Application Placement on a Cluster of Servers

(extended abstract)

Bhuvan Urgaonkar, **Arnold Rosenberg** and **Prashant Shenoy**

Department of Computer Science,

University of Massachusetts, Amherst, MA 01003

{*bhuvan, rsnbrg, shenoy*}*@cs.umass.edu*

The APPLICATION PLACEMENT PROBLEM (APP) *arises in clusters of servers that are used for hosting large, distributed applications such as Internet services. Such clusters are referred to as hosting platforms. Hosting platforms imply a business relationship between the platform provider and the application providers: the latter pay the former for the resources on the platform. In return, the platform provider gives some kind of guarantees on resource availability to the applications. This implies that a platform should host only applications for which it has sufficient resources. The objective of the APP is to maximize the number of applications that can be hosted on the platform while satisfying their resource requirements. We show that the APP is NP-hard. We show that restricted versions of the APP admit polynomial-time approximation schemes. Finally, we present algorithms for the online version of the APP.*

1 Introduction

Server clusters built using commodity hardware and software are an increasingly attractive alternative to traditional large multiprocessor servers for many applications, in part due to rapid advances in computing technologies and falling hardware prices. We call such server clusters *hosting platforms*. Hosting platforms can be shared or dedicated. In dedicated hosting platforms [1, 15], either the entire cluster runs a single application (such as a web search engine), or each individual processing element in the cluster is dedicated to a single application (as in the hosting services provided by some data centers). In contrast, shared hosting platforms [3, 17] run a large number of different third-party applications (web-servers, streaming media servers, multi-player game servers, e-commerce applications, etc.), and the number of applications typically exceeds the number of nodes in the cluster. More specifically, each application runs on a subset of the nodes and these subsets may overlap. Whereas dedicated hosting platforms are

used for many niche applications that warrant their additional cost, economic reasons of space, power, cooling and cost make shared hosting platforms an attractive choice for many application hosting environments.

Shared hosting platforms imply a business relationship between the *platform provider* and the *application providers*: the latter pay the former for the resources on the platform. In return, the platform provider gives some kind of guarantees of resource availability to applications. This implies that a platform should admit only applications for which it has sufficient resources. In this work, we take the number of applications that a platform is able to host (admit) to be an indicator of the revenue that it generates from the hosted applications. The number of applications that a platform admits is related to the *application placement algorithm* used by the platform. A platform's application placement algorithm decides where on the cluster the different components of an application get placed. In this paper we study properties of the *application placement problem (APP)* whose goal is to maximize the number of applications that can be hosted on a platform. We show that APP is NP-hard and present approximation algorithms.

The rest of the paper is organized as follows. Section 2 develops a formal setting for the APP and discusses related work. Section 3 establishes the NP-hardness of the APP. Section 4 presents polynomial-time approximation algorithms for various restrictions of the APP. Section 5 begins to study the online version of the APP. Section 6 discusses directions for further work.

2 The Application Placement Problem

2.1 Notation and Definitions

Say that we have a cluster of n servers (also called nodes), $N_1, N_2, \ldots, N_n$. Each node has a given *capacity* (of *available resources*). Unless otherwise noted, nodes are *homogeneous*, in the sense of having the same initial capacities. The APP appropriates portions of nodes' capacities; a node that still has its initial capacity is said to be *empty*. Let m denote the number of applications to be placed on the cluster and let us represent them as A_1, ..., A_m. Further, each application is composed of one or more *capsules*. A capsule may be thought of as the smallest component of an application for the purposes of placement — all the processes, data etc., belonging to a capsule must be placed on the same node. Capsules provide a useful abstraction for logically partitioning an application into sub-components and for exerting control over the distribution of these components onto different nodes. If an application wants certain components to be placed together on the same node (e.g., because they communicate a lot), then it could bundle these as one capsule. Some applications may want their capsules to be placed on different nodes. An important reason for doing this is to improve the availability of the application in the face of node failures — if a node hosting a capsule of the application fails, there would still be capsules on other nodes. An example of such an application is a replicated web server. We refer to this requirement as *the capsule placement restriction.* In

what follows, we look at the APP both with and without the capsule placement restriction.

In general, each capsule in an application would require guarantees on access to multiple resources. In this work, we consider just one resource, such as the CPU or the network bandwidth. We assume a simple model where a capsule specifies its resource requirement as a fraction of the resource capacity of a node in the cluster (i.e., we assume that the resource requirement of each capsule is less than the capacity of a node). A capsule can be placed on a node just when the sum of its resource requirement and those of the capsules already placed on the node does not exceed the resource capacity of the node. We say that an application can be *placed* only if *all* of its capsules can be placed simultaneously. It is easy to see that there can be more than one way in which an application may be placed on a platform. We refer to the total number of applications that a placement algorithm could place as the *size* of the placement.

Definition 1 The offline APP: *Given a cluster of n empty nodes $N_1, \ldots, N_n$, and a set of m applications $A_1, \ldots, A_m$, determine a placement of maximum size.*

Definition 2 The on-line APP: *Given a cluster of n empty nodes $N_1, \ldots, N_n$, and a set of m applications $A_1, \ldots, A_m$, determine a placement of maximum size while satisfying the following conditions: (1) the applications should be considered for placement in increasing order of their indices, and (2) once an application has been placed, it cannot be moved while the subsequent applications are being placed.*

Definition 3 Single-Capsule Application Placement Problem (DEC_MAX_CAP): *Given n empty nodes $N_1, \ldots, N_n$, a set of m single-capsule applications $C_1, \ldots, C_m$, and an integer k, determine if a placement of size k exists.*

Lemma 1 *DEC_MAX_CAP is NP-complete.*

Proof: The proof consists of two parts.

- DEC_MAX_CAP is in NP: Given an instance of DEC_MAX_CAP and a placement, we can in polynomial time verify — (a) if this is a valid placement — this involves checking for each node that the sum of the requirements of all the capsules placed on it does not exceed the node capacity , and (b) if the size of the placement is k, i.e., could k capsules be placed. Thus, we have shown that DEC_MAX_CAP is in NP.

- BIN-PACKING reduces to DEC_MAX_CAP: Let us first state the decision version of the bin-packing problem which is known to be NP-complete [11].

  BIN-PACKING: Given a set of m objects $O_1, \ldots, O_m$ of sizes $s_1, \ldots, s_m$ respectively, and an integer k, determine if all the objects can be placed into k bins, where each bin has unit capacity.

