Sharing-Aware Online Virtual Machine Packing in Heterogeneous Resource Clouds

Safraz Rampersaud, *Student Member, IEEE* and Daniel Grosu, *Senior Member, IEEE*

Abstract—One of the key problems that cloud providers need to efficiently solve when offering on-demand virtual machine (VM) instances to a large number of users is the VM Packing problem, a variant of Bin Packing. The VM Packing problem requires determining the assignment of user requested VM instances to physical servers such that the number of physical servers is minimized. In this paper, we consider a more general variant of the VM Packing problem, called the Sharing-Aware VM Packing problem, that has the same objective as the standard VM Packing problem, but allows the VM instances collocated on the same physical server to share memory pages, thus reducing the amount of cloud resources required to satisfy the users' demand. Our main contributions consist of designing several online algorithms for solving the Sharing-Aware VM Packing problem, and performing an extensive set of experiments to compare their performance against that of several existing sharing-oblivious online algorithms. For small problem instances, we also compare the performance of the proposed online algorithms against the optimal solution obtained by solving the offline variant of the Sharing-Aware VM Packing problem (i.e., the version of the problem that assumes that the set of VM requests are known a priori). The experimental results show that our proposed sharing-aware online algorithms activate a smaller average number of physical servers relative to the sharing-oblivious algorithms, directly reduce the amount of required memory, and thus, require fewer physical servers to instantiate the VM instances requested by users.

Index Terms—Clouds, heterogeneous resource, multilinear, online algorithm, sharing-aware, vector bin packing, virtual machine

1 Introduction

Cloud adoption by government, industrial, and academic institutions have created opportunities for providers to offer services through flexible infrastructures based on virtualization technologies. Industry forecasts predict that by 2018 approximately 80 percent of all workloads will be managed through data center virtualization services [1]. One of the key problems that cloud providers need to efficiently solve when offering on-demand virtual machine (VM) instances to a large number of users is the *VM Packing problem*, a variant of Bin Packing. The VM Packing problem requires determining the assignment of user requested VM instances to physical servers such that the number of physical servers is minimized.

Current virtualization technologies incorporate mechanisms that perform *memory reclamation*, i.e., mechanisms that regulate/conserve memory resources when multiple VMs are instantiated through a hypervisor layer. The deduplication of similar memory pages between two or more VMs instantiated through the same hypervisor layer, i.e., *page-sharing*, is an example of such mechanisms which are common to both open source and proprietary platforms.

In this paper, we take into account page-sharing and consider a more general variant of the VM Packing problem [2], called the *Sharing-Aware VM Packing problem*. The more general problem has the same objective as the standard VM Packing problem, but allows the VM instances collocated on the same physical server to share memory pages, thus reducing the amount of cloud resources required to satisfy the users' demand. We focus on the online version of the Sharing-Aware VM Packing problem (called *Sharing-Aware Online Virtual Machine Packing* (SA-OVMP)) problem which considers assigning the VMs, whose resource requests are unknown until they arrive to the system, such that the number of active servers is minimized. Traditionally, VM allocation problems with multiple resource requirements have been modelled as vector bin packing problems [3], where each resource is represented as a vector component. Classical sharing-oblivious vector bin packing algorithms in an online setting where VMs request multiple types of resources, will result in less efficient allocations since they do not leverage memory sharing opportunities. Therefore, in this paper, we design and investigate algorithms for solving the sharing-aware online VM Packing problem which results in a minimum number of active servers used to instantiate arriving VMs, where page-sharing occurs relative to VMs already instantiated on the servers. Since hypervisors used by cloud providers employ memory reclamation, our sharing-aware online algorithms leverage this utility; significantly reducing the number of servers needed to satisfy the user requests and implicitly reducing energy and service costs.

We design several sharing-aware online algorithms for solving the *Sharing-Aware Online Virtual Machine Packing* problem in cloud environments with heterogeneous server capacities and heterogeneous resource VM requests. Our proposed sharing-aware online algorithms are improved

- *The authors are with the Department of Computer Science, Wayne State University, 5057 Woodward Avenue, Detroit, MI 48202 E-mail: {safraz, dgrosu}@wayne.edu.*

Manuscript received 15 Jan. 2016; revised 27 Nov. 2016; accepted 13 Dec. 2016. Date of publication 20 Dec. 2016; date of current version 14 June 2017. Recommended for acceptance by I. Brandic.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2641937

designs of classical sharing-oblivious online algorithms for vector bin packing [3] which take page sharing into account when making allocation decisions. We introduce a new *server resource scarcity* metric necessary for designing sharing-aware online Best-Fit and Worst-Fit type algorithms. Our *server resource scarcity* metric considers all VM resource requirements, server's available resource capacities and page-sharing to identify a server with the highest priority to instantiate an online VM request. We perform extensive experiments to compare the performance of our sharing-aware online VM packing algorithms against several sharing-oblivious packing algorithms and against the optimal solutions obtained by solving the offline version of the Sharing-Aware VM Packing. In order to solve the offline version of the problem we provide a novel multilinear programming formulation of it. To the best of our knowledge, no sharing-aware online algorithms for packing VMs with multiple heterogeneous resource capacities and requirements have been proposed to date.

The rest of the paper is organized as follows. In Section 2, we discuss the related work. In Section 3, we define the Sharing-Aware Online VM Packing problem. In Section 4, we present the design of our proposed online sharing-aware algorithms. In Section 5, we present and solve the "offline" version of the sharing-aware VM packing problem. In Section 6, we compare the performance of our proposed algorithms against that of several sharing-oblivious algorithms through extensive experiments. In Section 7, we summarize our results and present possible directions for future research.

2 Related Work

Several variants of online vector bin packing problem modeling the allocation of resources in clouds have been recently investigated. Song et al. [4] proposed a semi-online bin packing algorithm for resource allocation. Their proposed setup allows VMs to be reshuffled through live migration among the servers if resource conservation can be achieved. Li et al. [5] introduced novel variants of bin packing algorithms which attempt to minimize the total cost associated with a server's utilization. Kamali and Ortiz [6] improved upon the upper bound for Next-Fit and introduced a new algorithm, Move To Front, which performed the best in the average case for the online dynamic bin packing total cost minimization problem. Azar et al. [7] proposed vector-bin packing algorithms, analyzed their performance under various VM sequences, and established lower competitive ratios. Panigrahy et al. [8] studied heuristic variants of the First-Fit-Decreasing algorithm for "offline" VM allocation. A recent extensive survey on Multidimensional Bin Packing and related problems is by Christensen et al. [3].

Resource awareness is a prevalent topic in designing resource allocation algorithms for cloud environments. Carli et al. [9] formulated a variant of the bin packing problem, called Variable-Sized Bin Packing with Cost and Item Fragmentation, which is energy-aware when attempting to pack cloud resource requests onto servers in both online and "offline" settings. Breitgand and Epstein [10] considered a variant of the bin packing problem called Stochastic Bin Packing (SBP) which is risk-aware of network bandwidth consumption, and designed both online and approximation algorithms to solve it. Kleineweber et al. [11] investigated a variant of the multi-dimensional bin packing problem which is QoS-aware relative to cloud file systems, specific to storage virtualization. Zhao et al. [12] designed online VM algorithms specific to energy and SLA-violation awareness to increase a cloud provider's revenue. Xu et al. [13] developed a hardware heterogeneity, VM-inference aware provisioning technique which focused on predicting MapReduce performance in the cloud. Xiao et al. [14] modeled the scaling of internet applications in the cloud as a class of constrained bin packing problem and solved the problem using an efficient semi-online algorithm which supports green-computing. Hao et al. [15] proposed an online, generalized VM placement strategy which considers variation on cloud architectures, resource demand duration and data-center location. While these contributions focus on VM allocation, none of them takes into account the potential for memory sharing when making allocation decisions.

Several systems such as Satori [16], Memory Buddies [17], and Difference Engine [18] considered hypervisor-based VM page-sharing, but did not address the design of sharing-aware online algorithms for VM packing. Sindelar et al. [2] were the first to propose and analyze *"offline" sharing-aware algorithms* for the VM Maximization and VM Packing problems under hierarchical page sharing models. Our work in this paper differs substantially from Sindelar et al. [2] in that we design algorithms for an online setting, consider multiple-type VM resource requests, assume heterogeneous server capacities and operate under a general sharing model which frees the limitation of page sharing due to grouping VMs via hierarchical models.

In our previous work [19], [20], we considered the design of sharing-aware "offline" algorithms for the VM Maximization problem under the general sharing model. The VM Maximization problem considered in our previous work is different from the problem of VM Packing considered in this paper. The objective of the VM Maximization problem is to allocate VM instances onto a set of servers such that the profit is maximized, while the objective of the VM Packing problem is to minimize the number of servers used to host user requested VM instances.

3 SA-OVMP Problem

We now introduce the Sharing-Aware Online Virtual Machine Packing problem from the perspective of a cloud service provider. The notation used in the paper is presented in Table 1.

We consider a cloud service provider that offers resources in the form of VM instances to cloud users. A VM instance is denoted by V_j and is characterized by a tuple $[q_j^u, q_j^m, q_j^s]$, where q_j^u is the number of requested CPUs, q_j^m is the amount of requested memory, and q_j^s is the amount of requested storage. The cloud service provider has a set $\mathcal{S}$ of servers available for instantiating user requested VMs. Each server $S_k \in \mathcal{S}$ is characterized by a tuple $[C_k^u, C_k^m, C_k^s]$, where C_k^u is the number of available CPUs, C_k^m is the available memory capacity, and C_k^s is the available storage capacity. We denote by $\mathcal{R}$ the subset of resource types composed of CPUs (type denoted by u) and storage (type denoted by s), that is, $\mathcal{R} = \{u, s\}$. The memory resource (type denoted

TABLE 1
Notation

Expression	Description
$\mathcal{S}$	Set of available servers.
V_j	Virtual machine j.
S_k	Server k.
$\overline{\mathcal{S}}$	Set of inactive servers; $\overline{\mathcal{S}} \subset \mathcal{S}$.
N	Maximum number of pages between S_k and V_j.
M	Number of servers in configuration; $\mid \mathcal{S} \mid = M$.
q_j^u	Requested number of CPUs by V_j (cores).
q_j^m	Requested amount of memory by V_j (GB).
q_j^s	Requested amount of storage by V_j (GB).
C_k^u	CPU capacity of server S_k (cores).
C_k^m	Memory capacity of server S_k (GB).
C_k^s	Storage capacity of server S_k (GB).
$\mathcal{R}$	Subset of server resource types u and s; $\mathcal{R} = \{u, s\}$.
e_j^k	Server scarcity metric relative to S_k and V_j.
s_j^k	Shared pages requested for V_j and managed by S_k.
$\mathcal{V}$	Set of available "offline" virtual machines.
$\mathcal{P}(\mathcal{V})$	Power set of "offline" virtual machines $\mathcal{V}$.
$\mathcal{J}$	Index of "offline" virtual machines in $\mathcal{P}(\mathcal{V})$.

by m) is not included in $\mathcal{R}$ since in the design of our algorithms we will treat the memory resource differently by considering memory sharing among the VMs collocated on the same server. For simplicity of presentation, we only consider these three types of resources; but the SA-OVMP problem and our algorithms in Section 4 can be easily extended to a general setting with any number of resources.

