International Journal of Foundations of Computer Science
© World Scientific Publishing Company

APPLICATION PLACEMENT ON A CLUSTER OF SERVERS[*]

BHUVAN URGAONKAR
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, 16802, USA

ARNOLD L. ROSENBERG
Department of Computer Science
University of Massachusetts
Amherst, MA, 01003, USA

and

PRASHANT SHENOY *Department of Computer Science*
University of Massachusetts
Amherst, MA, 01003, USA

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

The Application Placement Problem (APP, for short) arises in *hosting platforms:* clusters of servers that are used for hosting large, distributed applications such as Internet services. Hosting platforms imply a business relationship between an entity called *the platform provider* and a number of entities called *the application providers.* The latter pay the former for the resources on the hosting platform, in return for which, the former provides guarantees on resource availability for the applications. This implies that a hosting 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. The complexity of the APP is studied here, with the following results. The general APP is NP-hard; indeed, even restricted versions of the APP may not admit polynomial-time approximation schemes. However, several significant variants of the *online* version of the APP admit efficient approximation algorithms.

Keywords: Approximation algorithm; Application placement; Hosting platform

1. Introduction

[*]A portion of this work was presented at the *17th Intl. Conf. on Parallel and Distributed Computing Systems (PDCS'2004).*

Advances in computing technologies and falling hardware prices have combined to make server clusters built using commodity hardware and software an increasingly attractive alternative to large multiprocessor servers for many applications. We study a mapping problem that arises in one such application, whose commercial significance is growing rapidly. We consider the use of a server cluster as a *hosting platform* (*HP*, for short) wherein the servers host large, distributed applications such as Internet services. HPs can be shared or dedicated. In dedicated mode [1, 15], either all servers in the HP run a single application (such as a Web search engine), or each individual server is dedicated to a single application (such as dedicated Web hosting services where each node runs a single application). In contrast, in shared mode, [3, 17], the HP's servers run a large number of different third-party applications (Web servers, streaming media servers, multi-player game servers, e-commerce applications, etc.), with the number of applications typically exceeding the number of servers. More specifically, in shared mode, each application runs on a subset of the HP's servers, and these subsets may overlap. Whereas dedicated HPs 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 HPs imply a business relationship between the *platform provider* (the "owner" of the HP) and the *application providers*. The latter pay the former for the resources on the platform, in return for which, the former gives some kind of guarantees of resource availability for applications. The existence of guarantees implies that an HP should admit only applications for which it has sufficient resources. In the current paper, we take the number of applications that an HP able to host (or, admit) to be an indicator of the revenue that it generates from the hosted applications. The number of applications that an HP admits is related to the *application placement algorithm*—that decides where on the cluster the different components of an application get placed—that the HP employs. In this paper we study properties of the *Application Placement Problem* (*APP*, for short): the problem of maximizing the number of applications that can be hosted on the HP. We show that APP is NP-hard and, indeed, that even restricted versions of the APP may not admit polynomial-time approximation schemes. We design and analyze several approximation algorithms for the APP and present algorithms for its online version.

Roadmap. In Section 2, we develop a formal setting for the APP and discuss related work. Section 3 establishes the hardness of even approximately solving the APP. Section 4 develops polynomial-time approximation algorithms for various restriction versions of the APP. Section 5 begins to study the *online* version of the APP, wherein new applications arrive dynamically. Section 6 discusses directions for further work on this topic of increasing importance.

2. The Application Placement Problem

2.1. Notation and Definitions

We consider a hosting platform *HP* consisting of *n servers* (or, *nodes*), $N_1, N_2, \ldots, N_n$, each having a given *(initial) capacity* C_i (of "available resources"). Unless otherwise noted, nodes are *homogeneous*, in the sense of having the same initial capacities. The APP allocates portions of each node's capacity to (one or more) applications.

Let $A_1, \ldots, A_m$ denote the *applications* to be placed on *HP*. For our purposes, an application can be viewed as a set of *requirements*, i.e., demands for node capacity, expressed in discrete uniform units called *capsules*. For instance, a typical online bookstore application may consist of three capsules—a Web server responsible for HTTP processing, a middle-tier Java application server that implements the application logic, and a back-end database that stores catalogs and user orders. A capsule is the smallest component of an application for the purposes of placement; i.e., all 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-applications and for controlling the allocation of these sub-applications onto nodes. For instance, if an application wants to ensure that certain components are allocated to the same node (e.g., because they communicate a lot), then it could bundle them as one capsule. Some applications—e.g., replicated Web servers—may want all capsules to be allocated to distinct nodes, say to improve resilience in the face of node failures: if a node hosting a capsule fails, there would still be capsules on other nodes. We refer to this as the *capsule placement restriction*. We study the computational complexity of APP both with and without the capsule placement restriction.

Although each capsule in an application would generally require guarantees on access to multiple resources, we consider here just a single resource, such as the CPU or the network bandwidth. We adopt a simple model wherein a capsule specifies its resource requirement as a fraction of the resource capacity of a node in cluster *HP*; we thus assume that the resource requirement of each capsule is less than the capacity of a node. A capsule C can be placed on a node N only if the sum of C's resource requirements and those of the capsules already placed on N does not exceed N's resource capacity. We say that an application *can be placed* only if *all* of its capsules can be placed simultaneously. Easily, there may be more than one way in which an application can be placed on a cluster *HP*. We call the total number of applications that a placement algorithm could place on *HP* as the *size* of the placement. A node N_i, none of whose resources have been reserved (so that it still has capacity C_i), is said to be *empty*.