Consider the following polynomial-time reduction from BIN-PACKING to DEC_MAX_CAP. Given an input to BIN-PACKING, we construct an input to DEC_MAX_CAP as follows. Corresponding to each object in the input to BIN-PACKING, we construct a capsule whose requirement is equal to the size of the object. Next, we construct k nodes, each with unit capacity. These node- and capsule-sets along with the integer m comprise the input to DEC_MAX_CAP.

It is easy to see that the above is a reduction. Assume the input to BIN-PACKING had m objects and the integer k. The input to DEC_MAX_CAP that we construct would have k nodes, m capsules and the integer m. If the m objects can fit into k bins, then clearly we can place the m capsules in k nodes. On the other hand, if the m objects cannot fit into k bins, then the m capsules cannot all be placed into the k nodes.

This completes the proof. ■

Definition 4 Decision Version of the APP (DEC_MAX_APP): *Given n empty nodes N_1, ..., N_n, a set of m applications A_1, ..., A_m, and an integer k, determine if a placement of size k exists.*

Lemma 2 *DEC_MAX_APP is NP-complete.*

Proof: Restrict DEC_MAX_APP to DEC_MAX_CAP by allowing only applications with one capsule. ■

Finally, we can show the NP-hardness of the APP.

Theorem 1 *The APP is NP-hard.*

Proof: DEC_MAX_APP is the decision version of offline APP. Therefore, the NP-hardness of DEC_MAX_APP shown in Lemma 2 proves the NP-hardness of OFF_PLACE. ■

Definition 5 Polynomial-time approximation scheme (PTAS): *A set of algorithms $A_\epsilon, \epsilon > 0$, where each A_ϵ is a $(1 + \epsilon)$-approximation algorithm and the execution time is bounded by a polynomial in the length of the input. The execution time may depend on the choice of ϵ.*

2.2 Related Work

Two generalizations of the classical knapsack problem are relevant to our discussion of the APP. These are the *Multiple Knapsack Problem* (MKP) and the *Generalized Assignment Problem* (GAP). In MKP, we are given a set of n items and m bins (knapsacks) such that each item i has a profit $p(i)$ and a size $s(i)$, and each bin j has a capacity $c(j)$. The goal is to find a subset of items of maximum profit that has a feasible packing in the bins. MKP is a special case of GAP where the profit and the size of an item can vary based on the specific bin that it is assigned to.

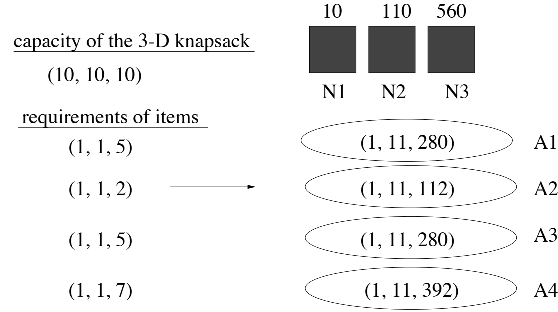

Figure 1: An example of the gap-preserving reduction from the Multi-dimensional Knapsack problem to the general offline placement problem.

GAP is APX-hard (see [13] for a definition of APX-hardness) and [16] provides a 2-approximation algorithm for it. This was the best result known for MKP until a polynomial-time PTAS was presented for it in [5]. It should be observed that the offline APP is a generalization of MKP where an item may have multiple components that need to be assigned to different bins (the profit associated with an item is 1). Further, [5] shows that slight generalizations of MKP are APX-hard. This provides reason to suspect that the APP may also be APX-hard (and hence may not have a PTAS). In Sections 3 and 4 we show that certain restrictions of the APP admit PTAS.

Another closely related problem is a "multidimensional" version of the MKP where each item has requirements along multiple dimensions, each of which must be satisfied to successfully place it. The goal is to maximize the total profit yielded by the items that could be placed. A heuristic for solving this problem is described in [12]. However, the authors evaluate this heuristic only through simulations and do not provide any analytical results on its performance.

3 Hardness of Approximating the APP

In this section, we demonstrate that a restricted version of the APP admits a PTAS. The capsule placement restriction is assumed to hold throughout this section.

Definition 6 Gap-preserving reduction: *[8] Let Π and Π' be two maximization problems. A gap-preserving reduction from Π to Π' with parameters (c,ρ), (c',ρ') is a polynomial-time algorithm f. For each instance I of Π, algorithm f produces an instance $I' = f(I)$ of Π'. The optima of I and I', say $OPT(I)$ and $OPT(I')$ respectively, satisfy the following property:*

$$OPT(I) \geq c \Longrightarrow OPT(I') \geq c', \tag{1}$$

$$OPT(I) < \frac{c}{\rho} \Longrightarrow OPT(I') < \frac{c'}{\rho'}. \tag{2}$$

Here c and ρ are functions of $|I|$, the size of instance $|I|$, and c', ρ' are functions of $|I'|$. Also, $\rho(I)$, $\rho'(I') \geq 1$.

Suppose we wish to prove the inapproximability of problem Π'. Suppose further that we have a polynomial time reduction τ from SAT to Π that ensures, for every boolean formula ϕ:

$$\phi \in SAT \Longrightarrow OPT(\tau(\phi)) \geq c,$$

$$\phi \notin SAT \Longrightarrow OPT(\tau(\phi)) < \frac{c}{\rho}.$$

Then composing this reduction with the reduction of Definition 6 gives a reduction $f \circ \tau$ from SAT to Π' that ensures:

$$\phi \in SAT \Longrightarrow OPT(f(\tau(\phi))) \geq c',$$

$$\phi \notin SAT \Longrightarrow OPT(f(\tau(\phi))) < \frac{c'}{\rho'}.$$

In other words, $f \circ \tau$ shows that achieving an approximation ratio ρ' for Π' is NP-hard. So a gap-preserving reduction can be used to exhibit the hardness of approximating a problem. We now give a gap-preserving reduction from the *Multi-dimensional* 0-1 *Knapsack Problem* [2] to a restricted version of the APP. We begin with definition of the former problem (which is also known as the *Packing Integer Problem* [4]).

