An Approximation Algorithm for Sharing-Aware Virtual Machine Revenue Maximization

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

Abstract—Cloud providers face the challenge of efficiently managing their infrastructure through minimizing resource consumption while allocating service requests such that their revenue is maximized. Solutions addressing this challenge should consider the sharing of memory pages among virtual machines (VMs) and the available capacity of each type of requested resources. We provide such solution by designing a greedy approximation algorithm for solving the sharing-aware virtual machine revenue maximization (SAVMRM) problem. The SAVMRM problem requires determining the set of VMs that can be instantiated on a given server such that the revenue derived from hosting the VMs is maximized. In addition, we model the SAVMRM problem as a multilinear binary program and optimally solve it, while accounting for page sharing and multiple resource constraints. We determine and analyze the approximability properties of our proposed greedy algorithm and evaluate it by performing extensive experiments using Google cluster workload traces. The experimental results show that under various scenarios, our proposed algorithm generates higher revenue than other VM allocation algorithms while achieving significant reduction of allocated memory.

Index Terms—approximation algorithm, multilinear programming, multidimensional knapsack, sharing-aware, virtual machine.

1 INTRODUCTION

Virtualization embodies all the positive characteristics of a technology that minimizes administrative effort, energy consumption, and infrastructure investment. The process of virtualizing applications, servers, networks, etc., as a service benefits consumers and providers alike. Consumers enjoy the fulfillment of their requests and are protected, in a sense, by Service Level Agreements (SLAs). Providers, on the other hand, must ensure that essential resources are thoroughly available and that they generate the highest revenue from providing the services.

Cloud service providers face many challenges concerning the availability of resources to host user specified services. One of the major challenges is how to allocate and manage resources in large scale systems such that the revenue is maximized and the user requests are satisfied. To meet these challenges, several platforms and systems such as Mesos [1], Borg [2], and Kubernetes [3] have been developed and presented in the research literature.

Resource-based sharing, which lies at the heart of virtualization, is a way for service providers to alleviate scarcity, improve utilization and make available an enormous amount of services to users. In this paper, we focus our attention on exploiting the benefits of sharing memory pages among co-located VMs. Sharing at the level of memory pages, page sharing, is a standard memory reclamation technique where the hypervisor removes identical memory pages between the co-located VMs and manages a single page to be shared between them. Hypervisors use an assortment of memory reclamation techniques, e.g., ballooning, compression, swapping, etc., to conserve the memory resource and implement them in different ways. For instance, the Xen hypervisor [4] manages the sharing of pages at the application level, whereas IBM’s PowerVM [5] manages page sharing at the logical partition level. When a server hosts several VM instances that 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 Red Hat Enterprise Linux VM instances are hosted by 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. If service providers could allocate their resources by taking into account the utilization and sharing of those resources, then the potential for higher revenues could be increased due to attracting more consumers to portions of resources which have been freed by sharing.

In this paper, we address the sharing-aware virtual machine revenue maximization (SAVMRM) problem which requires determining the set of VMs that can be instantiated on a given server such that the revenue derived from hosting the VMs is maximized. The solution to the SAVMRM problem takes into account the sharing of memory pages among the VMs and the available capacity of each type of resource requested by the VMs. If memory sharing is not considered, a cloud provider could employ classical multidimensional knapsack algorithms (with the knapsack as the server and the items as the VMs) to solve the virtual machine revenue maximization problem. The classical knapsack algorithms [6] assume that items are distinct and are characterized by dimension and weight. When the items are treated as non-distinct and can be shared, as is the case for SAVMRM, the classic knapsack algorithms produce allocations which generate less revenue than specially designed sharing-aware algorithms. Our focus is on designing such sharing-aware algorithms that solve SAVMRM.
1.1 Our Contribution

We formulate SAVMRM as a multilinear binary program and optimally solve for maximized revenue in the case of small instances. Since solving the multilinear program is not feasible for large scale instances of SAVMRM, we propose and design a greedy approximation algorithm for solving SAVMRM. The algorithm allocates a set of requested VM instances to the server resource such that the revenue of the provider is maximized while the sharing of memory pages and the constraints on the capacity of each type of resource are taken into account. The greedy order employed by the algorithm is based on an efficiency metric that considers multiple types of resources and the page sharing potential among the VMs. We analyze the properties of our proposed greedy algorithm and determine its approximation ratio. Lastly, we investigate the performance of our proposed algorithm by comparing it with the performance of several other greedy allocation algorithms on Google cluster workload traces [7]. To the best of our knowledge, no multi-resource sharing-aware greedy approximation algorithms for solving the SAVMRM problem have been proposed in the research literature to date.

1.2 Related Work

Previous research on the VM resource allocation problem has focused on the optimization of various utility functions under multiple VM resource constraints and on the design of incentive-based mechanisms for VM allocation. Heo et al. [8] employed feedback control techniques to design a resource control system for allocating CPU and memory resources to co-located VMs. Nathuji et al. [9] also employed feedback control techniques to develop a QoS-aware resource allocation framework that minimizes the effect of performance interference among collocated applications in virtual environments. The use of auction-based mechanisms for the VM allocation problem considering multiple resource types has been investigated by several researchers. Zaman and Grosu [10] designed combinatorial auction-based greedy mechanisms for VM provisioning and allocation in clouds. Nejad et al. [11] proposed a family of truthful greedy heuristic mechanisms for dynamic VM provisioning for the auction-based model. Agmon Ben-Yehuda et al. [12] designed Ginseng, a truthful market-driven memory allocation framework, which finds an efficient memory allocation and reallocates physical memory to users according to their valuation. Ghodsi et al. [13] were the first to propose the Dominant Resource Fairness (DRF) allocation policy for multiple types of resources in clusters. DRF policy satisfies a number of desired properties including strategy-proofness, envy-freeness, and Pareto-efficiency. It also incentivizes the sharing of resources by guaranteeing that no request is better off if the resources are equally partitioned among the set of users’ requests. Wang et al. [14] extended the DRF policy concept to multiple heterogeneous server resources in a cloud environment. An alternative notion of fairness for allocating multiple resources, called Bottleneck-Based Fairness (BFF), was proposed by Dolev et al. [15]. They showed that BFF provides incentives for sharing and promotes high utilization of resources. While these allocation methods do take multiple resources into consideration, they do not take into account the benefits of page sharing in their design and implementation.

The majority of research on page sharing focused on developing page sharing systems. Bugnion et al. [16] proposed the transparent page sharing technique for minimizing redundancy and memory overhead. Wood et al. [17] proposed Memory Buddies, a sharing-aware VM memory allocation system which uses the VMWare ESX Server to identify page sharing opportunities. This is achieved by employing hashing algorithms that capture the potential for sharing between multiple VMs. Commercial systems such as VMWare’s ESX Server [18] enable transparent page sharing in addition to other memory reclamation techniques [19]. The open source Xen hypervisor [4], has incorporated page sharing in Versions 4.0 and above for Hardware Virtual Machines (HVM) [20]. Milos et al. et al. [21] designed Satori, a system for sharing memory in virtualized environments. Their design enables efficient detection of short-lived sharing opportunities and incentivizes VMs to share pages. Gupta et al. [22] developed the Difference Engine system which incorporates sub-page sharing, i.e., sharing pages that are nearly identical, and uses compression techniques for pages that are not similar, thereby further reducing the overall memory footprint. Pan et al. [23] proposed the use of a memory de-duplication engine in coordination with a hypervisor to promote the sharing of memory among the co-located VMs. Our work focuses on developing sharing-aware VM allocation algorithms that maximize the revenue obtained from hosting the VMs and take into account page sharing.

To the best of our knowledge, the existing research on the design and analysis of sharing-aware VM allocation algorithms consists of only one paper by Sindelar et al. [24], who introduced and investigated VM packing and maximization problems under hierarchical sharing models. They developed several algorithms to solve these problems assuming two hierarchical sharing models: tree and cluster-tree. In a hierarchical sharing model the sharing occurs in a hierarchical fashion at several levels, OS, OS version, libraries and applications. As an example, in the tree model, the pages shared across all VMs are captured at the root, those shared among the VMs running the same OS version are captured at the next level of the tree, and so on. Our research on the sharing-aware VM maximization problem focuses on the general sharing model which differs from Sindelar et al. [24]. By focusing on the general sharing model, further memory reclamation can occur when VMs request similar operating systems with different overlapping subsets of applications or libraries, which are not captured by hierarchical models. In a previous paper [25], we developed a greedy algorithm for solving the sharing-aware VM maximization problem where only one type of resource, the memory, is considered. Moreover, both contributions [24] and [25] do not consider the allocation of multiple types of resources. In one of our more recent work [26], we proposed sharing-aware algorithms for solving the VM packing problem. VM packing is a different problem from the VM revenue maximization problem considered here since its objective is minimizing the number of servers utilized to serve the requests and not maximizing the revenue from allocating VMs.
1.3 Organization

The rest of the paper is organized as follows. In Section 2, we define the sharing-aware virtual machine revenue maximization problem. In Section 3, we formulate SAVMRM problem as a binary multilinear program. In Section 4, we present our proposed greedy algorithm for solving the SAVMRM problem. In Section 5, we determine the approximation ratio of our proposed greedy algorithm. In Section 6, we describe the experimental setup and investigate the performance of our proposed algorithm by performing extensive experiments on Google Cluster Usage trace data [7]. In Section 7, we summarize our results and present directions for future research.