We study two versions of APP, both requiring one to determine a maximum-size placement of a set of m applications, $A_1, \ldots, A_m$ on a cluster *HP* of n empty nodes.[a]

offline-APP: Determine a placement in any way.

online-APP: Determine a placement, one A_i at a time, while satisfying the following:

[a]Node N_i is *empty* if none of its resources have been reserved (so that it still has capacity C_i).

1. The A_i should be considered for placement in increasing order of their indices i.
2. Once an A_i has been placed, it cannot be moved as subsequent A_j are being placed.

An easy reduction from BIN-PACKING [13] (see Section 7) shows that APP is NP-hard.

Lemma 1 *The Application Placement Problem APP is NP-hard.*

2.2. Related Work

We presume familiarity with standard notions of computational complexity, referring the reader to a source such as [13].

Two generalizations of the classical knapsack problem are relevant to our study of APP: the *Multiple Knapsack Problem* (MKP) and the *Generalized Assignment Problem* (GAP) [13]. In MKP, we are given a set of n items and m bins (knapsacks), with each item i having a *profit* $p(i)$ and a *size* $s(i)$, and each bin j having a *capacity* $c(j)$. The goal is to find a subset of the 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. GAP is APX-hard (meaning that a PTAS [Polynomial-time approximation scheme] is unlikely), but it admits a 2-approximation algorithm [16]. This was the best result known for MKP until a 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). It is further shown in [5] that slight generalizations of MKP are APX-hard, leading one to suspect that APP may also be APX-hard (and, hence, may not have a PTAS). Another closely related problem is a "multidimensional" version of MKP wherein each item has requirements along multiple dimensions, all of which must be satisfied to successfully place it. Again, one seeks to maximize the total profit yielded by the items that can be placed. A heuristic for this problem is described in [12]; however, only simulated evaluations appear, with no analytical performance guarantees.

To the best of our knowledge, our work is the first to formulate and study the version of APP that arises in hosting platforms.

3. The Hardness of Approximating APP

This section is devoted to exploring the complexity of APP. We show that even a restricted version of APP may not admit a PTAS. The capsule placement restriction is assumed to hold throughout this section.

The main technical tool here is the *gap-preserving reduction.* We paraphrase from [8].

Gap-preserving reduction: Let Π and Π' be two maximization problems.[b] A *gap-preserving reduction from* Π *to* Π' *with parameters* (c, ρ) *and* (c', ρ') is a poly-

[b] We choose *maximization* problems only for definiteness.

time algorithm f with the following property. For each instance I of Π, algorithm f produces an instance $I' = f(I)$ of Π'. The maxima of I and I', say $MAX(I)$ and $MAX(I')$, respectively, satisfy the following property:

$$MAX(I) \geq c \quad \Rightarrow \quad MAX(I') \geq c' \tag{1}$$
$$MAX(I) < c/\rho \quad \Rightarrow \quad MAX(I') < c'/\rho'. \tag{2}$$

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

Gap-preserving reductions can be used to demonstrate the hardness of approximating optimization problems, as follows. Say that we wish to establish the (likely) poly-time inapproximability of problem Π'. Say that we have a polynomial-time reduction τ from SAT to Π that ensures, for every boolean formula ϕ:

$$\phi \in SAT \quad \Rightarrow \quad MAX(\tau(\phi)) \geq c$$
$$\phi \notin SAT \quad \Rightarrow \quad MAX(\tau(\phi)) < c/\rho.$$

Then composing this reduction with a gap-preserving reduction f from Π to Π' gives a reduction $f \circ \tau$ from SAT to Π' that ensures:

$$\phi \in SAT \quad \Rightarrow \quad MAC(f(\tau(\phi))) \geq c'$$
$$\phi \notin SAT \quad \Rightarrow \quad MAX(f(\tau(\phi))) < c'/\rho'.$$

In other words, $f \circ \tau$ shows that achieving an approximation ratio ρ' for Π' is NP-hard.

We now give a gap-preserving reduction from the *Multidimensional* 0-1 *Knapsack Problem* [2] (also known as the *Packing Integer Problem* [4]) to a restricted version of APP.

The Multidimensional 0-1 Knapsack Problem (MDK):

Given: the fixed positive integer k, the *dimension* of the problem

Maximize: $\sum_{i=1}^{n} c_i x_i$ *Subject to:* $\sum_{i=1}^{n} a_{ij} x_i \leq b_j$, for $j = 1, \ldots, k$

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

Define $B = \min_i b_i$.

To see MDK as a *knapsack* problem, think of $(b_1, \ldots, b_k)$ as a *capacity vector:* along each dimension d, the knapsack has capacity b_d. Think of n items $I_1, \ldots, I_n$, with each I_j having the *requirement vector* $(a_{j1}, \ldots, a_{jk})$. MKDP thus maximizes the number of k-dimensional items that can be packed in the k-dimensional knapsack such that along each dimension, the sum of the requirements of the packed items does not exceed the capacity of the knapsack.