Definition 7 Multi-Dimensional 0-1 Knapsack Problem (MDKP)*: For a fixed positive integer k, the k-dimensional knapsack problem is the following:*

$$\text{Maximize} \sum_{i=1}^{n} c_i x_i$$

Subject to

$$\sum_{i=1}^{n} a_{ij} x_i \leq b_j, j = 1, \ldots, k,$$

where: n is a positive integer; each $c_i \in \{0,1\}$ and $\max_i c_i = 1$; the a_{ij} and b_i are non-negative real numbers; all $x_i \in \{0,1\}$. Define $B = \min_i b_i$.

To see why the above maximization problem models a multi-dimensional knapsack problem, think of a k-dimensional knapsack with the capacity vector $(b_1, \ldots, b_k)$. That is, the knapsack has capacity b_1 along dimension 1, b_2 along dimension 2 etc. Think of n items $I_1, \ldots, I_n$, each having a k-dimensional requirement vector. Let the requirement vector for item I_j be $(a_{j1}, \ldots, a_{jk})$. It is easy to see that the above maximization problem is equivalent to the problem of maximizing the number of k-dimensional items that can be packed in the k-dimensional knapsack such that for any d ($1 \leq d \leq k$) the sum of the requirements along dimension d of the packed items does not exceed the capacity of the knapsack along dimension d.

Hardness of approximating MDKP: For fixed k there is a PTAS for MDKP [10]. For large k the randomized rounding technique of [14] yields integral solutions of value $\Omega(OPT/d^{1/B})$. [4] establishes that MDKP is hard to approximate within a factor of $\Omega(k^{\frac{1}{B+1}-\epsilon})$ for every fixed B, thus establishing that randomized rounding essentially gives the best possible approximation guarantees.

Theorem 2 *For a fixed integer k, there exists a PTAS for the offline placement problem that has the following restrictions: (1) all the capsules have a positive requirement and (2) there exists a constant M, such that $\forall i, j(1 \leq j \leq k, 1 \leq i \leq n), M \geq b_j/a_{ji}$.*

Proof: We explain later in this proof why the two restrictions mentioned above arise. We begin by describing the reduction.

The reduction: Consider the following mapping from instances of k-MDKP to offline APP:

Suppose the input to k-MDKP is a knapsack with capacity vector $(b_1, \ldots, b_k)$. Also let there be n items $I_1, \ldots, I_n$. Let the requirement vector for item I_j be $(a_{j1}, \ldots, a_{jk})$. We create an instance of offline APP as follows. The cluster has k nodes $N_1, \ldots, N_k$. There are n applications $A_1, \ldots, A_n$, one for each item in the input to k-MDKP. Each of these applications has k capsules. The k capsules of application A_i are denoted $c_i^1, \ldots, c_i^k$. Also, we refer to c_i^j as the jth capsule of application A_i. We now describe how we assign capacities to the nodes and requirements to the applications we have created. This part of the mapping proceeds in k stages. In stage s, we determine the capacity of node N_s and the requirements of the sth capsule of all the applications. Next, we describe how these stages proceed.

Stage 1: Assigning capacity to the first node N_1 is straightforward. We assign it a capacity $C(N_1) = b_1$. The first capsule of application A_i is assigned a requirement $r_i^1 = a_{i1}$.

Stage s ($1 < s \leq k$)*:* The assignments done by stage s depend on those done by stage $s-1$. We first determine the smallest of the requirements along dimension s of the items in the input to k-MDKP, that is, $r_{min}^s = \min_{i=1}^{n}(a_{is})$. Next we determine the scaling factor for stage s, SF_s as follows:

$$SF_s = \lfloor C(N_{s-1})/r_{min}^s \rfloor + 1. \tag{3}$$

Recall that we assume that $\forall s, r_{min}^s > 0$. Now we are ready to do the assignments for stage s. Node N_s is assigned a capacity $C(N_s) = b_i \times SF_s$. The sth capsule of application A_i is assigned a requirement $r_i^s = a_{is} \times SF_s$.

This concludes our mapping. Let us now take a simple example to better explain how this mapping works. Consider the instance of input T to MDKP shown on the left of Figure 1. Here we have $k = 3, n = 4$. We create 3 nodes N_1, N_2 and N_3. We create 4 applications A_1, A_2, A_3 and A_4, each with 3 capsules. Let us now consider how the 3 stages in our mapping proceed.

Stage 1: We assign a capacity of 10 to N_1 and requirements of 1 each to the first capsules of all four applications.

Stage 2: The scaling factor for this stage, SF_2 is 11. So we assign a capacity of 110 to N_2 and requirements of 11 each to the second capsules of the four applications.

Stage 3: The scaling factor for this stage, SF_3 is $\lfloor 110/s \rfloor + 1 = 56$. So we assign N_3 a capacity of 560. The third capsules of the four applications are assigned requirements of $280, 112, 280$ and 392 respectively.

Correctness of the reduction: We show that the mapping described above is a reduction.

($\Longrightarrow$) Assume there is a packing P of size $m \leq n$. Denote the n items in the input to k-MDKP as $I_1, \ldots, I_n$. Without loss of generality, assume that the m items in P are $I_1, \ldots, I_m$. Therefore we have,

$$\sum_{i=1}^{m} a_{ij} \leq b_j, \;\; j = 1, \ldots, k. \tag{4}$$

Consider this way of placing the applications that the mapping constructs on the nodes $N_1, \ldots, N_k$. If item $I_i \in P$, place application A_i as follows: $\forall j, 1 \leq j \leq k$, place capsule c_i^j on node N_j. We claim that we will be able to place all m applications corresponding to the m items in P. To see why consider any node $N_i (1 \leq i \leq k$. The capacity assigned to N_i is SF_i times the capacity along dimension i of the k-dimensional knapsack in the input to k-MDKP, where $SF_i \geq 1$. The requirements assigned to the ith capsules of all the applications are also obtained by scaling by the same factor SF_i the sizes along the ith dimension of the items. Multiplying both sides of (4) by SF_i we get,

$$SF_i \times \sum_{i=1}^{m} a_{ij} \leq SF_i \times b_j, \;\; j = 1, \ldots, k.$$

Observe that the term on the right is the capacity assigned to N_i. The term on the left is the sum of the requirements of the ith capsules of the applications corresponding to the items in P. This shows that node N_i can accommodate the ith capsules of the applications corresponding to the m items in P. This implies that there is a placement of size m.