2 Sharing-Aware VM Revenue Maximization (SAVMRM)

We now present the SAVMRM (Sharing-Aware Virtual Machine Revenue Maximization) problem from the perspective of a service provider.

The allocation of multiple VMs that share a PM resource is controlled by the hypervisor software layer maintained by the service provider. The process of memory reclamation between the physical resource and the requesting VMs is also managed by the hypervisor. Moreover, the hypervisor is the only agent that has the ability to translate pages from PM to VM and/or from VM to VM. We assume the use of an external mechanism, outside of, but in coordination with the hypervisor, capable of managing a library of memory pages, denoted by $\Pi$, required for the services offered by the provider. The use of an external mechanism, outside of, but in coordination with the hypervisor was proposed by Pan et al [23]. Such an approach allows for service flexibility and minimizes any performance degradation resulting from taxing the hypervisor more than it is necessary. The mechanism runs concurrently with the hypervisor on the PM server $\Omega$ that provides the resources. The instantiation of a VM implementing a virtualized service offered by the provider, requires a given number of memory pages. In order to identify the memory pages within $\Pi$, we denote by $\pi_0^i$, the $i$-th memory page in $\Pi$. We assume that $\Pi$ manages a finite number $N$ of pages, i.e., $\Pi = \bigcup_{i=1}^{N} \{\pi_0^i\}$. The notation used in the paper is presented in Table 1.

We assume that there is a set $V$ of $M$ VMs that are candidates for instantiation. We call this set, the set of "offline" VMs. We denote by $V_j$, the VM instance $j$, where $j=1,\ldots,M$, and $V_j \subseteq V$, and by $\pi_j^i$, the $i$-th memory page required by VM $V_j$. The provider allocates and instantiates a subset of VMs, denoted by $V^H$, onto $\Omega$. The hypervisor employs content-based page sharing, which is a technique that detects duplicate copies of a page required by the VMs collocated on the same server, removes the duplicates, and keeps only one copy of the page which is shared among the VMs. One example of a hypervisor implementing such a technique is the VMware ESX server [19]. We assume that the server employs a fingerprinting technique based on hashing or Bloom filters [17] to determine the sharing potential among the VMs considered for allocation (i.e., the amount of memory sharing expressed as number of pages). Our model for the problem is most suitable for cases in which sharing happens at the OS level, but it can also be used in cases where sharing is predictable (e.g., the same database shared by two users running different OSs on two collocated VMs). The allocation should be determined based on how efficient in terms of revenue is to allocate a VM given the availability of PM resources. In general, our model can handle any number of resource types, but for simplicity of presentation and the relevance to practical settings, we specifically consider three main types of resources: (i) memory, where the PM memory capacity is denoted by $C^m$; (ii) virtual CPUs (vCPUs), where the PM vCPU capacity is denoted by $C^s$; and (iii) storage, where the PM storage capacity is denoted by $C^s$. We denote by $R$ the subset of resource types composed of vCPUs (type denoted by $u$) and storage (type denoted by $s$), that is, $R = \{u, s\}$. We do not include the memory resource type in $R$ since it is not treated differently, due to page sharing. Each VM $V_j$ requires a given amount of each resource type as follows: $q_j^m$ amount of memory, $q_j^s$ amount of vCPUs, and $q_j^s$ amount of storage. We assume that the requests for resources from any single VM can be satisfied by the provider (i.e., $q_j^m \leq C^m$, $q_j^s \leq C^s$, and $q_j^s \leq C^s$, for any $j=1,\ldots,M$). We also assume that multiple VM memory pages are in the main memory simultaneously as is the case for multi-core systems. Some environments may not require that all the memory pages be in memory simultaneously and, in those cases, we consider for each VM the minimum number of pages that are required to be in main memory. We now introduce the SAVMRM problem as follows:

**SAVMRM problem:** Given a set of $M$ “offline” VMs $V$, with each VM $V_j$ yielding a revenue $p_j$ upon allocation of the required amount of memory, $q_j^m$, number of vCPUs, $q_j^s$, and amount of storage, $q_j^s$, determine a subset $V^H \subseteq V$ of VMs that can be allocated onto server $\Omega$, considering the PM memory capacity $C^m$, the available number of vCPUs, $C^s$, the PM storage capacity, $C^s$, and the sharing of memory pages, such that the total revenue, $P = \sum_{j:V_j \in V^H} p_j$, obtained by the provider is maximized.

Here, we consider that the revenue $p_j$ derived from
hosting VM $V_j$ is the price paid by the user for using $V_j$. Therefore, the total revenue $P$ obtained by the provider is the sum of the prices paid by the users for each of the requested VM instances hosted by the service provider. Because the majority of the current providers employ fixed-price mechanisms to decide the prices of VM instances, we assume that price for each VM instance is known before allocation and can be used as input to an algorithm that solves SAVMRM.

The formulation of SAVMRM is novel in that it considers the allocation of multiple types of resources and, most importantly, it considers page sharing for the memory resource. If the formulation disregarded page sharing, then the problem could have been reduced to the standard multi-dimensional knapsack problem [6], for which the VMs are the items and the PM is the multi-dimensional knapsack (with dimensions given by the capacities of the multiple resource types). Existing algorithms for solving the multi-dimensional knapsack problem would not be appropriate for solving SAVMRM, leading to revenue loses. SAVMRM represents a new class of multidimensional-knapsack problems with overlapping items. By considering page sharing, more VMs may be allocated to utilize more efficiently the memory resource. Therefore, the service provider may increase its potential for revenue as a result of implementing sharing-aware based allocations. To the best of our knowledge, no algorithms for solving the multi-resource sharing-aware VM allocation problem have been proposed in the literature.

### 3 BMP-SAVMRM: Binary Multilinear Program Formulation

In this section, we propose a multilinear programming formulation of SAVMRM. The objective of the service provider is to instantiate a number of VMs which maximizes the revenue relative to the amount of available resources. Therefore, we formulate the SAVMRM problem as a binary multilinear program (BMP), called BMP-SAVMRM, as follows:

- maximize: $P = \sum_{j : V_j \in \mathcal{V}} p_j x_j$  
  \hspace{2cm} \text{(1)}

subject to: $\sum_{j : V_j \in \mathcal{V}} q^r_j x_j \leq C^r$, $\forall r \in \mathcal{R}$ \hspace{2cm} \text{(2)}

$\sum_{I \in \Pi(\mathcal{V})} (-1)^{|I|+1} \sigma_I \prod_{k \in I} x_k \leq C^m$ \hspace{2cm} \text{(3)}

$x_j \in \{0, 1\}$, $\forall j : V_j \in \mathcal{V}$. \hspace{2cm} \text{(4)}

where $I$ is an element of the power set $\Pi(\mathcal{V})$ of $\mathcal{V}$, and $\sigma_I$ is the sharing parameter that will be defined in the following. The solution to this problem is a boolean decision vector $x \in \{0, 1\}^M$, where $x_j$ corresponds to service provider’s decision to instantiate $V_j$, i.e., $x_j = 1$, if $V_j$ is instantiated, and $x_j = 0$, otherwise. The objective function in Equation (1) corresponds to revenue, $P$, aggregated from the subset of instantiated VMs. The constraint in Equation (2) ensures that the subset of instantiated VMs do not request more resources than the service provider has available, that is, $C^r$, where $r = u$ for vCPUs, and $r = s$ for storage. The constraint in Equation (3) ensures that the subset of instantiated VMs does not request more memory than the service provider has available and takes into account the reclaimed memory through page sharing. Lastly, the constraint in Equation (4) expresses the fact that $x_j$‘s are binary decision variables.