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

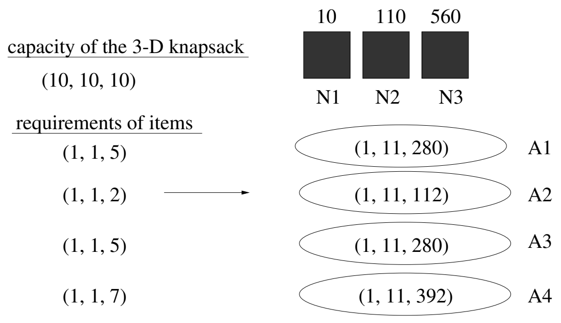

Figure 1: The gap-preserving reduction from Multidimensional Knapsack to offline-APP.

Theorem 1 *Given any fixed $\epsilon > 0$, it is NP-hard to approximate to within the factor $(1+\epsilon)$ offline-APP with the restrictions:*

1. each capsule has a positive requirement;

2. there is a fixed constant M such that: $(\forall i \in \{1,\ldots,n\})(\forall j \in \{1,\ldots,k\})$ $[M \geq b_j/a_{ji}]$.

Proof. Deferring an explanation of the two restrictions, we describe a gap-preserving reduction from k-MDK to offline-APP.

The reduction. Consider an instance of k-MDK with capacity vector $(b_1,\ldots,b_k)$ and with n items, $I_1,\ldots,I_n$, each I_j having requirement vector $(a_{j1},\ldots,a_{jk})$. We create an instance of offline-APP as follows. The cluster has k nodes $N_1,\ldots,N_k$. There are n applications $A_1,\ldots,A_n$, each having k capsules. A_i's capsules are denoted $c_i^1,\ldots,c_i^k$; we call c_i^j A_i's jth capsule. We assign capacities to the N_i and requirements to the A_j in k stages. In stage s, we determine the capacity of N_s and the requirements of the sth capsules of all applications.

Stage 1. We assign N_1 capacity $C(N_1) = b_1$; for each i, we assign the first capsule of A_i requirement $r_i^1 = a_{i1}$.

Stage s $(1 < s \leq k)$. The assignments at stage s depend on those at stage $s-1$. We let $r^s_{min} = \min_{i=1}^n(a_{is})$ denote the smallest requirement along dimension s of the items in the input to k-MDK. We then specify the *scaling factor* for stage s to be:

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

(Recall that $(\forall s) r^s_{min} > 0$.) Now we are ready to do the assignments for stage s. Node N_s is assigned capacity $C(N_s) = b_i \times SF_s$, and capsule c_i^s is assigned requirement $r_i^s = a_{is} \times SF_s$.

This completes our mapping. A simple example will illustrate how the mapping works. Consider the instance of input T to MDK shown in the left of Fig. 1, wherein $k = 3$, $n = 4$. We create three nodes, N_1, N_2, N_3, and four applications, A_1, A_2, A_3, A_4, each with 3 capsules. Consider how the three stages in our mapping proceed.

Stage 1. We assign N_1 capacity 10 and unit requirements to the first capsules of all

applications.
Stage 2. The scaling factor SF_2 is 11, so we assign N_2 capacity 110 and requirements 11 to the second capsules of all applications.
Stage 3. The scaling factor SF_3 is $\lfloor 110/s \rfloor + 1 = 56$, so we assign N_3 capacity 560. The third capsules of the applications are assigned respective requirements $280, 112, 280, 392$.

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

($\Rightarrow$) Consider an instance of k-MDK with n items, $I_1, \ldots, I_n$. Say that there is a packing P of size $m \leq n$, involving (with no loss of generality) items $I_1, \ldots, I_m$. We thus have:

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

Consider the following placement of applications $A_1, \ldots, A_n$ on nodes $N_1, \ldots, N_k$. If item I_i appears in packing P, then place A_i as follows: for $j \in \{1, \ldots, k\}$, place capsule c_i^j on N_j. We claim that we can place all m applications corresponding to the m items in P. To wit, each node N_i, for $i \in \{1, \ldots, k\}$, has capacity ($SF_i \times$ (the capacity along dimension i of the k-dimensional knapsack)), with $SF_i \geq 1$. The requirements assigned to the ith capsules of all applications are ($SF_i \times$ (the sizes along the ith dimension of the items)). Multiplying both sides of (4) by SF_i, we get

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

The term on the right in this inequality is N_i's capacity, while. the term on the left is the sum of the requirements of the ith capsules of the applications corresponding to the m items in P. Thus, N_i can accommodate all of these applications, so there is a placement of size m.

($\Leftarrow$) Consider an instance of APP with n applications, $A_1, \ldots, A_n$. Say that there is a placement L of size $m \leq n$, involving (with no loss of generality) $A_1, \ldots, A_m$. For $s \in \{1, \ldots, k\}$, let Cap_s denote the set of sth capsules of the placed applications.

We make two key observations.
1. *An application can be successfully placed, only if its ith capsule is placed on node N_i.* The inductive verification relies on the fact that the scaling factor (3) renders the requirements of all sth capsules, for $s > 1$, larger than the capacities of nodes $N_1, \ldots, N_{s-1}$. To see this, consider the kth capsules first. The only node with sufficient capacity for these 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 may be placed only on N_{k-1}. Proceeding inductively verifies the claim.
2. *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 s^{th} dimension.* This is immediate from the fact that, for all $s \in \{1, \ldots, k\}$, the capacity of N_s and the requirements of all sth capsules are scaled by the same factor.

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

Time- and space-complexity. This reduction works in time polynomial in the size of the input. To wit, it proceeds in k stages, each involving computing a scaling factor (which requires a division) and multiplying $n+1$ numbers (the capacity of the knapsack and the requirements of the n items along the relevant dimension).