($\Longleftarrow$) Assume that there is a placement L of size $m \leq n$. Let the n applications be denoted $A_1, \ldots, A_n$. Without loss of generality, let the m applications in L be $A_1, \ldots, A_m$. Also denote the set of the sth capsules of the placed applications by $Cap_s, 1 \leq s \leq k$.

We make the following key observations:

- For any application to be successfully placed, its ith capsule must be placed on node N_i. Due to the scaling by the factor computed in Eq. (3), the requirements assigned to the sth ($s > 1$) capsules of the applications are strictly greater than the capacities of the nodes $N_1, \ldots, N_{s-1}$. Consider the kth capsules of the applications first. The only node these can be placed on is N_k. Since no two capsules of an application may be placed on the same node, this implies that the $k - 1^{th}$ capsules of the applications may be placed only on N_{k-1}. Proceeding in this manner, we find that the claim holds for all the capsules.

- Since for all s ($1 \leq s \leq k$), the node capacities and the requirements of the sth capsules are scaled by the same multiplicative factor, the fact that the m capsules in Cap_s could be placed on N_s implies that the m items $I_1, \ldots, I_m$ can be packed in the knapsack in the sth dimension.

Combining these two observations, we find that a packing of size m must exist.

Time and space complexity of the reduction: This reduction works in time polynomial in the size of the input. It involves k stages. Each stage involves computing a scaling factor (this involves performing a division) and multiplying $n+1$ numbers (the capacity of the knapsack and the requirements of the n items along the relevant dimension).

Let us consider the size of the input to the offline placement problem produced by the reduction. Due to the scaling of capacities and requirements described in the reduction, the magnitudes of the inputs increase by a multiplicative factor of $O(M^j)$ for node N_j and the jth capsules. If we assume binary representation this implies that the input size increases by a multiplicative factor of $O(M^{j/2}), 1 < j \leq k$. Overall, the input size increases by a multiplicative factor of $O(M^k)$. For the mapping to be a reduction, we need this to be a constant. Therefore, our reduction works only when we impose the following restrictions on the offline APP: (1) k and M are constants, and (2) all the capsule requirements are positive.

Gap-preserving property of the reduction: The reduction presented is gap-preserving because the size of the optimal solution to the offline placement problem is *exactly equal* to the size of the optimal solution to MDKP. More formally, in terms of the terminology used in Definition 6, we can set $c = c' = \rho = \rho' = 1$. Putting these values in Equations 1 and 2, we find that the following conditions hold:

[OPT(MDKP) ≥ 1] $\Longrightarrow$ [OPT(offline APP) ≥ 1]

[OPT(MDKP) < 1] $\Longrightarrow$ [OPT(offline APP) < 1]

This proves that the reduction is gap-preserving. Together, these results prove that the restricted version of the offline APP described in Theorem 2 admits a PTAS. ■

4 Offline Algorithms for APP

In this section we present and analyze offline approximation algorithms for several variants of the placement problem. Except in Section 4.4, we assume that the cluster is homogeneous, in the sense specified earlier.

4.1 Placement of Single-Capsule Applications

We consider a restricted version of offline APP in which every application has exactly one capsule. We provide a polynomial-time algorithm for this restriction of offline APP, whose placements are within a factor 2 of optimal.

The approximation algorithm works as follows. Say that we are given n nodes $N_1, \ldots, N_n$ and m single-capsule applications $C_1, \ldots, C_m$ with requirements $R_1, \ldots, R_m$. Assume that the nodes have unit capacities. The algorithm first sorts the applications in nondecreasing order of their requirements. Denote the sorted applications by $c_1, \ldots, c_m$ and their requirements by $r_1, \ldots, r_m$. The algorithm considers the applications in this order. An application is placed on the "first" node where it can be accommodated (i.e., the node with the smallest index that has sufficient resources for it). The algorithm terminates once it has considered all the applications or it finds an application that cannot be placed, whichever occurs earlier. We call this algorithm FF_SINGLE.

Lemma 3 *FF_SINGLE has an approximation ratio of 2.*

Proof: Denote by k_{FF} the number of single-capsule applications that FF_SINGLE could place on n nodes. Denote by k_{OPT} the number of single-capsule applications that an optimal algorithm could place.

If FF_SINGLE places all the applications on the given set of nodes, then it has matched the optimal algorithm and we are done.

Consider the case when there is at least one application that FF_SINGLE could not place. Since all capsules have requirements less than the capacity of a node, this implies that there is no empty node after the placement. Our proof is based on the following key observation: *if FF_SINGLE could not place all the applications, then there can be at most one node that is more than half empty*. To see why, assume that there are two nodes n_i and n_j that are more than half empty, $i < j$. Since the application(s) (equivalently, capsule(s)) placed on n_j can be accommodated in n_i, the assumed situation can never arise in a placement found by FF_SINGLE.

As a result we have the following:

$$r_1 + \ldots + r_{k_{FF}} \geq n/2$$

The best that an optimal algorithm can do is to use up all the capacity on the nodes. So we have:

$$r_1 + \ldots + r_{k_{FF}} + \ldots + r_{k_{OPT}} \leq n$$

Since $r_{k_{OPT}} \geq \ldots \geq r_{k_{FF}} \geq \ldots \geq r_1$, the set $\{c_1, \ldots, c_{FF}\}$ would have at least as many applications as the set $\{c_{k_{FF}}, \ldots, c_{k_{OPT}}\}$. Consequently, FF_SINGLE has placed at least half as many applications as an optimal algorithm. This gives us the desired performance ratio of 2. ■

4.2 Placement without the Capsule Placement Restriction

Now we show that an approximation algorithm based on first-fit gives an approximation ratio of 2 for multi-capsule applications, provided that they don't have the capsule placement restriction.

The approximation algorithm works as follows. Say that we are given n nodes $N_1, \ldots, N_n$ and m applications $A_1, \ldots, A_m$ with requirements $R_1, \ldots, R_m$ (the requirement of an application is the sum of the requirements of its capsules). Assume that the nodes have unit capacities. The algorithm first orders the applications in nondecreasing order of their requirements. Denote the ordered applications by $a_1, \ldots, a_m$ and their requirements by $r_1, \ldots, r_m$. The algorithm considers the applications in this order. An application is placed on the "first" set of nodes where it can be accommodated (i.e., the nodes with the smallest indices that have sufficient resources for all its capsules). The algorithm terminates once it has considered all the applications or it finds an application that cannot be placed, whichever occurs first. We call this algorithm FF_MULTIPLE_RES.