When several VM instances are hosted on a server S_k, and they use a common subset of memory pages, the total amount of memory allocated to those VM instances can be reduced through page-sharing. For example, when two Microsoft Windows 8 VM instances are collocated on the same server, they can share a significant amount of pages and the total allocated memory to those two VM instances can be reduced significantly compared to the case in which page sharing is not considered. To determine the amount of memory sharing among collocated VM instances, the cloud provider uses a *staging* server that computes the memory fingerprints [17] of the VM instance that is ready for allocation on one of the servers. The fingerprint of the VM instance is then used to determine the amount of memory sharing (in pages), denoted by s_j^k, which occurs among the currently considered VM instance, V_j, and the VM instances that are already hosted by server S_k. Bloom filters [17] are used to identify the number of shared pages s_j^k between VM V_j requested pages and pages already allocated to server S_k. This process has runtime complexity of $\mathcal{O}(N)$; where N is the maximum between the number of pages managed by server S_k and those pages required by V_j.

The cloud provider is interested in hosting all VM instances requested by the users while activating the minimum amount of servers. The requests for VM instances arrive one by one and the cloud provider decides the assignment of a newly arrived VM request without knowing any information about future requests. Thus, this is an online setting and the cloud provider must rely on online algorithms to assign VMs to servers. Our goal is to design such online algorithms for VM packing that take the sharing of memory into account when making allocation decisions. We formulate the Sharing-Aware Online VM Packing problem as follows.

SA-OVMP problem: We consider a cloud provider having a set of servers, $\mathcal{S} = \{S_1, S_2, \dots, S_{|\mathcal{S}|}\}$, where each server $S_k \in \mathcal{S}$ is characterized by [C_k^u, C_k^m, C_k^s], and a sequence of VM requests $\{V_1, V_2 \dots, V_j, \dots\}$, arriving one by one, where each VM request V_j is characterized by [q_j^u, q_j^m, q_j^s]. A VM request must be assigned to a server $S_k \in \mathcal{S}$ upon arrival, so that the following capacity constraints are satisfied:

$$C_k^m - q_j^m + s_j^k \geq 0 \tag{1}$$

$$C_k^r - q_j^r \geq 0, \quad \forall r \in \mathcal{R}, \tag{2}$$

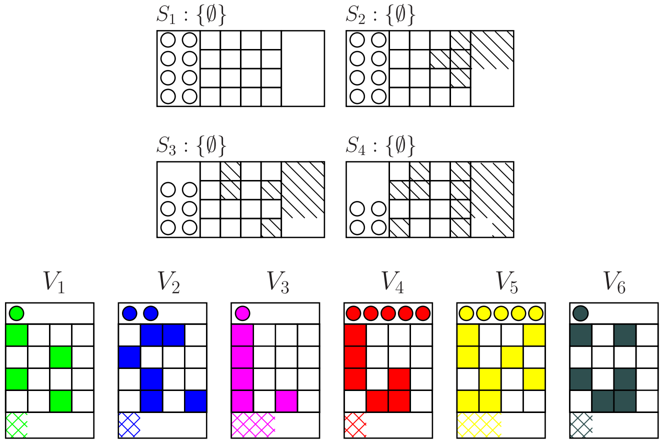
where s_j^k is the amount of memory sharing among the currently considered instance V_j and the VM instances that are already hosted by server S_k. The *objective* is to minimize the total number of active servers necessary to serve the requests.

Equation (1) is the memory capacity constraint, guaranteeing that the available memory capacity of server S_k is not exceeded. The available capacity $C_k^m - q_j^m$ is adjusted for the amount of sharing, s_j^k, between V_j and the VM instances already hosted by S_k. The constraints in Equation (2) guarantee that the capacities of the other types of resources of server S_k are also not exceeded.

4 SA-OVMP Algorithms

In this section, we design sharing-aware online algorithms for solving the SA-OVMP problem. Before describing the algorithms we introduce few definitions and assumptions concerning the servers. The servers managed by the cloud provider are in one of the following two states: *active* and *inactive*. An *active* server is a server that is powered on and is currently considered for allocation by the algorithms. An *inactive* server is a server that is not powered on and is not currently considered for allocation by the algorithms. We denote by $\overline{\mathcal{S}}$ the set of inactive servers. When all the VMs hosted by a server are terminated the server becomes an inactive server and can be activated in the future. Initially, all servers are inactive servers, i.e., $\overline{\mathcal{S}} = \mathcal{S}$. All the sharing-aware algorithms presented in the paper assume that the amount of sharing, s_j^k, among the currently arrived VM V_j and the VMs hosted by active server S_k, was already determined through memory fingerprinting on the staging servers as described in Section 3.

To illustrate how each of our sharing-aware online algorithms works, we consider an instance of the SA-OVMP problem with the resource configuration presented in Fig. 1. Each server in Fig. 1, $\mathcal{S}_1$ through $\mathcal{S}_4$, is characterized by the number of CPUs (each circle corresponds to 4 CPU cores available in the left rectangle within each server image), memory in MB (each small square corresponds to 4 MB of available, in the middle, larger square within each server image) and storage in GB (to which, a mesh block will correspond to 256 GB of available memory and fill the empty space in the right rectangle within each server image). The diagonal lines in each of the servers correspond to either unavailable memory or storage. By representing the servers in this way, we can capture the heterogeneity of available server resource capacities. Initially, there are no VMs allocated to the servers. This is represented by $S_k : \{\emptyset\}$ placed



V_j	units	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$	$\{6\}$	S_k	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$
q_j^m	4 MB	4	6	5	6	8	6	C_k^m	16	12	12	8
q_j^u	4 CPUs	1	2	1	5	5	1	C_k^u	8	8	6	4
q_j^s	256 GB	1	1	2	1	2	1	C_k^s	8	4	2	1

Fig. 1. VM requests and resource configuration.

above each server image. Each VM in Fig. 1, V_1 through V_6, is characterized by the same set of resource types as the servers and their requests are identified by shaded circles, shaded squares, and shaded mesh blocks (using the same units of measure as used for the servers, where one circle corresponds to 4 CPUs, one square corresponds to 4 MB, and one mesh block corresponds to 256 GB of storage). For instance, VM V_4 requests 20 CPUs, 24 MB of memory for a specific set of applications, libraries, etc., in exactly the memory pattern illustrated within the middle square and, lastly, it requests 256 GB of storage identified by one mesh block at the bottom of the VM image. When we illustrate how our sharing-aware online algorithms work, the server resources will be reduced incrementally in the included table and the space within the server for each resource type will be shaded according to the respective VM requests. Lastly, page sharing is identified when two or more VMs request memory by imposing a shaded rhombus on top of the memory block which is shared. Page sharing is illustrated in Fig. 2 through Fig. 8 for each of the proposed algorithms.

4.1 Next-Fit-Sharing (NFS) Algorithm

In order to design NFS, we need to introduce a third type of state for servers, called *closed*. A *closed* server is already hosting VM instances and is not currently considered for allocation by the algorithm. The NFS algorithm is given in Algorithm 1 and works as follows. Upon arrival of VM request V_j, the cloud provider determines if V_j can be packed onto the active server denoted by $S_{\tilde{k}} \in \mathcal{S} \setminus \overline{\mathcal{S}}$. Only one server is active at any time and server S_1 is initially activated upon the first VM arrival. If active server $S_{\tilde{k}}$ has enough capacity for every resource type to instantiate V_j while considering the sharing of memory, $s_j^{\tilde{k}}$, then V_j is packed onto server $S_{\tilde{k}}$ (lines 3 and 4). Else, server $S_{\tilde{k}}$ is closed using a function close (line 6) and the search begins for finding a server which has enough resource capacity to instantiate V_j. We note that for problem instances with servers having the same resource types and size characteristics, the next server will automatically suffice if every server has enough capacity for every VM type. For servers with heterogeneous resource characteristics (which is the case in our SA-OVMP problem), a search must ensue to find a server which meets the V_j's resource demand.

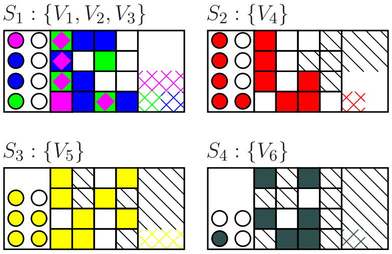


Fig. 2. NFS: VM assignment.

Algorithm 1. NFS

```
1: Input: VM instance arrival ($V_j$)
2: {$S_{\tilde{k}}$: currently active server.}
3: if ($[C_{\tilde{k}}^m, C_{\tilde{k}}^u, C_{\tilde{k}}^s] - [q_j^m - s_j^{\tilde{k}}, q_j^u, q_j^s] \geq [0,0,0]$) then
4:     $S_{\tilde{k}} \leftarrow S_{\tilde{k}} \cup \{V_j\}$
5: else
6:     close($S_{\tilde{k}}$)
7:     $\tilde{k} \leftarrow \tilde{k} + 1$
8:     while ($\tilde{k} \leq |\mathcal{S}|$) do
9:         if ($[C_{\tilde{k}}^m, C_{\tilde{k}}^u, C_{\tilde{k}}^s] - [q_j^m - s_j^{\tilde{k}}, q_j^u, q_j^s] \geq [0,0,0]$) then
10:            activate($S_{\tilde{k}}$)
11:            $\overline{\mathcal{S}} \leftarrow \overline{\mathcal{S}} \setminus \{S_{\tilde{k}}\}$
12:            break
13:        $\tilde{k} \leftarrow \tilde{k} + 1$
14:    if ($\tilde{k} > |\mathcal{S}|$) then
15:        exit
16:    $S_{\tilde{k}} \leftarrow S_{\tilde{k}} \cup \{V_j\}$
17: $[C_{\tilde{k}}^m, C_{\tilde{k}}^u, C_{\tilde{k}}^s] \leftarrow [C_{\tilde{k}}^m, C_{\tilde{k}}^u, C_{\tilde{k}}^s] - [q_j^m - s_j^{\tilde{k}}, q_j^u, q_j^s]$
```

Following server $S_{\tilde{k}}$'s closure, server index $\tilde{k}$ is incremented (line 7). The algorithm enters a while loop to search for a server among the inactive servers which can host V_j (line 8). If the V_j's resource demand can be satisfied by server $S_{\tilde{k}}$, then the server is activated by a function activate, removed from the set of inactive servers, and the algorithm leaves the while loop (lines 10-12). Else, the search continues within the while loop by incrementing server index $\tilde{k}$ until a server is found with enough resources to host V_j (line 13). Following the while loop, if the server index exceeds the number of available servers, V_j cannot be hosted and the algorithm exits (lines 14 and 15). Otherwise, the algorithm found a suitable server $S_{\tilde{k}}$ within the available servers and V_j is allocated to $S_{\tilde{k}}$ (line 16). Lastly, server $S_{\tilde{k}}$'s resource capacities are reduced accordingly (line 17).