Consider the size of the input to offline-APP produced by our reduction. Due to the mandated scaling of capacities and requirements, the magnitudes of the inputs for node N_j and all jth capsules ($j \in \{1, \ldots, k\}$) increase by a factor of $O(M^j)$. This implies that the input size (using binary representations) increases by a factor of $O(M^{j/2})$. Overall, the input size thus increases by a factor of $O(M^k)$. For the mapping to be a reduction, we need this to be a constant. This explains our two restrictions on offline-APP: (1) k and M must be constants; (2) all capsule requirements must be positive.

Gap-preservation. Our reduction is gap-preserving because the size of the optimal solution to offline-APP is *exactly equal* to the size of the optimal solution to MDK. Formally, using the terminology in our definition of gap-preserving reduction, we can set $c = c' = \rho = \rho' = 1$. Putting these values in (1,2), we find that the following conditions hold:

$\text{MAX(MDK)} \geq 1 \quad \Rightarrow \quad \text{MAX(offline-APP)} \geq 1$

$\text{MAX(MDK)} < 1 \quad \Rightarrow \quad \text{MAX(offline-APP)} < 1$

This proves that the reduction is gap-preserving, completing the proof that offline-APP, restricted as described in the theorem does not admit a PTAS unless $P = NP$. □

4. Algorithms for Offline-APP

This section is devoted to approximation algorithms for several variants of offline-APP. Except in Section 4.2.2, we focus on clusters that are homogeneous, in the sense specified earlier.

4.1. Placement without the Capsule Placement Restriction

We present 2-approximate algorithms for two variants of offline-APP without the capsule placement restriction: (1) when any capsule may be placed on any node; (2) when each application's capsules must be placed on the same node.

4.1.1. A *first-fit* based approximation algorithm

We consider the most general form of APP, in which a capsule may be placed on any node that has enough capacity. We show that a placement algorithm based on first-fit gives an approximation ratio that approaches 2 as the size of the cluster grows.

The approximation algorithm FF_MULTIPLE_RES. Let us be given n unit-capacity nodes, $N_1, \ldots, N_n$, and m applications, $A_1, A_2, \ldots, A_m$, having respective requirements[c] $R_1 \leq R_2 \leq \cdots \leq R_m$.[d] The algorithm considers the applications in increasing order of their indices (which means nondecreasing order of their requirements). It places an application using the "first-fit" criterion, i.e., on the "first" set of nodes where it can be accommodated, i.e., the smallest-index nodes that have sufficient resources for all its capsules. The algorithm terminates once it has either placed all applications or found an application that cannot be placed.

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

Proof. Consider an instance of APP in which the optimal algorithm can place all m applications on n nodes. If FF_MULTIPLE_RES matches this placement, then we are done.

Otherwise, FF_MULTIPLE_RES can completely place only $k_{FF} < m$ applications (i.e., with all capsules placed). Since all capsules have requirements less than the capacity of a node, this implies that there is no empty node after the placement. The placement regimen used by FF_MULTIPLE_RES implies that the completely placed applications are $A_1, \ldots, A_{k_{FF}}$, and that some (but not all) capsules of $A_{k_{FF}+1}$ *may* have been placed. Importantly, in this case: *at most one node can be more than half empty.* To wit, if two nodes, N_i and N_j (with $i < j$), were both more than half empty, then, by our assumptions about node-capacities, FF_MULTIPLE_RES would have placed all capsules from N_j into N_i. We thus have

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

where $R'_{k_{FF}+1} \geq 0$ is the sum of the requirements of the capsules of application $A_{k_{FF}+1}$ that *could be* placed on the cluster. Since an optimal algorithm can do no better than use up *all* node-capacity, we have $R_1 + \cdots + R_m \;\leq\; n$. Our assumption about the ordering of the R_i implies that

$$|\{A_1, \ldots, A_{k_{FF}}\}| \geq |\{A_{k_{FF}+1}, \ldots, A_m\}|,$$

so that FF_MULTIPLE_RES places at least one less than half as many applications as an optimal algorithm. Hence, the performance ratio of FF_MULTIPLE_RES tends to 2 as the number of nodes grows. □

4.1.2. On placing applications whose capsules must be colocated

We now provide a polynomial-time algorithm for a restriction of offline-APP, whose placements are within a factor 2 of optimal. The restriction requires that all of an application's capsules be placed on the same node. This is equivalent to packaging each application's capsules into a single one which assumes all of the application's requirements.

Motivation. Highly parallel scientific applications that require a significant amount of interprocess communication may insist on single-node placements. The overheads for communication may render dispersed placements prohibitively expensive.

[c] The requirement of an application is the sum of the requirements of its capsules.

[d] We thus assume that applications are indexed in nondecreasing order of their requirements.

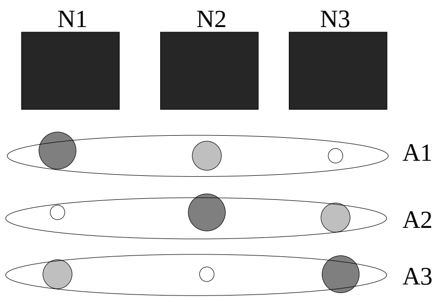

Figure 2: An example of striping-based placement.

