View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digital Commons@Wayne State University

Wayne State University

Wayne State University Dissertations

1-1-2016

Sharing-Aware Resource Management Algorithms For Virtual Computing Environments

Safraz Rampersaud
Wayne State University,

Follow this and additional works at: https://digitalcommons.wayne.edu/oa_dissertations

Part of the Computer Sciences Commons

Recommended Citation
Rampersaud, Safraz, "Sharing-Aware Resource Management Algorithms For Virtual Computing Environments" (2016). *Wayne State University Dissertations*. 1477.
https://digitalcommons.wayne.edu/oa_dissertations/1477

This Open Access Dissertation is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Dissertations by an authorized administrator of DigitalCommons@WayneState.

SHARING-AWARE RESOURCE MANAGEMENT ALGORITHMS FOR VIRTUAL COMPUTING ENVIRONMENTS

by

SAFRAZ RAMPERSAUD

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

2016

MAJOR: COMPUTER SCIENCE

Approved By:

____

Advisor   Date

____

____

____

____

©COPYRIGHT BY

SAFRAZ RAMPERSAUD

2016

All Rights Reserved

DEDICATION

In memory of Gerald Keith Larson. We started the Ph.D. program together but unfortunately you passed on before completing the journey.

ACKNOWLEDGEMENTS

I am indebted to Dr. Daniel Grosu for his dedication, direction and time while leading this body of research as my Ph.D. adviser. I hope we always keep the lines of communication between us open following the completion of this fundamental stage. I would like to thank the members of my dissertation committee Dr. Nathan Fisher, Dr. Boris Mordukhovich, Dr. Robert Reynolds and Dr. Loren Schwiebert for their support over the years. I have gained advice, ability, and perspective from this committee on both personal and professional levels and hope that in some way I can pay the investment forward.

I would like to acknowledge my mother and father, Farida and Feizal Rampersaud. I offer my sincerest gratitude for their unconditional support; only they know how far of a comeback I've had to make to get here. I would like to acknowledge my brother, Reza, and my sister, Farah, for keeping life interesting. I am glad I was home to share in their experiences which motivated me to persist onward. I would also like to acknowledge my significant other, Bernadette, for the emotional support and for keeping me in good spirits.

My research was supported in part by NSF grants DGE-0654014 and CNS-1116787 and from opportunities made available by the College of Engineering in both the Department of Chemical Engineering and the Department of Computer Science.

TABLE OF CONTENTS

Dedication i

Acknowledgements ii

LIST OF FIGURES vii

LIST OF TABLES viii

CHAPTER 1: INTRODUCTION 1

1.1 Background 1

1.1.1 The Dawn of the Hypervisor 2

1.1.2 The Practice of Page Sharing 4

1.1.3 Foundations of Sharing-Aware Resource Management 7

1.1.4 Our Contributions 12

1.2 Organization 15

CHAPTER 2: SINGLE-RESOURCE VM MAXIMIZATION 16

2.1 Introduction 16

2.1.1 Our Contribution 17

2.1.2 Related Work 18

2.1.3 Organization 19

2.2 Sharing-Aware VM Maximization 20

2.3 Greedy Approximation Algorithm (G-SAVMM) 20

2.4 G-SAVMM Properties 26

2.5 Experimental Results. 29

2.5.1 Experimental Setup 30

2.5.2 Analysis of Results 31

2.6 Summary 34

CHAPTER 3: MULTI-RESOURCE VM MAXIMIZATION 35

3.1 Introduction 35

3.1.1 Our Contribution 36

3.1.2 Related Work 37
3.1.3 Organization 39
3.2 Multi-Resource Sharing-Aware VM Maximization 40
3.3 Binary Multilinear Program Formulation 42
3.4 Greedy Approximation Algorithm (G-MSAVMM) 46
3.5 G-MSAVMM Properties 53
3.6 Experimental Results 57
3.6.1 Experimental Setup 57
3.6.2 Analysis of Results 64
3.7 Summary 70
CHAPTER 4: MULTI-RESOURCE VM PACKING **73**
4.1 Introduction 73
4.1.1 Our Contribution 74
4.1.2 Related Work 74
4.1.3 Organization 76
4.2 SA-OVMP: Problem 77
4.3 SA-OVMP: Algorithms 79
4.3.1 Next-Fit-Sharing (NFS) Algorithm 81
4.3.2 First-Fit-Sharing (FFS) Algorithm 84
4.3.3 Best-Fit-Sharing (BFS) Algorithm 86
4.3.4 Worst-Fit-Sharing (WFS) Algorithm 92
4.4 Offline Sharing-Aware VM Packing 92
4.5 Experimental Results 95
4.5.1 Experimental Setup 95
4.5.2 Analysis of Results 100
4.6 Summary 105

CHAPTER 5: CONCLUSION **107**
5.1 Summary of Contributions 107
5.2 Future Research Directions 108
5.2.1 Analyzing Sharing-Aware Online VM Packing Performance 108
5.2.2 Sharing-Aware Algorithms for Container Management 109
5.2.3 Sharing-Aware Streaming Resource Management 110
APPENDIX **111**
REFERENCES **123**
ABSTRACT **124**
AUTOBIOGRAPHICAL STATEMENT **125**

LIST OF FIGURES

Figure 1.1 Page sharing among two VM tenants. 5

Figure 2.1 G-SAVMM: Execution Example. 26

Figure 2.2 G-SAVMM: Revenue Ratios vs. Sharing Stratifications. 32

Figure 2.3 G-SAVMM: Capacity Ratios vs. Sharing Stratifications. 33

Figure 3.1 Page Sharing Among VMs. 44

Figure 3.2 G-MSAVMM Efficiency Metric Calculation: Iteration 0 51

Figure 3.3 G-MSAVMM Efficiency Metric Calculation: Iteration 1 51

Figure 3.4 G-MSAVMM Efficiency Metric Calculation: Iteration 2 52

Figure 3.5 Page Sharing Percentages Table: OS. 61

Figure 3.6 Distribution of Google Type VMs in Experiment. 65

Figure 3.7 Sharing vs. non-Sharing Memory Utilization. 66

Figure 3.8 Average Aggregate Revenue Ratios. 67

Figure 3.9 Average Revenue Per Server. 68

Figure 3.10 Memory / CPU Utilization. 69

Figure 3.11 W30: G-MSAVMM behavior for different VM request configurations. . . 70

Figure 3.12 W50: G-MSAVMM behavior for different VM request configurations. . . 71

Figure 3.13 W100: G-MSAVMM behavior for different VM request configurations. . 72

Figure 4.1 SA-OVMP: VM Requests and Resource Configuration. 80

Figure 4.2 NFS: VM Assignment 83

Figure 4.3 FFS: VM Assignment 85

Figure 4.4 BFS: Init 89

Figure 4.5 BFS: VM 1 Assignment 89

Figure 4.6 BFS: VM 2 Assignment 90

Figure 4.7 BFS: VM 3 Assignment 90

Figure 4.8 BFS: VM 4 Assignment 91

Figure 4.9 BFS: VM 5 Assignment 91

Figure 4.10 BFS: VM Final Assignment 91
Figure 4.11 Optimal VM Assignment 93
Figure 4.12 Sharing parameter values among V_3, V_4 and V_6 95
Figure 4.13 85% Low Request 1000 VM Stream. 97
Figure 4.14 15% Low Request 1000 VM Stream. 98
Figure 4.15 Server Configurations. 99
Figure 4.16 Average Memory Reduction: 500 VM Stream. 101
Figure 4.17 Average Memory Reduction: 1000 VM Stream. 102
Figure 4.18 Excess Active Servers: 500 VM Stream. 103
Figure 4.19 Excess Active Servers: 1000 VM Stream. 103
Figure 4.20 Average Active Servers Over All 500 VM Streams. 104
Figure 4.21 Average Active Servers Over All 1000 VM Streams. 105

LIST OF TABLES

Table 2.1 SAVMM Notation. 21

Table 2.2 Efficiency Metric Calculation Example. 26

Table 3.3 MSAVMM Notation. 41

Table 3.4 VM Characteristics and Sharing Relationships. 43

Table 3.5 G-MSAVMM Experiment: VM Instance Types. 59

Table 3.6 Algorithms Used in Experiments. 63

Table 4.7 SA-OVMP Notation. 77

Table 4.8 SA-OVMP Experiment: VM Instance Types. 96

CHAPTER 1: INTRODUCTION

Virtualization, i.e., the process of abstracting a state from a primal resource such that multiple instances of the abstraction may operate within a single environment simultaneously, has played a dominant role in distributed computing over the past two decades. Cloud service providers, public and private institutions, etc., derive significant value by extending the breadth of their virtualization technology in order to optimize the use of their resources. For many of these entities, this directly translates to cost savings and/or an increase of revenue. Our inquiry focuses on increasing the efficiency of resource management strategies within a *virtual computing environment* by exploiting the potential for sharing resources. Our interpretation of virtual computing environment corresponds to any computing environment where resources can be virtualized.

Our research focuses on virtual memory reclamation techniques, specifically *page sharing*, and how this process influences resource management strategies when providers are bound to allocate resources in a variety of settings within a virtual computing environment. From the algorithmic perspective, inquiries of this nature have only been investigated through a single paper, Sindelar *et al.* [86], outside of our own contributions. At a time when cutting-edge technologies such as "wearable" devices and the internet-of-things (IoT) are heavily dependent on large-scale virtualization of services for operability, service providers, now and in the future, should improve resource utilization at every opportunity to support these innovations at scale. Therefore, designing efficient resource management strategies in virtual computing environments is pivotal to a growing industry.

1.1 Background

In this section, we introduce the concepts that will serve as the foundation for this dissertation. The contents therein are an introduction to virtualization, an explanation of how page-sharing operates, a motivation for formulating page sharing relationships, and a review of relevant approximation algorithm concepts and models used throughout our work.

We then present our contributions which make up the building blocks of this dissertation and close outlining the chapters within this dissertation.

1.1.1 The Dawn of the Hypervisor

In 1974, Popek and Goldberg [75] proposed sufficient conditions for the efficient exploitation of unused computing resources within a computer architecture. First-generation computers offered computing capabilities for mostly single tasks and second-generation computing extended usability by dedicating more specialized instructions to the hardware and allowed users more freedom to design processes and applications through high-level programming languages. In the third-generation of computing, internal relocation and trap mechanisms, time-sharing and operating system multitasking were used to manage computing machine resources in order to perform tasks fast without having to utilize all the available machine resources; paving the way for system resource redistribution.

Popek and Goldberg envisioned an update to the third-generation computing era where physical machines (PMs) could abstract a duplicate of themselves and isolate their processes from other abstractions on the same PM efficiently. Their ideas motivated the use of a software layer known as the virtual machine monitor (VMM), or *hypervisor*, which would support three main functionalities: (i) creates a virtual machine environment nearly identical to an environment directly supported by a PM, (ii) instantiation of the abstractions would only suffer minimal performance degradation, and (iii) the system resources would be controlled by the VMM software layer; situated between the abstractions and the PM resources from which it is supported. Then, any abstraction under the control of the VMM would be known as a *virtual machine* (VM).

In order for VMs to operate, they must satisfy three main properties: (i) *efficiency*, the VM should be able to execute user processes without requiring VMM support outside of acquiring resources; (ii) *resource control*, the VMs may not access or modify the system resources directly; and (iii) *equivalence*, not considering timing or lack of resources, the VM execution under a VMM should be near indistinguishable from process execution natively on

a PM. In order to characterize these properties, Popek and Goldberg classified the types of machine instructions used in Instruction Set Architectures (ISA) into three categories: (a) *privileged*, processor instructions which perform a trap in user mode and do not perform a trap if they are in system (kernel) mode; (b) *control sensitive*, processor instructions which attempt to change system resource configurations; and (c) *behavior sensitive*, processor instructions which are dependent on the system resource configurations. Under these categories of instruction types, Popek and Goldberg [75] introduced the first theorem of virtualization as follows:

Theorem 1.1.1. *For any conventional third-generation computer, an effective VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

Theorem 1.1.1 states that if an architecture satisfies all properties (i) through (iii) by classifying processor instructions into (a) through (c), and if the VMM sensitive instructions are a subset of its privileged instructions, then the architecture is virtualizable. Since Theorem 1.1.1 is only a sufficient condition, architectures which do not satisfy the stated requirements may still be virtualizable either through further modifications, e.g., binary-translation, or only be partially virtualizable, e.g., para-virtualization. Popek and Goldberg's second theorem corresponds to recursive virtualization, i.e., abstracting a VMM through a VM abstraction. Their theorem is as follows:

Theorem 1.1.2. *A conventional third generation computer is recursively virtualizable if it is: (1) virtualizable, and (2) a VMM without timing dependencies can be constructed for it.*

The first component of Theorem 1.1.2 follows from Theorem 1.1.1. The second component of Theorem 1.1.2 constrains the VMM to execute without timing dependencies. If timing dependencies exist for the abstracted VMM, then this could lower performance which would violate the *equivalence* property.

1.1.2 The Practice of Page Sharing

Page sharing is a memory reclamation technique which hypervisors use in order to reduce memory utilization from among a group of VM tenants residing on the same PM. The process, managed by the hypervisor, entails identifying two or more VM tenants which run similar processes such as applications, libraries, and/or operating systems; all consisting of physical blocks of memory, where a lower level of granularity for these physical blocks of memory are known as *pages*. If two or more VM tenants execute similar processes on the same PM, then the hypervisor can support the deduplication of identical pages for multiple VM tenants without interrupting their intended processes. When deduplication occurs, a single page survives and is used as the reference page, or is *shared*, among VM tenants executing similar processes. As an example, Figure 1.1 illustrates the end result of a page being *shared* among two VM tenants. Both VM tenants necessitate six pages of memory, where the fifth page within VM1's memory block is identical to the third page within VM2's memory block. The hypervisor identifies this equivalence, deduplicates the similar pages among the VM tenants, manages a copy of the page within its own block of memory and provides references from that page to the appropriate locations within the VMs memory block in lieu of managing multiple, identical physical memory pages; hence, the process of *page sharing* has occurred. The concept of memory sharing was introduced in 1972 by Parmelee *et al.* [73]. Shortly thereafter, system implementations of memory sharing features were proposed by Bagley *et al.* [4]. Motivated by the authors' desire to develop a centralized library management database among a group of users, the VMM would not move physical memory from one user to another, but rather changes to the references, addresses and privileges of the users page table entries would occur in order to share the memory features. The users could then access and modify content within the database without the VMM transferring memory from one user to another through managed pointer references to the data of interest.

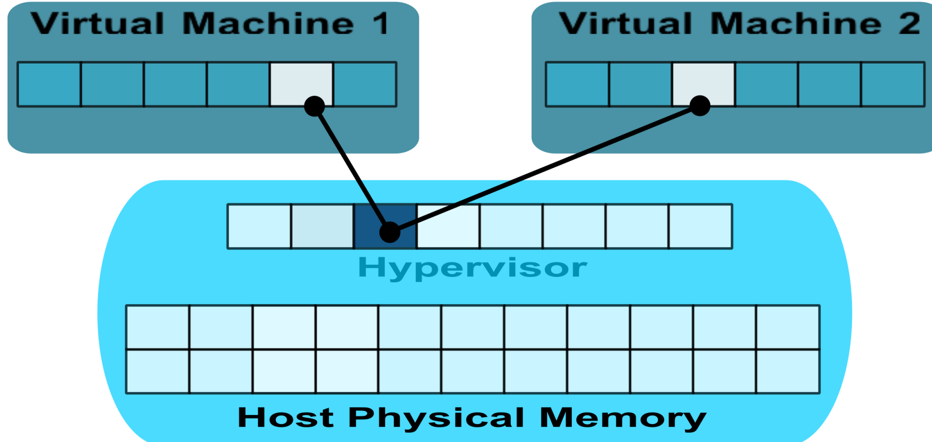


Figure 1.1: Page sharing among two VM tenants.

In the late 1990s, a different motivation lead to a resurgence of considering how resources can be shared through the VMM. In 1997, research brought forth by Bugnion *et al.* [15] was motivated by the need to manage large-scale, shared-memory multiprocessor operating system resources. From their perspective, operating system software was not developing as fast as needed to accommodate large-scale systems for new memory and processor hardware. A feature of their proposed solution was to modify the hypervisor layer to take advantage of shared memory among VM tenants in the form of *transparent page sharing* (TPS). This technique based page sharing on page characteristics such as origin and location within the hard disk. The VM tenant had opportunities to access the shared pages but issues would occur if the memory pages were modified. As a result, Bugnion *et al.* [15] implemented a system composed of copy-on-write disks and operations to allow VM tenants to share the original pages; yet, for the VM of interest desiring to modify memory through a shared page, a private copy was created by the hypervisor and accessed strictly by the modifying VM only.

Transparent page sharing lead the way for large systems to minimize their memory resources; yet, in order to operate correctly, modifications to the VM tenant operating system would have to occur. Recognizing this as a potential liability, Waldspurger [98] is credited

with the introduction of a new page sharing technique called *content-based page sharing.* In order to implement content-based page sharing, any hypervisor will routinely perform a search which scans for memory pages among VM tenants which are identical. A brute-force search through all VM tenants for determining identical pages is expensive with a runtime of $\mathcal{O}(n^2)$, where n is the number of VM tenants. Instead of a brute-force method, a hash table of VM tenant pages is managed by the hypervisor in order to determine identical pages in less time. Early on, page-sharing systems implemented hashing algorithms such as Jenkins hashing function by Jenkins [45], then later implemented a more efficient algorithm, SuperFastHash by Hsieh [44], in order to capture potential page sharing opportunities within a hash table.

Typically, hypervisor implementations operate on blocks of memory pages in sizes of either 4 KB or 20 MB. Research has shown that operating on the former size makes finding identical page blocks more difficult than in the latter size [5]. Each memory page, which is evaluated for sharing, will have a generated hash value associated with it based on its bit content. The page hash value is then checked against other hash values in a hash table, where the table entries consist of both the hash value and a page number which identifies the original page, managed by the hypervisor, to be shared. If a match is determined, a comparison between the potential and the original page ensues to determine if they are bit-wise identical. If the bits match exactly, a reference to the original page is created for the potential page and the potential page memory is reclaimed. Lastly, the original page is flagged as read-only and then marked as copy-on-write by the hypervisor. A shared page may be accessed by VM tenants but not modified explicitly. In the case a VM tenant requires a write operation relative to the shared page, the hypervisor generates a private copy of the shared page to be accessed by the VM tenant and provided with read-write access. Other VM tenants which share the page will not have access to the private copy.