The constraint in Equation (3) requires a more detailed explanation since it captures the sharing of memory pages. To explain it, we consider an example in which four VMs request instantiation onto the server, where the requested resources are given in the second column of Table 2. We consider that only a total of 16 different pages ($\pi_0^2, \pi_0^3, \ldots, \pi_0^{16}$) are going to be requested by these VMs. The pages requested by each of the four VMs are given in Figure 1. For example $V_1$ requests a total of 4 pages (pages marked with hatched boxes in Figure 1, the row corresponding to $V_1$). The vertical bold lines connecting the hatched boxes in the figure mark the pages that are shared. For example, page $\pi_0^2$ is required by $V_1$, $V_2$ and $V_3$, and thus, the hatched boxes corresponding to it in the three VMs are connected with a vertical bold line indicating that $\pi_0^2$ is shared among the three VMs. We now show how the sharing parameter $\sigma_I$ used in constraint (3) is determined. We denote by $\mathcal{P}(\mathcal{V})$ the power set of the set $\mathcal{V}$ of available VMs and by $I$ an element of the power set $\mathcal{V}$. The sharing parameter represents the number of shared pages among the VMs in set $I$. For example for $I = \{1, 2, 3\}$, $\sigma_{123} = 2$, that is, two pages, $\pi_0^2$ and $\pi_0^5$, are shared among the three VMs considered. We calculate the sharing parameter $\sigma_I$ for all the sets $I$ of the power set $\mathcal{P}(\mathcal{V})$ and organize them by the cardinality of $I$ in Table 2. When $|I| = 1$, the sharing parameter $\sigma_I$ represents the amount of memory resource $q^m$ in number of pages requested by $V_j$, that is, $\sigma_j = q^m_j$. 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., those pages which are required to instantiate a subset of VMs, are managed only once in $\Pi$, and are available to be shared among requesting VMs. To calculate the number of unique pages in equation (3) we need to introduce an adjustment parameter, $(-1)^{|I|+1}$, which adjusts the calculation of the number of unique pages according to the cardinality of $I$. By referencing the data in Table 2, we can calculate how many unique pages are required in order to instantiate the entire set of VMs and compare this value to the available
service provider’s memory capacity $C^m$ as follows:

$$
(+1)(\sigma_1 + \sigma_2 + \sigma_3 + \sigma_4) + $$

$$(-1)(\sigma_{12} + \sigma_{13} + \sigma_{14} + \sigma_{23} + \sigma_{24} + \sigma_{34}) + $$

$$(+1)(\sigma_{123} + \sigma_{124} + \sigma_{134} + \sigma_{234}) + (-1)(\sigma_{1234}) \leq C^m \tag{5}
$$

By substituting the values for $\sigma_j$ from Table 2 and performing the calculation above in Equation 5, we arrive at 16 unique pages which is consistent with the number of grey boxes, i.e., those pages required to be managed by $\Pi$ in order to instantiate all four VMs, from Figure 1. In order for the service provider to support the memory requests of all four VMs, they would have to have an available memory capacity which can support the management of at least 16 pages. 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 (3) consists of the product of boolean decision variables, $x_k$, where $k$ is an index corresponding to any VM within the VM subset combination $\mathcal{I}$, on the sharing parameter $\sigma_j$, and the unique page adjustment parameter $(-1)^{|(I_j)|+1}$. BMP-SAVMRM problem is a new and more complex variant of the multidimensional knapsack problem which is strongly $\mathcal{NP}$-hard [6]. Therefore, we infer that BMP-SAVMRM is also strongly $\mathcal{NP}$-hard.

4 Greedy Approximation Algorithm (G-SAVMRM)

In this section, we present the design of our greedy algorithm for solving the SAVMRM problem. Our algorithm orders the candidate VMs according to an efficiency metric which considers the revenue of allocating the VMs, the capacity of the multiple resource types (e.g., memory, vCPU and storage), and the potential for page sharing. Since the focus is on maximizing the revenue of the service provider, the metric should take into account the revenue as the main factor. After each allocation, the efficiency metric is recalculated and the greedy order is adjusted accordingly. Each allocation represents an iteration (denoted by $k$) of the greedy allocation process. The efficiency metric, $E_j^k$, corresponding to VM $V_j$ at iteration $k$ is defined as follows:

$$
E_j^k = \frac{P_j}{\sqrt{\sum_{r \in R} q_{r}^j + \frac{q_{r}^m - q_{r}^j + 1}{C^m}}} \tag{6}
$$

The efficiency metric $E_j^k$ represents the relative value of allocating VM $V_j$ onto $\Omega$ by considering the revenue, the number of resource types requested, and the potential for sharing pages. More specifically, the efficiency metric represents the unit price per square root of normalized resource.

The initial step in the allocation process, at iteration $k = 0$, selects the first VM to be allocated onto $\Omega$, based on the order induced by the efficiency metric. More specifically, it allocates first the VM that has the maximum value for the efficiency metric. The efficiency metric at $k = 0$ for all $V_j \in \mathcal{V}$ depends on the number of shared pages, $s_j^k$, relative to all $V_j \in \mathcal{V}$, since no other VMs have been allocated yet to share pages. At later iterations (i.e., $k > 0$) the efficiency metric considers the potential for sharing among the candidate VM and the VMs that are currently scheduled to be allocated (i.e., VMs that are currently in $\mathcal{V}^H$). An interesting property of our efficiency metric is that as $k$ increases, $s_j^k \leq s_j^{k+1}$, that is, the potential for sharing monotonically increases with $k$, for any $k > 0$.

We now describe the proposed algorithm, called G-SAVMRM, for solving the SAVMRM problem. The algorithm is presented in Algorithm 1. G-SAVMRM consists of two phases distinguished by how the potential for sharing is determined. In the first phase, the potential for page sharing is determined considering the sharing among all the VMs in

<table>
<thead>
<tr>
<th>Algorithm 1 G-SAVMRM: Greedy Algorithm for SAVMRM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Input: Set of offline VM instances ($\mathcal{V}$)</td>
</tr>
<tr>
<td>2: [Phase I: Initial VM Allocation based on the potential for sharing in $\mathcal{V}$]</td>
</tr>
<tr>
<td>3: $[A] \leftarrow 0$</td>
</tr>
<tr>
<td>4: $\mathcal{V}^H \leftarrow \emptyset$</td>
</tr>
<tr>
<td>5: $i, j \leftarrow 0$</td>
</tr>
<tr>
<td>6: for $i = 1, \ldots, N$ do</td>
</tr>
<tr>
<td>7: for all $j : V_j \in \mathcal{V}$ do</td>
</tr>
<tr>
<td>8: if (activePage($\pi_i^{(j)}$)) then</td>
</tr>
<tr>
<td>9: $A^i = A^i + 1$</td>
</tr>
<tr>
<td>10: $i = \text{argmax} {A^i}$</td>
</tr>
<tr>
<td>11: for all $j : V_j \in \mathcal{V}$ do</td>
</tr>
<tr>
<td>12: if (activePage($\pi_i^{(j)}$)) then</td>
</tr>
<tr>
<td>13: $\mathcal{V}^H = \mathcal{V}^H \cup {V_j}$</td>
</tr>
<tr>
<td>14: for $i = 1, \ldots, N$ do</td>
</tr>
<tr>
<td>15: for all $j : V_j \in \mathcal{V}^H$ do</td>
</tr>
<tr>
<td>16: if ($A^i &gt; 1$) and (activePage($\pi_i^{(j)}$)) then</td>
</tr>
<tr>
<td>17: $s_j^0 = s_j^0 + 1$</td>
</tr>
<tr>
<td>18: for all $j : V_j \in \mathcal{V}^H$ do</td>
</tr>
<tr>
<td>19: $E_j^0 = \frac{P_j}{\sqrt{\sum_{r \in R} q_{r}^j + \frac{q_{r}^m - q_{r}^j + 1}{C^m}}}$</td>
</tr>
<tr>
<td>20: $j = \text{argmax} {E_j^0}$</td>
</tr>
<tr>
<td>21: $\mathcal{V}^H = {V_j}$</td>
</tr>
<tr>
<td>22: $\mathcal{V} = \mathcal{V} \setminus {V_j}$</td>
</tr>
<tr>
<td>23: $[C^m, C^u, C^s] = [C^m, C^u, C^s] - [q_j^m, q_j^u, q_j^s]$</td>
</tr>
<tr>
<td>24: for $i = 1, \ldots, N$ do</td>
</tr>
<tr>
<td>25: if (activePage($\pi_i^{(j)}$)) then</td>
</tr>
<tr>
<td>26: allocatePage($\pi_i^{(j)}$)</td>
</tr>
<tr>
<td>27: $k \leftarrow 1$</td>
</tr>
<tr>
<td>28: [Phase II: VM Allocation based on explicit page sharing in $\mathcal{V}^H$]</td>
</tr>
<tr>
<td>29: while ($[C^m, C^u, C^s] &gt; 0$) and ($</td>
</tr>
<tr>
<td>30: $flag \leftarrow 1$</td>
</tr>
<tr>
<td>31: for $i = 1, \ldots, N$ do</td>
</tr>
<tr>
<td>32: for all $j : V_j \in \mathcal{V}$ do</td>
</tr>
<tr>
<td>33: if (activePage($\pi_i^{(j)}$)) and (activePage($\pi_i^{(j)}$)) then</td>
</tr>
<tr>
<td>34: $s_j^k = s_j^k + 1$</td>
</tr>
<tr>
<td>35: for all $j : V_j \in \mathcal{V}$ do</td>
</tr>
<tr>
<td>36: $E_j^k = \frac{P_j}{\sqrt{\sum_{r \in R} q_{r}^j + \frac{q_{r}^m - q_{r}^j + 1}{C^m}}}$</td>
</tr>
<tr>
<td>37: $j = \text{argmax} {E_j^k}$</td>
</tr>
<tr>
<td>38: if ($C^m - (q_j^m - s_j^k) &lt; 0$) or ($C^u - q_j^u &lt; 0$) or ($C^s - q_j^s &lt; 0$) then</td>
</tr>
<tr>
<td>39: $flag \leftarrow 0$</td>
</tr>
<tr>
<td>40: $\mathcal{V} = \mathcal{V} \setminus {V_j}$</td>
</tr>
<tr>
<td>41: if ($flag$) then</td>
</tr>
<tr>
<td>42: $\mathcal{V}^H = \mathcal{V}^H \cup {V_j}$</td>
</tr>
<tr>
<td>43: $\mathcal{V} = \mathcal{V} \setminus {V_j}$</td>
</tr>
<tr>
<td>44: $[C^m, C^u, C^s] = [C^m, C^u, C^s] - [(q_j^m - s_j^k), q_j^u, q_j^s]$</td>
</tr>
<tr>
<td>45: for $i = 1, \ldots, N$ do</td>
</tr>
<tr>
<td>46: if (activePage($\pi_i^{(j)}$)) then</td>
</tr>
<tr>
<td>47: allocatePage($\pi_i^{(j)}$)</td>
</tr>
<tr>
<td>48: $P = P + p_j$</td>
</tr>
<tr>
<td>49: $k = k + 1$</td>
</tr>
<tr>
<td>50: $\Omega \leftarrow \mathcal{V}^H$</td>
</tr>
<tr>
<td>51: exit</td>
</tr>
</tbody>
</table>
the offline set of VMs, $\mathcal{V}$ (Lines 3-27). In the second phase, the potential for sharing is determined by considering the sharing among the candidate VM and the VMs that are currently scheduled to be allocated onto $\Omega$ (Lines 28 - 49).