The approximation algorithm FF_SINGLE. Let us be given n unit-capacity nodes $N_1, \ldots, N_n$ and m single-capsule applications $C_1, \ldots, C_m$ with respective requirements $R_1 \leq \cdots \leq R_m$. The algorithm considers the applications in the indicated order, placing each application using the "first-fit" criterion. The algorithm terminates once it has either placed all applications or found an application that cannot be placed.

The following result, which is left to the reader, yields to a proof similar to that of Lemma 2.

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

4.2. Placement with the Capsule Placement Restriction

We now consider APP with the capsule placement restriction. We first study a special case—"identical" applications—and then consider the general case.

Motivation. A replicated Web server might impose the capsule placement restriction, since the dispersal of capsules over multiple nodes would enhance tolerance to node failures.

4.2.1. On placing "identical" applications

Two applications are "identical" if their sets of capsules are identical. We now present and anayze a "striping"-based placement algorithm.

Striping-based placement. Let there be n nodes,[e] $N_0, \ldots, N_{n-1}$, and m applications, $A_0, \ldots, A_{m-1}$, each having k "identical" capsules, with respective requirements $r_0 \leq r_1 \leq \cdots \leq r_{k-1}$. The algorithm partitions the nodes into $(t+1)$ sets, $S_1, \ldots, S_t$,, where $t = \lfloor n/k \rfloor \geq 1$. All sets contain $k \leq n$ nodes with sequential indices. The algorithm considers the S_i in turn and "stripes" as many unplaced applications on them as it can; i.e., each application is placed on nodes of the form $N_i, N_{i+1}, \ldots, N_{i+k-1 \bmod n}$. The set of nodes receiving capsules at any moment is the *current* set of nodes. Fig. 2 illustrates three nodes and a number of identical 3-capsule applications. 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

[e]It is useful to use 0-based indexing here.

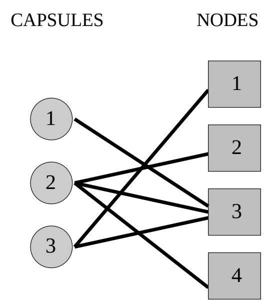

Figure 3: A bipartite graph indicating which capsules can be placed on which nodes

continues. The algorithm terminates when either the nodes in S_t are filled or all applications have been placed. Note that there may be unused nodes if k does not divide n.

A standard "striping'-oriented" analysis yields the following.

Lemma 4 *The striping-based placement algorithm yields an approximation ratio of* $\left(\dfrac{t+1}{t}\right)$ *for identical applications, where* $t = \lfloor n/k \rfloor$.

4.2.2. On placing arbitrary applications

We have thus far considered restrictions of offline-APP and that yield to heurisitics with approximation ratios of 2 or better. We now find out that the general offline-APP is much harder to compute approximately optimally (within the current state of knowledge). For the first time, we consider clusters with *heterogeneous* nodes.

Our first heuristic for offline-APP yields an approximation ratio k, the maximum number of capsules in any application.

The Max-First **heuristic**. This heuristic gives each application a *weight* which is the requirement of its *largest* capsule. The heuristic considers applications in nondecreasing order of weight and uses a bipartite graph G to model the placement problem. G has one vertex for each capsule in the application and one for each node in the cluster. There is an edge connecting a capsule and a node precisely when the node is *feasible* for the capsule, i.e., has sufficient capacity to host it; cf. Fig. 3. We show now that an application can be placed on the cluster if, and only if, G admits a matching whose size equals the number of capsules in the application. This suggests using the Maximum Matching Problem on G [7] to derive a placement. If the maximum matching has size equal to the number of capsules in the application, then we place each capsule on the nodes that the matching connects it to. Otherwise, we terminate and say that the application cannot be placed.

Lemma 5 *An application with k capsules can be placed on a cluster if, and only if, there is a matching of size k in the bipartite graph G modeling its placement on the cluster.*

Proof. ($\Rightarrow$) A matching of size k in G specifies a one-to-one correspondence between capsules and nodes. Since edges connote feasibility, this correspondence is a valid placement.

($\Leftarrow$) If G admits no matching of size k, then any placement of capsules on nodes must end up with distinct capsules sharing the same node. This means that the application cannot be placed without violating the capsule placement restriction. $\square$

Lemma 6 *The Max-First placement heuristic has an approximation ratio k, where k is the maximum number of capsules in an application.*

Proof. Let us be given n nodes, $N_1, \ldots, N_n$, and a set A of m applications. Denote by H the set of applications that Max-First places, and by O the set of applications placed by any optimal placement algorithm. Clearly, $|H| \leq |O| \leq m$. Let $I = H \cap O$ comprise the applications that both H and O place, and let R be the set of applications that neither places.