Lemma 4 *FF_MULTIPLE_RES has an approximation ratio that approaches 2 as the number of nodes in the cluster grows.*

Proof: Denote by k_{FF} the number of applications that FF_MULTIPLE_RES could place on n nodes, completely (meaning all the capsules of the application could be placed) or partially (meaning at least one capsule of the application could not be placed). Denote by k_{OPT} the number of applications that an optimal algorithm could place on the same set of nodes.

If FF_MULTIPLE_RES places all the applications on the given set of nodes, then it has matched the optimal algorithm and we are done.

Consider the case when there is at least one application that FF_MULTIPLE_RES could not place. Since all capsules have requirements less than the capacity of a node, this implies that there is no empty node after the placement. The set of applications placed by FF_MULTIPLE_RES is $\{a_1, \ldots, a_{k_{FF}}\}$. Observe that except for the last of these applications, namely $a_{k_{FF}}$, all the applications would have been placed completely. The application $a_{k_{FF}}$ may or may not have been completely placed. In either case, the following key observation would hold: *if FF_MULTIPLE_RES could not place all the applications, then there can be at most one node that is more than half empty*. To see why, assume that there are two nodes N_i and N_j that are more than half empty, $i < j$. Since the capsules placed on N_j can be accommodated in N_i, the assumed situation can never arise in a placement found by FF_MULTIPLE_RES.

As a result we have the following:

$$R_1 + \ldots + R_{k_{FF}-1} + {R'}_{k_{FF}} \geq n/2,$$

where ${R'}_{k_{FF}}$ is the sum of the requirements of the capsules of application $a_{k_{FF}}$ that could be placed on the cluster.

Since ${R'}_{k_{FF}} \leq R_{k_{FF}}$, this implies the following:

$$R_1 + \ldots + R_{k_{FF}} \geq n/2$$

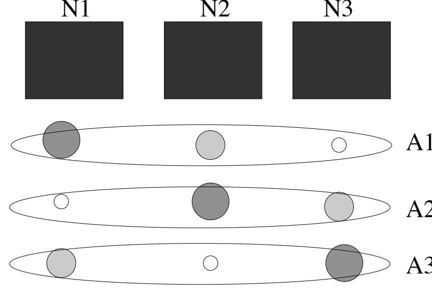

Figure 2: An example of striping-based placement.

The best that an optimal algorithm can do is to use up all the capacity on the nodes. So we have:

$$R_1 + \ldots + R_{k_{FF}} + \ldots + R_{k_{OPT}} \leq n$$

Since $R_{k_{OPT}} \geq \ldots \geq R_{k_{FF}} \geq \ldots \geq R_1$, the set $\{c_1, \ldots, c_{FF}\}$ would have at least as many applications as the set $\{a_{k_{FF}}, \ldots, a_{k_{OPT}}\}$. Discounting $a_{k_{FF}}$ which may not have been completely placed, we find that FF_MULTIPLE_RES guarantees to place one less than half as many applications as an optimal algorithm can place. As the number of nodes grows, the performance ratio of FF_MULTIPLE_RES tends to 2. ■

4.3 Placement of Identical Applications

Two applications are identical if their sets of capsules are identical. Below we present a placement algorithm based on "striping" applications across the nodes in the cluster and determine its approximation ratio.

Striping-based placement: Assume that the applications have k capsules each, with requirements $r_1, \ldots, r_k$ ($r_1 \leq \ldots \leq r_k$). The algorithm works as follows. Let us denote the nodes as $N_1, \ldots, N_m$. The nodes are divided into sets of size k each. Since $m \geq k$, there will be at least one such set. The number of such sets is $\lceil m/k \rceil$. Let $t = \lfloor m/k \rfloor, t \geq 1$. Let us denote these sets as $S_1, \ldots, S_{t+1}$. Note that S_{t+1} may be an empty set, $0 \leq |S_{t+1}| \leq k-1$. The algorithm considers these sets in turn and "stripes" as many unplaced applications on them as it can. The set of nodes under consideration is referred to as the *current set of k nodes*.

We illustrate the notion of striping using an example. In Figure 2, we have three nodes and a number of identical 3-capsule applications to be placed on them. Striping places the first capsule of A_1 on N_1, second on N_2 and third on N_3. For the next application A_2, it places the first capsule on N_2, second on N_3 and third on N_1.

When the current set of k nodes gets exhausted and there are more applications to place, the algorithm takes the next set of k nodes and continues. The algorithm terminates when the nodes in S_t are exhausted, or all applications have been placed, whichever occurs earlier. Note that none of the nodes in the (possibly empty) set S_{t+1} are used for placing the applications.

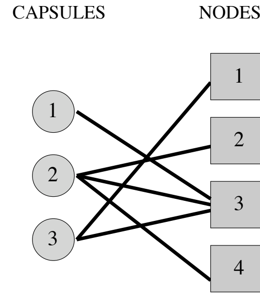

Figure 3: A bipartite graph indicating which capsules can be placed on which nodes

Lemma 5 *The striping-based placement algorithm yields an approximation ratio of* $\left(\frac{t+1}{t}\right)$ *for identical applications, where* $t = \lfloor m/k \rfloor$.

Proof: It is easy to observe that the striping-based placement algorithm places an optimal number of identical applications on a homogeneous cluster of size k (due to symmetry). Since the striping-based algorithm places applications on the sets $S_1, \ldots, S_t$ and lets S_{t+1} go unused, and since the nodes are homogeneous and the applications are identical, its approximation ratio is strictly less than $\left(\frac{t+1}{t}\right)$.

■

4.4 Max-First Placement

We have considered so far restricted versions of the offline APP and have presented heuristics that have approximation ratios of 2 or better. In this section we turn our attention to the general offline APP. We let the nodes in the cluster be heterogeneous. We find that this problem is much harder to approximate than the restricted cases. We first present a heuristic that works differently from the first-fit based heuristics we have considered so far. We obtain an approximation ratio of k for this heuristic, where k is the maximum number of capsules in any application.