The input to G-SAVMRM is a set of "offline" VMs, $\mathcal{V}$. First, G-SAVMRM initializes the shared page counter array, $A$ (Line 3), the subset of allocated VMs, $\mathcal{V}^H_i$ (Line 4), and the indices used for selecting VMs (Line 5). The shared page counter array $A$ is used to determine the potential for sharing pages among the VMs in $\mathcal{V}$, that is, entry $A_i$ is the number of occurrences of page $\pi_j^i$ requested by the VMs in $\mathcal{V}$. The pages requested by the VMs in $\mathcal{V}$ are identified and $A$ is updated accordingly (Lines 6 - 9). Function $\text{activePage}()$ (Line 8), determines whether memory page $\pi_j^i$ from VM $V_j$ is requested. If $\pi_j^i$ is requested, then $\text{activePage}()$ returns 1, otherwise it returns 0. The $\text{activePage}()$ function uses information from a pre-processing stage in which the server determines the memory fingerprints of the VMs. The cloud provider could implement a memory fingerprinting technique similar to the one presented by Wood et al. [17]. Then, the $i$-th memory page that is requested the most, is selected, and every $V_j$ which requests the $i$-th memory page is included in the VM subset $\mathcal{V}^H_i$ (Lines 10-13). The next task is to calculate the number of shared pages for each $V_j \in \mathcal{V}^H_i$. If there are memory pages shared by at least two VMs, (i.e., $A_i > 1$), and $V_j$ requests the $i$-th memory page, then the VM shared page counter at the initial iteration $s_j^i$ is updated (Lines 14-17). Then, our proposed efficiency metric is calculated for each $V_j \in \mathcal{V}^H_i$ (Lines 18 and 19), where the VM corresponding to the highest efficiency value is identified by index $j$ (Line 20). $V_j^*$ is then allocated to $\mathcal{V}^H_i$ (Line 21) and removed from $\mathcal{V}$ (Line 22). The three PM resource capacities are then reduced by the amount of resource requests from $V_j$ (Line 23). Note, we do not add the shared pages $s_j^i$ back into the PM resource capacity $C^m$ since at $k = 0$, $V_j^*$ is the first VM allocated and only has a potential for sharing pages with other VMs to be allocated later. Any memory pages which are deemed active according to $\text{activePage}()$ are then allocated onto PM server $\Omega$ through $\text{allocatePage}()$ (Lines 24 - 26). After the initial allocation according to the potential for sharing, $k$ is updated to 1 (Line 27).

The second phase starts by checking the availability of resources of each type on the server $\Omega$ (Line 29). A variable flag is set to 1 (Line 30) which indicates a valid VM allocation upon identifying the VM that is allocated later in the algorithm. The major difference between the first phase that considers potential sharing and the second phase is that in the second phase the sharing is determined relative to the VMs that are already scheduled to be allocated on the server. The algorithm identifies the pages which can be shared relative to memory pages already allocated, for every page requested in each remaining $V_j \in \mathcal{V}$. For those memory pages requested by $V_j \in \mathcal{V}$ which are already allocated, the shared page counter $s_j^i$ is updated (Lines 31 - 34). Next, the efficiency metric is calculated for all $V_j \in \mathcal{V}$ (Lines 35 and 36) and the VM with the highest efficiency value is identified by the index $j$ (Line 37). Prior to allocating $V_j^*$, a check must determine if the allocation will fully deplete any of the multiple types of resources provided by the PM (Line 38). If any of those resources are fully depleted, the flag variable is set to 0 (Line 39) and $V_j^*$ is removed from $\mathcal{V}$ (Line 40) since it cannot be allocated. If flag is still 1, then $V_j^*$ is stored in $\mathcal{V}^H$ and removed from $\mathcal{V}$ (Lines 42 and 43). The capacities of each of the multiple resources of the PM are then reduced according to the resources requested by $V_j^*$ (Line 44), that is, the PM memory capacity $C^m$ is reduced by $q_{j}^{m}$ and $s_{j}^{m}$ pages are added back to the capacity because those pages are already allocated and do not count against $C^m$ since they will be shared as a result of a previous VM allocation. Any new pages requested by $V_j^*$, if they are not already allocated, are then allocated by calling $\text{allocatePage}()$ (Lines 45 - 47). Next, the revenue $p_j^m$ from allocation of $V_j \in \mathcal{V}^H$ is accumulated into $P_i$ (Lines 48).

Lastly, the iteration count $k$ is incremented (Line 49) and the process continues until either one of the PM resources are fully depleted, or until $\mathcal{V} = \emptyset$, and then the VMs in the set $\mathcal{V}^H$ are instantiated on the PM server $\Omega$ (Line 50).

We present an example to show how G-SAVMRM works. We consider a single server with resource capacities: vCPU, $C^v = 6$ vCPUs; storage, $C^s = 8$ GB; and memory, $C^m = 16$ pages. We consider four VM requests characterized by the parameters given in Table 3 (derived revenue, $p_j$; vCPU request, $q_{j}^{v}$; storage request, $q_{j}^{s}$; and memory request, $q_{j}^{m}$, translated into number of pages). Figure 2 along with Table 3 show the details of each iteration $k$ of G-SAVMRM. In Figure 2, page $\pi_j^i$, $(i = 1, \ldots, 16$ and $j = 1, \ldots, 4)$, is identified by a gray block, if it is requested by $V_j$, or by an empty block, if the page is not requested by $V_j$. The number of gray blocks per VM corresponds to the total number of pages translated from the requested amount of memory, $q_j^m$.

The first column in both Figure 2 and Table 3 corresponds to the first phase of G-SAVMRM. The array $A$ in Figure 2, stores these values per page and only the values where $A_i > 1$ indicate potential for page sharing. The maximum value in $A$ corresponds to the page that is shared the most among all the pages in $\mathcal{V}$. Based on the parameters of our example, $\pi_0^i$, where the max count is identified in bold in array $A$ (Figure 2), would be shared the most and all VMs which request $\pi_0^i$ would be considered candidates for instantiation in the first phase of G-SAVMRM. The highest efficiency metric, 1.6040, is associated with $V_3$ and therefore, all pages requested by $V_3$ are activated in $\mathcal{V}$ and added to subset $\mathcal{V}^H$. The activated pages under provider management in $\mathcal{V}$ are marked by gray boxes which are connected with vertical lines to the pages required by $V_3$. Lastly, the server resource capacities are reduced as follows: vCPUs, $C^v = 4$, storage, $C^s = 6$, and memory, $C^m = 9$, according to $V_3$ resource requests. The service provider then updates the derived revenue from instantiating $V_3$, amounting to 1.35.