We estimate the sizes of the sets $(H \setminus I)$ and $(O \setminus I)$, which comprise the applications that only one algorithm places. (If both $(H \setminus I)$ and $(O \setminus I)$ are empty, then we have the claimed ratio trivially.) Construct $(H \setminus I)$ by removing from H all applications in I. Deduct from all nodes the resources reserved for the removed applications. Denote the resulting nodes by $N_1^{H\setminus I}, \ldots, N_n^{H\setminus I}$. Analogously, construct $(O \setminus I)$ and deduct all resources reserved for the removed applications, thereby producing the nodes $N_1^{O\setminus I}, \ldots, N_n^{O\setminus I}$. Say that Max-First places y applications from the set $(H \setminus I)$ on nodes $N_1^{H\setminus I}, \ldots, N_n^{H\setminus I}$. Let the applications in $(A \setminus I)$ be denoted $B_1, \ldots, B_y, \ldots, B_{|A\setminus I|}$ when arranged in nondecreasing order of the size of their largest capsule: letting $l(X)$ be the requirement of the largest capsule in application X, we have $l(B_1) \leq \cdots \leq l(B_y) \leq \cdots \leq l(B_{|A\setminus I|})$. By definition, the y applications that Max-First places are $B_1, \ldots, B_y$. Also, the applications that the optimal algorithm places on nodes $N_1^{O\setminus I}, \ldots, N_n^{O\setminus I}$ must be from the set $\{B_{y+1}, \ldots, B_{|A\setminus I|}\}$. We observe that *for each application in the set* $\{B_{y+1}, \ldots, B_{|A\setminus I|}$, *the requirement of the largest capsule is* $\geq l(B_y)$. We then infer that Max-First will exhibit its worst approximation ratio when all applications in $(H \setminus I)$ have k capsules, each with requirement $l(B_y)$, and all applications in $(O \setminus I)$ have $(k-1)$ capsules with requirement 0 and one capsule with requirement $l(B_y)$. Since the total capacities remaining on node $N_1^{H\setminus I}, \ldots, N_n^{H\setminus I}$ and on nodes $N_1^{O\setminus I}, \ldots, N_n^{O\setminus I}$ are equal, this implies that, in the worst case, $O \setminus I$ would contain k times as many applications as $H \setminus I$. We can finally establishprove an approximation ratio of k for Max-First:

$$|O| \ = \ |O \setminus I| + |I| \ \leq \ k \cdot |H \setminus I| + |I| \leq \ k \cdot (|H \setminus I| + |I|) \ = \ k \cdot |H| \qquad \square$$

5. Algorithms for Online-APP

In this section, we develop algorithms for online-APP that always find a placement for an application if one exists. Such algorithms place applications that arrive one by one. They place a newly arrived application only if they can do so without moving any already placed capsule. We assume a heterogeneous cluster throughout this section.

5.1. Online Placement Algorithms

We use a bipartite *feasibility graph* G_A to model the situation an online placement algorithm faces when a new application, A, arrives. G_A has one vertex for each of A's capsules and one for each node in the cluster. An edge connects a capsule C with a node N just when N *is feasible for* C, i.e., has sufficient resources to host C; cf. Fig. 3. As described in Section 4.2.2, a maximum matching on G_A can be used to find a placement for A if one exists.

Let $\mathcal{A}$ denote any online placement algorithms that operate *greedily*, in the following sense. $\mathcal{A}$ processes the capsules of a new application A in nondecreasing order of their degrees in the feasibility graph G_A. $\mathcal{A}$ places a capsule C on one of its feasible nodes, say N, terminating if no such node exists. After successfully placing C, $\mathcal{A}$ removes from G_A all edges connecting N to any as-yet unplaced capsules. $\mathcal{A}$ repeats this process until it either completes placing A or it terminates for inability to place a capsule. We now consider two specific greedy online placement algorithms.

The best-fit placement algorithm BF. This greedy algorithm places a capsule on a random feasible node whose remaining capacity is *minimum.*

The worst-fit placement algorithm WF. This greedy algorithm places a capsule on a random feasible node whose remaining capacity is *maximum.*

We now study the approximation ratios of BF and WF, R_{BF} and R_{WF}, respectively.

Lemma 7 *BF can perform arbitrarily worse than WF, so R_{BF} admits no upper bound.*

Proof. Let there be m applications and $n < m$ unit- capacity nodes. Say that the first n applications to arrive have one capsule each, with requirement $1/n$. BF will place these applications on the first node. Say that the next $m-n$ applications to arrive have n capsules each, each having non-zero requirement. Since the first node has no capacity left, BF will not be able to place any of these. In contrast, WF would have placed each of the first n single-capsule applications on a separate node, so that every node would have residual capacity $(1-1/n)$ available for the n-capsule applications. On this input sequence, then,

$$P(BF) \;\leq\; \frac{n}{m} P(WF), \tag{5}$$

where $P(\mathcal{A})$ is the number of applications that algorithm $\mathcal{A}$ places.

Since WF can never outperform an optimal algorithm, and since we can make the requirements for the last $m-n$ applications arbitrarily small, inequality (5) implies that

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

hence cannot be bounded from above. □

Lemma 8 *$R_{WF} \;\geq\; (2-1/n)$ for an n-node cluster.*

Proof. Consider a cluster with n unit-capacity nodes. Say that the first n applications to arrive have one capsule each, with a tiny requirement ϵ. WF places

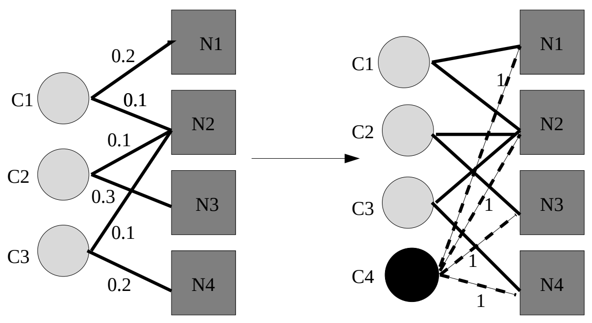

Figure 4: Reducing Minimum-Weight Maximum Matching to Minimum-Weight Perfect Matching.