1.1.3 Foundations of Sharing-Aware Resource Management

Our research focuses on the design and analysis of *sharing-aware* resource management algorithms. The differences between our proposals and the existing techniques are that existing techniques do not focus on capturing the utility of memory sharing when allocating VM tenants onto PM resources and they restrict the relationship between VM tenants and their memory pages to a specific model when attempting to identify page sharing opportunities. Therefore, if we consider page-sharing within a variety of more traditional VM allocation problems, the process becomes more difficult to manage and further modifications to existing algorithms are required. Considering the example from Figure 1.1, we formalize a sharing relationship where both V_1 and V_2 are composed of six pages and an identical page is shared between them. If we aggregate the amount of memory required to host the VM tenants and include the pages managed by the hypevisor, we can derive the following relationship,

$$\begin{aligned} &|\pi(V_1) \cup \pi(V_2)| \leq |\pi(V_1)| + |\pi(V_2)|, \text{ where} \\ &|\pi(V_1) \cup \pi(V_2)| = 11 \;\&\; |\pi(V_1)| + |\pi(V_2)| = 12 \end{aligned} \tag{1.1}$$

and $\pi(V_i)$ represents the set of memory pages required by VM V_i. The right-hand side of Equation 1.1 corresponds to the number of memory pages requested by each VM, while the left-hand side corresponds to pages allocated by the mechanism, that is allocating the shared pages only once in memory. While this is a small example, it nonetheless expresses how, through page sharing, the aggregate number of memory pages which are required to be managed is less than the total number of requested memory pages by the VM tenants; reflecting a triangle-like inequality on the number of required pages. Moreover, greater insight into how many pages are required by the hypervisor to host both VMs can be obtained by re-expressing the union of pages between the two VM memory page sets as,

$$|\pi(V_1)| \cup |\pi(V_2)| = |\pi(V_1)| + |\pi(V_2)| - |\pi(V_1 \cap V_2)|, \text{ or} \tag{1.2}$$

$$\left|\bigcup_{j=1}^{2}\pi(V_j)\right| = |\pi(V_1)| + |\pi(V_2)| - |\pi(V_1 \cap V_2)|. \tag{1.3}$$

Naturally, we can extend the relationship to the general case for M VM tenants, where the aggregate memory pages required to host all the tenants by the hypervisor is identified as the union of all pages requested. Due to the properties of sets, only unique pages will be elements of the union; whereby, any of these pages are *shareable*. Similar in form to Equation 1.3, we can expand the right side for the general case as follows,

$$\begin{aligned}\left|\bigcup_{j=1}^{M}\pi(V_j)\right| = &\sum_{j=1}^{M}\pi(V_j) - \sum_{j_1<j_2}\pi(V_{j_1}\cap V_{j_2}) + \\ &\cdots + (-1)^{r+1}\sum_{j_1<j_2<\cdots<j_r}\pi(V_{j_1}\cap V_{j_2}\cap\cdots\cap V_{j_r}) + \\ &\cdots + (-1)^{M+1}\pi(V_{j_1}\cap V_{j_2}\cap\cdots\cap V_{j_M})\end{aligned} \tag{1.4}$$

where $\sum_{j_1<j_2<\cdots<j_r}\pi(V_{j_1}\cap V_{j_2}\cap\cdots\cap V_{j_r})$ is taken over all $\binom{M}{r}$ possible subsets of size r from the set $\{V_1, V_2, \ldots, V_M\}$. Based on the *inclusion-exclusion identity* from probability theory [85], Equation 1.4 can be simplified and re-expressed in set notation form on the indices in the right hand side as follows,

$$\left|\bigcup_{j=1}^{M}\pi(V_j)\right| = \sum_{J\in\mathcal{P}(\mathcal{V})}(-1)^{(|J|+1)}\left|\bigcap_{j\in J}\pi(V_j)\right|. \tag{1.5}$$

The set notation index on J in Equation 1.5 corresponds to an index from the power set of the set of VMs, $\mathcal{P}(\mathcal{V})$, where $|\mathcal{V}| = M$. The right hand side of Equation 1.5 serves as a basis to characterize the general page sharing relationship between M VM tenants and their subsets in "offline" environments. In order to determine the optimal VM allocation in "offline" environments while considering page sharing, optimization programs which exhibit characteristics of nonlinearity and nonconvexity can be modeled and solved for by considering the right hand side of Equation 1.5 as the program's memory constraint shown in Chapters 3 and 4. If enough memory pages can be shared and all other resources are available, then

more VMs may be allocated to utilize more efficiently the memory resource. Unfortunately, calculating the right hand side of Equation 1.5 to determine the number of pages required among a set of M VM tenants requires an exponential number of operations, making the computation infeasible. Therefore, we have to rely on approximation algorithms which can determine VM allocations while considering page sharing and can execute in reasonable time and generate reasonable results. In the following subsections, we review the approximation algorithms concepts and system models which underpin the design of our sharing-aware resource management algorithms.

The Knapsack Problem

We now briefly describe the classic knapsack problem and its application to sharing-aware resource management. The *knapsack* problem [95] is a classic combinatorial optimization problem described as follows:

> The Knapsack Problem: Given a set $\mathcal{S} = \{a_1, \ldots, a_n\}$ of objects, with size(a_i), revenue(a_i)$\in \mathbb{Z}^+$, and a "knapsack capacity" $\mathcal{B} \in \mathbb{Z}^+$, find a subset of objects whose total size is bounded by $\mathcal{B}$ and the total revenue is maximized.

Problems of this combinatorial nature are $\mathcal{NP}$-hard [32] and have been investigated well before the turn of the 20th century. In 1957, Dantzig coined the term *knapsack* in observation of certain classes of combinatorial problems which could be modeled as discrete-valued, linear programming problems. The standard 0-1 integer programming version of the knapsack problem can be formulated as follows [60]:

$$\max \sum_{j=1}^{n} p_j x_j$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_j \leq c$$

$$\text{where } x_j \in \{0,\ 1\},\ \forall j \in \{1, 2, \ldots, n\}$$

and p_j is the revenue of the jth item, w_j is the size of the jth item, c is the knapsack capacity and x_j is a boolean decision variable which determines if the jth item should be included

in the "knapsack", $x_j = 1$, or should not be included, $x_j = 0$. Many variations of the standard formulation have have been investigated in the research literature when framing knapsack-like problems with specific qualities, e.g., fractional items, multi-dimensional, non-linear objectives, etc. Heuristic solution techniques have been formulated early on in order to solve knapsack problems based on dynamic programming [25], greedy algorithms [58] and branch & bound techniques [55]. A comprehensive treatment of knapsack variant problems, approximation algorithms for solving them, and performance analyses can be found in Vazirani [95], Martello and Toth [60], and Kellerer [52].

Specific to our research, we investigate *VM Maximization* which describes the problem of allocating VMs onto a single server to maximize the revenue, where the revenue is the sum of the revenue derived from hosting each individual VM; which in the most general form, can be modeled as the knapsack problem. When the sharing of pages among the VMs is considered, the problem of VM revenue maximization is no longer directly equivalent to the knapsack problem and existing algorithms will produce less than the maximum revenue due to not allocating additional VMs on the extraneous server resources. Thus, the VM Maximization problem is considered a new variant of the knapsack problem in which the items can share space in the knapsack.

The Bin-Packing Problem

We now briefly describe the classic bin packing problem and its application to sharing-aware resource management. The origins of the bin packing problem were inspired by the knapsack problem through applications of the cutting stock, Gilmore and Gomory [34], and job-shop scheduling, Conway *et al.* [22], problems from the 1960s. Both of these applications previously modeled their problems as knapsack variants in order to maximize a specific objective. When the objective shifts from identifying the subcollection of items which maximizes a value, to minimizing the number of "knapsacks" required to complete an assignment of items, the problem is then reformulated into a *bin packing* problem. The *bin packing* [95] problem is a classic combinatorial optimization problem which is described as follows:

The Bin Packing Problem: Given a bin $\mathcal{S}$ of size $\mathcal{V}$ and a list of n items with sizes $a_1, a_1, \ldots, a_n$ to pack, find an integer number of bins $\mathcal{B}$ and a $\mathcal{B}$-partition $\mathcal{S}_1 \bigcup \cdots \bigcup \mathcal{S}_{\mathcal{B}}$ of the set $\{1, 2, \ldots, n\}$ such that $\sum_{i \in \mathcal{S}_i} a_i \leq \mathcal{V}, \; \forall \; k \; = \; 1, 2, \ldots, \mathcal{B}$ and the number of bins is minimized.

The standard 0-1 integer programming version of the bin packing problem can be formulated as follows [60]:

$$\min \sum_{i=1}^{n} y_i$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_{ij} \leq c y_i, \; \forall \; i \in \{1, 2, \ldots, n\}$$

$$\text{s.t.} \sum_{j=1}^{n} x_{ij} = 1, \; \forall \; i \in \{1, 2, \ldots, n\}$$

$$\text{where } x_{ij} \in \{0, \; 1\}, \; \forall \; i, j \in \{1, 2, \ldots, n\}$$

$$\text{and } y_j \in \{0, \; 1\}, \; \forall \; j \in \{1, 2, \ldots, n\}$$

and c is the capacity of each bin, w_j is the weight of the jth item, y_i is a boolean decision variable which determines if the ith bin should be used, $y_i = 1$, or should not be used, $y_i = 0$, and x_{ij} is also a boolean decision variable which determines if the jth item should be assigned to the ith bin, $x_{ij} = 1$, or should not be assigned accordingly, $x_{ij} = 0$. Due to combinatorial nature of assigning items for every combination of bins, the bin packing problem is also $\mathcal{NP}$-hard [32]. As a result, a suite of heuristic algorithms were developed which solve the classic bin packing problem. In 1972, Garey *et al.* [31] designed and analyzed several algorithms for the bin-packing problem; namely, First-Fit, Best-Fit, First-Fit-Decreasing and Best-Fit-Decreasing. Further research in this domain naturally followed in Johnson [49]; broadening the class of heuristic algorithms solving the bin packing problem in which algorithms belonging to the same class were characterized by similar worst case behavior. In 1974, a thorough analysis of the aforementioned works was published by Johnson *et al.* [48] which designed and analyzed a suite of approximation algorithms for the bin packing problem.

Several variations on the standard formulation have appeared in the literature for framing bin packing problems with specific qualities, e.g., bin packing with variable sized bins, bin packing with item rejection, bin packing with item fragmentation, etc. Approximation algorithms have been studied rigorously over half a century for solving bin packing problems and their variants. A comprehensive survey on approximation algorithms for classic bin packing problems is by Coffman *et al.* [20]. Approximately three decades later, Coffman *et al.* [19] provided an updated survey of bin-packing problems.

Specific to our research, we investigate *VM Packing* which describes the assignment of VM requests onto a minimum number of active servers required to instantiate the requests; which in the most general form, can be modeled as the bin packing problem. When the sharing of pages among the VMs is considered, the problem of determining the minimum set of active servers is no longer directly equivalent to the bin packing problem and existing algorithms will activate more servers than necessary; resulting in wasted server resource utilization. Thus, the VM Packing problem is considered a new variant of the bin-packing problem in which the items can share space in the bins.

Sindelar *et al.* [86] 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 dissertation differs substantially from Sindelar *et al.* in that we design algorithms for both online and "offline" settings, consider multiple type VM resource requests, assume heterogeneous server capacities and operate under a general sharing model. 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.

1.1.4 Our Contributions

In this section, we present the summary of our contributions and the outline of our dissertation. We summarize below the three research projects that we accomplished as part of this dissertation.

- **Sharing-Aware Virtual Machine Maximization.** Service providers face multiple challenges in hosting an increasing number of virtual machine (VM) instances. Minimizing the utilization of system resources while maximizing the potential for revenue are among the most common challenges. Recent studies have investigated memory reclamation techniques focused on virtual technologies, specifically page sharing, for minimizing the utilization of system resources. By incorporating page sharing into the challenge of scheduling VMs on physical machines, we formulate the sharing-aware VM maximization (SAVMM) problem. The SAVMM 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 when VMs consist of only the memory resource. The SAVMM problem has been shown to be $\mathcal{NP}$-hard. Therefore, we address this challenge by developing a greedy algorithm for solving this problem. We determine the approximation ratio of our greedy algorithm and perform extensive experiments to investigate its performance against other VM allocation algorithms. This is the first algorithm proposed in the literature which solves the VM maximization problem under a general sharing model. A paper describing this research was published in the Proceedings of the 13th IEEE International Symposium on Network Computing and Applications (NCA'14) [77]. We present this research in Chapter 2.
- **Multi-Resource Sharing-Aware Virtual Machine Maximization.** 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 an approximation algorithm for solving the multi-resource sharing-aware virtual machine maximization (MSAVMM) problem. The MSAVMM 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 MSAVMM problem as a multilinear binary program and optimally solve for maximized revenue, 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. This is the first algorithm proposed in the literature which solves the multi-resource VM maximization problem under a general sharing model. A paper describing this research was published in the Proceedings of the 3rd IEEE International Conference on Cloud Engineering (IC2E'15) [79] and an extended version of this paper has been submitted to IEEE Transactions on Computers for publication. We present this work in detail in Chapter 3.

- **Sharing-Aware Online Algorithms for Virtual Machine Packing in Cloud Environments.** Cloud service providers offer on-demand computing resources to a large number of users by employing virtualization technologies. A key challenge faced by cloud service providers is to develop efficient algorithms for assigning Virtual Machine (VM) instances to server resources such that the number of required servers which meet the users' demand is minimized. This challenge has been referred in the literature as the VM Packing problem, a variant of bin packing that is $\mathcal{NP}$-hard. The VM Packing problem differs from other packing problems in that, through virtualization, the VM instances collocated on the same server can share memory pages which reduces the amount of cloud resources required to satisfy users' demand. By focusing on the opportunity for collocated VMs to virtually share memory through a hypervisor, we design a family of sharing-aware online algorithms for solving the VM Packing problem. We also introduce a new multilinear program which captures the essence of sharing memory and optimally solves the "offline" VM Packing problem. Lastly, we evaluate our

sharing-aware online algorithms through extensive experiments and compare them not only against themselves but also against their sharing-oblivious counterparts. These algorithms are the first algorithms proposed in the literature which solve the multi-resource VM packing problem under a general sharing model. The results of this research were published in Proceedings of the 8th IEEE International Conference on Cloud Computing (CLOUD'15) [80] and an extended version of this paper has been submitted to IEEE Transactions on Parallel and Distributed Systems for publication. We present this work in detail in Chapter 4.

1.2 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we present our research on the design of a new sharing-aware greedy approximation algorithm for the "offline" VM Maximization (SAVMM) problem under a general memory sharing model. In Chapter 3, we present our research on the design of a new multi-resource sharing-aware approximation algorithm which solves the "offline" multi-resource VM Maximization (MSAVMM) problem and introduce the optimal multilinear boolean program which models this problem and can be solved for under a general sharing model. In Chapter 4, we present our research on the design of a family of multi-resource sharing-aware online algorithms for the online VM Packing (SA-OVMP) problem and introduce the optimal multilinear boolean program which models this problem and can be solved for in an "offline" environment under a general sharing model. In Chapter 5, we describe the possible future directions of our research, and conclude the dissertation.

CHAPTER 2: SINGLE-RESOURCE VM MAXIMIZATION

2.1 Introduction

Virtualization, the process of abstracting a software layer which decouples the physical hardware from the operating system to deliver greater resource utililization and flexibility [97], serves as a means to increase productivity, lower power consumption, reduce hardware installation, and overall, minimize the need for increasing the resource capacity to meet the demand [46]. The application of virtualization technologies is ubiquitous in data centers around the world which must consider operational costs and guarantee fast delivery of a variety of profitable services. Specifically, the service provider must ensure the efficiency of their virtualized service in a competitive environment where fast entry to market, technology advancement, and service pricing differentials can separate sustaining providers from antiquated ones. Proprietary virtualization platforms, such as VMWare's ESX Suite, Microsoft's Hyper-V and IBM's PowerVM, vary in their methods of operations, e.g., full-, para- and hardware assisted-virtualization, overhead and available number of guest OS hosting capacities among other features. Open-source alternatives, e.g., Xen, KVM and Linux-VServer, offer comparable features and operations to the proprietary platforms while being supported by a large online community. Moreover, open-source virtualization systems such as Xen [6] have improved the user experience by implementing safe resource management strategies without losing performance and/or functionality.

Virtualization has undergone a significant evolution spanning approximately half a century. Innovations within virtualization technology were initially focused on overcoming the limitations of third-generation computing architectures [35]. Within this context, virtualization solved the problem of protecting non-privileged references to end users when multiple end users attempted to access non-privileged instructions through a privileged mode on the base machine [35]. Invocation of a software layer to access the non-privileged instructions, known at the time as the *privileged software nucleus*, suffered from single access to the non-privileged references limiting the potential for multiple users. Hence, virtualization

was born out of these limitations and fulfilled the opportunity to replicate the privileged and non-privileged instruction sets from the base machine, known as the *host*, for multiple end users through a transformed software layer referred to as a *hypervisor*.

Minimizing resource consumption has been a key driver in the overall advancement of virtualization technologies. Memory reclamation techniques such as ballooning, hypervisor swapping, memory compression, and page sharing all attempt to efficiently utilize virtual machine (VM) memory [98]. Page sharing creates new challenges in the development of algorithms which allocate VMs onto server resources. The problem of allocating VMs onto a single server to maximize the revenue, where the revenue is the sum of the revenues derived from hosting each individual VM, is equivalent to the knapsack problem. The equivalence is made by associating each VM as an object and by quantifying the number of memory pages required to host each VM as the weight. Therefore, each VM can be treated as a distinct object having a weight and a utility given by the revenue derived from hosting it. As a result of this equivalence, knapsack heuristic algorithms can be successfully applied to solve the above VM allocation problem when page sharing is not considered. When the sharing of pages among the VMs is considered, the problem of VM revenue maximization is no longer equivalent to the knapsack problem. Existing knapsack algorithms will produce less than the maximum revenue due to not allocating additional VMs on the extraneous server resources which becomes available when VM pages are shared; resulting in loss of revenue. Therefore, new algorithms for VM maximization that take into account the sharing of pages among VMs must be developed.

2.1.1 Our Contribution

We address the problem of sharing-aware VM maximization in a general sharing model which has as objective finding a subset of VMs that can be hosted by a server with a given memory capacity such that the total revenue derived from hosting the subset of VMs is maximized. This problem has been shown to be $\mathcal{NP}$-hard [86]. Therefore, we design a greedy approximation algorithm based on a new efficiency metric which considers

both revenue-seeking and page sharing opportunities in the VM allocation process. We determine the approximation ratio of our greedy algorithm that solves the sharing-aware VM maximization problem in the general sharing model, a model that does not assume any hierarchical or other structured form of sharing. We perform extensive experiments to evaluate the performance of our greedy algorithm against other VM allocation algorithms.

2.1.2 Related Work

The sharing-aware VM maximization problem has been introduced by Sindelar *et al.* [86]. Their main contributions lie in the development of hierarchical sharing models for VM colocation for both the VM maximization and packing problems. They were the first to propose and investigate algorithms for solving the sharing-aware VM maximization problem. Their research is the closest to our research. Our research on the sharing-aware VM maximization problem focuses on the general sharing model which differs from the shared hierarchical models investigated by Sindelar *et al.* [86].

The sharing-aware VM maximation problem has been shown to be $\mathcal{NP}$-hard [86]. Thus, solving it optimally is not feasible and we have to resort to approximation algorithms, more specifically greedy algorithms. Greedy algorithms have been extensively investigated for different classical problems such as the knapsack [52], subset-sum, partition [56], as well as, facility location [91]. Greedy algorithms for VM provisioning and dynamic allocation in clouds have been investigated by Zaman and Grosu [106] [107] [108], who designed combinatorial auction-based mechanisms. Nejad *et al.* [69] designed a family of truthful greedy heuristic mechanisms for dynamic VM provisioning. Other research on greedy heuristics for VM provisioning focused on minimizing bandwidth-constraint VM placement in data centers [21], minimizing power consumption [92], federated clouds [62], and physical machine resourcing in clouds by implementing a mechanism design approach [63]. All these works focused on designing algorithms for provisioning VMs on multiple physical machines within a cloud computing system, and for allocation of VMs to users. Our work focuses on developing algorithms that maximize the revenue derived from hosting VMs on a single physical

machine that can be employed in making decisions at the physical machine level and work in conjunction with higher level resource management algorithms such as the ones discussed above.