The difference between NFS and a standard sharing-oblivious Next-Fit (NF) algorithm modified for VM allocation is that page sharing is accounted for in NFS and a search is performed to find a server which meets the incoming VM request. The standard sharing-oblivious NF algorithm has a runtime of $\mathcal{O}(1)$ when allocating a VM request to servers, where each server has the same initial resource type capacities. In the case of NFS, the run time increases due to the search for the *next* server which can host V_j; resulting in a run time of $\mathcal{O}(M)$ in the worst case, where M is the number of servers under management. Lastly, allocating V_j requires searching for page sharing relative to only one active server $S_{\tilde{k}}$ as described in Section 3, thus resulting in a total run time of $\mathcal{O}(NM)$ for NFS.

Fig. 2 illustrates the assignment of VMs to servers according to NFS for the SA-OVMP instance presented in Fig. 1. All six VMs are assigned sequentially from V_1 to V_6. VMs V_1, V_2 and V_3 are assigned to S_1; which is initially active. When V_4 arrives, it cannot be assigned to S_1 due to over-committing the CPU capacity. Server S_1 is then closed, S_2 is found to satisfy V_4's resource request at which time S_2 is activated and V_4 is assigned to it. Next, V_5 arrives and cannot be assigned to S_2 due to over-committing the memory capacity. Server S_2 is then closed, S_3 is found to satisfy V_5's resource request at which time S_3 is activated and V_5 is assigned to it. Lastly, V_6 arrives and cannot be assigned to S_3 due to over-committing the storage capacity. Server S_3 is then closed, S_4 is found to satisfy V_6's resource request at which time S_4 is activated and V_6 is assigned to it. NFS requires all four servers in order to assign the VMs. For the SA-OVMP problem instance considered here, the sharing-oblivious NF implementation would also require all four servers to assign the VMs; albeit, more memory would be consumed on server S_1.

Algorithm 2. FFS

```
1: Input: VM instance arrival ($V_j$)
2: $\tilde{k} \leftarrow 0$
3: $flag \leftarrow 1$
4: while ((active($S_{\tilde{k}}$) = true) and ($\tilde{k} \leq |\mathcal{S}|$)) do
5:     if ($[C^m_{\tilde{k}}, C^u_{\tilde{k}}, C^s_{\tilde{k}}] - [q^m_j - s^{\tilde{k}}_j, q^u_j, q^s_j] \geq [0,0,0]$) then
6:         $flag \leftarrow 0$
7:         break
8:     $\tilde{k} \leftarrow \tilde{k} + 1$
9: if ($flag$) then
10:    while ($\tilde{k} \leq |\mathcal{S}|$) do
11:        if ($[C^m_{\tilde{k}}, C^u_{\tilde{k}}, C^s_{\tilde{k}}] - [q^m_j - s^{\tilde{k}}_j, q^u_j, q^s_j] \geq [0,0,0]$) then
12:            activate($S_{\tilde{k}}$)
13:            $\overline{\mathcal{S}} \leftarrow \overline{\mathcal{S}} \setminus \{S_{\tilde{k}}\}$
14:            break
15:        $\tilde{k} \leftarrow \tilde{k} + 1$
16: if ($\tilde{k} > |\mathcal{S}|$) then
17:     exit
18: $S_{\tilde{k}} \leftarrow S_{\tilde{k}} \cup \{V_j\}$
19: $[C^m_{\tilde{k}}, C^u_{\tilde{k}}, C^s_{\tilde{k}}] \leftarrow [C^m_{\tilde{k}}, C^u_{\tilde{k}}, C^s_{\tilde{k}}] - [q^m_j - s^{\tilde{k}}_j, q^u_j, q^s_j]$
```

4.2 First-Fit-Sharing (FFS) Algorithm

We now introduce the FFS algorithm which is similar to NFS except that servers are never closed when a VM request cannot fit into a server. Rather, any server that cannot accommodate the current VM request will remain active in anticipation of another VM request which can be accommodated. FFS is given in Algorithm 2 and works as follows.

Upon arrival of VM request V_j, a search ensues to determine the first active server $S_{\tilde{k}}$ from the set of active servers $\mathcal{S} \setminus \overline{\mathcal{S}}$, which has enough capacity for every resource type to host V_j while considering memory sharing in the amount of s^k_j. To simplify the description of the algorithm, we assume that all active servers are placed before any of the inactive servers in the search sequence. The algorithm executes a while loop to search for the first active server $S_{\tilde{k}}$ that meets V_j's resource demand in consideration of memory sharing (line 4). If a suitable server is found among the active servers, then *flag* is set to 0, and the algorithm leaves the while loop (lines 5-7). Else, the search continues within the while loop by incrementing server index $\tilde{k}$ until a server with enough resources to host V_j is found (line 8). If there are no active servers which can host V_j, *flag* is still 1, signalling the need to search for a suitable server among the set of inactive servers. The search process among the inactive servers (lines 10-15) is similar to NFS (Algorithm 1, lines 8-16) except that upon reaching the *flag* if condition, server index $\tilde{k}$ has already been incremented to the first inactive server. If $\tilde{k}$ is greater than the number of available servers in the active or inactive server search, the algorithm exits (lines 16-17). If a suitable server $S_{\tilde{k}}$ has been found from either the active or inactive servers, V_j is assigned to $S_{\tilde{k}}$, and $S_{\tilde{k}}$'s resource capacities are reduced accordingly (lines 18-19).

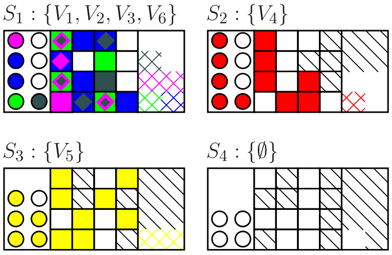


Fig. 3. FFS: VM assignment.

The difference between FFS and the standard sharing-oblivious First-Fit (FF) algorithm modified for VM allocation is that page sharing is accounted for in FFS and a search for a server which meets the incoming VM request is performed. FFS undergoes the same fingerprinting process mentioned in Section 3 to determine similar pages (taking $\mathcal{O}(N)$ time) and searches for either the first active server which meets the VM resource request over the set of active servers, or determines the first inactive server to activate in order to satisfy the VM resource request. Since the run time of the search can be at most $O(M)$, FFS has a run time complexity of $\mathcal{O}(NM)$ for allocating one VM request.

In Fig. 3, we present the assignment of VMs using FFS for the SA-OVMP instance from Fig. 1. VMs V_1, V_2 and V_3 are assigned to S_1; which is initially activated. When V_4 arrives, it cannot be assigned to $\mathcal{S}_1$ due to over-committing the CPU capacity. Server S_2 is found to satisfy V_4's resource request at which time server S_2's state is changed from inactive to active and V_4 is assigned to it. Next, V_5 arrives and cannot be assigned to either S_1 or S_2 due to over-committing the CPU capacity. Server S_3 is found to satisfy V_5's resource request at which time server S_3's state is changed from inactive to active and V_5 is assigned to it. Lastly, V_6 arrives and according to the search, V_6 can be assigned to S_1 since it is still in an active state. By consolidating the VM request to an already activated server which was not *closed*, FFS activates fewer servers, and thus, achieves better performance than NFS.

4.3 Best-Fit-Sharing (BFS) Algorithm

In order to design BFS, we introduce the *server resource scarcity* metric which characterizes the scarcity of the resources at a given server relative to the requested resources by a VM. The classical sharing-oblivious Best-Fit (BF) packing algorithm places a new item into the bin with the least remaining current capacity according to one dimension, i.e., the size of the item in one dimension. Since the SA-OVMP problem considers multiple resource requirements, we have to consider all required resources and available capacities when determining the appropriate server for allocating the VM request. To be able to achieve this, we define the *server resource scarcity* metric as follows:

$$e_j^k = \begin{cases} \max\left\{\frac{q_j^m - \sqrt{s_j^k}}{C_k^m}, \frac{q_j^u}{C_k^u}, \frac{q_j^s}{C_k^s}\right\} & \text{if } C_k^m - q_j^m + s_j^k \geq 0 \;\& \\ & C_k^u - q_j^u \geq 0 \;\& \\ & C_k^s - q_j^s \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The metric characterizes the scarcest resource among all resource types from server S_k relative to V_j's resource requirements. Each resource request type is expressed as a remaining resource capacity ratio in Equation (3) relative to the available server capacity type, if V_j were to be instantiated on S_k. These ratios are only relevant if the V_j's resource requests do not over-commit any of the resource capacities on server S_k. The maximum remaining resource ratio among the three resource types reflects the scarcest remaining resource after server S_k instantiates VM V_j.

The scarcity metric takes into account the sharing of memory pages among the current VM requesting resources and the VMs already allocated to server S_k. This is achieved by subtracting $\sqrt{s_j^k}$ from the amount of memory requested by VM V_j in the numerator of the remaining ratio for memory, (i.e., $\frac{q_j^m - \sqrt{s_j^k}}{C_k^m}$). This allows us to decrease the amount of memory requested by a VM by the amount of pages already shared among it and the collocated VMs. As an example let us consider that the current VM requests $q_j^m = 6$ pages on server S_k. The current VM has the potential to share $s_j^k = 4$ pages with the VMs already collocated at S_k, and therefore these pages will not require new memory allocation and, in the scarcity metric, the amount of requested pages by V_j has to be decreased by the number of shared pages, $s_j^k = 4$. We chose to subtract $\sqrt{s_j^k}$ from the amount of pages requested instead of s_k^j to avoid situations where VM V_j has a sizable memory request which shares a significant amount of pages with already hosted VMs making the memory resource appear less scarce when compared to the other resources. For our example, the numerator of the remaining resource ratio for memory is $q_j^m - \sqrt{s_j^k} = 6 - 2 = 4$. Lastly, if V_j's resource demand over-commits any of the server S_k's capacities, then the value of the server resource scarcity metric will be 0 indicating an absence of opportunity to assign V_j to S_k.