each of these applications on a separate node, leaving each node with residual capacity $(1-\epsilon)$. Next, n applications arrive, each having one capsule with requirement 1. Since no node is empty, none of these applications can be placed. In contrast, BF would place the first n applications on the first node. It could then place $(n-1)$ of the subsequent applications on the $(n-1)$ empty nodes, failing to place only the last application. On this input sequence, therefore,

$$P(BF) \ \geq \ \left(2 - \frac{1}{n}\right) P(WF).$$

Since BF can never outperform an optimal algorithm, we have

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

as n grows without bound. □

5.2. Online Placement with Variable Node-Preferences

In some scenarios, especially involving heterogeneous clusters, certain capsules may "prefer" one feasible node over another. We now study online-APP wherein such preferences are honored. We model such scenarios by enhancing the feasibility graph's edges with positive weights; cf. Fig. 4 wherein *lower weights mean higher preference*. The online placement problem in such scenarios is to find a maximum matching of minimum weight in this weighted graph. This placement problem reduces to the *Minimum-Weight Perfect Matching Problem.*

Minimum-Weight Perfect Matching (MWPM). A *perfect matching* in a graph G is a subset of edges that "touch" each vertex exactly once. Given a real weight c_e for each edge e, the **MWPM** requires one to find a perfect matching M whose weight, $\sum_{e \in M} c_e$, is minimum.

The reduction works as follows. By normalization, we may assume that each weight ω in the feasibility graph G lies in the range $0 \leq \omega \leq 1$ and that the weights sum to 1. Let there be m capsules and $n \geq m$ nodes. Create an augmented graph

$\widetilde{G}$ by adding $n - m$ new, *dummy* capsules and unit-weight edges connecting each with *all* nodes. Fig. 4 exemplifies this reduction. The lefthand graph G shows the normalized preferences of the capsules $C1, C2, C3$ for their feasible nodes. The righthand graph $\widetilde{G}$ adds a new capsule, $C4$, to equalize the numbers of capsules and nodes, and adds unit-weight edges between $C4$ and all nodes. (Since the weights of the original edges do not change, they are not shown in $\widetilde{G}$.)

Lemma 9 *A matching of size m and cost c exists in the feasibility graph G of an application with m capsules and a cluster with $n \geq m$ nodes if, and only if, a perfect matching of cost $(c + n - m)$ exists in the augmented graph $\widetilde{G}$.*

Proof. ($\Rightarrow$) Given a matching M of size m and cost c in G, we construct a perfect matching $\widetilde{M}$ in $\widetilde{G}$ as follows. $\widetilde{M}$ contains all edges in M, in addition to edges that "touch" each dummy capsule. To choose these latter edges, we consider dummy capsules one by one (in any order), and, for each, we add to $\widetilde{M}$ an edge connecting it to any as-yet "untouched" node. Since G admits a matching of size m, and since each dummy capsule connects to all n nodes, $\widetilde{G}$ is certain to have a size-n (hence, perfect) matching. Further, since each edge that "touches" a dummy capsule has unit weight, and there are $(n - m)$ such edges, the cost of $\widetilde{M}$ is $c + (n - m)$.

($\Leftarrow$) Let $\widetilde{G}$ admit a perfect matching $\widetilde{M}$ of cost $c + n - m$. Consider the set M comprising all m edges in $\widetilde{M}$ that do not "touch" a dummy capsule. Since $\widetilde{M}$ is a matching in $\widetilde{G}$, M is a matching in G. Moreover, the cost of M is just $n - m$ less than the cost of $\widetilde{M}$, namely, c. □

Thus, the reduction preceding Lemma 9 yields the desired placement algorithm. We construct the feasibility graph G_A for each arriving application A and augment it to $\widetilde{G}_A$. If $\widetilde{G}_A$ contains a perfect matching, then we remove the edges that "touch" dummy capsules and obtain the desired placement. If $\widetilde{G}_A$ does not contain a perfect matching, then we know that A cannot be placed. One finds in [9, 6] poly-time algorithms for MWPM.

6. Conclusions

6.1. Summary of Results

We have considered the offline and 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 Multidimensional Knapsack Problem to show that even a restricted version of offline-APP may not have a PTAS. A heuristic that considered applications in nondecreasing order of their "largest component" was found to provide an approximation ratio of k, the maximum number of capsules in any application. We also considered restricted versions of offline-APP in a *homogeneous* cluster, finding that heuristics based on "first-fit" or "striping" could provide approximation ratios of 2 or better.

For the online version of APP, we provided algorithms based on solving matching problems on bipartite graphs that model the placement of a new application on a *heterogeneous* cluster. These algorithms guarantee to find a placement for each arriving application, if one exists. When one allows the capsules of an application to have variable preference for the nodes of a cluster, we found that a minimum weight perfect matching algorithm will allow one to find the “most preferred” of all possible placements for such an application.

6.2. Ongoing and Future Work

Excluding results for some special cases, we currently have algorithms with approximation ratios of k and 2 for APP with and without the capsule placement restriction (k is the number of capsules in an application). We now describe ongoing work on devising an improved approximation algorithm for APP and some directions for future work.

6.2.1. LP-Relaxation based placement

APP can be formulated as an integer linear program (ILP). We can, therefore, construct an approximation algorithm for APP that relaxes the ILP to a linear program. We are currently working on determining if this algorithm can provide a better approximation that our current algorithms.