Much of the work on page sharing focused on system development. Bugnion *et al.* [15] proposed the transparent page sharing technique for minimizing redundancy and memory overhead. Commercial systems such as VMWare's ESX Server [5] enable transparent page sharing in addition to other memory reclamation techniques [98]. Wood *et al.* [101] 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. The open source Xen hypervisor [6], has incorporated page sharing in Versions 4.0 and above for Hardware Virtual Machines (HVM) [76]. Gupta *et al.* [41] 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. 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.

2.1.3 Organization

The rest of the chapter is organized as follows. In Section 2.2, we describe the sharing-aware VM maximization problem. In Section 2.3, we present the design of our proposed efficiency metric and our greedy algorithm for the sharing-aware VM maximization problem. In Section 2.4, we characterize the properties of the proposed greedy algorithm. In Section 2.5, we evaluate our greedy algorithm against other VM allocation algorithms by extensive experiments. In Section 2.6, we summarize our results and present possible directions for future research.

2.2 Sharing-Aware VM Maximization

We now introduce the SAVMM (Sharing-Aware Virtual Machine Maximization) problem as it applies to a service provider resource environment.

We assume that a service provider maintains a server Ω, and a library Π of all memory pages required for each service it offers. Thus, the provider can identify and manage all memory pages required by a VM. We denote by π_i, the i-th memory page under the provider's management. Library Π is comprised of N distinct pages, i.e., $\Pi = \bigcup_{i=1}^{N} \{\pi_i\}$.

Each VM instance requires a set of memory pages which virtualizes a service offered by the provider. We denote by V_j, the VM instance j, by Λ_j, the set of indices of pages required by V_j, and by π_i^j, the i-th memory page required by VM V_j. We denote by $\mathcal{V}$, the set of "offline" VM instances that are possible candidates for allocation and hosting on server Ω. Given this setup, we define the SAVMM problem as follows:

> SAVMM problem: Given a set of M "offline" VMs $\mathcal{V}$ with each VM V_j yielding a revenue of p_j, determine a subset $\mathcal{V}^H \subset \mathcal{V}$ of VMs that can be allocated on the server, considering the memory capacity C of the server and the sharing of pages within library Π, such that the total revenue, $P = \sum_{j:V_j \in \mathcal{V}^H} p_j$, obtained by the provider is maximized.

The SAVMM problem may appear similar to the standard knapsack problem [52], but it is not the same, because the items (VMs) in the SAVMM problem are shared, while the items in the standard knapsack problem are not. Server Ω can host all the VMs in $\mathcal{V}$, if all the VMs in the set share the same pages and the total number of allocated pages does not exceed the capacity C of the server. The notation we use throughout the paper is summarized in Table 2.1.

2.3 Greedy Approximation Algorithm (G-SAVMM)

In this section, we present the design of our greedy algorithm for solving the SAVMM problem. The main idea used in the design of our greedy algorithm is to order the candidate VMs according to a metric which characterizes their potential for revenue and page-sharing

Table 2.1: SAVMM Notation.

Expression	Description
Π	Set of pages under provider's management.
N	Number of memory pages under provider's management.
$\mathcal{V}$	Set of "offline" VMs.
M	Number of "offline" VMs.
$\mathcal{V}^H$	Subset of VMs maximizing provider's revenue, $\mathcal{V}^H \subset \mathcal{V}$.
V_j	Virtual machine j.
π_i	The i-th memory page under provider's management.
π_i^j	The i-th memory page requested by VM V_j.
p_j	Revenue generated from allocating VM V_j.
Λ_j	Set of indices of pages requested by VM V_j.
Ω	Provider's server resource.
C	Memory capacity of server resource Ω.
k	Iteration number.
$\mathcal{E}_j^k$	Efficiency metric of VM V_j at iteration k.
S_j^k	Number of pages VM V_j shares with Ω at iteration k.

and then allocates them one by one according to the greedy order. The greedy metric and the greedy order is updated after allocating each VM. This represents an iteration in the greedy allocation process and will be denoted by k.

We first introduce the proposed metric we use in our greedy algorithm to establish the greedy order among the candidate VMs. At every iteration k, we order the candidate VMs, $V_j \in \mathcal{V}$, according to an efficiency metric, $\mathcal{E}_j^k$, defined as follows:

$$\mathcal{E}_j^k = \frac{p_j}{\sqrt{K_j - S_j^k + 1}}. \tag{2.1}$$

where j is the index corresponding to VM V_j, K_j is the number of pages required by VM V_j (i.e., $K_j = |\Lambda_j|$), and S_j^k is the number of shared pages between VM V_j and the VMs that are already allocated to the server. The efficiency metric $\mathcal{E}_j^k$ represents the relative value of allocating VM V_j onto Ω by considering the revenue p_j and the potential for sharing pages characterized by S_j^k, where k corresponds to the current greedy iteration. Prior to allocating the first VM onto Ω (i.e., at iteration $k = 0$), the efficiency metric for the "offline" set $\mathcal{V}$ of VMs is calculated using S_j^0 determined relative to the number of shared pages within all the VMs in $\mathcal{V}$ and not relative to the VMs that are allocated on the server. Once a VM

has been selected and allocated (i.e., for all iterations $k > 0$) then $\mathcal{E}_j^k$ is calculated using S_j^k, the number of shared pages between VM V_j and the VMs that are already allocated onto the server. As k increases and VMs are allocated onto Ω, we have $S_j^k \leq S_j^{k+1}$, that is S_j^k monotonically increases with k, for $k > 0$.

Since $\mathcal{E}_j^k$ needed to be well defined for all possible cases, we add 1 to the denominator. The reason for this is that, if VM V_j shares all its pages with another VM already allocated onto Ω, (i.e., $K_j = S_j^k, \forall k$), and if we do not consider adding 1 to the denominator of $\mathcal{E}_j^k = \dfrac{p_j}{\sqrt{K_j - S_j^k}}$, then the efficiency metric would produce an indeterminate value. We also reduce the magnitude of the sharing potential in the efficiency metric against the revenue by applying a square root to the denominator. Revenue has the largest effect when calculating the efficiency metric and therefore we want to capture as much effect as possible, while still allowing for the influence of page sharing. Similar metrics to our efficiency metric have been experimented with in studies focusing on the knapsack problem [52] and have led to good approximation ratios.

The G-SAVMM algorithm for solving the SAVMM problem are presented in Algorithms 1 and 2. G-SAVMM consists of two phases, executed one after the other: (i) a pre-processing phase, for $k = 0$ (Algorithm 1); and, (ii) a greedy allocation phase, for $k > 0$ (Algorithm 2). The input of G-SAVMM is an "offline" set of VMs $\mathcal{V}$. G-SAVMM determines the set $\mathcal{V}^H$ of VMs allocated onto the server, which is an approximate solution to the SAVMM problem.

In the pre-processing phase, G-SAVMM scans every VM V_j to identify its required pages, denoted by π_i^j. activePage() (Line 8) is a function that returns 1, if page π_i^j is requested, or returns 0 if page π_i^j is not requested. For every active page π_i^j the algorithm increments the variable K_j, the number of pages required by VM V_j, and A_i, the number of page π_i occurrences among all VMs in $\mathcal{V}$ (Lines 6 through 10). After calculating A, the algorithm determines the page from $\mathcal{V}$ that has the maximum number of requests which is identified by index $\tilde{i}$ (Line 11). If a VM requests page $\pi_{\tilde{i}}$, that VM will be placed in the

Algorithm 1 G-SAVMM: *Phase I*

1: **Input:** Set of "offline" VM instances ($\mathcal{V}$)
2: {*Phase I: Pre-processing*}
3: $\mathcal{V}^H \leftarrow \emptyset$
4: $A \leftarrow \mathbf{0}$
5: $\tilde{i}, \tilde{j}, k \leftarrow 0$
6: **for** $i = 1, \ldots, N$ **do**
7:     **for** $j = 1, \ldots, |\mathcal{V}|$ **do**
8:         **if** (activePage(π_i^j)) **then**
9:             $A_i = A_i + 1$
10:             $K_j = K_j + 1$
11: $\tilde{i} = \underset{i}{\operatorname{argmax}}\{A_i\}$
12: **for** $j = 1, \ldots, |\mathcal{V}|$ **do**
13:     **if** (activePage($\pi_{\tilde{i}}^j$)) **then**
14:         $\mathcal{V}^H = \mathcal{V}^H \cup \{V_j\}$
15: **for all** $j \in \mathcal{V}^H$ **do**
16:     **for** $i = 1, \ldots, N$ **do**
17:         **if** $(A_i > 1)$ & (activePage(π_i^j)) **then**
18:             $S_j^0 = S_j^0 + 1$
19: **for all** $j \in \mathcal{V}^H$ **do**
20:     $\mathcal{E}_j^0 = \dfrac{p_j}{\sqrt{K_j - S_j^0 + 1}}$
21: $\tilde{j} = \underset{j}{\operatorname{argmax}}\{\mathcal{E}_j^0\}$
22: $C = C - K_{\tilde{j}}$
23: $\mathcal{V}^H = \mathcal{V}^H \cap \{V_{\tilde{j}}\}$
24: $\mathcal{V} = \mathcal{V} \setminus \{V_{\tilde{j}}\}$
25: **for** $i = 1, \ldots, N$ **do**
26:     **if** (activePage($\pi_i^{\tilde{j}}$)) **then**
27:         activate(π_i)
28: $k \leftarrow 1$

subset $\mathcal{V}^H$ (Lines 12 through 14). The algorithm then calculates S_j^0, the number of shared pages among the VMs in $\mathcal{V}^H$, by identifying the active pages where $A_i > 1$, implying more than one VM is requesting memory page i (Lines 15 through 18). The efficiency metric (Eq. 2.1) is then calculated for all VMs in subset $\mathcal{V}^H$ (Lines 19 and 20). Once the VM with the largest efficiency value, denoted by $V_{\tilde{j}}$, is identified (Line 21), the server capacity C is reduced by the number of pages $K_{\tilde{j}}$ in $V_{\tilde{j}}$ (Line 22). Following the server capacity reduction, the subset $\mathcal{V}^H$ is modified by eliminating all VMs with the exception of VM $V_{\tilde{j}}$ (Line 23) and then VM $V_{\tilde{j}}$ is removed from $\mathcal{V}$ (Line 24). Following the allocation of VM $V_{\tilde{j}}$, every requested page $\pi_i^{\tilde{j}}$ is identified, and π_i is activated on the server resource through a function we denote as activate() (Lines 25 through 27). The activate() function implements the actions that need

Algorithm 2 G-SAVMM: *Phase II*

1: **Output:** Subset of VM instances maximizing provider revenue ($\mathcal{V}^H$)
2: {*Phase II: Greedy allocation*}
3: **while** $(C > 0)$ & $(|\mathcal{V}| > 0)$ **do**
4: $\quad flag \leftarrow 1$
5: $\quad$ **for** $i = 1, \ldots, N$ **do**
6: $\quad\quad$ **for** $j = 1, \ldots, |\mathcal{V}|$ **do**
7: $\quad\quad\quad$ **if** (activePage(π_i^j)) & (activePage(π_i)) **then**
8: $\quad\quad\quad\quad S_i^k = S_i^k + 1$
9: $\quad$ **for** $j = 1, \ldots, |\mathcal{V}|$ **do**
10: $\quad\quad \mathcal{E}_j^k = \dfrac{p_j}{\sqrt{K_j - S_j^k + 1}}$
11: $\quad \tilde{j} = \underset{j}{\operatorname{argmax}}\{\mathcal{E}_j^k\}$
12: $\quad$ **if** $C - (K_{\tilde{j}} - S_{\tilde{j}}^k) < 0$ **then**
13: $\quad\quad flag \leftarrow 0$
14: $\quad\quad \mathcal{V} = \mathcal{V} \setminus \{V_{\tilde{j}}\}$
15: $\quad$ **if** $(flag)$ **then**
16: $\quad\quad \mathcal{V}^H = \mathcal{V}^H \cup \{V_{\tilde{j}}\}$
17: $\quad\quad \mathcal{V} = \mathcal{V} \setminus \{V_{\tilde{j}}\}$
18: $\quad\quad C = C - (K_{\tilde{j}} - S_{\tilde{j}}^k)$
19: $\quad\quad$ **for** $i = 1, \ldots, N$ **do**
20: $\quad\quad\quad$ **if** (activePage($\pi_i^{\tilde{j}}$)) **then**
21: $\quad\quad\quad\quad$ activate(π_i)
22: $\quad k = k + 1$
23: $\Omega \leftarrow \mathcal{V}^H$
24: **exit**

to be performed in order to make a page active on the server. The implementation of this function is platform specific and is out of the scope of this study. The pre-processing phase is completed with an update of the iteration number k to 1 (Line 28).

The greedy allocation phase of G-SAVMM, (i.e., Algorithm 2 where iteration $k > 0$), is similar to the pre-processing phase (Algorithm 1 where iteration $k = 0$). At the beginning of the greedy phase, a test is performed to ensure that server capacity C is never exceeded and that there is at least one VM in $\mathcal{V}$ (Line 3). The differences between the two phases consists on how sharing is checked. In the first phase, the pages in each VM from set $\mathcal{V}^H$ are checked against the pages of all other VMs in $\mathcal{V}^H$ (Algorithm 1 Lines 15 through 18), while in the second phase the pages of each VM from $\mathcal{V}$ are checked against the active pages on server resource Ω (Algorithm 2, Lines 5 through 8). Every time a new VM V_j is inserted into $\mathcal{V}^H$, a new efficiency value is calculated (Lines 9 and 10) for every $k > 0$. A test is then

performed to recalculate the server capacity reduced by number of pages, K_j, less the shared pages, S_j^k, in common with the active pages on the server resource Ω.

If, by allocating VM V_j onto Ω, the capacity is exceeded, V_j is removed from the "offline" set $\mathcal{V}$ with no opportunity for inclusion in $\mathcal{V}^H$ (Lines 13 through 14). Else, VM V_j is allocated, the server capacity is reduced, and both $\mathcal{V}$ and $\mathcal{V}^H$ are updated accordingly (Lines 15 through 18). Next, pages within the library Π are updated to active, if they have not been already, relative to VM V_j (Lines 19 through 21) and the iterator k is updated (Line 22). Lastly, upon exiting the while loop, server Ω is allocated the subset $\mathcal{V}^H$ of VMs which represents the solution to the SAVMM problem (Line 23).

In the following, we present an example to show how G-SAVMM works. We consider a server with memory capacity $C = 10$ pages. There are twelve distinct pages in the library Π and four VM candidates for allocation onto the server. Figure 2.1 along with Table 2.2 show the details of each iteration k of G-SAVMM. The first column in both Figure 2.1 and Table 2.2 corresponds to the pre-processing phase, where a scan occurs for identical, requested pages within the set of VMs $\mathcal{V}$. In Figure 2.1, page π_i^j, ($i = 1, \ldots, 12$ and $j = 1, \ldots, 4$), is identified by a block labeled by 1, if it is requested, and by 0, otherwise. The aggregate value of blocks per VM corresponds to the total number of requested pages K_j. The highlighted blocks in Figure 2.1, correspond to identical pages found between the set of VMs, where $A_i > 1$. The maximum value in A corresponds to the page that is shared the most among all the pages in $\mathcal{V}$. The efficiency metric value is calculated for those VMs sharing this most shared page (i.e., the page with the greatest A_i). Based on the values given in Table 2.2, the highest efficiency metric, 4.772, is associated with V_4, and V_4 is selected for allocation to subset $\mathcal{V}^H$.

The next iteration of G-SAVMM, corresponding to the first iteration of the greedy phase, is illustrated in the second column of both Figure 2.1 and Table 2.2. In this iteration, a scan occurs for identical, requested pages between VMs and the active pages within library Π. Once the initial VM has been selected for allocation based on the efficiency metric, the provider activates all pages within Π requested by the selected VM. The active pages are

	$k=0$				$k=1$				$k=2$				$k=3$			
	p_j	K_j	S_j^0	$\mathcal{E}_j^0$	p_j	K_j	S_j^1	$\mathcal{E}_j^1$	p_j	K_j	S_j^2	$\mathcal{E}_j^2$	p_j	K_j	S_j^3	$\mathcal{E}_j^3$
V_1	–	–	–	–	6.00	3	0	3.000	6.00	3	1	3.464	6.00	3	1	3.464
V_2	6.50	5	3	3.753	6.50	5	2	**3.250**	–	–	–	–	–	–	–	–
V_3	7.00	5	2	3.500	7.00	5	1	3.131	7.00	5	2	**3.500**	–	–	–	–
V_4	6.75	3	2	**4.772**	–	–	–	–	–	–	–	–	–	–	–	–

Table 2.2: Efficiency Metric Calculation Example.

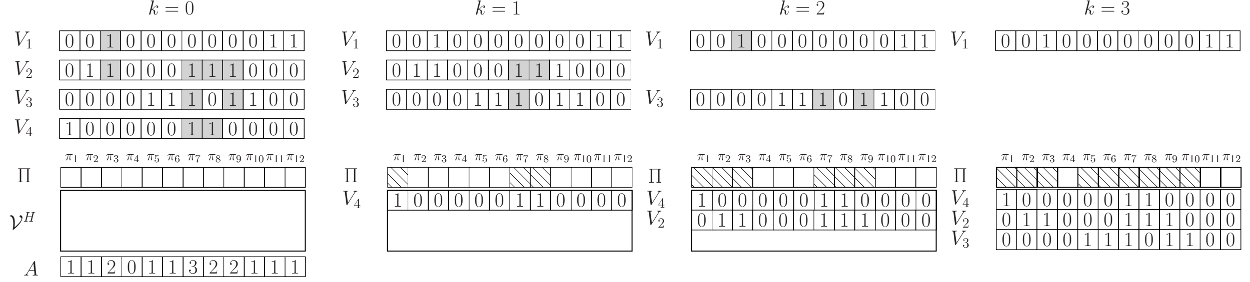


Figure 2.1: G-SAVMM: Execution Example.

identified by blocks with diagonal line filling underneath each page π_i from Π. The active pages correspond to all pages from V_4. The highlighted blocks for VMs in iteration $k = 1$, correspond to those pages that are identical to the active pages in Π. Even though V_1 does not share any active pages with the active pages in Π at $k = 1$, the efficiency metric is calculated and V_1 may be considered a candidate for allocation since at some $k > 1$, there may be active pages that are identical to pages in V_1 in later allocations. The largest efficiency value is 3.250, which corresponds to V_2, and the new server capacity is 6. VM V_2 consists of six pages, where three of them are shared with the active pages in Π and therefore do not have to be accounted for against the capacity. G-SAVMM proceeds until $k = 3$, where the remaining capacity is 1. The total revenue obtained by G-SAVMM is 20.25.