In our previous work [21] we employed a different scarcity metric (given by, $e_j^k = \frac{q_j^u}{C_k^u} + \frac{q_j^s}{C_k^s} + \frac{q_j^m - s_j^k}{C_k^m}$) in the design of a BFS-type algorithm. The metric is defined as the sum of the remaining resource ratios and characterizes the scarcity of resources in an aggregated way, which does not capture very well the scarcity of resources. As an example let us consider two servers having two types of resources, CPU and memory, and the following values for the remaining resource capacity ratios: Server 1 (0.1 for CPU and 0.8 for memory) and Server 2: (0.4 for CPU and 0.5 for memory). The scarcity metric used in [21] has the same value for both servers (i.e., 0.9) even though the memory is very scarce at Server 1. Our new scarcity metric (Equation (3)) which uses the maximum among the remaining capacity ratios determines two different values for the two servers, that is, 0.8 and 0.5, and therefore the BFS algorithm selects the first server for allocation. The new metric is able to capture the scarcity of resources in a much better way.

Algorithm 3. BFS

```
1:  **Input:** VM instance arrival ($V_j$)
2:  $\tilde{k} \leftarrow 0$
3:  $flag \leftarrow 0$
4:  while ((active($S_{\tilde{k}}$) = true) and ($\tilde{k} \leq |\mathcal{S}|$)) do
5:      if ($[C_k^m, C_k^u, C_k^s] - [q_j^m - s_j^k, q_j^u, q_j^s] \geq [0, 0, 0]$) then
6:          $flag \leftarrow 1$
7:          $e_j^k \leftarrow \max\left\{\frac{q_j^m - \sqrt{s_j^k}}{C_k^m}, \frac{q_j^u}{C_k^u}, \frac{q_j^s}{C_k^s}\right\}$
8:      else
9:          $e_j^k \leftarrow 0$
10:     $\tilde{k} \leftarrow \tilde{k} + 1$
11: if ($flag$) then
12:         $\tilde{k} \leftarrow \text{argmax}\{e_j^k\}$
13: else
14:     while ($\tilde{k} \leq |\mathcal{S}|$) do
15:         if ($[C_{\tilde{k}}^m, C_{\tilde{k}}^u, C_{\tilde{k}}^s] - [q_j^m - s_j^{\tilde{k}}, q_j^u, q_j^s] \geq [0, 0, 0]$) then
16:             activate($S_{\tilde{k}}$)
17:             $\overline{\mathcal{S}} \leftarrow \overline{\mathcal{S}} \setminus \{S_{\tilde{k}}\}$
18:             break
19:         $\tilde{k} \leftarrow \tilde{k} + 1$
20: if ($\tilde{k} > |\mathcal{S}|$) then
21:     exit
22: $S_{\tilde{k}} \leftarrow S_{\tilde{k}} \cup \{V_j\}$
23: $[C_{\tilde{k}}^m, C_{\tilde{k}}^u, C_{\tilde{k}}^s] \leftarrow [C_{\tilde{k}}^m, C_{\tilde{k}}^u, C_{\tilde{k}}^s] - [q_j^m - s_j^{\tilde{k}}, q_j^u, q_j^s]$
```

BFS is given in Algorithm 3 and works as follows. Upon the arrival of VM request V_j, a search ensues to determine the active server $S_{\tilde{k}} \in \mathcal{S} \setminus \overline{\mathcal{S}}$ which would have the least remaining single resource after instantiating VM V_j (i.e., the scarcest resource). The algorithm calculates the resource scarcity metric for each server in the set of active servers through a while loop (line 4). If at least one active server has enough resource capacities to meet the V_j's resource demand (line 5), then *flag* will be set to 1, which guarantees that V_j will be assigned to one of the active servers, and the V_j resource scarcity metric is calculated relative to S_k (lines 6 and 7). Else, at least one of the resource requests violate at least one of the current active server capacities, and then the server resource scarcity metric would be 0 for those servers (line 9). Calculating the resource scarcity metric among the active servers continues within the while loop by incrementing server index $\tilde{k}$ until the first inactive server is found (line 10). If *flag* is set to 1 following the while loop, then the index of the server with the maximum resource scarcity metric is determined and stored in $\tilde{k}$ (line 12). If no active servers have enough resources available to host V_j according to resource scarcity metric, then a search for a suitable server among the set of inactive servers occurs

V_1	s_1^k	e_1^k	C_k^m	C_k^u	C_k^s
S_1	0	0.250	16	8	8
S_2	0	0.333	12	8	4
S_3	0	0.5	12	6	2
S_4	0	**1.000**	8	4	1

V_2	s_2^k	e_2^k	C_k^m	C_k^u	C_k^s
S_1	0	0.375	16	8	8
S_2	0	**0.500**	12	8	4
S_3	0	0.500	12	6	2
S_4	0	0.000	4	3	0

V_3	s_3^k	e_3^k	C_k^m	C_k^u	C_k^s
S_1	0	0.313	16	8	8
S_2	2	0.774	6	6	3
S_3	0	**1.000**	12	6	2
S_4	3	0.000	4	3	0

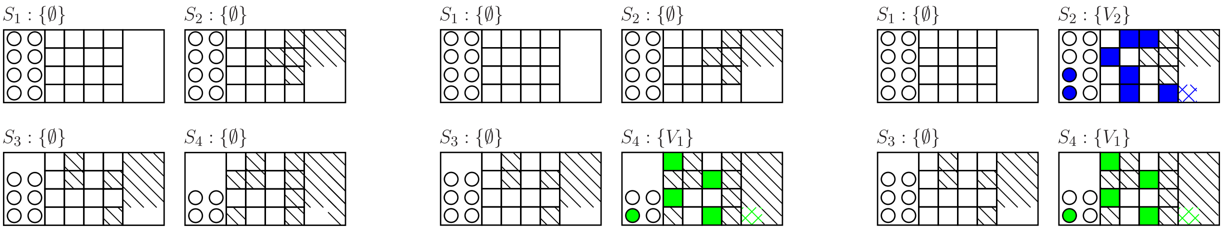


Fig. 4. BFS: VMs 1 through 3 assignment.

V_4	s_4^k	e_4^k	C_k^m	C_k^u	C_k^s
S_1	0	0.625	16	8	8
S_2	2	**0.942**	6	6	3
S_3	4	0.000	7	5	0
S_4	3	0.000	4	3	0

V_5	s_5^k	e_5^k	C_k^m	C_k^u	C_k^s
S_1	0	**0.438**	16	8	8
S_2	4	0.000	2	1	2
S_3	2	0.000	7	5	0
S_4	2	0.000	4	3	0

V_6	s_6^k	e_6^k	C_k^m	C_k^u	C_k^s
S_1	0	0.627	8	7	6
S_2	4	**1.000**	2	1	2
S_3	2	0.000	7	5	0
S_4	2	0.000	4	3	0

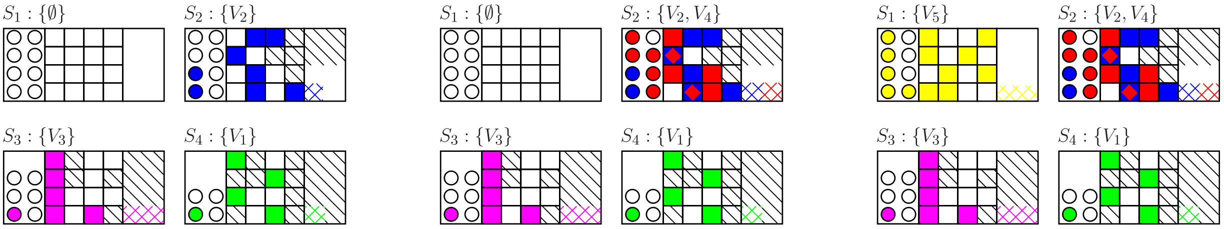


Fig. 5. BFS: VMs 4 through 6 assignment.

(lines 14-21) exactly as in FFS (Algorithm 2, lines 10-17). Lastly, VM V_j is then assigned to server $S_{\tilde{k}}$ which would have the least remaining resource following instantiation and server $S_{\tilde{k}}$'s resource capacities are reduced according to V_j's resource demand (lines 22-23).

There are several differences between BFS and the sharing-oblivious version of the BF algorithm. From a general point of view, BF assigns items into bins based on the least remaining space after item placement. When considering BF for VM allocation, the algorithm would only account for a single resource. When multiple resources are considered, BF can have several interpretations for allocating VMs to servers based on various resources. BFS is more precise in that it is guided by the least remaining resource among all resources identified by the metric in Equation (3). Another difference is that BFS accounts for page sharing within each server when allocating the incoming VMs, whereas the standard BF algorithm does not. Provided the similarities between BFS and FFS, the run time complexity of BFS is also $\mathcal{O}(NM)$, which includes calculating the resource scarcity metric for any incoming VM relative to the available, active servers.

We now illustrate the assignment process of BFS using the SA-OVMP instance from Fig. 1. Fig. 4 illustrates the process for VMs V_1 through V_3. The amount of sharing, s_1^k, and the server resource scarcity metric, e_1^k, are calculated relative to V_1 and the servers within the configuration. Since there are no VMs assigned to the server, s_1^k is zero and a server which will leave the least amount of a single resource following instantiation is selected (i.e., the *best* fit server). Server S_4 has the highest value for the resource scarcity metric since the resource capacities are lower than the rest of the servers. Therefore, V_1 is assigned to S_4 and S_4's capacities are reduced accordingly and updated. Next, V_2 is ready for instantiation. All s_2^k, are 0 since no pages are shared with V_1. The server resource scarcity metric is the same for both S_2 and S_3. The resource which will yield the least remaining space per our metric is the memory, where both S_2 and S_3 offer the same memory capacities. To break the tie, we select the lowest indexed server with the highest server resource scarcity metric, e.g., S_2, to host V_2 and the resource capacities of S_2 are updated. Relative to server S_4, $e_2^4 = 0$ since there is not enough memory available. Next, V_3 is ready for instantiation. With V_1 assigned to S_4 and V_2 assigned to S_2, V_3 has two opportunities to share pages, leading to $s_3^2 = 2$ and $s_3^4 = 3$. Upon calculating the server resource scarcity metrics, it is determined that V_3 should be assigned to S_3 due to the scarcity of storage which occurs following instantiation against the other servers. The BFS assignment for VMs V_4 through V_6 is illustrated in Fig. 5. VM V_4 will be assigned to S_2 due to the CPU resource being the most scarce resource following instantiation when compared to S_1. The assignment of V_5 to server S_1 is by default since the other servers do not have enough CPU capacities to instantiate the request. Lastly, V_6 arrives and due to both the CPU and storage requests, the resource scarcity metric has a value of 1.0 relative to S_2 which is the largest. Thus, V_6 is assigned to S_2. The final VM assignment for the SA-OVMP instance considered here is illustrated in Fig. 6.