Our heuristic works as follows. It associates with each application a *weight* which is equal to the requirement of the *largest* capsule in the application. The heuristic considers the applications in nondecreasing order of their weights. We use a bipartite graph to model the problem of placing an application on the cluster. In this graph, we have one vertex for each capsule in the application and for each node in the cluster. Edges are added between a capsule and a node if the node has sufficient capacity for hosting the capsule. We say that the node is *feasible* for the capsule. An example is shown in Figure 3. In Lemma 6 we show that an application can be placed on the cluster if and only if there is a matching of size equal to the number of capsules in the application. We solve the maximum matching problem on this bipartite graph [7]. If the matching has size equal to the number of capsules, we place the capsules of the application on the nodes that the maximum matching connects them to. Otherwise, the application

cannot be placed and the heuristic terminates. We refer to this heuristic as *Max-First*.

Lemma 6 *An application with k capsules can be placed on a cluster if and if only there is a matching of size k in the bipartite graph modeling its placement on the cluster.*

Proof: We prove each direction in turn.

($\Longrightarrow$) Consider a matching of size k in the bipartite graph. It must have an edge connecting each capsule to a node. Further, no two capsules could be connected to the same node (since this is a matching). Since edges denote feasibility, this is clearly a valid placement.

($\Longleftarrow$) Suppose there is no matching of size k in the bipartite graph. Then there must be at least one capsule that can not be assigned to a node independent of the other capsules. In other words, there must be at least one capsule that would need to share a node with some other capsule(s). Therefore this application can not be placed without violating the capsule placement restriction.

This concludes the proof. ■

Lemma 7 *The placement heuristic Max-First described above has an approximation ratio of k, where k is the maximum number of capsules in an application.*

Proof: Let A represent the set of all the applications and $|A| = m$. Denote by n the number of nodes in the cluster and the nodes themselves by $N_1, \ldots, N_n$. Let us denote by H the set of applications that Max-First places. Let O denote the set of applications placed by any optimal placement algorithm. Clearly, $|H| \leq |O| \leq m$. Represent by I the set of applications that both H and O place; that is, $I = H \cap O$. Further, denote by R the set of applications that neither H nor O places.

The basic idea behind this proof is as follows. We focus in turn on the applications that only Max-First and the optimal algorithm place (that is, applications in $(H - I)$ and $(O - I)$), and compare the sizes of these sets. A relation between the sizes of these sets immediately yields a relation between the sizes of the sets H and O. (Observe that $(H - I)$ and $(O - I)$ may both be empty, in which case we have the claimed ratio trivially.)

Consider the placement given by Max-First. Remove from this all the applications in I, and deduct from the nodes the resources reserved for the capsules of these applications. Denote the resulting nodes by $N_1^{H-I}, \ldots, N_n^{H-I}$. Do the same for the placement given by the optimal algorithm, and denote the resulting nodes by $N_1^{O-I}, \ldots, N_n^{O-I}$. To understand the relation between the applications placed on these node sets by Max-First and the optimal algorithm, suppose Max-First places y applications from the set $(H - I)$ on the nodes $N_1^{H-I}, \ldots, N_n^{H-I}$. Let us denote the

applications in $(A - I)$ by $B_1, \ldots, B_y, \ldots, B_{|A-I|}$, where the applications are arranged in nondecreasing order of the size of their largest capsule. That is, $l(B_1) \leq \ldots \leq l(B_y) \leq \ldots \leq l(B_{|A-I|})$, $l(x)$ being the requirement of the largest capsule in application x. From the definition of Max-First, the y applications that it places are $B_1, \ldots, B_y$. Also, the applications that the optimal algorithm places on the set of nodes $N_1^{O-I}, \ldots, N_n^{O-I}$ must be from the set $B_{y+1}, \ldots, B_{|A-I|}$. We make the following useful observation about the applications in the set $B_{y+1}, \ldots, B_{A-I}$: *for each of these applications, the requirement of the largest capsule is at least* $l(B_y)$. Based on this we infer the following: Max-First will exhibit the worst approximation ratio when all the applications in $(H - I)$ have k capsules, each with requirement $l(B_y)$, and all the applications in $(O - I)$ have $(k-1)$ capsules with requirement 0, and one capsule with requirement $l(B_y)$. Since the total capacities remaining on the node sets $N_1^{H-I}, \ldots, N_n^{H-I}$ and $N_1^{O-I}, \ldots, N_n^{O-I}$ are equal, this implies that in the worst case, the set $O - I$ would contain k times as many applications as $H - I$. Based on the above, we can prove an approximation ratio of k for Max-First as follows:

$$|O| \;=\; |O - I| + |I| \;\leq\; k \cdot |H - I| + |I|$$

$$\leq\; k \cdot (|H - I| + |I|) \;=\; k \cdot |H|$$

This concludes our proof. ∎

4.5 LP-Relaxation Based Placement

Say that we have n nodes and m applications. Each application can be thought of as having n capsules (we can add some capsules with requirement 0 to an application with fewer than n capsules). Denote by r_{ij} the requirement of capsule j of application i and by C_k the capacity of node k. We construct the variable x_{ijk} with the following meaning:

$$x_{ijk} = \begin{cases} 1 & \text{if capsule } j \text{ of app } i \text{ is placed on node } k \\ 0 & \text{otherwise} \end{cases}$$

Additionally, define:

$$x_{ij} = \sum_{k=1}^{n} x_{ijk} \;\text{ and }\; x_i = \sum_{j=1}^{n} x_{ij}$$

The placement problem can be recast as the following Integer Linear Program:

$$\text{Maximize } \sum_{i=1}^{m} x_i$$

Subject to

$$\forall i, k : \sum_{j=1}^{n} x_{ijk} \leq 1$$

$$\forall k : \sum_{i=1}^{n} \sum_{j=1}^{m} x_{ijk} \times r_{ik} \leq C_k$$

$$\forall i : x_{i1} = ... = x_{in}$$

The first step of the LP-relaxation based placement consists of solving the Linear Program obtained by removing the restriction $x_{ijk} \in \{0, 1\}$ and instead allowing x_{ijk} to take real values in $[0, 1]$. Denote the value assigned to x_{ijk} in this step by x'_{ijk}. This is followed by a step in which x_{ijk} are converted back to integers using the following rounding:

$$x_{ijk} = \begin{cases} 1 & \text{if } x'_{ijk} \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Finally, the capacities of some nodes may have been exceeded due to the above rounding. For such nodes, we remove the capsules placed on them in nonincreasing order of their requirements till the remaining capsules fit in the node. Observe that removing a capsule of an application implies also removing all of its other capsules.