2.4 G-SAVMM Properties

In this section, we determine the approximation ratio of G-SAVMM and characterize its computational complexity. To develop insight into the properties of G-SAVMM, we design and analyze a worst-case VM instance as follows. Let $\mathcal{V}^W$ denote an instance of the SAVMM problem where VM $V_{\hat{j}}$ does not share any memory pages with the other VMs in $\mathcal{V}^W$. Then,

let at least one VM $V_{\hat{j}^c} \in \mathcal{V}^W$ be comprised of pages which are a complement set of pages to VM $V_{\hat{j}}$. In addition, let the remaining VMs in $\mathcal{V}^W$ be comprised of either a subset of pages in VM $V_{\hat{j}^c}$ or be equivalent to VM $V_{\hat{j}^c}$. In either case, the remaining VMs would be allocated onto Ω if $V_{\hat{j}^c}$ were to be allocated first since they all share the same memory pages and would not reduce capacity.

We investigate this instance on a server Ω with capacity C such that either VM $V_{\hat{j}}$ or VM $V_{\hat{j}^c}$ can be allocated, but not both. If VM $V_{\hat{j}^c}$ is allocated, then all remaining VMs in $\mathcal{V}^W \setminus \{V_{\hat{j}}\}$, will be allocated as well due to page sharing. Else, VM $V_{\hat{j}}$ is allocated and utilizes the server resource capacity enough to not allow any other VM to be allocated from $\mathcal{V}^W$. Our last consideration of the problem instance $\mathcal{V}^W$ corresponds to revenue. G-SAVMM is inherently sensitive to revenue values when calculating the efficiency metric. In the following theorem, we determine the approximation ratio for G-SAVMM based on the worst case instance $\mathcal{V}^W$.

Theorem 2.4.1. *The approximation ratio of G-SAVMM is M, where M is the number of VMs.*

Proof. Let the revenue obtained from an optimal solution be denoted as P^*. Then, let the optimal set of VMs which generate P^* from $\mathcal{V}^W$ be denoted by $\mathcal{V}^W_{OPT}$, where $P^* = \sum_{j:V_j \in \mathcal{V}^W_{OPT}} p_j$. Let the revenue obtained by G-SAVMM be denoted by P, and the set of VMs which generate revenue P from $\mathcal{V}^W$ be denoted by $\mathcal{V}^W_{GRD}$, $\mathcal{V}^W_{GRD} \subset \mathcal{V}^W$, where $P = \sum_{j:V_j \in \mathcal{V}^W_{GRD}} p_j$. At $k = 0$, allocate VM $V_{\hat{j}}$ onto Ω; admitting $\mathcal{E}^0_j < \mathcal{E}^0_{\hat{j}}$. Then, by Equation 2.1, $\frac{p_j}{\sqrt{K_j - S^0_j + 1}} < \frac{p_{\hat{j}}}{\sqrt{K_{\hat{j}} - S^0_{\hat{j}} + 1}}$. Since VM $V_{\hat{j}}$ does not share pages with VMs in $\mathcal{V}^W$, $S^0_{\hat{j}} = 0$, resulting in $\frac{p_j}{\sqrt{K_j - S^0_j + 1}} < \frac{p_{\hat{j}}}{\sqrt{K_{\hat{j}} + 1}}$, where

$$\frac{\sqrt{K_{\hat{j}} + 1}}{\sqrt{K_j - S^0_j + 1}} p_j < p_{\hat{j}} \tag{2.2}$$

establishes the lower bound for $p_{\hat{j}}$ selected according to our efficiency metric at $k = 0$. This implies that for any $p_{\hat{j}}$ greater than the established lower bound, VM $V_{\hat{j}}$ will be allocated first onto Ω from $\mathcal{V}^W$ by G-SAVMM. Considering the server utilization of $V_{\hat{j}}$ and capacity C, no other VM allocations can be performed and k stops at 0. Since $P = \sum_{j:V_j \in \mathcal{V}^W_{GRD}} p_j$, the aggregate revenue is expressed as $P = p_{\hat{j}}$.

Suppose that through an exhaustive search, the optimal value P^*, is calculated whereby VM $V_{\hat{j}^c}$ is allocated first onto Ω at $k = 0$. Since every remaining VM in $\mathcal{V}^W$ is comprised of a subset of pages in VM $V_{\hat{j}^c}$, not including VM $V_{\hat{j}}$, then the exhaustive search allocates all remaining VMs onto Ω from $k = 1$ to at most $k = M - 1$. Thus, the optimal revenue expressed as $P^* = \sum_{j:V_j \in \mathcal{V}^W_{OPT}} p_j$ implies $P^* = \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} p_j$. In order to determine the approximation ratio for this instance of SAVMM, we must show that $P^* \leq P\alpha$, where α is the multiplicative factor that will give the approximation ratio of G-SAVMM. Therefore,

$$\frac{P^*}{P} = \frac{\sum_{j:V_j \in \mathcal{V}^W_{OPT}} p_j}{\sum_{j:V_j \in \mathcal{V}^W_{GRD}} p_j} \tag{2.3}$$

$$= \frac{\sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} p_j}{p_{\hat{j}}} \tag{2.4}$$

By substituting p_j from Eq. 2.2, we further determine

$$\frac{P^*}{P} < \frac{\sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \frac{\sqrt{K_j - S_j^k + 1}}{\sqrt{K_{\hat{j}} + 1}} p_{\hat{j}}}{p_{\hat{j}}} \tag{2.5}$$

$$= \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \frac{\sqrt{K_j - S_j^k + 1}}{\sqrt{K_{\hat{j}} + 1}} \tag{2.6}$$

$$= \frac{1}{\sqrt{K_{\hat{j}} + 1}} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \sqrt{K_j - S_j^k + 1} \tag{2.7}$$

For $k > 0$ and $\forall$ VM $V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}$, S_j^k will be at least 1 when VM $V_{\hat{j}^c}$ is allocated first onto Ω. Every remaining VM in $\mathcal{V}^W \setminus \{V_{\hat{j}}\}$, will be allocated onto Ω, where the remaining

VMs may only consist of a single shared page with $V_{\hat{j}^c}$ in the worst case. Then,

$$\frac{P^*}{P} \leq \frac{1}{\sqrt{K_{\hat{j}}+1}} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \sqrt{K_j - 1 + 1} \tag{2.8}$$

$$= \frac{1}{\sqrt{K_{\hat{j}}+1}} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \sqrt{K_j} \tag{2.9}$$

Following the allocation of VM $V_{\hat{j}^c}$, we consider $M-1$ maximum number of VMs left to allocate in the optimal solution. Since VM $V_{\hat{j}^c}$ exists and is the complement page set to $V_{\hat{j}}$, then for N pages, $1 \leq K_{\hat{j}} \leq N-1$. In addition, since there exists at least 1 shared page index between Λ_j and $\Lambda_{\hat{j}^c}$ $\forall j : V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}$, then for $K_j = 1$ we have

$$\frac{P^*}{P} \leq \frac{(M-1)\sqrt{1}}{\sqrt{K_{\hat{j}}+1}} = \frac{M-1}{\sqrt{2}} \leq M-1 < M \tag{2.10}$$

Therefore, $\frac{P^*}{P}$ is bounded by $\alpha = M$, which results in an approximation ratio of M for the G-SAVMM algorithm. □

We now investigate the time complexity of G-SAVMM. The running time is dominated by the second phase, the greedy phase. The while-loop (Algorithm 2 Line 3) may execute a maximum of $M-1$ iterations since one VM has already been inserted into $\mathcal{V}^H$. 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 Ω (Algorithm 2 Lines 5 through 8). 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 Ω, 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-SAVMM has an asymptotic running time of $O(NM^2)$ which is linear in the total number of pages and quadratic in the total number of VMs in the set of "offline" VMs.

2.5 Experimental Results.

In this section, we perform extensive experiments investigating the performance of G-SAVMM against other VM allocation algorithms considering their obtained revenue and the utilization of the server's memory.

2.5.1 Experimental Setup

We perform our experiments on a 2.4 GHz Intel Core® i7-3630 QM CPU 64-bit system. All simulations are implemented in C++ and are compiled with GCC Version 4.9.0. Our evaluation of G-SAVMM consists of comparing its performance against two other VM allocation algorithms: (i) Highest Revenue (HR-Oblivious); and, (ii) Maximum Shared Pages (MS-Sharing). The first allocation algorithm, HR-Oblivious, is a greedy algorithm which allocates VMs in decreasing order of their revenue and is page sharing oblivious. The second allocation algorithm, MS-Sharing, is a greedy algorithm which allocates VMs in decreasing order of their number of shared pages. The page sharing consideration in MS-Sharing mirrors that of G-SAVMM, but it does not take into account the revenue.

Our environment assumes page sharing within each simulation we evaluate. We consider the degree of sharing among the VMs and categorize the SAVMM instances into four categories, called *sharing stratifications*: (i) *Low-Share* (no greater than 20% of the active pages on the server are shared with VMs); (ii) *Mid-Share* (no greater than 50% of the active pages on the server are shared with VMs); (iii) *High-Share* (no greater than 80% of the active pages on the server are shared with VMs); and, (iv) *Full-Share* (approx. all active pages on the server are shared with VMs). Our experiments consist of 1000 simulations per sharing stratification. In our simulations, each sharing stratification is defined within the following ranges: (i) 15%−20% for Low-Share; (ii) 38%−50% for Mid-Share; (iii) 70%−80% for High-Share; and, (iv) 92%−99% for Full-Share.

Each instance of SAVMM considered in the simulation consists of 10 VMs. Each VM is assigned a revenue value randomly ranging from $1 to $20. The number of pages is also generated randomly with a maximum of 1000 pages possible per VM. Our server capacity C is fixed at 60% of the total number of pages for each simulation. Based on our experiments, operating at 60% capacity provides enough resources to accommodate a wide variety of simulations.

Our criterion for identifying the best performing algorithm is based on the calculation of revenue ratios. In our experiments, we execute the three greedy algorithms HR-Oblivious, MS-Sharing and G-SAVMM on instances of the SAVMM problem. The set of VMs therein will vary in their revenue generated from being hosted according to the range specified in the previous paragraph. Comparing then aggregating the actual values of the revenue generated by each of these greedy algorithms over a number of simulations is artificial since it may mislead the attainment of a defined value of revenue. Instead, we compare the revenues generated by each greedy algorithm over the maximum revenue generated in that instance and aggregate those ratios for a specific number of simulations. For example, suppose after simulating an instance of the SAVMM problem, HR-Oblivious generates a revenue value of 100, MS-Shaing generates a revenue value of 200 and G-SAVMM generates a revenue value of 250. Then, the maximum revenue generated in that instance would be 250. The calculated revenue ratios would be .4, or $\frac{100}{250}$, for HR-Oblivious, .8, or $\frac{200}{250}$, for MS-Sharing and 1, or $\frac{250}{250}$, for G-SAVMM. The revenue ratios indicate each greedy algorithm's proximity to the maximum revenue attained in that instance. These revenue ratios will never be larger than 1 for any of the algorithms in any instance. By aggregating these ratios over 1000 simulations, we identify the best performing algorithm as the one with the highest revenue ratio aggregate. The revenue ratio aggregate for each algorithm over the course of 1000 simulations will never be larger than 1000. In addition, these 1000 simulations are performed for each sharing stratification to determine the best performing algorithm under the various sharing scenarios.

2.5.2 Analysis of Results

We now compare the performance of G-SAVMM against both HR-Oblivious and MS-Sharing algorithms. In Figure 2.2, we plot the aggregate revenue ratios of all three algorithms under different sharing stratifications. For sharing stratifications Low-Share, Mid-Share and High-Share, G-SAVMM outperforms both HR-Oblivious and MS-Sharing algorithms. In Low-Share, G-SAVMM resulted in either the revenue maximum over or equal to the revenues

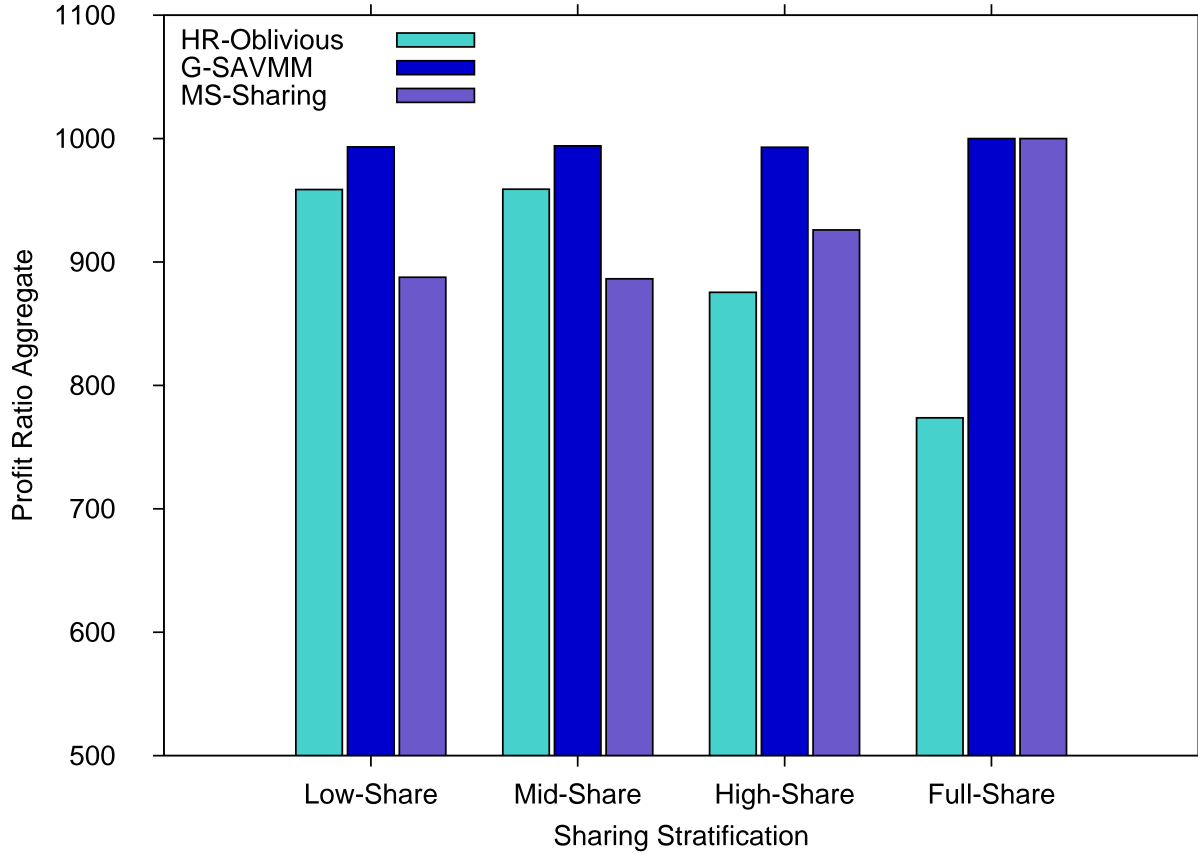


Figure 2.2: G-SAVMM: Revenue Ratios vs. Sharing Stratifications.

obtained using HR-Oblivious and MS-Sharing, in 852 of the 1000 simulations. In Mid-Share, G-SAVMM resulted in either the revenue maximum over or equal to the revenues obtained using HR-Oblivious and MS-Sharing in 875 of the 1000 simulations. In High-Share, G-SAVMM resulted in either the revenue maximum over or equal to the revenues obtained using HR-Oblivious and MS-Sharing in 816 of the 1000 simulations. In the Low-Share and Mid-Share stratifications, our experiments have shown that HR-Oblivious outperforms MS-Sharing. In the High-Share and Full-Share stratifications, our experiments have shown that MS-Sharing outperforms HR-Oblivious. As the sharing potential in the stratification increases, MS-Sharing generates an increased revenue since more VMs may be allocated. In the Full-Share stratification, G-SAVMM and MS-Sharing generate the same revenue resulting in a revenue maximum in 1000 out of 1000 simulations. Based on our results, G-SAVMM attains a revenue ratio aggregate of: (i) 993.2759 for Low-Share; (ii) 994.0514 for Mid-Share; (iii) 992.9242

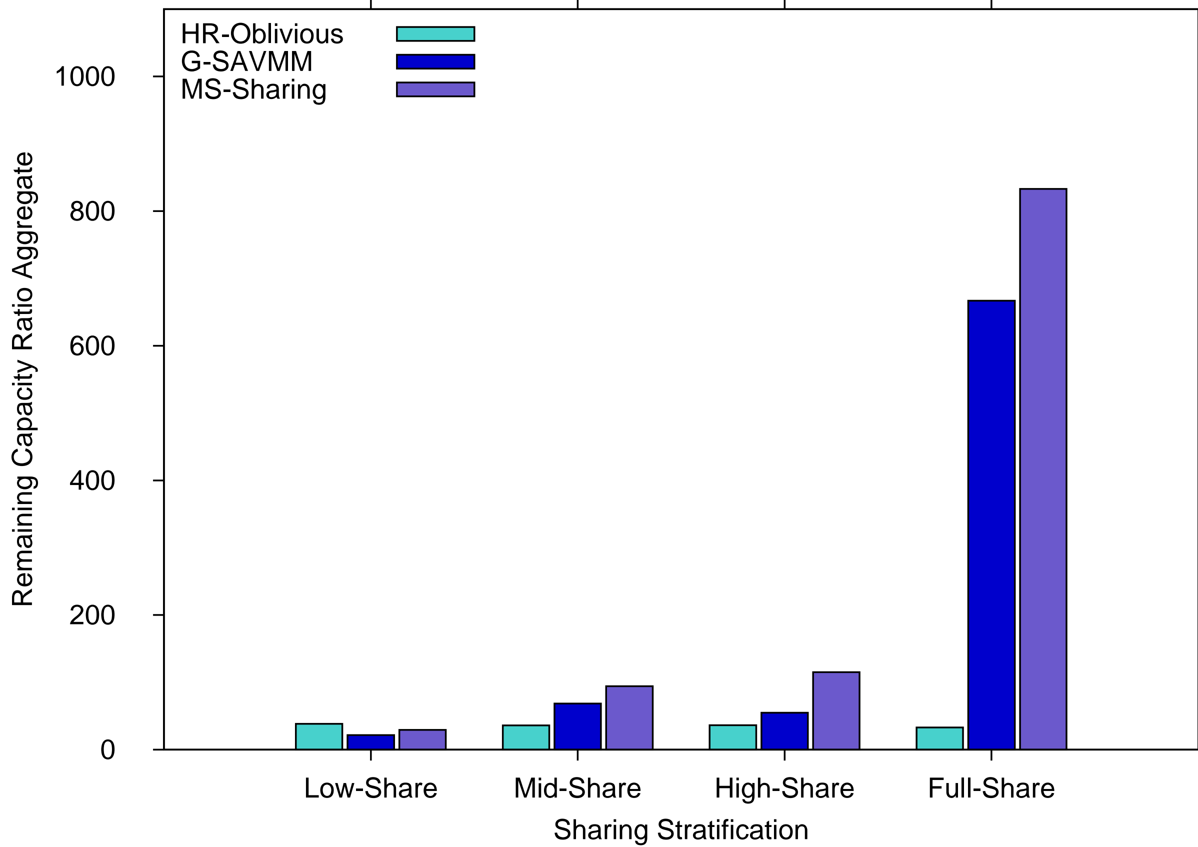


Figure 2.3: G-SAVMM: Capacity Ratios vs. Sharing Stratifications.

for High-Share; and, (iv) 1000 for Full-Share. When a simulation contains VMs with full-sharing potential, G-SAVMM or MS-Sharing returns the same result. When the simulated instance consists of VMs with less opportunity to share pages, G-SAVMM is the preferred algorithm with respect to revenue maximization. Therefore, according to our experiments, G-SAVMM should be the chosen algorithm for solving SAVMM. In Figure 2.3, we plot the aggregate remaining memory capacity ratios, after the VMs have been allocated, for all three algorithms under different sharing stratifications. We have shown the efficacy of G-SAVMM for revenue maximization now we show that from the point of view of preserving resources, G-SAVMM also performs well. The remaining capacities are slightly larger for HR-Oblivious in the Low-Share and are larger for MS-Sharing in Mid-Share and High-Share. The significant differences between these algorithms occur in the Full-Share stratification. MS-Sharing dominates the amount of unused capacity with G-SAVMM also experiencing a higher unused

capacity; albeit not as significant as MS-Sharing, yet well above HR-Oblivious. Therefore, choosing G-SAVMM as the algorithm for solving SAVMM leads to a considerable saving of memory which can be utilized for other purposes.