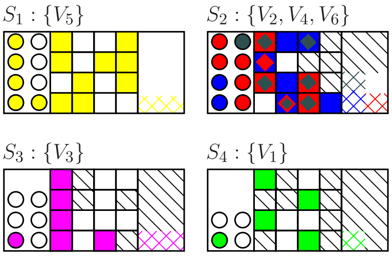


Fig. 6. BFS: VM final assignment.

4.4 Worst-Fit-Sharing (WFS) Algorithm

Since WFS can be viewed as the dual of BFS and thus, its structure and implementation are nearly identical to that of BFS, we will not provide a formal algorithmic description of it. The only difference between the two algorithms is that WFS allocates the new VM request to an active server with the minimum server resource scarcity metric, i.e., assigns the VM to the server which leaves the most remaining single resource following instantiation. WFS requires a change from $\text{argmax}\{e_j^k\}$ to $\text{argmin}\{e_j^k\}$ in BFS (line 11) and the maximum operator in Equation (3) is changed to the minimum operator. Due to the similarity to BFS, the run time complexity of WFS is also $\mathcal{O}(NM)$.

5 Offline Sharing-Aware VM Packing

In this section, we present a multilinear programming formulation of the "offline" Sharing-Aware VM Packing problem. This problem differs from the online version in Section 3 since it assumes that the set of VM requests, $\mathcal{V}$, is known a priori. In order for a solution to exists, we have to guarantee that enough servers are available to host all $V_j \in \mathcal{V}$. The objective of the service provider is to host all $V_j \in \mathcal{V}$, while minimizing the number of active servers necessary for instantiating the VMs in $\mathcal{V}$. We formulate this problem as a multilinear boolean program as follows:

$$\text{minimize: } B = \sum_{k:S_k \in \mathcal{S}} y_k \tag{4}$$

$$\text{subject to:} \tag{5}$$

$$\sum_{k:S_k \in \mathcal{S}} x_{jk} = 1, \ \forall j : V_j \in \mathcal{V} \tag{6}$$

$$\sum_{j:V_j \in \mathcal{V}} q_j^r x_{jk} \leq y_k C_k^r, \ \forall k : S_k \in \mathcal{S}, \ \forall r \in \mathcal{R} \tag{7}$$

$$\sum_{\mathcal{J} \in \mathcal{P}(\mathcal{V})} (-1)^{(|\mathcal{J}|+1)} \sigma_{\mathcal{J}} \prod_{\hat{j} \in \mathcal{J}} x_{\hat{j}k} \leq y_k C_k^m, \ \forall k : S_k \in \mathcal{S} \tag{8}$$

$$\forall \ y_k \in \{0, \ 1\}, \quad \forall \ x_{jk} \in \{0, \ 1\}. \tag{9}$$

A boolean decision vector $y \in \{0,1\}^M$ is the solution to our program from Equation (4); where the active servers are identified by $y_k = 1$, inactive servers are identified by $y_k = 0$, and B is the sum of the total number of active servers over all components of y. The constraint in Equation (6) ensures that V_j is not assigned to more than one server, where x_{jk} reflects the assignment of VM V_j to a single server S_k. Equation (7) is a resource capacity constraint which ensures that the subset of instantiated VM requests do not violate the server capacities, C_k^r, the provider has available in terms of CPUs, $r = u$, and storage, $r = s$. Equation (8) is the memory capacity constraint and ensures that the VMs requesting memory do not violate the service provider's memory capacities which considers the effect of page deduplication. Equation (9) ensures that decision variables y_k and x_{jk} are boolean. Fig. 7 shows the solution of our multilinear program for the SA-OVMP instance from Fig. 1. The optimal solution packs VMs V_1 through V_6 onto two servers, leading to a lower number of active servers than any of the online algorithms proposed in Section 4. The novelty of our multilinear program formulation is in how the memory constraint takes into account the memory requests with regards to page sharing. To describe the constraint, we consider an example using VMs V_3, V_4 and V_6 and server S_2.

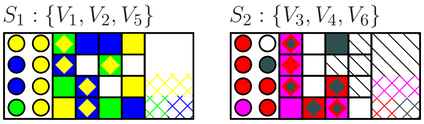
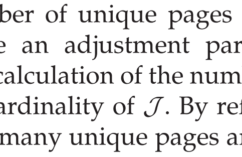


Fig. 7. Optimal VM assignment.

In Equation (8), we denote by $\mathcal{P}(\mathcal{V})$, the power set of the set of available VMs, $\mathcal{V}$, and index the elements from this power set using $\mathcal{J}$. We define the *sharing parameter* $\sigma_{\mathcal{J}}$ as the variable which represents the number of shared pages among the VMs in set $\mathcal{J}$. As an example, for $|\mathcal{J}| = 3$, we have $\sigma_{346} = 2$, i.e., all VMs in $\mathcal{J}$ which include V_3, V_4 and V_6 share three pages between them. We calculate the sharing parameter $\sigma_{\mathcal{J}}$ for all the sets of the power set $\mathcal{P}(\mathcal{V})$ indexed by $\mathcal{J}$, and organize them by cardinality in Fig. 8. When $|\mathcal{J}| = 1$, the sharing parameter $\sigma_{\mathcal{J}}$ represents the amount of memory resource in number of pages requested by V_j, i.e., $\sigma_j = q_j^m$. By combining the set of values representing the number of shared pages and the number of pages required by each VM, we can deduce the number of *unique* pages, i.e., pages which are required to instantiate a subset of VMs and are available to be shared among requesting VMs. To calculate the number of unique pages in Equation (8) we need to introduce an adjustment parameter, $(-1)^{(|\mathcal{J}|+1)}$, which adjusts the calculation of the number of unique pages according to the cardinality of $\mathcal{J}$. By referencing Fig. 8, we can calculate how many unique pages are required in order to instantiate VMs V_3, V_4 and V_6 and compare this to S_2's memory capacity, C_2^m, as follows:

$$\begin{aligned}&(+1)(\sigma_3 + \sigma_4 + \sigma_6) + (-1)(\sigma_{34} + \sigma_{36} + \sigma_{46}) \\ &\quad + (+1)(\sigma_{346}) \leq C_2^m.\end{aligned} \tag{10}$$



$\|\mathcal{J}\| = 1$	$\|\mathcal{J}\| = 2$	$\|\mathcal{J}\| = 3$
🟪 $\sigma_3 = 5$	◆ $\sigma_{34} = 4$	◆ $\sigma_{346} = 3$
🟥 $\sigma_4 = 6$	◆ $\sigma_{36} = 3$	
⬛ $\sigma_6 = 6$	◆ $\sigma_{46} = 5$	

Fig. 8. Sharing parameter values among V_3, V_4 and V_6.

TABLE 2
Google Compute Engine VM Instances

	Low Resource Request VMs in Experiments						High Resource Request VMs in Experiments					
Resource	{n1s1}	{n1s2}	{n1c2}	{n1m2}	{n1c4}	{n1c8}	{n1s4}	{n1m4}	{n1s8}	{n1m8}	{n1s16}	{n1c16}
Memory (GB)	3.75	7.50	1.80	13	3.6	7.20	15	26	30	52	60	14.40
CPU	1	2	2	2	4	8	4	4	8	8	16	16

By substituting the values for $\sigma_{\mathcal{J}}$ from Fig. 8 and performing the calculation above in Equation (10), we arrive at 8 unique pages which are required to allocate V_3, V_4 and V_6, when sharing pages is considered; consistent with the number of colored pages in Fig. 8. In most cases, only a subset of the VMs may be chosen for instantiation based on the service provider's memory resource. Therefore, the constraint in Equation (8) consists of the product of boolean decision variables, $x_{\tilde{j}k}$, where $\tilde{j}$ is an index corresponding to any VM $V_{\tilde{j}}$ within the VM subset combination $\mathcal{J}$, on the sharing parameter $\sigma_{\mathcal{J}}$, and the unique page adjustment parameter $(-1)^{(|\mathcal{J}|+1)}$.

The "offline" Sharing-Aware VM Packing problem is a new and more general variant of the vector bin packing problem. If sharing is ignored, the Sharing-Aware VM packing reduces to vector bin packing. Because vector bin packing is $\mathcal{NP}$-hard and it is a special case of the offline Sharing-Aware VM packing problem, the "offline" Sharing-Aware VM Packing problem is also $\mathcal{NP}$-hard. Therefore solving the "offline" Sharing-Aware VM Packing problem optimally is only practical for small instances. Solving the "offline" version of the SA-OVMP problem instance in Fig. 1 only takes a few seconds, but when we modestly increased the number of VMs and the number of servers, the time required to solve it took several minutes to complete. In Section 6.3 we present a detailed set of experiments in which we compare the execution times of the proposed online algorithms against those obtained when solving the offline version using a standard nonlinear programming solver.

6 Experimental Results

In this section, we describe the experimental setup including our strategy for generating VM streams, simulating server configurations, and modelling page sharing. We perform extensive simulation experiments with our sharing-aware online algorithms and their sharing-oblivious counterparts and then analyze the results. The main goal of our experiments is to compare the performance of the proposed algorithms against that of several sharing-oblivious packing algorithms. We are interested to determine the reduction in the total number of physical servers and the reduction in the amount of memory obtained by the proposed algorithms relative to the other algorithms when considering real heterogeneous servers and VM instance requests. We will also compare how well our algorithms are performing against the "offline" optimal solution in terms of average number of activated servers.

6.1 Experimental Setup

The simulation experiments were performed using a custom build simulation software implemented in C++ and run on 2.93 GHz Intel hexa-core dual-processor 64 bit systems within the Wayne State University HPC Grid [22]. The simulation software takes as input the server configurations and the VM streams and executes the VM packing algorithms to determine the allocations. Based on the obtained allocations it computes the values of several metrics that are used in our experimental analysis (e.g., average number of active servers, memory usage, and others).