We now describe our LP-relaxation based placement algorithm. Say that we have n nodes and m applications. By possibly adding dummy capsules with requirement 0, each application can be thought of as having 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:

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

Additionally, define:

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

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

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

Subject to

$$\begin{array}{lrcl} (\forall i,k) & \sum_{j=1}^{n} x_{ijk} & \leq & 1 \\ (\forall k) & \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ijk} \times r_{ik} & \leq & C_k \\ (\forall i) & x_{i1} = \cdots & = & x_{in}. \end{array}$$

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 each x_{ijk} to take real values in the range $0 \leq x_{ijk} \leq 1$. 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 preceding rounding. We remove capsules placed such nodes, in nonincreasing order of requirements, until the remaining capsules fit on the node. Observe that removing a capsule of an application implies also removing all of its other capsules.

6.2.2. Future directions

We have focused on applications that have requirements for a single resource. Realistic applications exercise multiple resources (such as CPU, memory, disk, network bandwidth) on a server, hence, may want guarantees for more than one resource. Our approach for online placement can be extended to this scenario in a straightforward manner. Recall that in online-APP 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 offline-APP, 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.

Acknowledgements

The research of A. Rosenberg was supported in part by NSF grant CCF-0342417. The research of P. Shenoy and B. Urgaonkar was supported in part by NFS grants CNS-0323597, CCR-9984030, and EIA-0080119.

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 and R. Doyle. Balance of Power: Energy Management for Server Clusters. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany*, May 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. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1991.
8. D. S. Hochbaum (Ed.). *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, Boston, MA, July 1996.
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. *Computers and Intractibility: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, New York, January 1979.
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, Miami, FL*, 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA*, December 2002.
18. M. J. Beeson, *Foundations of Constructive Mathematics* (Springer, Berlin, 1985).
19. K. L. Clark, "Negations as failure," in *Logic and Data Bases*, eds. H. Gallaire and J. Winker (Plenum Press, New York, 1973) pp. 293–306.
20. D. Dolve, "Unanimity in an unknown and unreliable environment," *Proc. 22nd Annual Symposium on Foundations of Computer Science*, Nashville, TN, Oct. 1981, pp. 159–168.
21. W. L. Gewirtz, "Investigations in the theory of descriptive complexity," Ph. D. Thesis, New York University, 1974.
22. M. Joliat, "A simple technique for partial elimination of unit productions from LR(k) parsers," *IEEE Trans. Comput.* **C-27** (1976) 753–764.
23. R. Lorentz and D. B. Benson, "Deterministic and nondeterministic flow-chart interpretations," *J. Comput. System Sci.* **27** (1983) 400–433.
24. R. Tamassia, C. Batini and M. Talamo, "An algorithm for automatic layout of entity relationship diagrams," in *Entity-Relationship Approach to Software Engi-*

neering, Proc. 3rd Int. Conf. on Entity-Relationship Approach, eds. C. G. Davis, S. Jajodia, P. A. Ng and R. T. Yeh (North-Holland, Amsterdam, 1983) pp. 421–439.

Appendix A:

7. NP-Hardness of APP

We begin with an auxiliary problem.

The Single-capsule APP (DEC_MAX_CAP).
Given: n empty nodes $N_1, \ldots, N_n$; m 1-capsule applications $C_1, \ldots, C_m$; target integer k
Decide: Does there exist a placement of size k?

Lemma A.1 *DEC_MAX_CAP is NP-complete.*

Proof sketch: Clearly, DEC_MAX_CAP is in NP. One merely must check the validity of a given putative placement of k capsules, which amounts to checking that the sum of the requirements of all the capsules placed on each N_i does not exceed C_i.

A simple reduction from BIN-PACKING [11] shows that DEC_MAX_CAP is NP-Hard.

BIN-PACKING.
Given: m objects $O_1, \ldots, O_m$ of sizes $s_1, \ldots, s_m$, respectively; target integer k
Decide: Can all the objects can be placed into k unit-capacity bins?.

The reader can verify easily that the following construction of an input to DEC_MAX_CAP from an input to BIN-PACKING is a poly-time reduction. First, for each object O_i, we provide a capsule C_i whose requirement equals O_i's size. Next, we provide k nodes, each with unit capacity. These node- and capsule-sets, along with the target integer m, comprise the input to DEC_MAX_CAP.

General APP (DEC_MAX_APP).
Given:: n empty nodes $N_1, \ldots, N_n$; m applications $A_1, \ldots, A_m$; a target integer k
Decide: Does there exist a placement of size k?

Lemma A.2 *DEC_MAX_APP is NP-complete.*

Proof sketch: One needs only mimic the proof of Lemma A.1, since DEC_MAX_APP becomes DEC_MAX_CAP when each application has just one capsule.

General APP with the Capsule Placement Restriction (DEC_MAX_APP_RES).
Given:: n empty nodes $N_1, \ldots, N_n$; m applications $A_1, \ldots, A_m$; a target integer k
Decide: Does there exist a placement of size k that satisfies the capsule placement restriction?

Lemma A.3 *DEC_MAX_APP_RES is NP-complete.*

Proof sketch: Again, DEC_MAX_APP_RES becomes DEC_MAX_CAP when each application has just one capsule.

Since DEC_MAX_APP_RES is the decision version of APP, we finally have

Theorem A.1 *APP is NP-hard.*