2.6 Summary

We designed a sharing-aware greedy approximation algorithm (G-SAVMM) for solving the sharing-aware VM maximization problem. We showed that G-SAVMM is a M-approximation algorithm, where M is the number of VM instances. The experimental results show that G-SAVMM outperforms two other VM allocation algorithms in terms of generated revenue.

CHAPTER 3: MULTI-RESOURCE VM MAXIMIZATION

3.1 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) that define Quality of Service (QoS) guarantees. 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 have been developed and presented in the research literature. An example of such a platform is Mesos [43], which allows sharing of cluster resources among various cluster computing frameworks. A more recent example is Borg [96], Google's large scale cluster management system, which schedules requests on what may well be the largest service infrastructure in the world [67]. While these systems represent significant contributions to resource management in large scale systems, both works identify extensions in search of greater efficiency, that is, leveraging more information about resource offerings in the case of Mesos and in the case of Google's next-generation container management system, Kubernetes [39].

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 chapter, 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 [6] manages the sharing of pages at the application level, whereas IBM's PowerVM [23] manages page sharing at the logical partition level. If service providers can adapt their pricing for services on the utilization and sharing of 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 chapter, we address the multi-resource sharing-aware virtual machine maximization (MSAVMM) problem. The MSAVMM 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. The solution to this 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 maximization problem. The classical knapsack algorithms [52] 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 MSAVMM, 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 MSAVMM.

3.1.1 Our Contribution

We formulate MSAVMM 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 MSAVMM, we propose and design a greedy approximation algorithm for solving MSAVMM. 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 [83]. To the best of our knowledge, no multi-resource sharing-aware greedy approximation algorithms for solving the MSAVMM problem have been proposed in the research literature to date.

3.1.2 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. Wei *et al.* [100] investigated physical machine (PM) provisioning for Infrastructure as a Service (IaaS) clouds and argued that service providers should offer flexible resource combinations when hosting VMs. Their research also suggested that the use of a single resource-type provisioning scheme by cloud providers when multiple resource types are requested, leads to PM over-provisioning and limits resource utilization. Therefore, the authors have developed a dynamic multiple resource provisioning approach which optimizes resource utilization for IaaS cloud providers. Minarolli and Freisleben [66] investigated the allocation of VMs requesting multiple resource types in IaaS clouds. Their proposal employs a utility function which maximizes the quality of service (QoS) and the service provider's revenue through resource managers running on PMs. The use of auction-based mechanisms for the VM allocation problem considering multiple resource types has been investigated by several researchers. Zaman and Grosu [107] designed combinatorial auction-based greedy mechanisms for VM provisioning and allocation in clouds. Nejad *et al.* [70] proposed a family of truthful greedy heuristic mechanisms for dynamic VM provisioning for the auction-based model. Mashayekhy *et al.* [64] formulated a PTAS mechanism for the provisioning and allocation of heterogeneous cloud 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.

Dominant Resource Fairness (DRF) has received significant attention in establishing fair resource allocation when multiple resources are requested. Ghodsi *et al.* [33] 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. Dolev *et al.* [27] considered an alternative fairness criterion for allocation of multiple resources and proved that fairness is guaranteed by any combination of user requests under multiple bottlenecks. Wang *et al.* [99] extended the DRF policy concept to multiple heterogeneous server resources in a cloud environment. Wong *et al.* [47] investigated the fairness-efficiency trade-off of allocating multiple resources in data-centers. Even though the above works considered multiple resource types, they did not consider page sharing when deciding the allocation.

The majority of research on page sharing focused on developing page sharing systems. Bugnion *et al.* [15] proposed the transparent page sharing technique for minimizing redundancy and memory overhead. Wood *et al.* [101] 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 [5] enable transparent page sharing in addition to other memory reclamation techniques [98]. The open source Xen hypervisor [6], has incorporated page sharing in Versions 4.0 and above for Hardware Virtual Machines (HVM) [76]. Gupta *et al.* [41] 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.* [71] 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.* [86], who introduced and investigated VM packing and maximization problems under hierarchical sharing models. They developed several algorithms to solve these problems assuming hierarchical sharing models. Our research on the sharing-aware VM maximization problem focuses on the general sharing model which differs from Sindelar *et al.* [86]. 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 Chapter 2 and our previous paper [78], 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 [86] and [78] do not consider the allocation of multiple types of resources.

3.1.3 Organization

The rest of this chapter is organized as follows. In Section 3.2, we define the multi-resource sharing-aware VM maximization problem. In Section 3.3, we formulate MSAVMM problem as a binary multilinear program. In Section 3.4, we present our proposed greedy algorithm for solving the MSAVMM problem. In Section 3.5, we determine the approximation ratio of our proposed greedy algorithm. In Section 3.6, we describe the experimental setup and investigate the performance of our proposed algorithm by performing extensive experiments on Google Cluster Usage trace data [83]. In Section 3.7, we summarize our results and present directions for future research.

3.2 Multi-Resource Sharing-Aware VM Maximization

We now present the MSAVMM (Multi-resource Sharing-Aware Virtual Machine 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 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 Π, 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* [71]. 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 Ω 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 Π, we denote by π^i, the i-th memory page in Π. We assume that Π manages a finite number N of pages, i.e., $\Pi = \bigcup_{i=1}^{N}\{\pi^i\}$. The notation used in this chapter is presented in Table 3.3.

We assume that there is a set $\mathcal{V}$ 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 $V_j \in \mathcal{V}$, and by π_j^i, the i-th memory page required by VM V_j. The provider allocates and instantiates a subset of VMs, denoted by $\mathcal{V}^H$, onto Ω. The allocation should be determined based on how efficient in terms of revenue it 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

Table 3.3: MSAVMM Notation.

Expression	Description
Π	Library of pages under provider's management.
N	Number of memory pages under provider's management.
V_j	Virtual machine j.
$\mathcal{V}$	Set of "offline" VMs.
M	Number of "offline" VMs.
$\mathcal{V}^H$	Subset of VMs maximizing provider's revenue, $\mathcal{V}^H \subset \mathcal{V}$.
π^i	The i-th memory page under provider's management.
s_j^k	Number of pages VM V_j shares at iteration k.
A^i	Shared page counter among M VMs for the i-th page.
π_j^i	The i-th memory page requested by VM V_j.
p_j	revenue generated from allocating VM V_j.
Ω	Provider's PM server resource.
C^m	Memory capacity (RAM) of PM server resource Ω (GB).
C^u	vCPU capacity of PM server resource Ω (cores).
C^s	Storage capacity of PM server resource Ω (GB).
R	Subset of PM resource types u and s, $R = \{u, s\}$.
q_j^m	Requested amount of memory (RAM) by V_j (GB).
q_j^u	Requested number of vCPU by V_j (cores).
q_j^s	Requested amount of storage by V_j (GB).
E_j^k	Efficiency metric of VM V_j at iteration k.
$\mathcal{P}(\mathcal{V})$	Power set of the set of "offline" virtual machines $\mathcal{V}$.
$\mathcal{I}$	Index of "offline" virtual machines in $\mathcal{P}(\mathcal{V})$.

denoted by C^u; 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 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^u 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., $q_j^m \leq C^m$, $q_j^u \leq C^u$, and $q_j^s \leq C^s$, for any $j = 1, \ldots, M$). We now introduce the MSAVMM problem as follows:

MSAVMM problem: Given a set of M "offline" VMs $\mathcal{V}$, 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^u, and amount of storage, q_j^s, determine a subset $\mathcal{V}^H \subset \mathcal{V}$ of VMs that can be allocated onto server Ω, considering the PM memory capacity C^m, the available number of vCPUs, C^u, the PM storage capacity, C^s, and the sharing

of memory pages, such that the total revenue, $P = \sum_{j:V_j \in \mathcal{V}^H} p_j$, obtained by the provider is maximized.

The formulation of MSAVMM 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 [52], 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 MSAVMM, leading to revenue loses. MSAVMM 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.

3.3 Binary Multilinear Program Formulation

In this section, we propose a multilinear programming formulation of MSAVMM. 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 MSAVMM problem as a binary multilinear program (BMP), called BMP-MSAVMM, as follows:

$$\text{maximize: } P = \sum_{j:V_j \in \mathcal{V}} p_j x_j \tag{3.1}$$

$$\text{subject to: } \sum_{j:V_j \in \mathcal{V}} q_j^r x_j \leq C^r, \ \forall\, r \in \mathcal{R} \tag{3.2}$$

$$\sum_{\mathcal{I} \in \mathcal{P}(\mathcal{V})} (-1)^{(|\mathcal{I}|+1)} \sigma_{\mathcal{I}} \prod_{k \in \mathcal{I}} x_k \leq C^m \tag{3.3}$$

$$x_j \in \{0,\ 1\}, \ \forall\, j:\ V_j \in \mathcal{V}. \tag{3.4}$$

The solution to this problem is a boolean decision vector $\mathbf{x} \in \{0,1\}^M$, 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 (3.1) corresponds to revenue, P, aggregated from the subset of instantiated VMs. The constraint in Equation (3.2) 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 (3.3) 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 (3.4) expresses the fact that x_j's are binary decision variables.

The constraint in Equation (3.3) 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 3.4. We consider that only a total of 16 different pages (π^1, π^2, ..., π^{16}) are going to be requested by these VMs.

V_j	$< q_j^m,\ q_j^u,\ q_j^s,\ p_j >$	$\lvert\mathcal{I}\rvert = 1$	$\lvert\mathcal{I}\rvert = 2$	$\lvert\mathcal{I}\rvert = 3$	$\lvert\mathcal{I}\rvert = 4$
V_1	$< 4,\ 1,\ 2,\ 0.95 >$	$\sigma_1 : 4$	$\sigma_{12} : 3$	$\sigma_{123} : 2$	$\sigma_{1234} : 1$
V_2	$< 5,\ 1,\ 2,\ 1.05 >$	$\sigma_2 : 5$	$\sigma_{13} : 3$	$\sigma_{124} : 2$	
V_3	$< 7,\ 2,\ 2,\ 1.35 >$	$\sigma_3 : 7$	$\sigma_{14} : 3$	$\sigma_{134} : 2$	
V_4	$< 14,\ 4,\ 2,\ 1.80 >$	$\sigma_4 : 14$	$\sigma_{23} : 2$	$\sigma_{234} : 1$	
			$\sigma_{24} : 4$		
			$\sigma_{34} : 5$		

Table 3.4: VM Characteristics and Sharing Relationships.

The pages requested by each of the four VMs are given in Figure 3.1. For example V_1 requests a total of 4 pages (pages marked with hatched boxes in Figure 3.1, 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 π^2 is required by V_1, V_2 and V_3, and

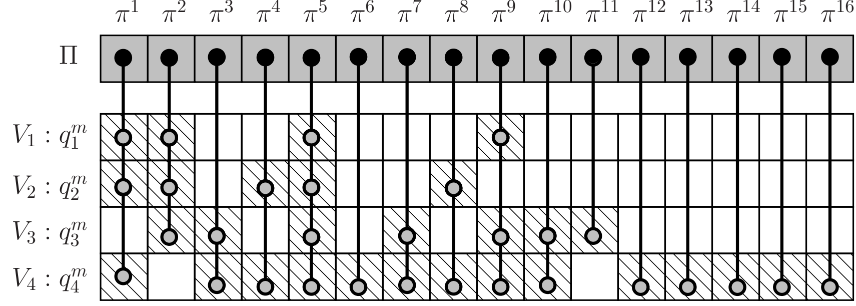


Figure 3.1: Page Sharing Among VMs.

thus, the hatched boxes corresponding to it in the three VMs are connected with a vertical bold line indicating that π^2 is shared among the three VMs.

We now show how the *sharing parameter* $\sigma_{\mathcal{I}}$ used in constraint (4.7) is determined. We denote by $\mathcal{P}(\mathcal{V})$ the power set of the set $\mathcal{V}$ of available VMs and by $\mathcal{I}$ an element of the power set $\mathcal{V}$. The sharing parameter represents the number of shared pages among the VMs in set $\mathcal{I}$. For example for $\mathcal{I} = \{1, 2, 3\}$, $\sigma_{123} = 2$, that is, two pages, π^2 and π^5, are shared among the three VMs considered. We calculate the sharing parameter $\sigma_{\mathcal{I}}$ for all the sets $\mathcal{I}$ of the power set $\mathcal{P}(\mathcal{V})$ and organize them by the cardinality of $\mathcal{I}$ in Table 3.4. When $|\mathcal{I}| = 1$, the sharing parameter $\sigma_{\mathcal{I}}$ represents the amount of memory resource q_j^m in number of pages requested by V_j, that is, $\sigma_j = q_j^m$. 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., those pages which are required to instantiate a subset of VMs, are managed only once in Π, and are available to be shared among requesting VMs. To calculate the number of unique pages in Equation (3.3) we need to introduce an adjustment parameter, $(-1)^{(|\mathcal{I}|+1)}$, which adjusts the calculation of the number of unique pages according to the cardinality of $\mathcal{I}$. By referencing the data in Table 3.4, 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:

$$\begin{aligned}
&(+1)(\sigma_1 + \sigma_2 + \sigma_3 + \sigma_4) + \\
&(-1)(\sigma_{12} + \sigma_{13} + \sigma_{14} + \sigma_{23} + \sigma_{24} + \sigma_{34}) + \\
&(+1)(\sigma_{123} + \sigma_{124} + \sigma_{134} + \sigma_{234}) + (-1)(\sigma_{1234}) \leq C^m
\end{aligned} \tag{3.5}$$

By substituting the values for $\sigma_{\mathcal{I}}$ from Table 3.4 and performing the calculation above in Equation 3.5, we arrive at 16 unique pages which is consistent with the number of grey boxes, i.e., those pages required to be managed by Π in order to instantiate all four VMs, from Figure 3.1. 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 (3.3) 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 $\sigma_{\mathcal{I}}$, and the unique page adjustment parameter $(-1)^{(|\mathcal{I}|+1)}$.

In order to solve BMP-MSAVMM, we use the AMPL [30] mathematical programming framework and an open-source solver, Couenne [8], capable of producing exact solutions for BMP-MSAVMM. Couenne employs a branch & bound algorithm for solving mixed integer nonlinear programs; which lends to our multilinear binary formulation. The constraint in Equation (3.3) of BMP-MSAVMM makes it a mixed integer nonlinear program. We submit our model, data, and preference for solver to NEOS [24], an internet-based optimization service, which solves BMP-MSAVMM.

We solved the BMP-MSAVMM instance in the example given in Table 3.4, and the solution consists of instantiating V_1, V_2 and V_4, generating \$4.05 as the optimal revenue. The execution takes approximately 9.6 milliseconds. The execution time increases dramatically for larger instances, for example for an instance of MSAVMM with 20 VMs and 256 pages, the execution time exceeds 20 minutes. These solvers can only be used for solving small instances of MSAVMM; for solving large instances of MSAVMM, we need to rely on

approximation algorithms. BMP-MSAVMM problem is a new and more complex variant of the multidimensional knapsack problem which is strongly $\mathcal{NP}$-hard [52]. Therefore, we infer that BMP-MSAVMM is also strongly $\mathcal{NP}$-hard.

3.4 Greedy Approximation Algorithm (G-MSAVMM)

In this section, we present the design of our greedy algorithm for solving the MSAVMM 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:

$$E_j^k = \frac{p_j}{\sqrt{\sum_{r \in R} \frac{q_j^r}{C^r} + \frac{q_j^m - s_j^k + 1}{C^m}}} \tag{3.6}$$

The efficiency metric E_j^k represents the relative value of allocating VM V_j onto Ω 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 normalized resource.

The initial step in the allocation process, at iteration $k = 0$, selects the first VM to be allocated onto Ω, 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 $\mathcal{V}^H$).

Algorithm 3 G-MSAVMM: *Phase I*

1: **Input:** Set of offline VM instances ($\mathcal{V}$)
2: $\{$*Phase I: Initial VM Allocation based on the potential for page sharing in* $\mathcal{V}\}$
3: $[A] \leftarrow \mathbf{0}$
4: $\mathcal{V}^H \leftarrow \emptyset$
5: $\tilde{i}, \tilde{j} \leftarrow 0$
6: **for** $i = 1, \ldots, N$ **do**
7:     **for all** $j: V_j \in \mathcal{V}$ **do**
8:         **if** (activePage(π_j^i)) **then**
9:             $A^i = A^i + 1$
10: $\tilde{i} = \underset{i}{\operatorname{argmax}}\{A^i\}$
11: **for all** $j: V_j \in \mathcal{V}$ **do**
12:     **if** (activePage($\pi_j^{\tilde{i}}$)) **then**
13:         $\mathcal{V}^H = \mathcal{V}^H \cup \{V_j\}$
14: **for** $i = 1, \ldots, N$ **do**
15:     **for all** $j: V_j \in \mathcal{V}^H$ **do**
16:         **if** $(A^i > 1)$ **and** (activePage(π_j^i)) **then**
17:             $s_j^0 = s_j^0 + 1$
18: **for all** $j: V_j \in \mathcal{V}^H$ **do**
19:     $E_j^0 = \dfrac{p_j}{\sqrt{\sum\limits_{r \in R} \frac{q_j^r}{C^r} + \frac{q_j^m - s_j^0 + 1}{C^m}}}$
20: $\tilde{j} = \underset{j}{\operatorname{argmax}}\{E_j^0\}$
21: $\mathcal{V}^H = \{V_{\tilde{j}}\}$
22: $\mathcal{V} = \mathcal{V} \setminus \{V_{\tilde{j}}\}$
23: $[C^m, C^u, C^s] = [C^m, C^u, C^s]$ - $[q_{\tilde{j}}^m, q_{\tilde{j}}^u, q_{\tilde{j}}^s]$
24: **for** $i = 1, \ldots, N$ **do**
25:     **if** (activePage($\pi_{\tilde{j}}^i$)) **then**
26:         allocatePage(π^i)
27: $k \leftarrow 1$

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-MSAVMM, for solving the MSAVMM problem. The algorithm is presented in phases by Algorithm 3 and Algorithm 4. G-MSAVMM consists of two phases distinguished by how the potential for sharing is determined. In the first phase (Algorithm 3), the potential for page sharing is determined considering the sharing among all the VMs in the offline set of VMs, $\mathcal{V}$. In the second phase (Algorithm 4), 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 Ω.