5 The Online APP

In the online version of the APP, the applications arrive one by one. We require the following from any online placement algorithm — *the algorithm must place a newly arriving application on the platform if it can find a placement for it without moving any already placed capsule*. This captures the placement algorithm's lack of knowledge of the requirements of the applications arriving in the future. We assume a heterogeneous cluster throughout this section.

5.1 Online Placement Algorithms

The online placement algorithms consider applications for placement one by one, as they arrive. Consider the situation the online placement algorithm is faced with on the arrival of a new application. We model this using a graph, in which we have one vertex for each capsule in the application and for each node in the cluster. Edges are added between a capsule and a node if the node has sufficient resources for hosting the capsule. We say that the node is *feasible* for the capsule. This gives us a bipartite graph that we call the *feasibility graph* of the new application. An example of a feasibility graph is shown in Figure 3. As described in Section 4.4, a maximum matching on this graph can be used to find a placement for the application if one exists.

Let us denote by $\mathcal{A}$ the class of greedy online placement algorithms that work as follows. Any such algorithm considers the capsules of the newly arrived application in nondecreasing order of their degrees in the feasibility graph of the application. If there are no feasible nodes for a capsule, the algorithm terminates. Otherwise, the capsule is placed on one of the nodes feasible for it. After this, all edges connecting any unplaced capsules to this node are removed from the graph. This is repeated until all capsules have been placed or the algorithm cannot find any feasible nodes for some capsule.

We define two members of $\mathcal{A}$ below.

Definition 8 Best-fit based Placement (BF): *When faced with a choice of more than one node to place a capsule on, BF chooses the node with the least remaining capacity.*

Definition 9 Worst-fit based Placement (WF): *When faced with a choice of more than one node to place a capsule on, WF chooses the node with the most remaining capacity.*

We can show the following regarding the approximation ratios of BF and WF, denoted R_{BF} and R_{WF} respectively.

Lemma 8 *BF can perform arbitrarily worse than the optimal.*

Proof: Let m be the total number of applications and n the number of nodes and let $m > n$. Let all the nodes have a capacity of 1. Suppose that n single-capsule applications arrive first, each capsule with a requirement $1/n$. BF puts them all on the first node. Next, $(m - n)$ n-capsule applications arrive with each capsule having non-zero requirement. Since the first node has no capacity left, BF will not be able to place any of these. WF would have worked as follows on this input. Each of the first n single-capsule applications would have been placed on a separate node, resulting in each of the n nodes having a remaining capacity $(1 - 1/n)$, available for the n-capsule applications. Therefore,

$$\exists\, input \quad s.t. \quad \frac{BF}{WF} \geq \frac{m}{n}$$

Also, since WF is optimal for this input, we have

$$R_{BF} \geq \frac{m}{n}$$

Since m can be arbitrarily larger than n (by making the n-capsule applications have capsules with requirements tending to 0), R_{BF} cannot be bounded from above. ■

Lemma 9 *$R_{WF} \geq (2 - 1/n)$ for an n-node cluster.*

Proof: Say that the cluster has n nodes, each with unit capacity. Consider the following sequence of application arrivals. Suppose that n single-capsule applications arrive first, each capsule with a requirement ϵ that approaches 0. WF places each of these applications on a separate node, resulting in each of the n nodes having a remaining capacity $(1 - \epsilon)$. Next, n single-capsule applications arrive, each capsule with a requirement of 1. Since no node

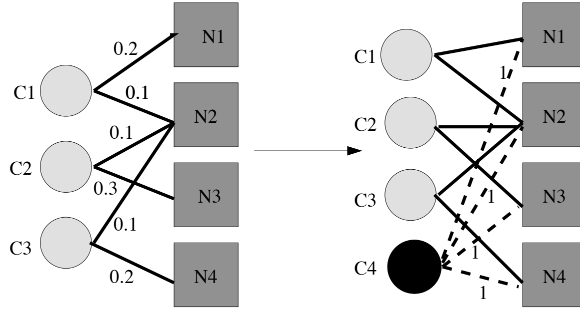

Figure 4: An example of reducing the minimum-weight maximum matching problem to the minimum-weight perfect matching problem.

is fully vacant, none of these applications can be placed. Here is how BF would work on this input. The n single-capsule nodes would be placed on the first node. Then, $(n-1)$ of the subsequently arriving applications would be placed on the $(n-1)$ fully vacant nodes, and the last application would be turned away. Therefore we have,

$$\exists\, input \quad s.t. \quad \frac{WF}{BF} \geq (2 - \frac{1}{n})$$

Since BF is optimal on this input, this gives us,

$$R_{WF} \geq (2 - \frac{1}{n})$$

This gives the claimed lower bound as n grows without bound. ■

5.2 Online Placement with Variable Preference for Nodes

In some scenarios, it may be useful to be able to honor any preference a capsule may have for one feasible node over another. In this section, we describe how online placement can take such preferences into account. We model such a scenario by enhancing the bipartite graph representing the placement of an application on the cluster by allowing the edges in the graph to have positive weights. An example of such a graph is shown in Figure 4. In this graph lower weights mean higher preference. A valid placement corresponds to a placement of size equal to the number of capsules k.

The online placement problem therefore is to find the maximum matching of minimum weight in this weighted graph. We show that this can be found by reducing the placement problem to the *Minimum-weight Perfect Matching Problem*. We will first define this problem and then present the reduction.

Definition 10 Minimum-weight Perfect Matching Problem*: A* perfect matching *in a graph G is a subset of edges such that each node in G is met by exactly one edge in the subset. Given a real weight c_e for each edge e of G, the* minimum weight perfect matching problem *is to find a perfect matching M of minimum weight* $\sum_{c \in M} c_e$.

Our reduction works as follows. Assume that all the weights in the original bipartite graph are in the range (0, 1) and that they sum to 1. This can be achieved by normalizing all the weights by the sum of the weights. If an edge e_i had weight w_i, its new weight would be $\frac{w_i}{\sum_{e \in E} w_e}$. Denote the number of capsules by m and the number of nodes by n, $m \leq n$. Construct $n - m$ capsules and add edges with weight 1 each between them and *all* the nodes. We call these the *dummy capsules*.

Figure 4 presents an example of this reduction. On the left is a bipartite graph showing the normalized preferences of the capsules $C1, C2, C3$ for their feasible nodes. We add another capsule $C4$ shown on the right to make the number of capsules equal to the number of nodes. Also shown on the right are the new edges connecting $C4$ to all the nodes. each of these edges has a weight of 1. The weights of the remaining edges do not change, so they have been omitted from the graph on the right.