6.1.1 VM Streams

Fairly recently, Google has made workload usage traces from Google compute cells [23] available to the public. Researchers have thoroughly investigated various components of the usage traces, such as applications [24] and workloads [25], [26], [27]. Significant to our experiments is the arrival pattern of VM resource requests and how our proposed algorithms behave under these patterns. Based on existing research [26], [28], it has been concluded that there are no standard distributions which fit the pattern of VM resource requests. Some statistical properties have been revealed such as, resource requests exhibiting a heavy-tailed distribution [26], requests reflecting degrees of fractal self-similarity [28], and the proportion of lower memory and CPU requests significantly outweigh higher memory and CPU requests within the trace [29]. Given the difficulties in identifying overall arrival and request characteristics from the traces, we design a broad range of VM *streams* which provide numerous variations on the mixture of requested VM types, arrival orderings (which is significant for online settings).

For our experiments, we consider the resource request characteristics from Google Compute Engine VM types which are listed in Table 2 and are available online [30]. We divide the VMs into two categories, *low resource request* and *high resource request*, based mostly on the memory and CPU request combinations. We keep n1m2 and n1c8 in the lower resource category since n1m2 only requests 2 CPUs and n1c8 requests a very low amount of memory compared to those VMs in the high resource request category. We define a *stream* as a sequence of either 500 or 1,000 VMs requests which exhibit various percentages of mixture between low and high VM resource requests. We design a set of VM streams accounting for various VM type mixtures in increments of 5 percent, ranging from 5 percent low (and 95 percent high) to 95 percent low (and 5 percent high) resource requests. Therefore, in order to test the performance of our algorithms, we consider common and uncommon workloads which span the VM resource request mixtures. For each VM stream, we randomly select VMs from each of the two requesting categories, until a desired percentage of mixture is achieved. As an example, for the 85 percent low request 1,000 VM stream, we select

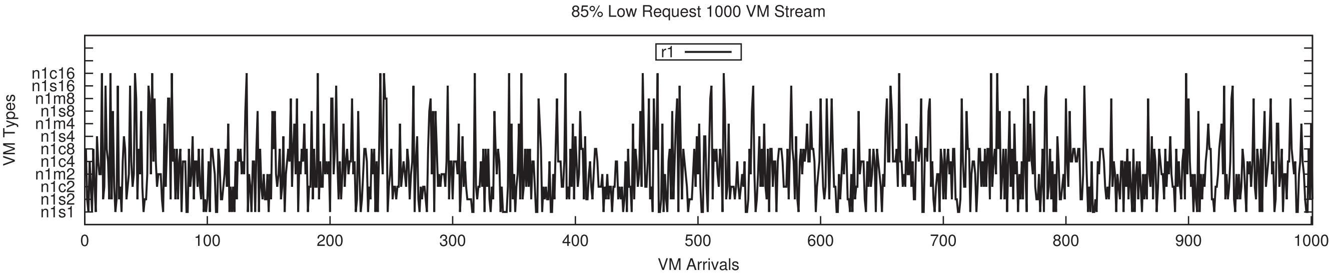


Fig. 9. 85 percent low request 1,000 VM stream.

uniformly at random 850 VMs from the low requesting category, leaving 150 VMs to be selected uniformly at random from the high requesting category in order to complete the stream. Once all the streams have been designed, we generate five copies of each stream and identify them by r1 through r5. Each r1 through r5 stream per mixture combination is then randomly shuffled using the C++ facility random_shuffle and the standard uniform random generator. Each r1 through r5 stream is shuffled a different number of times such that the stream sequences exhibit a fairly significant variability from each other. We account for 19 mixture combinations with five different orderings for each mixture per 500 and 1,000 VM streams; totaling 190 unique VM streams used in our experiments. Fig. 9 illustrates the 85 percent low requesting resource 1,000 VM r1 stream. We show the different VM types on the vertical axis and the arrival sequence of the 1,000 VMs in the stream on the horizontal axis. Stream r1 plot shows that the majority of the VM types correspond to our low resource requests (approximately 85 percent of the VM stream).

6.1.2 Server Configurations

Our experiments consider the heterogeneity of a cloud service provider's back-end infrastructure, i.e., infrastructure composed of multiple servers with various resource characteristics. Very few details have been revealed about the exact server configurations for major cloud service providers' infrastructure. Although, researchers studying the Google workload usage traces have provided fairly accurate results reflecting the number of and resource characteristics for servers within the compute cell from which the trace set was logged [26], [27]. It was determined that approximately 12,477 servers were used in hosting the requests captured in the Google usage trace. Determining the exact capacity specifications for these servers is not possible due to normalization and obfuscation techniques [31] used within the trace set; yet, each trace event within the set expresses a request ratio of CPU, RAM normalized to the largest server configuration (the values of which are not identifiable from the trace set).

Using these ratios, researchers have been able to derive representations for the distribution of machines and their resource characteristics. Liu and Cho [27] categorized these servers into 15 different capacity groups reflecting variations on (CPU, RAM) combinations, where each category reflects a percentage of the 12,477 servers. The capacity groups, identified by a tuple (CPU ratio, RAM ratio), are expressed as combinations of CPU and RAM server capacity ratios relative to the largest server capacities: .25, .50 and 1.00 for CPU; .125, .25, .50, .75 and 1.00 for RAM. For instance, the capacity group (.50, .25) exhibits server capacities that are 50 percent of the CPU resource, and 25 percent of the memory resource of the largest machine, and claims 31 percent of the 12,477 servers, or approximately 3,835 servers. For our experiments, we use the server capacity groups and percentage of group population from Liu and Cho [27], and consider that our largest server has resource capacities of 48 CPUs and 256 GB RAM. We determine all other server capacities relative to these values. We utilize 500 servers for the 500 VM streams and 1,000 servers for the 1,000 VM streams, where their grouping and percentage of population is consistent with the results from Liu and Cho [27]. Fig. 10 illustrates the number of servers per group for the 500 and 1,000 VM streams. For example, we consider 308 servers from the (24, 48) category (i.e., servers with 24 CPUs and 48 GB of RAM). Lastly, we make available the servers with the smallest capacities first throughout our experiments. In sequence, the server capacity groups ordering corresponds to: (12, 64), (24, 32), (24, 64), (24, 128), (24, 196), (24, 256), (48, 128) and (48, 256). We note that only a portion of the server capacity groups were activated in our experiments, but chose 500 and 1,000 servers as the maximum number of servers that can be activated. All the servers considered in the simulation are dedicated to handle only the requests from the VM streams considered in the simulation i.e., they do not handle exogenous workloads.

6.1.3 Modeling Page Sharing

For our experiments, we abstract a subset of the available software from Google Cloud Launcher [32] for the Google VM types. The software categories available to VMs in our experiments are content management, databases, developer tools, infrastructure and operating systems. Each application

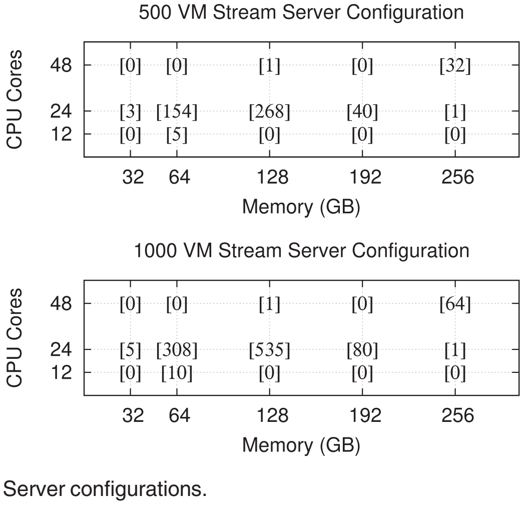


Fig. 10. Server configurations.

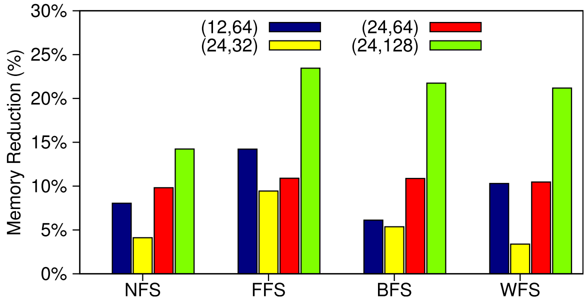


Fig. 11. Average memory reduction: 500 VM stream.

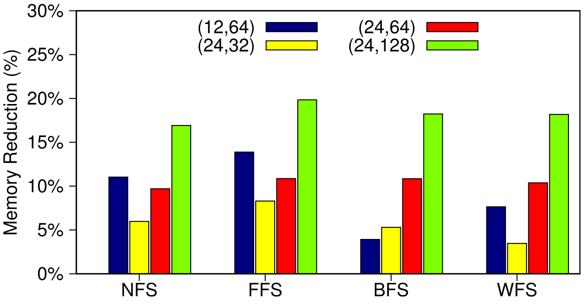


Fig. 12. Average memory reduction: 1,000 VM stream.

software category comprises eight different options, i.e., database software options such as MongoDB, MySQL, Cassandra, Redis, etc., as well as ten operating systems, where four are specific to server versions and six are desktop versions, i.e., operating system software options such as Ubuntu 15.04, Ubuntu Server 14.04 LTS, Windows Server 2008 R2, etc. Previous research on page sharing has uncovered that the majority of page sharing occurs between operating systems [2]. Operating systems and their versions can share a large amount of memory between them; yet, different operating systems may share almost no memory, e.g., collocating VMs which run Windows and Linux OS distributions [2]. Page sharing opportunities can be further identified between server and desktop distributions. In some cases, server distributions do not include desktop packages and the desktop distributions do not include server related packages; but can share kernel resources between them, e.g., Ubuntu 12.04 merges linux-image-server into linux-image-generic.

We model the memory pages requested by applications and OSs using boolean vectors. Each application or OS memory request is characterized by such a vector. The entries of the vectors represent memory pages, where an entry with value 1 signifies that the page represented by that entry is requested, while an entry with value 0 signifies that the page is not requested. Extensive effort has been exerted to build unique vectors reflecting the operating systems and applications memory requirements such that the sharing outcomes are fairly consistent with the results presented by Sindelar et al. [2] and Bazarbayev et al. [33]. For each VM in our experiments we select uniformly at random one operating system and one to four applications to run. We constrain some of the VM types to certain operating system and application combinations, e.g., low request VMs such as n1s1 will not choose OS server distributions since it is unlikely that a user would request a single cpu, low memory VM to host multiple instances. Each server memory pages are also modelled by a boolean vector which is populated with the corresponding entries from the application and OS vectors of the VMs hosted by the server. Once a VM has selected its software combination vectors and a server is identified to host the VM, the VM's vectors are compared to the server's vector to determine the pages that can be shared.