The next iteration of G-SAVMRM, corresponding to the first iteration of the greedy phase ($k = 1$), is illustrated in the second column of both Figure 2 and Table 3. In this iteration, G-SAVMRM finds identical, requested pages between VMs and the active pages within $\mathcal{V}$. The highest efficiency metric, 1.1514, is associated with $V_4$. Following the instantiation of $V_4$, the capacities are: vCPUs, $C^v = 0$, and storage, $C^s = 4$. For the server memory resource, $V_4$ consists of 14 pages, where 5 pages are shared with active pages in $\mathcal{V}$ (i.e., $\pi_0^i$, $\pi_5^i$, $\pi_7^i$, $\pi_9^i$, and $\pi_10^i$); thereby, the server memory resource only
needs to account for $\pi^1_0, \pi^4_0, \pi^6_0, \pi^8_0,$ and $\pi^{11}_0$ to $\pi^{14}_0$, in $\Pi$, which are required to instantiate $V_4$. Lastly, the revenue is updated to 3.15. At this iteration, G-SAVMRM stops because the memory resource has been exhausted and no further VM instantiation is possible. The total revenue obtained by G-SAVMRM for this example is 3.15, which is less than 4.05, the optimal revenue obtained by solving the BMP-SAVMRM. In the next section, we determine the approximation ratio for G-SAVMRM which will characterize how far the solution obtained by G-SAVMRM can be from the optimal solution.

### 5 G-SAVMRM Properties

In this section, we investigate the approximability properties of our proposed algorithm. We determine the approximation ratio of G-SAVMRM by considering a worst possible server setup, $\Omega^W$, for the SAVMRM problem. We consider $\Omega^W$ consisting of three resource types: memory, vCPU, and storage. We assume that $\Omega^W$ has a small capacity for the memory resource, a large capacity for the vCPU resource, and a large capacity for the storage resource.

Let $\mathcal{V}^W$ denote a worst-case instance of the SAVMRM problem, where VM $V_j \in \mathcal{V}^W$ does not share any pages with the other VMs in $\mathcal{V}^W$. Then, let at least one VM $V_j \in \mathcal{V}^W$ be comprised of pages which form the complement set of VM $V_j$’s pages. In addition, let the remaining VMs in $\mathcal{V}^W$ be comprised of either a subset of pages in VM $V_j$ or be equivalent to VM $V_j$. In either case, the remaining VMs would be allocated onto $\Omega^W$ if $V_j$ were to be allocated first since they all share the same memory pages and would not reduce the memory capacity of $\Omega^W$.

We investigate this instance on server $\Omega^W$ with a limited memory capacity such that either VM $V_j$ or VM $V_{j'}$ can be allocated, but not both, while not depleting the vCPU and storage capacities. If VM $V_j$ is allocated, then all remaining VMs in $\mathcal{V}^W \setminus \{V_j\}$, will be allocated as well due to page sharing and large capacities for both vCPU and storage. Else, VM $V_j$ is allocated and utilizes the memory capacity enough to not allow any other VM from $\mathcal{V}^W$ to be allocated. We assume that $\Omega^W$ has a large number of vCPUs available and a large storage capacity that allows a set of $M$ VMs to be allocated. If either the vCPU or storage capacities were small, then only a subset of VMs may be allocated due to vCPU or storage constraints in addition to the memory capacity.

The worst case instance $\mathcal{V}^W$ satisfies two additional conditions: (i) the sum of the revenues obtained by allocating all the other instances except $V_j$, should be greater than the revenue obtained by allocating $V_j$ (i.e., $\sum_{j:V_j \in \mathcal{V}^W \setminus \{V_j\}} p_j \geq p_j$), and (ii) the efficiency metric of instance $V_j$ should be greater than the efficiency metrics of all the other instances (i.e., $E^0_j > E^0_{j'}, \forall j' \neq j$). These conditions will result in G-SAVMRM selecting only VM $V_j$ for allocation and obtaining lower revenue than the optimal solution which allocates all the other VMs.

Our design of $\mathcal{V}^W$ and $\Omega^W$ will exhibit the greatest differences between the optimal revenue obtained by an optimal algorithm (e.g., exhaustive search) and the revenue generated from our greedy G-SAVMRM algorithm. If the memory capacity was larger than our proposed setup, then the revenue generated from G-SAVMRM could be closer to the optimal revenue generated by the optimal algorithm. Therefore, a server that has low memory capacity, high vCPU capacity, high storage capacity, and where page sharing occurs, represents the worst case scenario. In the following, we determine the approximation ratio for G-SAVMRM based on the worst case instance $\mathcal{V}^W$ and server $\Omega^W$.

**Theorem 5.1.** The approximation ratio of G-SAVMRM is $M \sqrt{C_{\max}([R] + 1)}$, where $C_{\max} = \max\{C^m, C^u, C^s\}$.

**Proof.** Let the revenue obtained from an optimal solution be denoted by $P^*$, and the optimal set of VMs which generates $P^*$ from $\mathcal{V}^W$ be denoted by $V_{OPT}^W$, $V_{OPT}^W \subseteq \mathcal{V}^W$, where $P^* = \sum_{j:V_j \in V_{OPT}^W} p_j$ under server resource $\Omega^W$.
Let the revenue obtained by G-SAVMRM be denoted by \( P \), and the set of VMs which generate \( P \) from \( \mathcal{V}^W \) be denoted by \( \mathcal{V}^W_{GRD} \), \( \mathcal{V}^W_{GRD} \subset \mathcal{V}^W \), where \( P = \sum_{j:V_j \in \mathcal{V}^W_{GRD}} p_j \) under server resource \( \Omega^W \).

Assume at \( k = 0 \), VM \( V_j \) is allocated by G-SAVMRM onto \( \Omega^W \); admitting the relationship \( E_j^0 < E_j^0 \) for any \( j \neq j \). Since VM \( V_j \) does not share pages with VMs in \( \mathcal{V}^W \), \( \sum_{j=0}^k \), and memory capacity of \( \Omega^W \), no other VM allocations can be performed and \( k \) stops at 0. Since \( P = \sum_{j:V_j \in \mathcal{V}^W_{GRD}, P_j} \), therefore \( P = p_j \).

Suppose through an exhaustive search, the optimal revenue value \( P^* \) is calculated whereby VM \( V_j \) is allocated first onto \( \Omega^W \). Since every remaining VM in \( \mathcal{V}^W \) is comprised of a subset of pages in VM \( V_j \), not including VM \( V_j \), then the exhaust search allocates all remaining VMs onto \( \Omega^W \) without depleting the vCPU and storage capacities. Therefore, the optimal value \( P^* = \sum_{j:V_j \in \mathcal{V}^W_{GRD}, P_j} p_j \).

In order to determine the approximation ratio for this instance of SAVMRM, we show that \( P^* \leq P_\alpha \), where \( \alpha \) is the multiplicative factor that will give the approximation ratio of G-SAVMRM. Therefore,

\[
P^* = \sum_{j:V_j \in \mathcal{V}^W_{GRD}, P_j} \frac{P_j}{p_j}
\]

By substituting \( p_j \) from Eq. 8, we obtain

\[
\frac{P^*}{P} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_j\}} \frac{q_j^m - s_j^k + 1}{C_m} < \frac{P^*}{P} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_j\}} \frac{q_j^m - s_j^k + 1}{C_m}
\]

Since

\[
\frac{\sum_{j:V_j \in \mathcal{V}^W \setminus \{V_j\}} q_j^m - s_j^k + 1}{C_m} \leq \frac{1}{C_{max}}
\]

where \( C_{max} = \max\{C_m, C_n, C^a\} \), we obtain

\[
\frac{\sum_{j:V_j \in \mathcal{V}^W \setminus \{V_j\}} q_j^m - s_j^k + 1}{C_m} \leq \frac{1}{C_{max}}
\]

Thus,

\[
\frac{P^*}{P} \leq \sqrt{C_{max} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_j\}} (\sqrt{|R|} + 1)}
\]

Therefore, \( P^* \) is bounded by \( \alpha = M \sqrt{C_{max} (|R| + 1)} \), which results in an approximation ratio of \( M \sqrt{C_{max} (|R| + 1)} \) for the G-SAVMRM algorithm.