The input to G-MSAVMM in Algorithm 3 is a set of "offline" VMs, $\mathcal{V}$. First, G-MSAVMM initializes the shared page counter array, A, (Line 3), the subset of allocated VMs, $\mathcal{V}^H$, (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 π^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 through 9). Function, activePage() (Line 8), determines whether memory page π_j^i from VM V_j is requested. If π_j^i is requested, then activePage() returns 1, otherwise it returns 0. The activePage() function uses information from a pre-processing stage in which the cloud provider uses a set of staging PMs to instantiate the requested VMs and determine their memory fingerprints. The cloud provider could implement a memory fingerprinting technique similar to the one presented by Wood *et al.* [101]. 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 $\mathcal{V}^H$ (Lines 10 through 13). The next task is to calculate the number of shared pages for each $V_j \in \mathcal{V}^H$. 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^0 is updated (Lines 14 through 17). Then, our proposed efficiency metric is calculated for each $V_j \in \mathcal{V}^H$ (Lines 18 and 19), where the VM corresponding to the highest efficiency value is identified by index $\tilde{j}$ (Line 20). $V_{\tilde{j}}$ is then allocated to $\mathcal{V}^H$ (Line 21) and removed from $\mathcal{V}$ (Line 22). The three PM resource capacities are then reduced by the amount of resource requests from $V_{\tilde{j}}$ (Line 23). Note, we do not add the shared pages $s_{\tilde{j}}^k$ back into the PM resource capacity C^m since at $k = 0$, $V_{\tilde{j}}$ 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 activePage() are then allocated onto PM server Ω through allocatePage() (Lines 24 through 26). After the initial allocation according to the potential for sharing, k is updated to 1 (Line 27).

The second phase of G-SAVMM in Algorithm 4 starts by checking the availability of resources of each type on the server Ω (Line 3). A variable *flag* is set to 1 (Line 4)

Algorithm 4 G-MSAVMM: *Phase II*

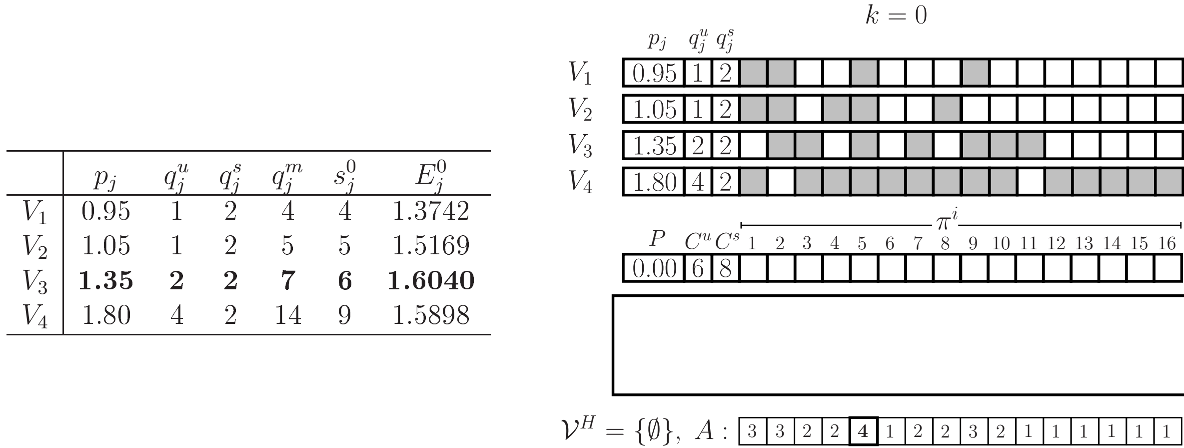
1: {*continued ...*}
2: {*Phase II: VM Allocation based on explicit page sharing in* $\mathcal{V}^H$}
3: **while** ($[C^m, C^u, C^s] > 0$) **and** ($|\mathcal{V}| > 0$) **do**
4: $flag \leftarrow 1$
5: **for** $i = 1, \ldots, N$ **do**
6: **for all** $j : V_j \in \mathcal{V}$ **do**
7: **if** (activePage(π_j^i)) **and** (activePage(π^i)) **then**
8: $s_j^k = s_j^k + 1$
9: **for all** $j : V_j \in \mathcal{V}$ **do**
10: $E_j^k = \dfrac{p_j}{\sqrt{\sum\limits_{r \in R} \frac{q_j^r}{C^r} + \frac{q_j^m - s_j^k + 1}{C^m}}}$
11: $\tilde{j} = \operatorname{argmax}_j \{E_j^k\}$
12: **if** $(C^m - (q_{\tilde{j}}^m - s_{\tilde{j}}^k) < 0)$ **or** $(C^u - q_{\tilde{j}}^u < 0)$ **or** $(C^s - q_{\tilde{j}}^s < 0)$ **then**
13: $flag \leftarrow 0$
14: $\mathcal{V} = \mathcal{V} \setminus \{V_{\tilde{j}}\}$
15: **if** ($flag$) **then**
16: $\mathcal{V}^H = \mathcal{V}^H \cup \{V_{\tilde{j}}\}$
17: $\mathcal{V} = \mathcal{V} \setminus \{V_{\tilde{j}}\}$
18: $[C^m, C^u, C^s] = [C^m, C^u, C^s]$ - $[(q_{\tilde{j}}^m$ - $s_{\tilde{j}}^k), q_{\tilde{j}}^u, q_{\tilde{j}}^s]$
19: **for** $i = 1, \ldots, N$ **do**
20: **if** (activePage($\pi_{\tilde{j}}^i$)) **then**
21: allocatePage(π^i)
22: $P = P + p_j$
23: $k = k + 1$
24: $\Omega \leftarrow \mathcal{V}^H$
25: **exit**

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 required by $V_j \in \mathcal{V}$ which are already allocated, the shared page counter s_j^k is updated (Lines 5 through 8). Next, the efficiency metric is calculated for all $V_j \in \mathcal{V}$ (Lines 9 and 10) and the VM with the highest efficiency value is identified by the index $\tilde{j}$ (Line 11). Prior to allocating $V_{\tilde{j}}$, a check must determine if the allocation will fully deplete any of the multiple types of resources provided by the PM

(Line 12). If any of those resources are fully depleted, the $flag$ variable is set to 0 (Line 13) and $V_{\tilde{j}}$ is removed from $\mathcal{V}$ (Line 14) since it cannot be allocated. If $flag$ is still 1, then $V_{\tilde{j}}$ is stored in $\mathcal{V}^H$ and removed from $\mathcal{V}$ (Lines 16 and 17). The capacities of each of the multiple resources of the PM are then reduced according to the resources requested by $V_{\tilde{j}}$ (Line 18), that is, the PM memory capacity C^m is reduced by $q_{\tilde{j}}^m$ and $s_{\tilde{j}}^k$ 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 $V_{\tilde{j}}$, if they are not already allocated, are then allocated by calling allocatePage() (Lines 19 through 21). Next, the revenue p_j from allocation of $V_j \in \mathcal{V}^H$ is accumulated into P (Line 22). Lastly, the iteration count k is incremented (Line 23) 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 Ω (Line 24).

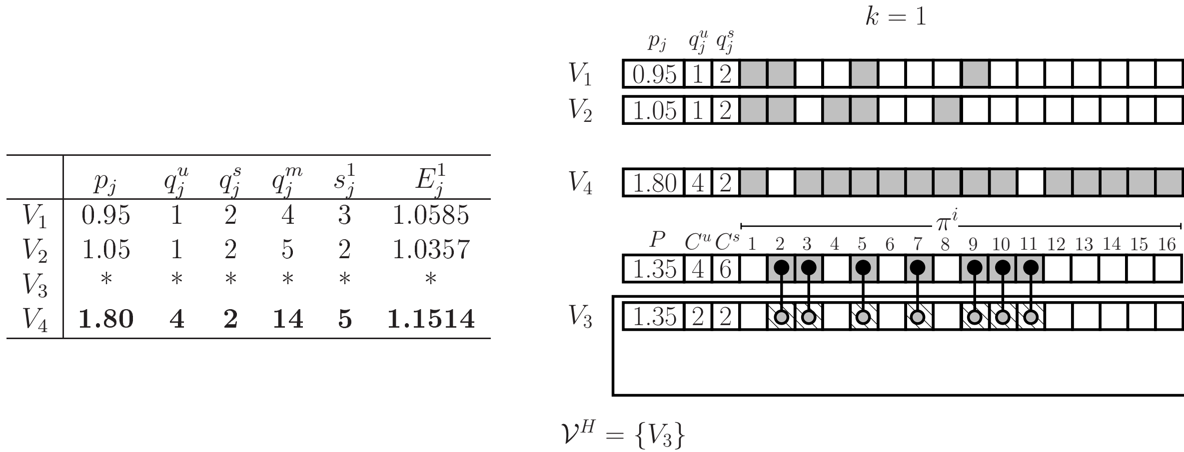
We now present an example to show how G-MSAVMM works. We consider a single server with resource capacities: vCPU, $C^u = 6$ vCPUs; storage, $C^s = 8$ GB; and memory, $C^m = 16$ pages. We consider four VM requests characterized by the parameters given in Table 2.2 (derived revenue, p_j; vCPU request, q_j^u; storage request, q_j^s; and memory request, q_j^m, translated into number of pages). Figures 3.2, 3.3, and 3.4 show the details of each iteration k of G-MSAVMM. Within the Figures, page π_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 phase of G-MSAVMM is illustrated Figure 3.2. The array A in Figure 3.2, 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, π^5, where the max count is identified in bold in array A (Figure 3.2), would be shared the most and all VMs which request π^5 would be considered candidates for instantiation in the first phase of G-MSAVMM. The efficiency



	p_j	q_j^u	q_j^s	q_j^m	s_j^0	E_j^0
V_1	0.95	1	2	4	4	1.3742
V_2	1.05	1	2	5	5	1.5169
V_3	**1.35**	**2**	**2**	**7**	**6**	**1.6040**
V_4	1.80	4	2	14	9	1.5898

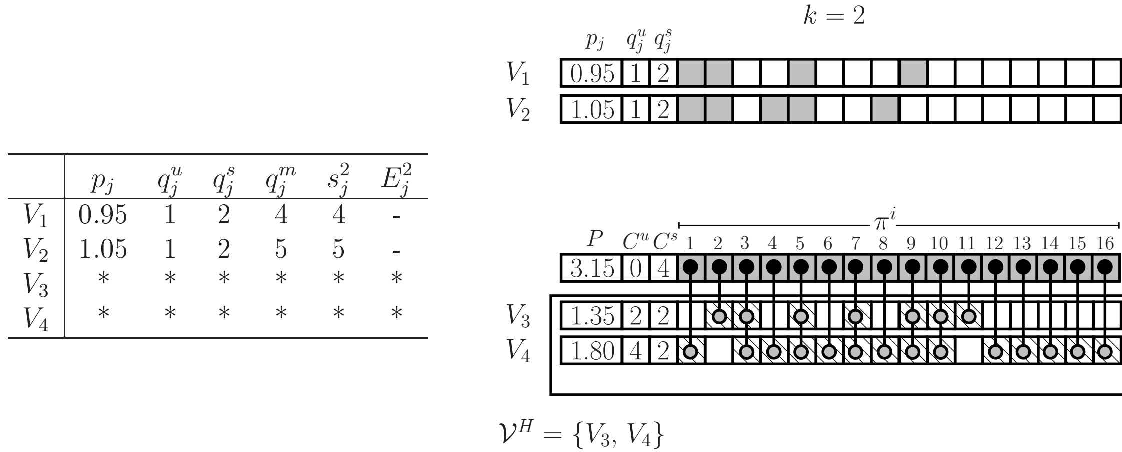
Figure 3.2: G-MSAVMM Efficiency Metric Calculation: Iteration 0



	p_j	q_j^u	q_j^s	q_j^m	s_j^1	E_j^1
V_1	0.95	1	2	4	3	1.0585
V_2	1.05	1	2	5	2	1.0357
V_3	*	*	*	*	*	*
V_4	**1.80**	**4**	**2**	**14**	**5**	**1.1514**

Figure 3.3: G-MSAVMM Efficiency Metric Calculation: Iteration 1

metric value is then calculated for those VMs sharing the most requested page and, based on the values given in Figure 3.2, the highest efficiency metric, 1.6040, is associated with V_3. All pages requested by V_3 are activated in Π and added to subset $\mathcal{V}^H$. The activated pages under provider management in Π 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, $C^u = 4$, 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.



	p_j	q^u_j	q^s_j	q^m_j	s^2_j	E^2_j
V_1	0.95	1	2	4	4	-
V_2	1.05	1	2	5	5	-
V_3	*	*	*	*	*	*
V_4	*	*	*	*	*	*

Figure 3.4: G-MSAVMM Efficiency Metric Calculation: Iteration 2

The next iteration of G-MSAVMM, corresponding to the first iteration of the greedy phase ($k = 1$), is illustrated in Figure 3.3. In this iteration, G-MSAVMM finds identical, requested pages between VMs and the active pages within Π. The efficiency metric value is calculated for all remaining VMs regardless of their potential for page sharing, where the highest efficiency metric, 1.1514, is associated with V_4. Following the instantiation of V_4, the algorithm reduces the server resource capacities according to V_4's resource request as follows: vCPUs, $C^u = 0$, 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 Π (i.e., π^3, π^5, π^7, π^9, and π^{10}); thereby, the server memory resource only needs to account for π^1, π^4, π^6, π^8, and π^{11} to π^{14}, in Π, which are required to instantiate V_4. Lastly, the revenue is updated to 3.15. At this iteration, G-MSAVMM stops because the memory resource has been exhausted and no further VM instantiation is possible (Figure 3.4). The total revenue obtained by G-MSAVMM for this example is \$3.15, which is less than \$4.05, the optimal revenue obtained by solving the BMP-MSAVMM.

A slightly larger MSAVMM instance consisting of 20 synthetically created VMs, where each VM may request up to 256 pages and considers multiple resource requests, shows a significant difference in performance between BMP-MSAVMM and G-MSAVMM. By generating, uniformly at random, VMs which are priced between \$.30 for a single vCPU, 4 GBs of

RAM, and 64 GBs of storage to \$2.45 for a VM which requests 16 vCPUs, 64 GBs of RAM, and 128 GBs of storage, our results show BMP-MSAVMM acquires 63% more revenue than G-MSAVMM. Specifically BMP-MSAVMM generated \$19.88 whereas G-MSAVMM generated \$12.18 when implemented on a single server consisting of 60 vCPUs, 1024 GBs of RAM, and approximately 1 TB of storage. In the next section, we determine the approximation ratio for G-MSAVMM which will characterize how far the solution obtained by G-MSAVMM can be from the optimal solution.

3.5 G-MSAVMM Properties

In this section, we investigate the approximability properties of our proposed algorithm. We determine the approximation ratio of G-MSAVMM by considering a worst possible server setup, Ω^W, for the MSAVMM problem. We consider Ω^W consisting of three resource types: memory, vCPU, and storage. We assume that Ω^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 MSAVMM problem, where VM $V_{\hat{j}} \in \mathcal{V}^W$ does not share any memory pages with the other VMs in $\mathcal{V}^W$. Then, let at least one VM $V_{\hat{j}^c} \in \mathcal{V}^W$ be comprised of pages which are a complement set of pages to VM $V_{\hat{j}}$. In addition, let the remaining VMs in $\mathcal{V}^W$ be comprised of either a subset of pages in VM $V_{\hat{j}^c}$ or be equivalent to VM $V_{\hat{j}^c}$. In either case, the remaining VMs would be allocated onto Ω^W if $V_{\hat{j}^c}$ were to be allocated first since they all share the same memory pages and would not reduce the memory capacity of Ω^W.

We investigate this instance on server Ω^W with a limited memory capacity such that either VM $V_{\hat{j}}$ or VM $V_{\hat{j}^c}$ can be allocated, but not both, while not depleting the vCPU and storage capacities. If VM $V_{\hat{j}^c}$ is allocated, then all remaining VMs in $\mathcal{V}^W \setminus \{V_{\hat{j}}\}$, will be allocated as well due to page sharing and the freedom in both vCPU or storage capacities. Else, VM $V_{\hat{j}}$ is allocated and utilizes the memory capacity enough to not allow any other VM from $\mathcal{V}^W$ to be allocated. We assume that Ω^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.

Our design of $\mathcal{V}^W$ and Ω^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-MSAVMM algorithm. If the memory capacity was larger than our proposed setup, then the revenue generated from G-MSAVMM 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-MSAVMM based on the worst case instance $\mathcal{V}^W$ and server Ω^W.

Theorem 3.5.1. *The approximation ratio of G-MSAVMM is $M\sqrt{C_{max}(|R|+1)}$, where $C_{max} = \max\{C^m, C^u, C^s\}$, R is the number of resources and M is the number of VMs.*

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 $\mathcal{V}^W_{OPT}$, $\mathcal{V}^W_{OPT} \subset \mathcal{V}^W$, where $P^* = \sum_{j:V_j \in \mathcal{V}^W_{OPT}} p_j$ under server resource Ω^W.

Let the revenue obtained by G-MSAVMM 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 Ω^W.

Assume at $k = 0$, VM $V_{\hat{j}}$ is allocated by G-MSAVMM onto Ω^W; admitting the relationship $E^0_j < E^0_{\hat{j}}$, for any $j \neq \hat{j}$. Since VM $V_{\hat{j}}$ does not share pages with VMs in $\mathcal{V}^W$, $s^0_{\hat{j}} = 0$, and by Equation 3.6,

$$\frac{p_j}{\sqrt{\sum_{r \in R} \frac{q^r_j}{C^r} + \frac{q^m_j - s^k_j + 1}{C^m}}} < \frac{p_{\hat{j}}}{\sqrt{\sum_{r \in R} \frac{q^r_{\hat{j}}}{C^r} + \frac{q^m_{\hat{j}} - s^k_{\hat{j}} + 1}{C^m}}} \tag{3.7}$$

$$\frac{\sqrt{\sum_{r\in R}\frac{q^r_{\hat{j}}}{C^r}+\frac{q^m_{\hat{j}}-s^k_{\hat{j}}+1}{C^m}}}{\sqrt{\sum_{r\in R}\frac{q^r_{j}}{C^r}+\frac{q^m_{j}-s^k_{j}+1}{C^m}}}p_j < p_{\hat{j}} \tag{3.8}$$

which establishes the lower bound for $p_{\hat{j}}$ in order for $V_{\hat{j}}$ to be selected according to our efficiency metric at $k = 0$. This implies that for any $p_{\hat{j}}$ greater than the established lower bound, VM $V_{\hat{j}}$ will be allocated first onto Ω^W from $\mathcal{V}^W$ by G-MSAVMM. Considering the memory utilization of VM $V_{\hat{j}}$ and memory capacity of Ω^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 $P = p_{\hat{j}}$.