6.2 Analysis of Results

We now compare the performance of our proposed sharing-aware online algorithms from Section 4 against their sharing-oblivious counterparts. Specifically, we show that by using our sharing-aware online algorithms the average number of activated servers is lower, and a substantial memory reduction occurs, which frees up resources for more VMs to be packed. We also analyze some worst-case scenarios for the two sets of algorithms.

In Figs. 11 and 12, we compare the average amount of memory reduction obtained when utilizing the sharing-aware over the sharing-oblivious algorithms for various server capacity categories and for 500 and 1,000 VM streams, respectively. We compare our sharing-aware algorithms, NFS, FFS, BFS, and WFS with sharing-oblivious algorithms, Next-Fit, First-Fit, Best-Fit, and Worst-Fit (WF). The server capacity categories that we sample are identified by a tuple (CPU, RAM). For instance, the server capacity category (24, 64) consists of the server capacity category which includes servers with 24 CPUs and 64 GB RAM. Along the horizontal axis for each sharing-aware algorithm we show the memory reductions for the following server capacity categories: (12, 64), (24, 32), (24, 64) and (24, 128). We note that only in very few instances servers outside of these categories were activated during our experiment. Along the vertical axis are the percentages of memory reduction obtained by our algorithms when compared with their sharing-oblivious counterparts. Quantifying the sharing directly was not straightforward as the sharing-aware and sharing-oblivious algorithms assigned different VMs to different servers. Therefore, we compare the overall memory utilization between each sharing-aware algorithm and its sharing-oblivious counterpart. For both 500 and 1,000 VM streams, all the sharing-aware algorithms tend to exhibit the greatest memory reduction on the server group with the largest amount of memory, i.e., (24, 128). This is because servers that offer more memory can accommodate more VMs as long as CPUs are available. When the number of assigned VMs increases, so does the opportunity to share pages, which leads to more VMs being assigned to the server, if sharing-aware algorithms are utilized. Lastly, when comparing the results for the 500 VM streams and the 1,000 VM streams, we note that the 500 VM stream tends to generate the larger reductions for the (24, 128) case. From our results, the sharing-aware algorithms can reduce the required memory by approximately 25 percent in the best case for the largest server capacity category, i.e., (24, 128), and can reduce the required memory by approximately 5 percent for the worst case in the smallest server capacity category, i.e., (12, 64).

In Fig. 13, we compare the average number of servers required to host the VMs for the 1,000 VM streams over the entire range of low-high requesting resource mixtures. Along the vertical axis are the acronyms for each of the sharing-aware and sharing-oblivious algorithms and along

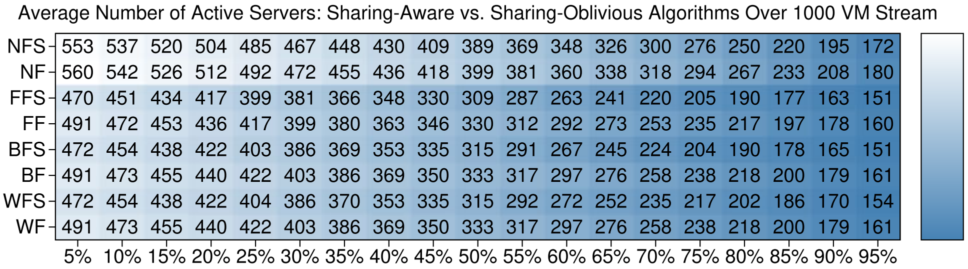


Fig. 13. Average active servers: 1,000 VM streams.

the horizontal axis are the percentages of low resource requesting VMs in the VM stream. The heat map representation has the lightest shade of gray when the highest number of servers are used, e.g., the maximum value is 560 by NF, and has the darkest shade of gray when the lowest number of servers are used, e.g., a minimum value of 151 by BFS. The average number of servers are calculated by aggregating the number of active servers from VM streams r1 through r5 for each requesting resource mixture, dividing by five and calculating the ceiling of the result. The figures show that all the sharing-aware online algorithms activate fewer servers than their respective sharing-oblivious analogues in all mixtures. When comparing the sharing-aware online algorithms among themselves, FFS slightly activates less servers than BFS. WFS tends to over-activate only slightly when compared to BFS in the lower requesting mixtures. As the number of lower requesting VMs outweigh the higher requesting VMs in the VM stream, WFS tends to diverge away from the BFS performance in most cases. Naturally, NFS performs the worst among the sharing-aware algorithms. Moreover, we find that the greatest differences in the 1,000 VM streams occur around the 60 to 85 percent low resource request VM streams which reflects the many low and fewer high resource requests found typically in usage traces from the current cloud service providers.

In Figs. 14 and 15, we show the number of servers activated by the sharing-oblivious algorithms in excess of those activated by our sharing-aware algorithms. We call these servers, the excess servers. In the plots, the sharing-oblivious algorithms have five bars, one for each resource mixtures ranging from 65 to 85 percent in increments of 5 percent. For each of the requesting resource mixtures, we plot the number of excess servers the sharing-oblivious algorithms required over that required by the sharing-aware algorithms. On the horizontal axis, for each sharing-oblivious algorithm we show the server capacity category which was found to exhibit the greatest differences. We note that in Fig. 14, NF exhibited the greatest differences for a different server capacity category, (24, 128), from the other algorithms in the experiment. For the VM 500 stream, NF filled most of the (24, 64) category servers. When comparing NF to NFS in the (24, 64) category, they were nearly identical. The greatest variance between the two algorithms in terms of the greatest number of excess active servers occurred in the next largest server capacity category, (24, 128). In the worst cases for the VM 500 stream, BF for (24, 64) and FF for (24, 64) at resource mixture 70 percent, required 16 to 17 extra servers when compared to our sharing-aware algorithms. The variability of excess servers in the case of BF for (24, 64), is not as pronounced as in the case of FF for (24, 64) among the represented resource mixtures. This implies that the difference in performance between FF and FFS is smaller than in BF and BFS for the worst cases. The results for the VM 1,000 stream are fairly similar in dynamics to the ones for the VM 500 stream, with the largest excesses occurring in the case of FF for (24, 64) with resource mixture 70 percent; accounting for 38 extra servers. From the results of our experiments, we conclude that the sharing-aware algorithms obtain a significant reduction of the number of active servers which implicitly leads to a significant reduction of the costs for the cloud provider.

6.3 Comparison with the Optimal Offline Solution

In order to optimally solve the “offline” Sharing-Aware VM Packing problem (formulated as a multilinear program in Equations (4), (5), (6), (7), (8), and (9)), we use the AMPL [34] mathematical programming framework and an open-source solver, Couenne [35], which employs a branch & bound algorithm for solving mixed integer nonlinear programs in general; which is applicable to solving our multilinear program.

In order to compare the online algorithms against the optimal solution obtained by solving our proposed multilinear program, we designed ten small problem instances with a small number of VM requests and a small number of physical servers and recorded the performance of the

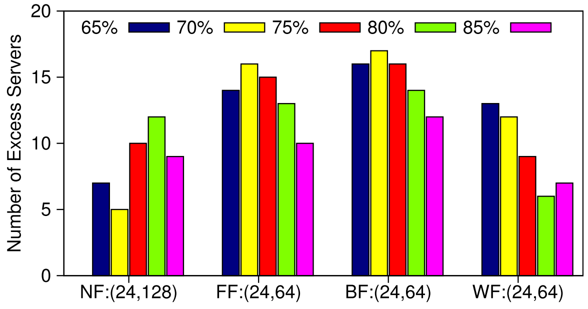


Fig. 14. Excess active servers: 500 VM stream.

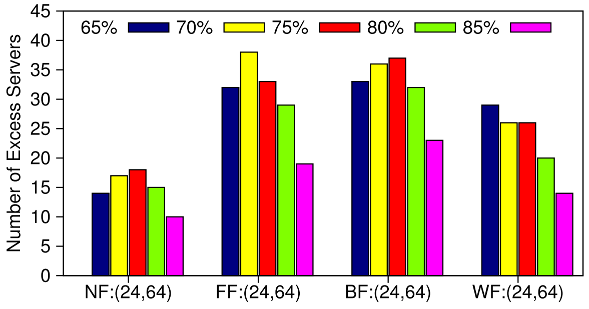


Fig. 15. Excess active servers: 1,000 VM stream.

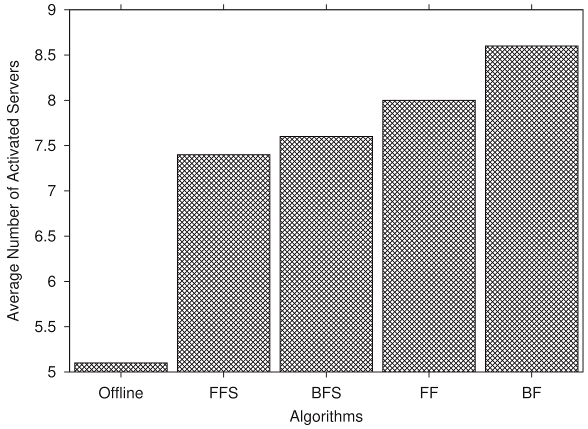


Fig. 16. Offline versus online: Average number of servers.

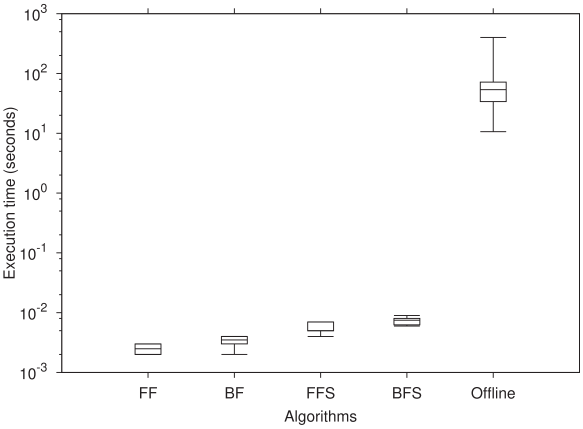


Fig. 17. Offline versus online: Execution times.

algorithms in terms of average number of activated servers and execution time. The environment consisted of 20 VMs and 15 physical servers of varied configurations, i.e., different VMs requesting various numbers of CPU and amounts of memory in addition to physical servers with various CPU and memory capacities. The algorithms we experimented with are, First-Fit-Sharing, Best-Fit-Sharing proposed in Section 4, and implementations of the classic First-Fit and Best-Fit algorithms from the literature which do not consider page-sharing when allocating VMs. We compare the performance obtained by these algorithms against the optimal solution obtained by solving the multilinear program modelling the "offline" Sharing-Aware VM Packing problem (introduced in Section 5).