The approximation ratio depends on the characteristics of the system on which the algorithm is deployed, that is, it depends on three parameters, the number of VMs requested \( M \), the maximum capacity of the available resources \( C_{max} \), and the number of resource types \( C_{max} \) provided by the cloud provider.

We now investigate the time complexity of G-SAVMRM. The running time is dominated by the second phase, the greedy phase. The while-loop (Line 29) is executed a maximum of \( M - 1 \) times since one VM has already been inserted into \( \mathcal{V}^H \) and there exists instances where \( \mathcal{V}^H \subset \mathcal{V} \). Within the while-loop, the running time is dominated by the search and calculation of shared pages between the VMs in \( \mathcal{V} \) and the active pages on \( \Omega \) (Lines 31 - 34). The search and calculation are executed a maximum of \( M - 1 \) times, corresponding to the possible number of VMs at \( k = 1 \), by the number of active pages to search on \( \Omega \), thus the running time is \( O(N(M - 1)) \). Then, the running time for the entire greedy phase is \( O(N(M - 1)^2) \). Thus, G-SAVMRM has an asymptotic running time of \( O(NM^2) \) which is linear in the total number of pages and quadratic in the number of VM requests.

### 6 Experimental Results

In this section, we describe the experimental setup and perform extensive experiments investigating the performance of G-SAVMRM against other VM revenue maximization algorithms.
6.1 Experimental Setup

The software used in the experiments and trace processing is implemented in C++ and run on a 3.10 GHz Intel Core i7 quad-core processor within a Mac operating system environment.

6.1.1 Utilizing Google Cluster Usage Traces

For our experiments, we used the cluster usage traces from workloads running on Google compute cells [7]. A compute cell is a set of machines within a single cluster, supported by a common cluster-management system. We used the publicly available ClusterData2011_1 data set which reports the activity for a 12k-machine cell during May 2011 from Google Cloud Storage [27]. While the data set is publicly available, extensive effort has been exerted in order to obfuscate information by normalizing, hashing and rescaling the data to not explicitly reveal actual information such as users, applications, server specifications, etc. [28]. As a result, research focusing on characterizing the many facets of the data set such as user behavior [29] and workloads [30], have already been thoroughly presented in the literature. The ClusterData2011_1 data set consists of tables grouped according to machines, jobs and tasks, which are further grouped into categories such as attributes, constraints, events, and usage. We focus on a single table, task_events, which provides normalized data of relevant requests for CPU, memory, and local disk resources. In order to generate a data set from task_events which is meaningful to our investigation, we employed a filtering strategy as follows:

(i) Eliminate traces which are missing information, i.e., acquire trace if missing info = 0.

(ii) Eliminate traces where task events are evicted, failed, killed, or lost, and eliminate any traces with update events, i.e., acquire trace if event type = 1.

(iii) Eliminate traces where tasks have a low scheduling class. The scheduling class field characterizes how sensitive a task is to latency. Since our investigation focuses on revenue maximization, we only concern ourselves with those tasks which are classified as high; reflecting a service to revenue generating user requests [7]. Due to obfuscation, we do not know exactly that every trace with a high scheduling class is a task with a high priority, which will be last to be evicted in the case of over-provisioning the machine resource, i.e., acquire trace if priority ≥ 8 and priority ≠ 10.

(iv) Eliminate any traces that allow for tasks within a job to be processed on different machines. Since our investigation only considers a single machine resource, we only consider traces where the job consists of tasks that must be allocated to a single machine, i.e., acquire trace if different machines restriction = 0.

While the trace usage events in ClusterData-2011-1 supply a considerable amount of information, our focus on revenue maximization requires each trace in our experiments to be augmented with a revenue value which a service provider would receive following the instantiation of a VM request. Since the trace usage data does not reveal the revenue acquired from hosting revenue generating user requests, we fit each trace request in our experiments to a priced Google Compute Engine VM Instance [31], relative to its normalized memory and cpu request values and server capacity values. The characteristics of Google Compute Engine VM instances are given in Table 4. Due to both data normalization and obfuscation techniques used in ClusterData-2011-1, identifying the exact server resources and extracting its technical specification is not possible solely on the data provided. Therefore, our experiments are conducted by simulating the resource capacities of a Lenovo Flex System x880 X6 Compute Node (Intel Xeon E7-8890 v2) PM server with the following resource specifications: 120 cores (8 chips × 15 cores per chip); 2 TB memory (128 × 16 GB DDR3) and 9.6 TB disk space (24 × 400 GB SSD). The Lenovo Flex System x880 X6 Compute Node is the highest rated server according to the SPECvirt_sc2013 benchmark which evaluates datacenter server performance and virtualized server consolidation conducted by the Standard Performance Evaluation Corporation c⃝ (SPEC), released in the 2nd quarter of 2015 [32].

Each VM instance used in our experiments reports its characteristics: memory, vCPU, storage, and price. In order to fit each VM request, t, from the trace usage set to a Google VM Instance, we first calculate the product of the normalized memory and CPU resource request values in the filtered data and the server’s memory and vCPU capacities, Cm and Cu respectively. The resulting products represent a specific amount of memory (in GB), denoted by tm, and a number of vCPUs, denoted by tu, relative to the server specifications. For every Google Compute Engine VM Instance g, y ∈ {1...16}, we denote its memory requirement by gy and its vCPU requirement by gu. We calculate ̃y, the index of the Google Compute Engine VM Instance that minimizes the 2-norm relative error between t’s requested amount of memory and vCPUs and gy’s requirements, as follows:

$$\tilde{y} = \arg\min_y \sqrt{\left(\frac{t^m - g_{y^m}}{C^m}\right)^2 + \left(\frac{t^u - g_{y^u}}{C^u}\right)^2}$$  (19)

Then, we map the trace request t to the Google Compute Engine VM Instance gy, that is, to the Google VM instance that fits the requested resources the best. Lastly, the storage

<table>
<thead>
<tr>
<th>[size]</th>
<th>n1-standard-[size]</th>
<th>n1-highmem-[size]</th>
<th>n1-highcpu-[size]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>[2]</td>
<td>[4]</td>
<td>[8]</td>
</tr>
<tr>
<td>Memory (GB)</td>
<td>3.75 7.50 15 30 60 120</td>
<td>13 26 52 104 208</td>
<td>1.80 3.60 7.20 14.40 28.80</td>
</tr>
<tr>
<td>vCPU</td>
<td>1 2 4 8 16 32</td>
<td>2 4 8 16 32</td>
<td>2 4 8 16 32</td>
</tr>
<tr>
<td>Price ($/hour)</td>
<td>0.050 0.100 0.200 0.400 0.800 1.600</td>
<td>0.126 0.252 0.504 1.008 2.016</td>
<td>0.760 0.152 0.304 0.608 1.216</td>
</tr>
</tbody>
</table>

Table 4: Google Compute Engine VM Instance Types.
usage values are not fully captured within ClusterData-2011-1 traces due to Google treating storage as a separate service from Google Compute Engine [7]. Therefore, we do not use the VM storage request information within our experiments.

6.1.2 Modeling Page Sharing

Leveraging page sharing to maximize revenue, requires the identification of applications and the operating system used by the instantiated VMs, which are not revealed within the ClusterData-2011-1 trace set. Although, each task event operates within its own container [7], we treat each task event as a VM instance under various operating system software.

For our experiments, we consider the page content similarity percentages among OSs reported by Bazarbayev et al [33]. These percentages are given in Figure 3. We consider fixed page sharing percentages for every possible OS combination considered in our experiments. Each entry in the sharing table represents a page sharing percentage value defined as the percentage of the OS memory of the already hosted VM that can be shared by the OS of the newly arrived VM. Each VM in our experiment will select uniformly at random one of three versions of three OSs: CentOS Server x86_64 (C6.0-6.2); Windows Server 64bit (W64b), Windows Server R2 (WR2), Windows Server R2 SQL (WR2S); and Red Hat Enterprise Linux x86_64 (R6.0-6.2). With each OS and OS versions considered here we associate a binary array which specifies the pages required by the OS. Each possible page has an entry in these arrays which is 1 if the page is requested by the OS, and 0, otherwise. We create these binary arrays such that the amounts of shared pages correspond to the sharing percentages presented in Figure 3.

