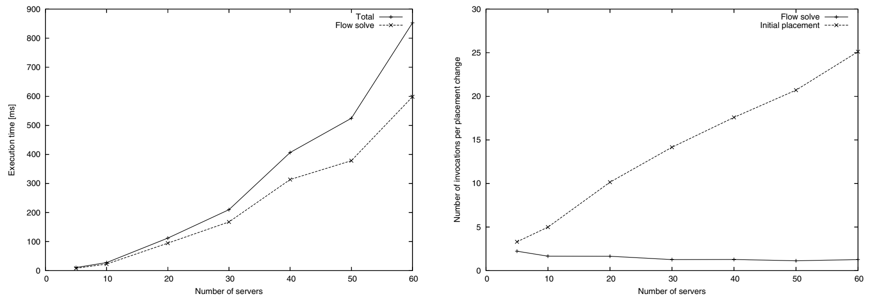
Running Time. In Figures 2(a) and 2(b), we show the execution time of a single placement recomputation. We observe that the incremental algorithm runs longer than the other two algorithms. In problem instances with a large number of applications and a high utilization factor, the running time performance of our algorithm is significantly worse than that of other algorithms. This is caused by the fact that in those hard problem instances, a placement satisfying the new demand might be obtained only by removing all or most of the current application instances, which requires the algorithm to execute a large number of iterations. However, even in these hard cases, the incremental algorithm allows the new placement for a 60-server cluster to be computed in 20 seconds. Note that starting a single application instance in our environment takes about 1 minute. A placement computation time that is a fraction of the application start-up time is acceptable in our application domain. Nevertheless, we investigate the computational time of the incremental algorithm further to understand the reasons for this lower performance.

Figure 3(a) presents the results obtained for utilization factor 0.9 and application to server ratio 2. It shows the total placement recomputation time and



(a) Utilization factor of 0.8 (b) Utilization factor of 0.99

Fig. 2. Execution time



(a) Total execution time (b) Number of invocations

Fig. 3. Contribution of flow-solve computation

the amount of time spent in flow-solve computation. Figure 3(b) presents the number of iterations and the number of flow-solve invocations per placement recomputation. We observe that solving the flow problem is the most significant contributor to the overall execution time. Even though the number of times the initial placement has to be invoked to compute the new placement is high in this scenario, compared to the number of times the flow problem has to be solved, the contribution of the actual placement computation remains low compared to that of the flow solver. It should be noted at this point that our flow-solve method uses the Simplex algorithm. Clearly, much more efficient techniques are available. We believe that, should one of the more efficient techniques be applied, the overall algorithm execution time would be significantly reduced.

6.2 Nonuniform Memory Requirements

Since in the case of non-uniform memory requirements γ_j there are no other heuristics to solve our problem, we need to compare the value of the solution provided by our algorithm either with the optimal value or with some lower bound on the optimal value. Finding a good lower bound seems a non-trivial task and remains an interesting open problem. The difficulty is that our objective function is essentially non-linear (concave) and it is hard (or impossible) to write a linear programming relaxation which does not have a large integrality gap. Finding good enumerative algorithms for our problem is also an interesting open question and again the main difficulty is to figure out how to construct a lower bound.

We compared the quality of solutions obtained by our algorithm with an optimal solution on instances of small size (5 servers and 14 applications). The optimal solution is found by complete enumeration with some additional tricks to cut down the search space.

The test instances are defined as follows: $m = 5$, $n = 14$. Server memory and capacity requirements are normalized and are equal to 1. Application memory requirements are chosen uniformly independently at random in the interval $[0, 2/3]$ and numbers of requests for each application are chosen uniformly independently at random in the interval $[0, 2\rho n/m]$, where ρ is the density which was 0.7, 0.8 or 0.9 in these experiments.

In the case of small density $\rho = 0.7$ the heuristic found an optimal solution in almost all the cases and sometimes it used one more assignment. Note that if the optimal value is zero our heuristic always finds an optimal solution.

In the case of higher density $\rho = 0.8$ the behavior of our heuristic becomes worse with $1-2$ additional changes of assignment, and occasionally our heuristic wipes out the previous solution completely, making $11 - 15$ changes. In the case of $\rho = 0.9$ the behavior is even worse. The difference between optimal value (which is $1 - 2$) and approximate value is significant $(5 - 6)$ in approximately 30% cases.

We believe that such behavior is due to a combination of bad factors such as small instance size, non-uniform capacities that sometimes do not allow us to utilize servers' memory well, and high density which decreases the number of "good" solutions.

7 Conclusion

In this paper we defined a new optimization problem and presented a heuristic algorithm to solve it. In this problem we produce copies of applications (at some cost) and assign them to servers. This is not a well-studied concept in classical scheduling where jobs or applications are usually simply assigned to machines. (A related concept is "duplication" that is popular in scheduling with communication delays [3, 4].) Another interesting feature of our problem is the objective function, which is the number of placement changes from time step to the next time step. This type of objective functions was not studied before in the optimization literature, to the best of our knowledge.

We think that it would be interesting to provide some theoretical evidence that our (or any other) algorithm performs well on a suitably defined class of instances for which finding feasible solutions is easy.

Another open problem, mentioned in the previous section, is to develop efficiently computable lower bounds which would allow more extensive and rigorous heuristic testing.

References

1. J. Kangasharju, J. Roberts, and K. W. Ross, Object replication strategies in content distribution networks, In 6th Int'l Workshop on Web Content Caching and Distribution (WCW), Boston, MA, 2001.
2. H. Kellerer, U. Pferschy and D. Pisinger, Knapsack Problems, Springer, 2004.
3. A. Munier and C. Hanen, Using duplication for scheduling unitary tasks on m processors with unit communication delays, Theoret. Comput. Sci. 178 (1997), 119–127.
4. C. Papadimitriou and M. Yannakakis, Towards an architecture-independent analysis of parallel algorithms. SIAM J. Comput. 19 (1990), 322–328.
5. D. N. Serpanos, L. Georgiadis, and T. Bouloutas, MMPacking: A load and storage balancing algorithm for distributed multimedia servers, IEEE Transactions on Circuits and Systems for Video Technology, 8(1):13 17, February 1998.
6. H. Shachnai and T. Tamir, On two class-constrained versions of the multiple knapsack problem, Algorithmica 29 (2001), 442–467.
7. H. Shachnai and T. Tamir, Noah Bagels - Some Combinatorial Aspects, International Conference on FUN with Algorithms (FUN), Isola d'Elba, June, 1998.
8. A. Turgeon, Q. Snell and M. Clement, Application placement using performance surfaces, in Proceedings of the Ninth International Symposium on High-Performance Distributed Computing, August 2000, Pittsburgh, PA, 229 - 236.
9. J. L. Wolf, P. S. Yu, and H. Shachinai, Disk load balancing for video-on-demand systems, ACM/Springer Multimedia Systems Journal, 5(6):358–370, 1997.
10. http://www.ibm.com/software/webservers/appserv/extend/