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

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

$$\frac{P^*}{P} = \frac{\sum_{j:V_j\in\mathcal{V}^W_{OPT}} p_j}{\sum_{j:V_j\in\mathcal{V}^W_{GRD}} p_j} \tag{3.9}$$

$$= \frac{\sum_{j:V_j\in\mathcal{V}^W\setminus\{V_{\hat{j}}\}} p_j}{p_{\hat{j}}} \tag{3.10}$$

By substituting p_j from Eq. 3.8, we obtain

$$\frac{P^*}{P} < \frac{1}{p_{\hat{j}}} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \frac{\sqrt{\sum_{r \in R} \frac{q_j^r}{C^r} + \frac{q_j^m - s_j^k + 1}{C^m}}}{\sqrt{\sum_{r \in R} \frac{q_{\hat{j}}^r}{C^r} + \frac{q_{\hat{j}}^m - s_{\hat{j}}^k + 1}{C^m}}} p_{\hat{j}} \tag{3.11}$$

$$= \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \frac{\sqrt{\sum_{r \in R} \frac{q_j^r}{C^r} + \frac{q_j^m - s_j^k + 1}{C^m}}}{\sqrt{\sum_{r \in R} \frac{q_{\hat{j}}^r}{C^r} + \frac{q_{\hat{j}}^m - s_{\hat{j}}^k + 1}{C^m}}} \tag{3.12}$$

$$= \frac{\sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \sqrt{\sum_{r \in R} \frac{q_j^r}{C^r} + \frac{q_j^m - s_j^k + 1}{C^m}}}{\sqrt{\sum_{r \in R} \frac{q_{\hat{j}}^r}{C^r} + \frac{q_{\hat{j}}^m - s_{\hat{j}}^k + 1}{C^m}}} \tag{3.13}$$

Since

$$\sqrt{\sum_{r \in R} \frac{q_{\hat{j}}^r}{C^r} + \frac{q_{\hat{j}}^m - s_{\hat{j}}^k + 1}{C^m}} \geq \sqrt{\frac{1}{C_{max}}} \tag{3.14}$$

where $C_{max} = \max\{C^m, C^u, C^s\}$, we obtain

$$\frac{P^*}{P} \leq \sqrt{C_{max}} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} \sqrt{\sum_{r \in R} \frac{q_j^r}{C^r} + \frac{q_j^m - s_j^k + 1}{C^m}} \tag{3.15}$$

Because

$$\sum_{r \in R} \frac{q_j^r}{C^r} \leq \sum_{r \in R} 1 \leq |R| \tag{3.16}$$

and

$$\frac{q_j^m - s_j^k + 1}{C^m} \leq 1 \tag{3.17}$$

we have

$$\frac{P^*}{P} \leq \sqrt{C_{max}} \sum_{j:V_j \in \mathcal{V}^W \setminus \{V_{\hat{j}}\}} (\sqrt{|R|+1}) \tag{3.18}$$

Thus,

$$\frac{P^*}{P} \leq (M-1)\sqrt{C_{max}}\sqrt{|R|+1} \leq M\sqrt{C_{max}(|R|+1)} \tag{3.19}$$

Therefore, $\frac{P^*}{P}$ 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-MSAVMM algorithm. □

We now investigate the time complexity of G-MSAVMM. 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 $\mathcal{V}^H \subseteq \mathcal{V}$. 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 Ω (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 Ω, 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-MSAVMM 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.

3.6 Experimental Results

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

3.6.1 Experimental Setup

The software used in the experiments and trace processing is implemented in C++ on 2.93 GHz Intel 64-bit Intel hexa-core dual-processor systems within the Wayne State University High Performance grid [102].

Utilizing Google Cluster Usage Traces

For our experiments, we used the cluster usage traces from workloads running on Google compute cells [83]. 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 [37]. 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. [84]. As a result, research focusing on characterizing the many facets of the data set such as applications [26], user behavior [1] and workloads [67] [81], 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:

- Eliminate traces which are missing information, i.e., acquire trace if missing info = 0.
- 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.
- 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 [83]. Due to obfuscation, we do not know exactly that every trace with a high scheduling task is a revenue generating user request; therefore, for our investigation we assume that traces at the highest level of scheduling class are revenue generating user requests, i.e., acquire trace if scheduling class = 3.

	n1-standard-{size} : (n1s{size})						n1-highmem-{size} : (n1m{size})					n1-highcpu-{size} : (n1c{size})				
{size}	{1}	{2}	{4}	{8}	{16}	{32}	{2}	{4}	{8}	{16}	{32}	{2}	{4}	{8}	{16}	{32}
Memory (GB)	3.75	7.50	15	30	60	120	13	26	52	104	208	1.80	3.60	7.20	14.40	28.80
vCPU	1	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
Price ($/hour)	0.050	0.100	0.200	0.400	0.800	1.600	0.126	0.252	0.504	1.008	2.016	0.760	0.152	0.304	0.608	1.216

Table 3.5: G-MSAVMM Experiment: VM Instance Types.

- Eliminate traces where tasks have a low priority and that are monitoring. We only consider traces corresponding to tasks classified as high priority, which will be last to be evicted in the case of over-provisioning the machine resource, i.e., acquire trace if $\mathsf{priority} \geq 8$ and $\mathsf{priority} \neq 10$.
- 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 $\mathsf{different\ machines\ restriction} = 0$.

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 [38], 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 3.5. 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 $\times$ 15 cores per chip); 2 TB memory (128 $\times$ 16 GB DDR3) and 9.6 TB disk space (24 $\times$ 400 GB SSD). The Lenovo Flex System x880 X6 Compute Node is the highest rated server according to the SPECvirt_sc2013 benchmark which evaluates data-

center server performance and virtualized server consolidation conducted by the Standard Performance Evaluation Corporation© (SPEC), released in the 2nd quarter of 2015 [89].

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 vCPUs capacities, C^m and C^u respectively. The resulting products represent a specific amount of memory (in GB), denoted by t^m, and a number of vCPUs, denoted by t^u, relative to the server specifications. For every Google Compute Engine VM Instance g_y, $y \in \{1 \dots 16\}$, we denote its memory requirement by g_y^m and its vCPU requirement by g_y^u. We calculate $\tilde{y}$, 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 g_y's requirements, as follows,

$$\tilde{y} = \operatorname*{argmin}_y \sqrt{\left(\frac{|t^m - g_y^m|}{C^m}\right)^2 + \left(\frac{|t^u - g_y^u|}{C^u}\right)^2} \tag{3.20}$$

Then, we map the trace request t to the Google Compute Engine VM Instance $g_{\tilde{y}}$, that is, to the Google VM instance that fits the requested resources the best. Lastly, the storage usage values are not fully captured within ClusterData-2011-1 traces due to Google treating storage as a separate service from Google Compute Engine [83]. Therefore, we do not use the VM storage request information within our experiments.

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 [83], 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.* [7]. These percentages are given in Figure 3.5. We con-

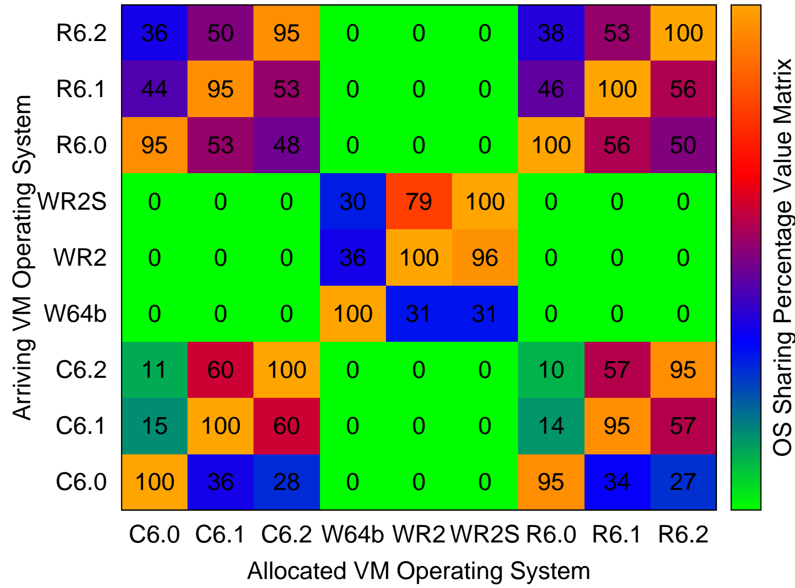


Figure 3.5: Page Sharing Percentages Table: OS.

sider 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).

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 220 MB, then the sharing percentage is calculated as $\frac{220\text{MB}}{.77\text{GB}} = 28\%$. The amount of memory sharing and image sizes are those determined by Bazarbayev *et. al* [7]. 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 $\frac{220\text{MB}}{1.96\text{GB}} = 11\%$. 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 3.5. Furthermore, we consider that CentOS and Red Hat Enterprise Linux (RHEL) distributions of the same version share approximately 95% of their content. CentOS is an open-source version of RHEL with the exception of proprietary updates and trademarks (see CentOS 6.2 Release Notes). We slightly scale down the page sharing percentages between two VMs with different versions of RHEL and CentOS according to the inter-OS version sharing percentages in Figure 3.5. Lastly, cases exist in which two operating systems will share very little memory, as was found by Sindelar *et. al* [86] 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.

Comparing G-MSAVMM

We compare our algorithm with other algorithms for VM 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 revenue, number of shared pages, vCPUs, and amount of memory, and use them in our experiments. Thus, we compare G-MSAVMM with four algorithms that are variants of G-MSAVMM: P-DO which allocates the VM requests in decreasing order of their revenue (this corresponds to G-MSAVMM with $E_j^k = p_j$); SP-DO which allocates the VM requests in decreasing order of the number of shared pages (this corresponds to G-MSAVMM where E_j^k is calculated with $p_j = 1$, and the first term under the square root equal to 0); C-IO

Table 3.6: Algorithms Used in Experiments.

Algorithm	Greedy ordering
G-MSAVMM	Decreasing order of E_j^k.
P-DO	Decreasing order of revenue.
SP-DO	Decreasing order of the number of shared pages.
C-DO	Decreasing order of the number of requested vCPUs.
C-IO	Increasing order of the number of requested vCPUs.
M-DO	Decreasing order of the amount of requested memory.
M-IO	Increasing order of the amount of requested memory.
DR-DO	Decreasing order of the dominant resource.
DR-IO	Increasing order of the dominant resource.

which allocates the VM requests in increasing order of the number of requested vCPUs (this corresponds to G-MSAVMM where E_j^k 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 increasing order of the amount of requested memory (this corresponds to G-MSAVMM where E_j^k is computed with $p_j = 1$, the first term under the square root equal to 0, and $s_j^k = 0$). We also compare G-MSAVMM with four other greedy algorithms that are not variants of G-MSAVMM: 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.

The last two algorithms 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. The algorithms used in our experiments are presented in Table 3.6. 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 is one exception, SP-DO algorithm, which uses the number of shared pages to establish the greedy ordering, and thus, the allocation.

3.6.2 Analysis of Results

We now compare the performance of G-MSAVMM 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 3.6.1. We use a portion of the transformed trace events which consists of 15,000 events. The distribution of VMs which are used in our experiments is illustrated in Figure 3.6.

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 3.7, we plot the increase of memory utilization when comparing G-MSAVMM against sharing-oblivious versions of the algorithms listed in Table 3.6. For each window within W30, W50, and W100, we implemented sharing-oblivious versions of these algorithms, meaning the hypervisor mechanism which performed the search for shared pages was turned off and duplicate pages could be present among collocated VMs' memory requests. We, then, recorded the amount of memory each sharing-oblivious algorithm utilized following the allocation of VMs within each window for W30, W50, and W100 to the available server resource. Lastly, we implemented G-MSAVMM for each window within W30, W50, and W100, then recorded the amount of memory that was utilized in the VM allocation. The

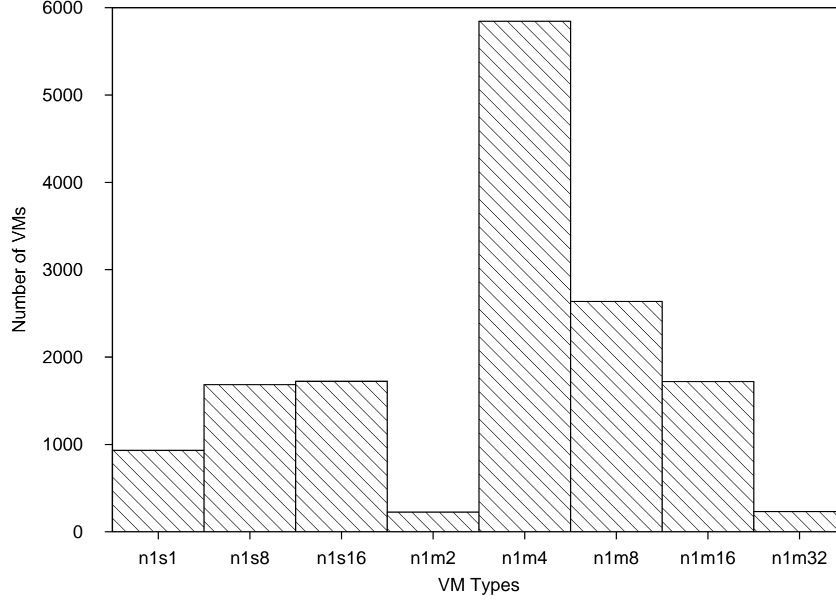


Figure 3.6: Distribution of Google Type VMs in Experiment.

increase in memory utilization is the difference between G-MSAVMM's memory utilization and the maximum memory utilization recorded among the sharing-oblivious algorithms. The algorithms which generated the maximum memory utilization fluctuated between sharing-oblivious versions of SP-DO, M-IO, and DR-IO for each window within W30, W50, and W100. Memory tends to be the extraneous resource which remains when the vCPU capacity has been exhausted on the server which hosts the VM requests. By taking page sharing into consideration, an increase of memory utilization can be achieved by a sharing-aware algorithm such as G-MSAVMM so that less memory lies dormant when vCPU resources have been exhausted. Based on our experiments, we have found that on average using G-MSAVMM increases the overall memory utilization by approximately 26% across W30, W50, and W100. In Figure 3.7, we show that by using G-MSAVMM, the increase in memory utilization is between 7% and 40% over all 500 windows in W30, between 10% and 41% over all 300 windows in W50, and between 11% to 42% over all 150 windows in W100.

In Figure 3.8, 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

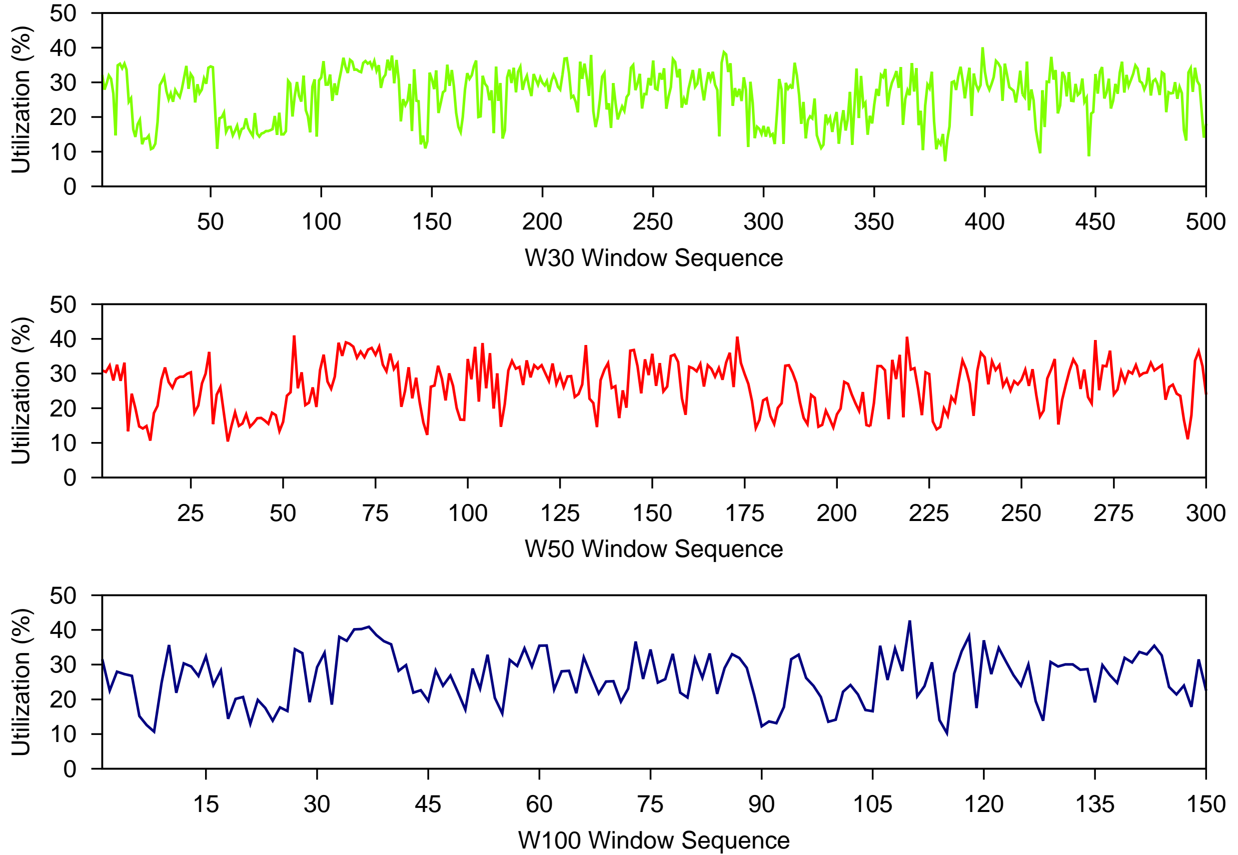


Figure 3.7: Sharing vs. non-Sharing Memory Utilization.

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-MSAVMM obtains the highest average aggregated revenue ratio for all three window intervals (Figure 3.8). Moreover, as the window size increases the eight competing algorithms exhibit a decrease in revenue which is in contrast to the increase in revenue exhibited by G-MSAVMM. Our experiments show that as the windows grow larger and contain greater VM resource type heterogeneity, G-MSAVMM makes better greedy allocation decisions for

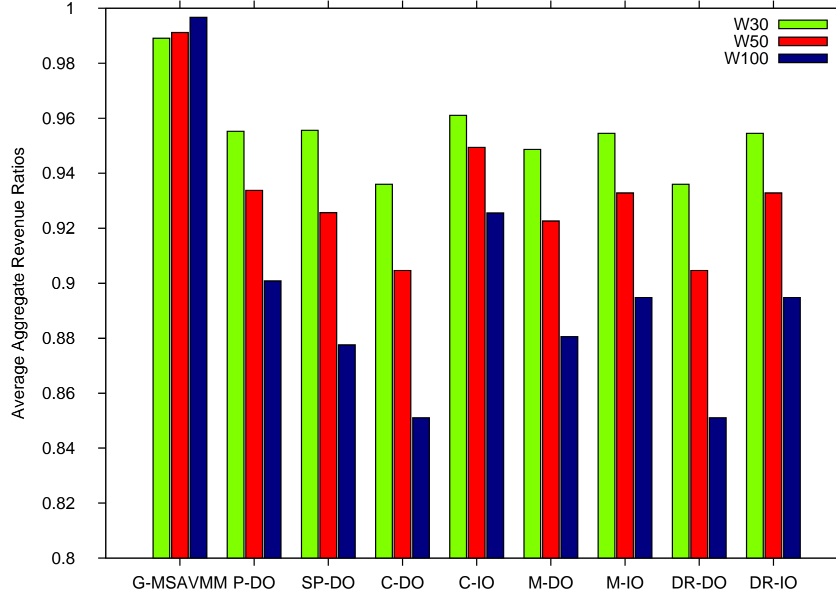


Figure 3.8: Average Aggregate Revenue Ratios.

revenue generation than the competing algorithms. The next best performing algorithm is C-IO which tends to have similar behavior to G-MSAVMM 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, 5% in W50, and 7% in W100.