To show how page sharing works in our experiment, if a server has a VM which has selected CentOS server 6.0 (C6.0) as its OS and another VM which is attempting to be collocated on the same server has selected CentOS server 6.2 (C6.2), then the VM which selected C6.0 will share 28% of C6.2’s OS pages. Since C6.0’s OS image size is .77 GB and the amount of memory that is shared between C6.0 and C6.2 is still 220 MB, then the sharing percentage is calculated as \( \frac{220MB}{0.77GB} = 28\% \). The amount of memory sharing and image sizes are those determined by Bazarbayev et. al [33]. On the other hand, if a server has a VM which has selected CentOS server 6.2 (C6.2) as its OS and another VM which is attempting to be collocated on the same server has selected CentOS server 6.0 (C6.0), then the VM which selected C6.2 will share 11% of C6.0’s OS pages. Since C6.2’s OS image size is 1.96 GB and the amount of memory that is shared between C6.0 and C6.2 is still 220 MB, then the sharing percentage is calculated as \( \frac{220MB}{1.96GB} = 11\% \). As can be seen from the above example, C6.0 and C6.2 share the same amount of memory in both cases, but the percentages are different because they are calculated relative to different bases, C6.2 in the first case and C6.0 in the second case. This asymmetry in terms of sharing percentages also occurs for other OS combinations given in Figure 3. Furthermore, we consider that CentOS and Red Hat Enterprise Linux (RHEL) versions sharing percentages in Figure 3. Lastly, cases exist in which two operating systems will share very little memory, as was found by Sindelar et al [24] for Windows and Linux OS distributions. Since the sharing is marginal in these cases, we assign a sharing percentage value of 0 when this occurs, i.e., a VM operating under Windows Server R2 (WR2) and a VM operating Red Hat Enterprise Linux 6.0 (R6.0) which are collocated on the same server will not share any OS pages between them.

6.1.3 Comparing G-SAVMRM

We compare G-SAVMRM with other algorithms for VM revenue maximization. Since such algorithms are not available in the literature, we decided to design several types of greedy algorithms that use various greedy ordering methods based on single parameters such as profit, number of shared pages, vCPUs, and amount of memory, and use them in our experiments. Thus, we compare G-SAVMRM with four algorithms that are variants of G-SAVMRM: P-DO which allocates the VM requests in decreasing order of their revenue (this corresponds to G-SAVMRM with \( E_k^j = p_j \)); SP-DO which allocates the VM requests in decreasing order of the number of shared pages (this corresponds to G-SAVMRM where \( E_k^j = s_j^f \) is calculated with \( p_j = 1 \), and the first term under the square root equal to 0); C-IO which allocates the VM requests in increasing order of the number of requested vCPUs (this corresponds to G-SAVMRM where \( E_k^j = s_j^f \) is calculated with \( p_j = 1 \), and the last term under the square root equal to 0); and, M-IO which allocates the VM requests in decreasing order of the amount of requested memory (this corresponds to G-SAVMRM where \( E_k^j = s_j^f \) is computed with \( p_j = 1 \), the first term under the square root equal to 0, and \( s_j^f = 0 \)). We also compare G-SAVMRM with four other greedy algorithms that are not variants of G-SAVMRM: C-DO which allocates the VM requests in decreasing order of the number of requested vCPUs; M-DO which allocates the VM requests in decreasing order of the amount of requested memory; DR-DO, which allocates VMs in decreasing order of the dominant resource request; and, DR-IO, which allocates VMs in increasing order of the dominant resource request. DR-IO and DR-DO are dynamic in the sense that their greedy order is dependent on the largest (dominant), normalized resource value given dynamic provisioning of the PM server resource. Lastly,
we compare G-SAVMRM with our previously developed greedy algorithm, G-SAVMM [25], which makes the allocation decisions solely based on the memory resource. This algorithm can be viewed as a special case of G-SAVMRM in which the efficiency metric is

\[ E_j^k = \frac{\psi_j}{\sqrt{q_j^k - s_j^k + 1}}. \]

The algorithms used in our experiments are presented in Table 5. Each greedy algorithm used for comparison is designed to benefit from page sharing at the hypervisor level (i.e., once the allocation is decided by the algorithms, the hypervisor identifies the pages that are shared among the allocated VMs), but they do not consider the sharing of pages in determining the allocation. There are two exceptions, SP-DO, which uses the number of shared pages to establish the greedy ordering, and thus, the allocation, and G-SAVMM which allocates VMs based on the scarcity of the memory resource and the sharing of memory pages among VMs already allocated to the server. A key difference between SP-DO and G-SAVMM is that SP-DO requires the VMs to be sorted in decreasing order based on how memory pages are shared, while G-SAVMM calculates an efficiency metric for all VMs and greedily allocates them according to the metric.

We also compare G-SAVMM performance against the optimal solution obtained by solving the Binary Multilinear Program (BMP-SAVMRM) presented in Section 3. In order to solve BMP-SAVMRM optimally, we use an open-source solver, Couenne [34] hosted by the NEOS Server [35] a free internet-based service for solving optimization problems.

### 6.2 Analysis of Results

We now compare the performance of G-SAVMRM against the other greedy algorithms considered in our experiments. Our experiments consist of using the filtered Google cluster-usage trace events according to our strategy described in Section 6.1. We use a portion of the transformed trace events which consists of 15,000 events. We partition our trace into windows, i.e., uniform interval partitions of the entire trace. Each algorithm in our experiments will operate and allocate VM requests to a server within a window according to its design and available server resources. Our experiments consider three types of windows: W30, W50 and W100, where a server will attempt to allocate a portion of the VMs. For example, in the case of W50, the trace is partitioned into 50 VM requests per window and each window is assigned a single server (300 servers total in W50). For W30 and W100, the trace is divided into sets of 30 and 100 VM requests, respectively. When at least one of the server resources has been exhausted in the current window, the server is considered closed and any VM which remains unallocated in the current window is rejected. Then, the next window becomes available and a new server comes online ready for each algorithm to undergo its allocation process until all 15,000 events have been considered.

In Figure 4, we show the average aggregated revenue ratios obtained by the algorithms using our trace. The revenue ratio is defined as an algorithm’s obtained revenue per window, over the revenue generated by the best performing algorithm within the same window. The revenue ratios indicate each algorithm’s performance proximity to the maximum revenue attained for that window within the window sequence. These revenue ratios will never be larger than 1 for any of the algorithms during any window within the window sequence. By aggregating these ratios and then dividing by the number of windows in the sequence (e.g., for W50, there will be 300 windows within the window sequence), we calculate the average aggregated revenue ratio, which provides insight into which algorithm exhibits the best performance in terms of revenue.

G-SAVMRM obtains the highest average aggregated revenue ratio for all three window intervals (Figure 4). As the window size increases, the nine competing algorithms exhibit a decrease in revenue which is in contrast to the increase in revenue exhibited by G-SAVMRM. Our experiments show that as the windows grow larger and contain greater VM resource type heterogeneity, G-SAVMRM makes better greedy allocation decisions for revenue generation than the competing algorithms. The next best performing algorithm is C-IO which tends to have similar behavior to G-SAVMRM due to the fact that vCPU is a scarce resource. G-SAVMM tends to outperform C-IO in terms of average aggregated revenue ratios by approximately 3% in W30, 4% in W50, and 7% in W100.

We also investigate the performance of the algorithms in terms of average generated revenue per server (Figure 5). The averages are calculated over 500 windows for W30, 300 windows for W50, and 150 windows for W100. The standard error determined for G-SAVMRM ranges from 0.06 for W30 (average revenue of 8.38), to 0.13 for W100 (average revenue of 9.37). The standard error for the other algorithms is within the same ranges as that of G-SAVMRM. The results are fairly consistent with those in Figure 4, in that G-SAVMRM generates the highest average revenue followed by C-IO.

### Table 5: Algorithms Used in Experiments.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-SAVMRM</td>
<td>Decreasing order of ( E_j^k )</td>
</tr>
<tr>
<td>G-SAVMM</td>
<td>Decreasing order of efficiency metric defined on memory resource only.</td>
</tr>
<tr>
<td>P-DO</td>
<td>Decreasing order of revenue.</td>
</tr>
<tr>
<td>SP-DO</td>
<td>Decreasing order of the number of shared pages.</td>
</tr>
<tr>
<td>C-DO</td>
<td>Increasing order of the number of requested vCPUs.</td>
</tr>
<tr>
<td>M-DO</td>
<td>Decreasing order of the amount of requested memory.</td>
</tr>
<tr>
<td>M-IO</td>
<td>Increasing order of the amount of requested memory.</td>
</tr>
<tr>
<td>DR-DO</td>
<td>Decreasing order of the dominant resource.</td>
</tr>
<tr>
<td>DR-IO</td>
<td>Increasing order of the dominant resource.</td>
</tr>
</tbody>
</table>
for all window types, G-SAVMM outperforms C-IO when comparing the average revenue generated per server by approximately 4% in W30 (or by $.024), 5% in W50 (or by $.041), and 8% in W100 (or by $.076). While these differences may be small; operating at scale with millions of VMs and tens of thousands of servers can lead to sizable losses of revenue if a less efficient algorithm is used. Our results reveal that G-SAVMRM is the best performing algorithm, obtaining greater revenue ratios and higher average revenue than the other nine algorithms.