In Fig. 16, we plot the average number of servers activated by the algorithms. The solution determined by solving our multilinear model using the Couenne solver (denoted in the figure by "Offline") resulted in the least number of activated servers, 4.6 servers on average. FFS and BFS activated 7.4 and 7.6 servers on average, respectively. FF and BF obtained the worst performance, activating 8.1 and 8.6 servers on average, respectively. FFS activated 1.60 times more servers than the "offline" solution, thus obtaining the best performance among the online algorithms, while BF activated 1.86 times more servers than the "offline" solution, the worst performance among the online algorithms.

In Fig. 17, we present a box-and-whiskers plot for the average execution time in seconds for each of the algorithms in our experiment on a log-scale. We also plot the execution time required by the Couenne solver to solve the multilinear program formulation of the "offline" problem. All online algorithms required time in the magnitudes of millisecond, whereas, the time required by the Couenne solver varied heavily ranging from seconds to minutes. This variance in execution times for the solver is due to the different computational work required for the instances we consider here, i.e., solving some of the instances required more linearization and convexification cuts when solving; thereby, increasing the computational time. The fastest algorithm is FF, requiring 2.48 milliseconds followed by BF. The online sharing-aware FFS, required 4.99 milliseconds while BFS required 7.48 milliseconds, all on average. The online sharing-aware algorithms (FFS and BFS) required slightly more time than the sharing-oblivious ones (FF and BF) due to the overhead of identifying page sharing opportunities. Moreover, the First-Fit class of algorithms are faster than the Best-Fit class since the First-Fit class does not require a search among all servers for a "best-fit" when allocating VMs. Even though FF and BF obtained slightly better performance in execution times, on average, compared to FFS and BFS, the differences in milliseconds are negligible when considering activating less servers, on average, which translates to direct savings for cloud service providers. The "offline" solution guarantees the smallest number of servers required for satisfying the VM requests; yet, obtaining it requires an execution time that makes it unfeasible for even slightly larger environments. Therefore, online algorithms such as FFS and BFS are the best alternatives to efficiently solve large problem instances which occur in real-world scenarios.

We believe that the results of the experiments are relevant to real cloud settings because the simulation was performed with input data derived from traces obtained from a real cloud system. A possible threat to validity of our experiments is the fact that the traces may not characterize all possible scenarios that can occur at different current and future cloud providers.

7 Conclusion and Future Work

We designed a family of sharing-aware online algorithms for solving the VM Packing problem. The experimental results showed that our proposed sharing-aware online algorithms activated a smaller average number of servers relative to their sharing-oblivious counterparts, directly reduced the amount of required memory, and thus, the packing of the VMs required fewer servers. Future work involves extending our algorithms to environments with lightweight virtual containers such as Docker containers on the Google Kubernetes infrastructure, and to streaming frameworks. Determining the theoretical performance bounds for the sharing-aware online algorithms is another open avenue for future research.

Acknowledgments

This paper is a revised and extended version of [21] presented at the 8th IEEE International Conference on Cloud Computing (CLOUD'15). This research was supported in part by US National Science Foundation grants DGE-0654014 and CNS-1116787.

REFERENCES

[1] CISCO, "Cisco global cloud index: Forecast and methodology," [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.html

[2] M. Sindelar, R. Sitaraman, and P. Shenoy, "Sharing-aware algorithms for virtual machine colocation," in *Proc. 23rd ACM Symp. Parallelism Algorithms Archit.*, 2011, pp. 367–378.

[3] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali, "Multidimensional bin packing and other related problems: A survey," 2016. [Online]. Available: https://people.math.gatech.edu/tetali/PUBLIS/CKPT.pdf

[4] W. Song, Z. Xiao, Q. Chen, and H. Luo, "Adaptive resource provisioning for the cloud using online bin packing," *IEEE Trans. Comput.*, vol. 63, no. 11, pp. 2647–2660, Nov. 2014.

[5] Y. Li, X. Tang, and W. Cai, "On dynamic bin packing for resource allocation in the cloud," in *Proc. 26th ACM Symp. Parallelism Algorithms Archit.*, 2014, pp. 2–11.

[6] S. Kamali and A. López-Ortiz, "Efficient online strategies for renting servers in the cloud," in *Proc. 41st Int. Conf. Theory Practice Comput. Sci.*, 2015, pp. 277–288.

[7] Y. Azar, I. R. Cohen, S. Kamara, and B. Shepherd, "Tight bounds for online vector bin packing," in *Proc. 45th Annu. ACM Symp. Theory Comput.*, 2013, pp. 961–970.

[8] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," Microsoft research., Cambridge, U.K., 2011.

[9] T. Carli, S. Henriot, J. Cohen, and J. Tomasik, "A packing problem approach to energy-aware load distribution in clouds," *Sustainable Comput.: Informat. Syst.*, vol. 9, pp. 20–32, Mar. 2016.

[10] D. Breitgand and A. Epstein, "Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds," in *Proc. IEEE INFOCOM*, 2012, pp. 2861–2865.

[11] C. Kleineweber, A. Reinefeld, and T. Schütt, "Qos-aware storage virtualization for cloud file systems," in *Proc. 1st ACM Int. Workshop Programmable File Syst.*, 2014, pp. 19–26.

[12] L. Zhao, L. Lu, Z. Jin, and C. Yu, "Online virtual machine placement for increasing cloud provider revenue," *IEEE Trans. Serv. Comput.*, vol. PP, no. 99, 2015.

[13] F. Xu, F. Liu, and H. Jin, "Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud," *IEEE Trans. Comput.*, vol. 65, no. 8, pp. 2470–2483, Aug. 2016.

[14] Z. Xiao, Q. Chen, and H. Luo, "Automatic scaling of internet applications for cloud computing services," *IEEE Trans. Comput.*, vol. 63, no. 5, pp. 1111–1123, May 2014.

[15] F. Hao, M. Kodialam, T. Lakshman, and S. Mukherjee, "Online allocation of virtual machines in a distributed cloud," in *Proc. IEEE INFOCOM*, Apr. 2014, pp. 10–18.

[16] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," in *Proc. USENIX Annu. Tech. Conf.*, 2009, pp. 1–1.

[17] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers," in *Proc. ACM Int. Conf. Virtual Execution Environ.*, 2009, pp. 31–40.

[18] D. Gupta, et al., "Difference engine: Harnessing memory redundancy in virtual machines," *Commun. ACM*, vol. 53, no. 10, pp. 85–93, Oct. 2010.

[19] S. Rampersaud and D. Grosu, "A sharing-aware greedy algorithm for virtual machine maximization," in *Proc. 13th IEEE Int. Symp. Netw. Comput. Appl.*, Aug. 2014, pp. 113–120.

[20] S. Rampersaud and D. Grosu, "A multi-resource sharing-aware approximation algorithm for virtual machine maximization," in *Proc. 3rd IEEE Int. Conf. Cloud Eng.*, Mar. 2015, pp. 266–274.

[21] S. Rampersaud and D. Grosu, "Sharing-aware online algorithms for virtual machine packing in cloud environments," in *Proc. 8th IEEE Int. Conf. Cloud Comput.*, Jun. 2015, pp. 718–725.

[22] Wayne State University HPC grid, 2015. [Online]. Available: http://www.grid.wayne.edu

[23] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: Format + schema," Google Inc., Mountain View, CA, USA, Nov. 2011.

[24] S. Di, D. Kondo, and C. Franck, "Characterizing cloud applications on a Google data center," in *Proc. 42nd Int. Conf. Parallel Process.*, Oct. 2013, pp. 468–473.

[25] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: Insights from Google compute clusters," *SIGMETRICS Performance Eval. Rev.*, vol. 37, no. 4, pp. 34–41, Mar. 2010.

[26] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2012, Art. no. 7.

[27] Z. Liu and S. Cho, "Characterizing machines and workloads on a Google cluster," in *Proc. 8th Int. Workshop Scheduling Res. Manag. Parallel Distrib. Syst.*, Sept. 2012, pp. 397–403.

[28] S. Chen, M. Ghorbani, Y. Wang, P. Bogdan, and M. Pedram, "Trace-based analysis and prediction of cloud computing user behavior using the fractal modeling technique," in *Proc. IEEE Int. Congr. Big Data*, Jun. 2014, pp. 733–739.

[29] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. ISTC-CC-TR-12–101, Apr. 2012.

[30] Google, "Google compute engine pricing," 2016. [Online]. Available: https://cloud.google.com/compute/pricing

[31] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Obfuscatory obscanturism: Making workload traces of commercially-sensitive systems safe to release," in *Proc. 3rd Int. Workshop Cloud Manage.*, Apr. 2012, pp. 1279–1286.

[32] Google, "Cloud laucher," 2015. [Online]. Available: https://cloud.google.com/launcher/explore

[33] S. Bazarbayev, M. Hiltunen, K. Joshi, W. H. Sanders, and R. Schlichting, "Content-based scheduling of virtual machines (VMS) in the cloud," in *Proc. IEEE 33rd Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 93–101.

[34] R. Fourer, D. M. Gay, and B. Kernighan, *AMPL: A Mathematical Programming Language*. Belmont, CA, USA: Duxbury Press/Brooks/Cole Publishing Company, 2003.

[35] P. Belotti, "Couenne, an exact solver for nonconvex minlps," 2015. [Online]. Available: https://projects.coin-or.org/Couenne

Safraz Rampersaud received the BSc degree in mathematics and the MSc degree in applied mathematics focusing on optimization from Wayne State University, Detroit, Michigan. He is currently working towards the PhD degree in computer science at Wayne State University. His research interests include applied mathematics, distributed systems, and virtualization. He is a student member of the IEEE and the SIAM.

Daniel Grosu received the diploma in engineering (automatic control and industrial informatics) from the Technical University of Iași, Romania, in 1994 and the MSc and PhD degrees in computer science from the University of Texas at San Antonio, in 2002 and 2003, respectively. Currently, he is an associate professor in the Department of Computer Science, Wayne State University, Detroit. His research interests include parallel and distributed computing, approximation algorithms, computer security, and topics at the border of computer science, game theory, and economics. He has published more than 100 peer-reviewed papers in the above areas. He has served on the program and steering committees of several international meetings in parallel and distributed computing such as ICDCS, CLOUD, ICPP and NetEcon. He is a senior member of the ACM, the IEEE, and the IEEE Computer Society.