Lemma 10 *In the weighted bipartite graph G corresponding to an application with m capsules and a cluster with n nodes $(m \leq n)$, a matching of size m and cost c exists if and only if a perfect matching of cost $(c + n - m)$ exists in the graph G' produced by reduction described above.*

Proof: $(\Longrightarrow)$ Suppose that there is a matching M of size m and cost c in G. We construct a perfect matching M' in G' as follows. M' has all the edges in M. Next we add to M' edges that have the dummy capsules incident on them. For this, we consider the dummy capsules one by one (in any order). For each such capsule, we add to M' an edge connecting it to a node that is not yet on any of the edges in M'. Since there is a matching of size m in G, and since each dummy capsule is connected to all n nodes, M' will have a matching of size n (that is a perfect matching). Further, since each edge with a dummy capsule as its end point has a weight of 1 and there are $(n - m)$ such edges, the cost of M' is $c + (n - m) \times 1 = c + n - m$.

$(\Longleftarrow)$ Suppose there is a perfect matching M' of cost $(c + n - m)$ in G'. Consider the set M that contains all the edges in M' that do not have a dummy capsule as one of their end points. There would be m such edges. Since M' was a perfect matching, M would be a matching in G. Moreover, the cost of M would be the cost of M' minus the sum of the costs of the $(n - m)$ edges that we removed from M' to get M. Therefore, the cost of M is $c + n - m - (n - m) \times 1 = c$.

This concludes the proof. ■

[9] gives a polynomial-time algorithm (called the *blossom* algorithm) for computing minimum-weight perfect matchings. [6] provides a survey of implementations of the blossom algorithm. The reduction described above, combined with Lemma 10, can be used to find the desired placement. If we do not find a perfect matching in the graph G', we conclude that there is no placement for the application. Otherwise, the perfect matching minus the edges incident on the newly introduced capsules gives us the desired placement.

6 Conclusions and Future Work

6.1 Summary of Results

In this work we considered the offline and the online versions of APP, the problem of placing distributed applications on a cluster of servers. This problem was found to be NP-hard. We used a gap preserving reduction from the Multi-dimensional Knapsack Problem to show that a restricted version of the offline placement problem has a PTAS. Designing a PTAS for the offline APP is part of our ongoing work. A heuristic that considered applications in nondecreasing order of their "largest component" was found to provide an approximation ratio of k, where k was the maximum number of capsules in any application. We also considered restricted versions of the offline APP in a homogeneous cluster. We found that heuristics based on "first-fit" or "striping" could provide an approximation ratio of 2 or better. Finally, an LP-relaxation based approximation algorithm was proposed.

For the online placement problem, we provided algorithms based on solving a maximum matching problem on a bipartite graph modeling the placement of a new application on a heterogeneous cluster. These algorithms guarantee to find a placement for a new application if one exists. We also allowed the capsules of an application to have variable preference for the nodes on the cluster and showed how a standard algorithm for the minimum weight perfect matching problem may be used to find the "most preferred" of all possible placements for such an application.

6.2 Directions for Future Work

There are several interesting directions along which we would like to work. Our reduction from the Multi-dimensional Knapsack Problem in Section 4.4 worked only when we assumed that k was fixed and made some assumptions about the behavior of the input to the offline APP. We would like to determine the hardness of approximating the APP in the absence of these assumptions. The second interesting direction is to analyze the approximation ratio of the LP-relaxation based approximation algorithm proposed in Section 4.5 and evaluate its performance through simulations. We have focused on the applications' requirement for a single resource. Realistic applications exercise multiple resources (such as CPU, memory, disk, network bandwidth) on a server, and hence may want guarantees on access to more than one resource. Our approach for online placement can be extended in a straightforward manner to this scenario. Recall that in the online version of the problem we were satisfied with finding a placement for a new application if one existed. We can ensure this even when applications have requirements for multiple resources. A node is now said to be feasible for a capsule if and only if it has enough resources of each type to be able to meet the capsule's requirement. A maximum matching on the resulting bipartite graph would yield a placement for a new application if one exists. For the offline placement, however, our goal was to maximize the number of

applications that we could place on the cluster. Solving the offline problem when multiple resources are involved would be interesting future work.

References

[1] K. Appleby, S. Fakhouri, L. Fong, M. K. G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA-based Management of a Computing Utility. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*, May 2001.

[2] A. K. Chandra, D. S. Hirschberg, and C. K. Wong. Approximate algorithms for some generalized knapsack problems. In *Theoretical Computer Science*, volume 3, pages 293–304, 1976.

[3] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, October 2001.

[4] C. Chekuri and S. Khanna. On Multi-dimensional Packing Problems. In *In Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 1999.

[5] C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. In *Proceedings of the eleventh annual ACM-SIAM Symposium on Discrete algorithms*, 2000.

[6] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. In *INFORMS Journal on Computing*, pages 138–148, 1999.

[7] T. Cormen, C. Leiserson, and R. Rivest. The MIT Press, Cambridge, MA.

[8] D. S. Hochbaum (Ed.). PWS Publishing Company, Boston, MA.

[9] J. Edmonds. Maximum matching and a polyhedron with 0,1 - vertices. In *Journal of Research of the National Bureau of Standards 69B*, 1965.

[10] A. M. Friexe and M. R. B. Clarke. Approximation algorithms for the m-dimensional 0-1 knapsack problem: worst-case and probabilistic analyses. In *European Journal of Operational Research 15(1)*, 1984.

[11] M. Garey and D. Johnson. W. H. Freeman and Company, New York.

[12] M. Moser, D. P. Jokanovic, and N. Shiratori. An Algorithm for the Multidimensional Multiple-Choice Knapsack Problem. In *IEICE Trans. Fundamentals Vol. E80-A No. 3*, March 1997.

[13] A compendium of NP optimization problems. `http://www.nada.kth.se/~viggo/problemlist/compendium.html`.

[14] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. In *Combinatorica*, volume 7, pages 365–374, 1987.

[15] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. QoS-Driven Server Migration for Internet Data Centers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)*, May 2002.

[16] D. B. Shmoys and E. Tardos. An Approximation Algorithm for the generalized assignment problem. In *Mathematical Programming A, 62:461-74*, 1993.

[17] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)*, December 2002.