When allocating VMs to server resources, the scarcest resource is the vCPU resource. Therefore, algorithms which maximize the vCPU and memory resources while generating higher revenues are desirable. In Figure 6, we compare the nine resource-centric algorithms against G-SAVMRM in terms of resource utilization. On the left side of Figure 6, we compare G-SAVMM and two memory-centric allocation algorithms, SP-DO and M-IO, against G-SAVMRM, and on the right, we compare G-SAVMM and two vCPU-centric allocation algorithms, P-DO and C-IO, against G-SAVMRM. P-DO is a vCPU-centric allocation algorithm since the value of a VM is more related to the scarcity of the vCPU resource. Focusing on memory, we plot the average utilization percentage for G-SAVMM and each memory-centric algorithm. SP-DO slightly outperforms G-SAVMM by .1% in W30, .2% in W50, and .4% in W100. While SP-DO utilizes slightly more memory than G-SAVMRM, choosing SP-DO as the allocation algorithm would lead to significantly less revenue generated on average per server. Focusing on vCPUs, we plot the average utilization percentage for G-SAVMM and each vCPU-centric algorithm. C-IO slightly outperforms G-SAVMRM by approximately .01% in W30, W50 and W100. The performance difference among G-SAVMM and P-DO is .01% in W30, W50 and W100; meaning, their performance is nearly identical on average. The performance difference among G-SAVMM and P-DO is approximately .01% in W30, W50 and W100; meaning, their performance is nearly identical on average when comparing their vCPU utilization. Nonetheless, both algorithms lag behind G-SAVMRM by approximately .01% in W30, W50 and .03% in W100. While C-IO utilizes slightly more vCPUs than G-SAVMRM, choosing C-IO as the allocation algorithm would lead to less revenue generated on average, $.29 instead of $.52 per server over the other algorithms. Although G-SAVMRM is a multi-resource allocation algorithm, its memory utilization is marginally close to the best memory-centric algorithm, SP-DO, and its vCPU utilization is marginally close to the best vCPU-centric algorithm, C-IO; subsequently generating the highest revenue among them.

Our algorithm obtains the best memory utilization among the algorithms (M-DO and M-IO) that consider only the amount of memory requests and not sharing when making decisions. This shows that memory sharing plays a significant role in our algorithm obtaining the best revenue among the other algorithms. On the other hand, SP-DO obtains higher memory utilization, but since it allocates the VMs in decreasing order of the number of shared pages it does not obtain revenue greater than our algorithm. This is because it does not consider the scarcity of memory (including sharing) and that of other resources. These show that the combined effect of both memory sharing and the consideration of multiple resources makes our algorithm the best performer in terms of revenue.

In Figure 7, we plot the average execution times for G-SAVMRM and all other algorithms considered in the experiments. The average execution times are calculated over 500 windows for W30, 300 windows for W50, and 150 windows for W100. The standard error determined for G-SAVMRM ranges from 0.003 for W30 (average execution time of 0.1026 seconds), to 0.077 for W100 (average execution time of 1.3580 seconds). The standard error for the other algorithms is within the same ranges as that of G-SAVMRM. In all windows, the smallest average execution times are obtained by the decreasing order resource algorithms: P-DO (0.0668 seconds), C-DO (0.0710 seconds), and M-DO (0.0717 seconds) in W30; P-DO (0.1584 seconds), M-DO (0.1585 seconds) and C-DO (0.1701 seconds) in W50; and P-DO (0.4801 seconds), C-DO (0.5771 seconds) and C-DO (0.6201 seconds) in W100. The average execution times for G-SAVMRM in W30, W50 and W100 are 0.1026 seconds, 0.3048 seconds and 1.3581 seconds, respectively. The worst average execution times are obtained for the increasing order resource allocation algorithms (C-IO, M-IO and DR-IO) in all windows. G-SAVMRM outperforms all increasing order resource allocation algorithms for all instances. Our previously proposed algorithm, G-SAVMM, outperforms G-SAVMRM in terms of execution time since G-SAVMRM requires more computation when determining the efficiency metric. While the execution times for the decreasing order resource algorithms are slightly better than those of G-SAVMRM, the revenue generated on average by G-SAVMRM surpasses that generated by the other algorithms overshadowing their negligible savings in...
execution time. To have an idea of how much time it takes to
obtain the optimal solution for the G-SAVMRM problem, we
use the Couenne solver to solve the BMP-SAVMRM program
presented in Section 3. The average execution time required
by the solver is about 32 minutes for small instances of
type W30. This is approximately four orders of magnitude
greater than the average execution times of G-SAVMRM and
its variants for those instances, making the use of optimal
solvers for large instances of the problem unfeasible in
practice.

We now compare the performance of G-SAVMM against
the optimal solution obtained by solving the Binary Mul-
tilinear Program (BMP-SAVMRM) presented in Section 3.
In order to solve BMP-SAVMRM optimally, we use the
AMPL [36] mathematical programming framework and an
open-source solver, Couenne [34] hosted by the NEOS
Server [35], a free internet-based service for solving opti-
mization problems. Since it is not possible to solve large
instances of SAVMRM optimally in reasonable amount of
time, we only consider small-scale instances of the problem
consisting of 10, 15 and 20 VMs and three types of host-
ing servers: Server1 (20 vCPUs and 256 GBs of memory),
Server2 (30 vCPUs and 512 GBs of memory), and Server3
(50 vCPUs and 1024 GBs of memory). The VM requests are
randomly selected from the W30 trace windows. In Figure 8,
we compare the average execution time of G-SAVMRM and
the time required to obtain the optimal solution by solving
BMP-SAVMRM. In the figure, we label the time to obtain
the optimal solution by OPT. We run ten simulations for
the types of instances with 10, 15 and 20 VMs, and calculate
the average execution time. As can be observed from the
results, the execution time of G-SAVMRM is several orders
of magnitude smaller than the execution time of OPT. As
an example for instances with 20 VMs the execution time of
OPT is about 16 minutes, while that of G-SAVMRM is only
about 10 milliseconds. As the size of the problem instances
increases, we expect very large increases in the time required
to obtain the optimal solution, making it unfeasible to obtain
in practice. In Figure 9, we plot the average ratio of the
revenue generated by the optimal solution and the revenue
generated by G-SAVMRM for the nine instances obtained
by combining the tree sizes of requests (10, 15, and 20 VMs)
and the three types of servers described above. We run ten
simulations for the nine instances and calculate the average
revenue ratios. G-SAVMRM obtains revenues that are 1.4 to
2.7 times lower than the optimal one. This shows that for

<table>
<thead>
<tr>
<th># of VMs</th>
<th>W30</th>
<th>W50</th>
<th>W100</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-SAVMRM</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>OPT</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 7: Execution Times.

Figure 8: OPT vs. G-SAVMRM: Execution Time.

Figure 9: OPT vs. G-SAVMRM: Revenue Ratio.

7 CONCLUSION AND FUTURE WORK

We designed a sharing-aware greedy approximation al-
gorithm (G-SAVMRM) for solving the sharing-aware VM
revenue maximization problem in single server environ-
ments. The experimental results showed that G-SAVMRM
outperforms nine other VM allocation algorithms in terms
of generated revenue and efficient utilization of resources.
Addressing the multi-server case would require solving a
variant of the multiple-multidimensional knapsack problem
with overlapping items for which no algorithms are readily
available. Extending our proposed solution to handle the
multi-server case would require redesigning the efficiency
metric and the greedy algorithm to take into account the
restrictions on the sharing of memory (e.g., the VMs have
to be collocated on the same server to be able to share
memory). In future work, we plan on extending the pro-
duced algorithm to manage the VM allocation process in
multi-server and online environments. Incorporating energy
consumption awareness and network virtualization into the
multi-resource VM allocation problem would be an interest-
ing extension.

ACKNOWLEDGMENT

This paper is a revised and extended version of [37] pre-
sented at the 3rd IEEE International Conf. on Cloud Engi-
neering (IEEE IC2E’15). This research was supported in part
by NSF grants DGE-0654014 and CNS-1116787.

<table>
<thead>
<tr>
<th># of VMs</th>
<th>W30</th>
<th>W50</th>
<th>W100</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-SAVMRM</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>OPT</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
References


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

Daniel Grosu received the Diploma in engineering (automatic control and industrial informatics) from the Technical University of Iasi, 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, resource allocation, computer security, and topics at the border of computer science, game theory and economics. He has published more than one hundred 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.