We also investigate the performance of the algorithms in terms of average generated revenue per server (Figure 3.9). The results are consistent with those in Figure 3.8, in that G-MSAVMM generates the highest average revenue followed by C-IO for all window types. G-SAVMM outperforms C-IO when comparing the average revenue generated per server by approximately 3% in W30 (or by $0.27), 5% in W50 (or by $0.43), and 8% in W100 (or by $0.73). While these differences maybe 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-MSAVMM is the best performing algorithm, obtaining greater revenue ratios and higher average revenue than the other eight algorithms.

When allocating VMs to server resources, the scarcest resource is the vCPU resource. Therefore, algorithms which conserve the vCPU resource and maximize the use of the less scarce memory resource while generating higher revenues are desirable. In Figure 3.10,

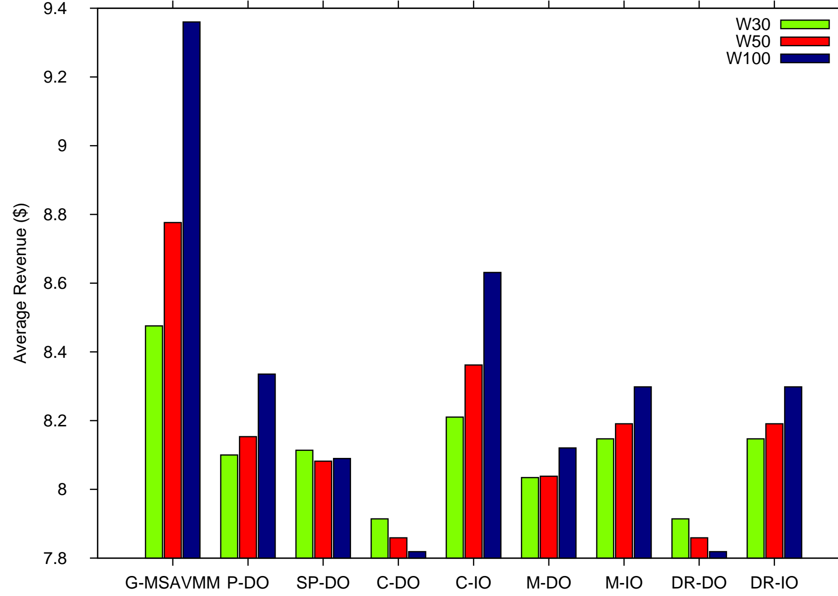


Figure 3.9: Average Revenue Per Server.

we compare the eight resource-centric algorithms against G-MSAVMM in terms of resource utilization. On the left side of Figure 3.10, we compare three memory-centric allocation algorithms, SP-DO, M-DO and M-IO, against G-MSAVMM, and on the right, we compare three vCPU-centric allocation algorithms, P-DO, C-DO and C-IO, against G-MSAVMM. 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 each memory-centric algorithm. SP-DO slightly outperforms G-MSAVMM by .5% in W30, .8% in W50, and 1% in W100. While SP-DO utilizes slightly more memory than G-MSAVMM, 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 each vCPU-centric algorithm. C-IO slightly outperforms G-MSAVMM by .5% in W30 (conserving .64 of a vCPU core), .7% in W50 (conserving .84 of a vCPU core), and 1% in W100 (conserving 1.16 vCPU cores). While C-IO utilizes slightly less vCPUs than G-MSAVMM, choosing C-IO as the allocation algorithm would lead to less revenue generated on average, $.27 instead of $.73 per server. Although G-MSAVMM is a multi-resource allocation algorithm, its memory utilization is marginally close to the best memory-centric algorithm,

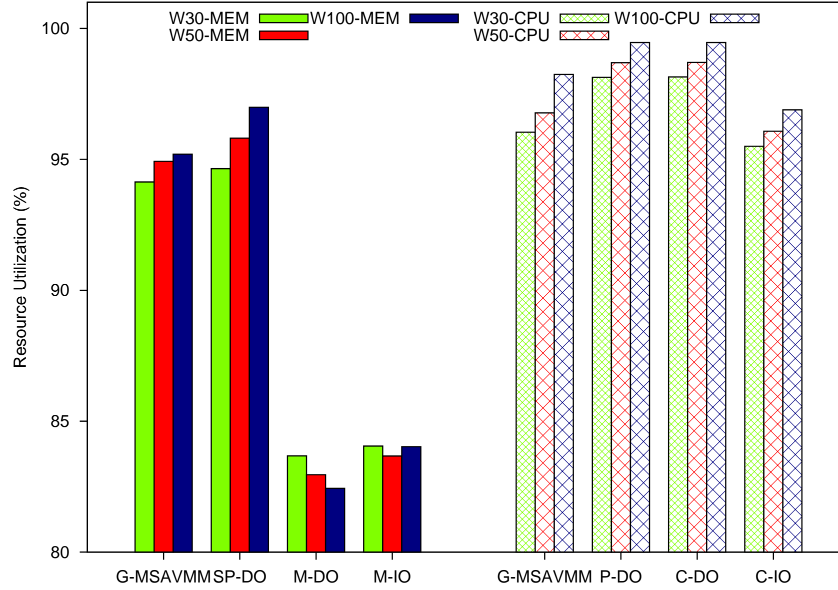


Figure 3.10: Memory / CPU Utilization.

SP-DO, and its vCPU utilization is marginally close to the best vCPU-centric algorithm, C-IO; subsequently generating the highest revenue among them.

Throughout our experiments, certain algorithms obtain greater revenue relative to G-MSAVMM for specific windows within W30, W50 and W100. The performance of the algorithms depends on the number and type of VMs requested within each window. For instance, when comparing G-MSAVMM to C-IO on a window with fairly homogeneous VM requests, their allocation behavior is nearly identical. In contrast, when the heterogeneity of VM types in a specific window increases, they behave differently with G-MSAVMM outperforming C-IO in terms of obtained revenue.

Lastly within our experiment, there are windows with specific VM type requests combinations which stifle G-MSAVMM performance against other algorithms. By analyzing the behaviors of these algorithms on specific sets of VM requests, we can identify under which set of VM requests should a specific allocation algorithm be used. In Figures 3.11, 3.12 and 3.13, we show the configurations of VM requests for specific W30, W50 and W100 windows. This illustrates the differences in allocation behavior between G-MSAVMM and its variants, P-DO, SP-DO, C-IO, and M-IO.

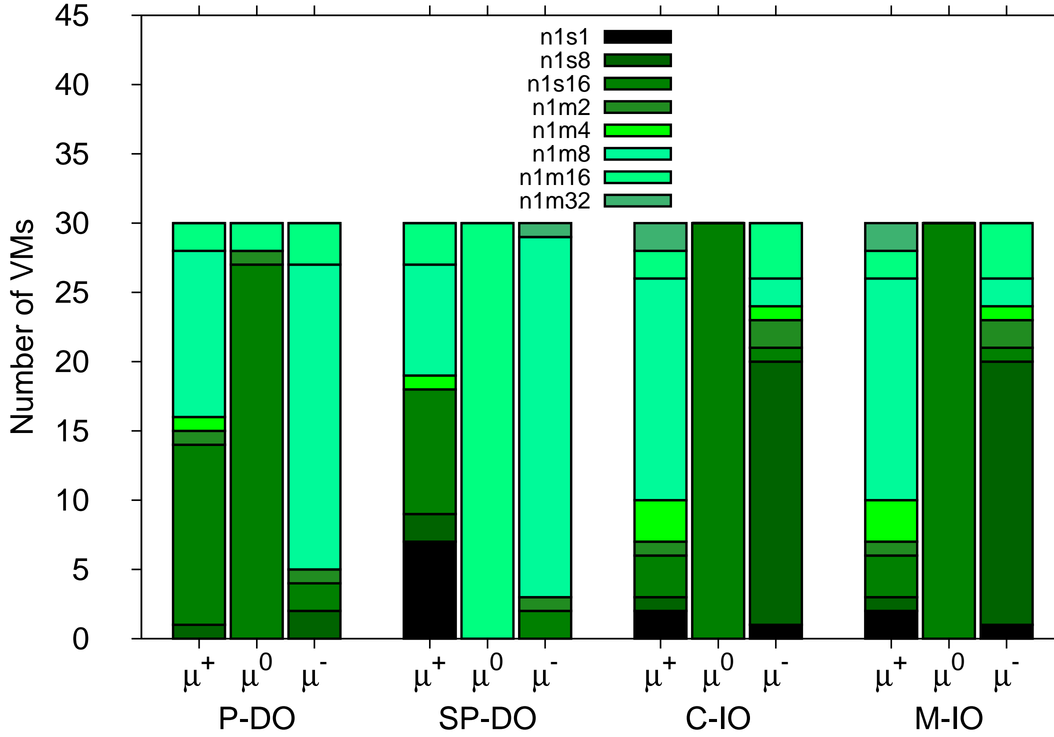


Figure 3.11: W30: G-MSAVMM behavior for different VM request configurations.

In each of the figures, we denote by μ^+ on the horizontal axis, the VM requests combinations in which the allocation results in the largest revenue for G-MSAVMM. Likewise, we denote by μ^-, the VM requests combinations in which the allocation results in the largest revenue for P-DO, SP-DO, C-IO and M-IO. Lastly, we denote by μ^0 the VM requests combinations in which G-MSAVMM's revenue is the same as that of P-DO, SP-DO, C-IO, and M-IO. While some outlier combinations exist (e.g., P-DO at μ^0 in W50), our results show that G-MSAVMM tends to outperform all other algorithms when VM requests are heterogeneous both with respect to the VM characteristics and the number of VMs of each type requested within the windows.

3.7 Summary

We designed a sharing-aware greedy approximation algorithm (G-MSAVMM) for solving the multi-resource sharing-aware VM maximization problem. We showed that G-MSAVMM is a $M\sqrt{C_{max}(|R|+1)}$-approximation algorithm, where M is the number of VM instances

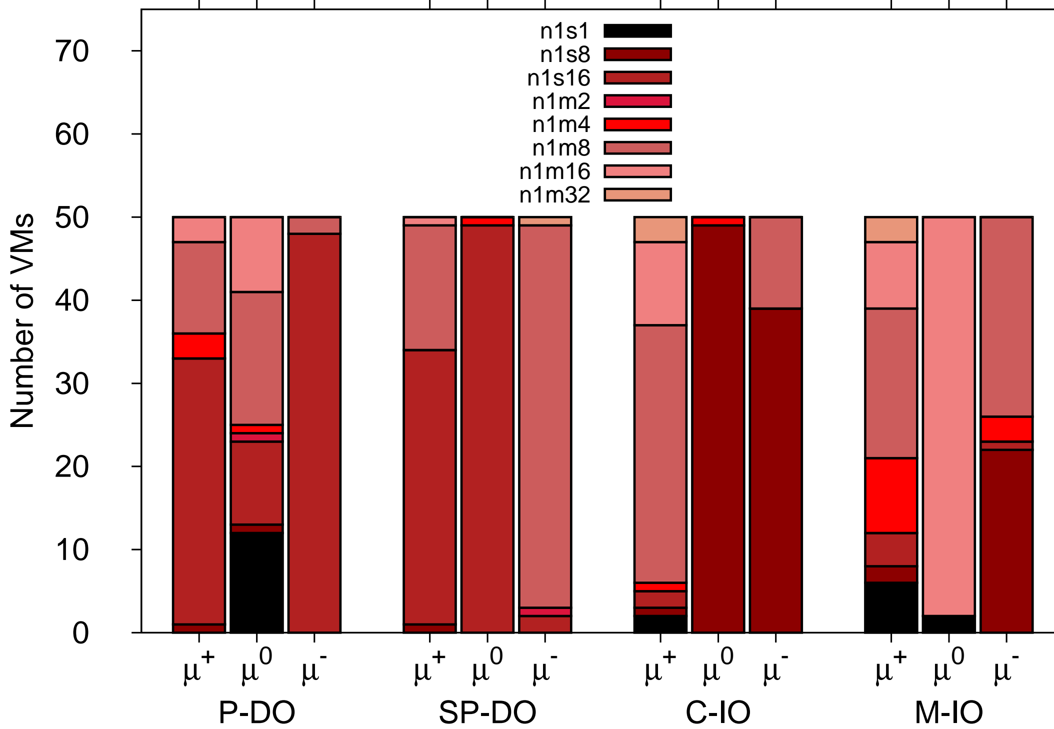


Figure 3.12: W50: G-MSAVMM behavior for different VM request configurations.

that are to be allocated, C_{max} is the maximum capacity among all types of resources, and R is the number of resource types except the memory resource. The experimental results showed that G-MSAVMM outperforms eight other VM allocation algorithms in terms of generated revenue and efficient utilization of resources. In future work, we plan on extending G-MSAVMM to manage the VM allocation process in online environments. Incorporating energy consumption awareness and network virtualization into the multi-resource type VM allocation problem would be an interesting extension.

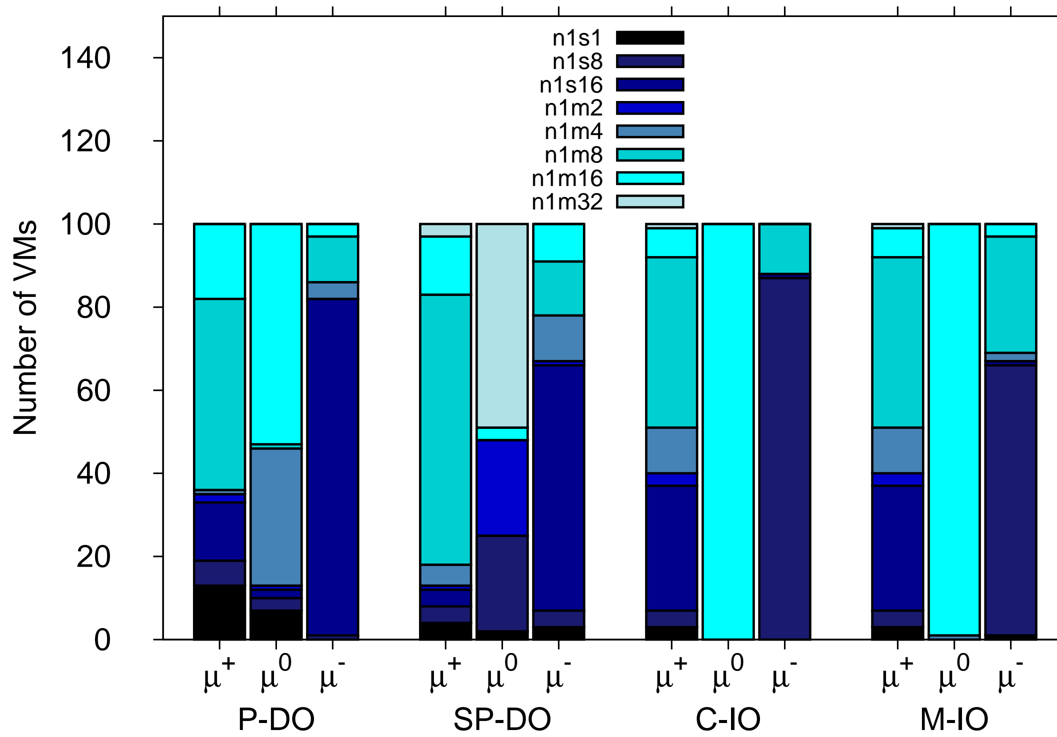


Figure 3.13: W100: G-MSAVMM behavior for different VM request configurations.

CHAPTER 4: MULTI-RESOURCE VM PACKING

4.1 Introduction

Cloud adoption by government, industrial, and academic institutions has created opportunities for providers to offer services through flexible infrastructures based on virtualization technologies. Industry forecasts predict that by 2019 approximately 80% of all workloads will be managed through data center virtualization services [18]. A challenge facing cloud service providers is the development of efficient resource allocation mechanisms allowing them to reduce the costs and increase their profits.

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. Page-sharing and similar mechanisms drive the development of more efficient algorithms suitable for resource management. A variant of the VM resource allocation problem motivated by these developments is the VM Packing problem [86].

The VM Packing problem considers instantiating multiple VMs in an "offline" setting which utilizes hypervisors as an architectural layer on top of physical servers, allowing for page-sharing; resulting in reduced utilization of the memory resource. Traditionally, VM allocation problems with multiple resource requirements have been modeled as vector bin packing problems, where each resource is represented as a vector component. The goal is to minimize the number of active servers used in order to instantiate a set of VMs according to server allocation policies and available resource capacities. The online VM Packing problem considers how to assign VMs, whose resource requests are unknown until they arrive to the cloud service provider, such that the number of active servers is minimized. 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 chapter, 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.

4.1.1 Our Contribution

We propose sharing-aware online algorithms for solving the VM Packing problem with multiple resource requirements and heterogeneous server capacities in an online setting. Our proposed sharing-aware online algorithms are improved designs of classical sharing-oblivious online algorithms for vector bin packing which take page sharing into account when making allocation decisions in cloud environments with heterogeneous server capacities and heterogeneous resource VM requests. 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 formulate the "offline" sharing-aware VM packing problem as a multilinear boolean program which when solved provides the optimal VM to server assignments. We perform extensive experiments to compare the performance of our sharing-aware online VM packing algorithms against several sharing-oblivious packing algorithms. 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.

4.1.2 Related Work

Several variants of online vector bin packing problem modeling the allocation of resources in clouds have been recently investigated. Song *et al.* [88] 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.* [57] introduced novel variants of bin packing algorithms which attempt to minimize the total cost associated with a server's utilization. Kamali and Ortiz [50] 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.* [3] proposed vector-bin packing algorithms, analyzed their performance under various VM sequences, and established lower competitive ratios. Panigrahy *et al.* [72] studied heuristic variants of the First-Fit-Decreasing algorithm for "offline" VM allocation.

Resource awareness is a prevalent topic in designing resource allocation algorithms for cloud environments. Carli *et al.* [16] 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 [14] 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.* [54] 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.* [109] designed online VM algorithms specific to energy and SLA-violation awareness to increase a cloud provider's revenue. Xu *et al.* [105] developed a hardware heterogeneity, VM-inference aware provisioning technique which focused on predicting MapReduce performance in the cloud. Xiao *et al.* [104] 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.* [42] proposed an online, generalized VM placement strategy which considers variation on cloud architectures, resource demand duration and data-center location. Mashayekhy *et al.* [61] designed an online mechanism for resource allocation and pricing in clouds. 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 [65], Memory Buddies [101], and Difference Engine [41] considered hypervisor-based VM page-sharing, but did not address the design of sharing-aware online algorithms for VM packing. Sindelar *et al.* [86] 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 chapter differs substantially from Sindelar *et al.* [86] 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 Chapters 2 and 3 and our previous work [77, 79], 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 chapter. 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.

4.1.3 Organization

The rest of the chapter is organized as follows. In Section 4.2, we define the Sharing-Aware Online VM Packing problem. In Section 4.3, we present the design of our proposed online sharing-aware algorithms. In Section 4.4, we present and solve the "offline" version of the sharing-aware VM packing problem. In Section 4.5, we compare the performance of our proposed algorithms against that of several sharing-oblivious algorithms through extensive experiments. In Section 4.6, we summarize our results and present possible directions for future research.

Table 4.7: SA-OVMP 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; $\lvert\mathcal{S}\rvert = M$.
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 $\mathcal{P}(\mathcal{V})$.

4.2 SA-OVMP: Problem

We now introduce the Sharing-Aware Online Virtual Machine Packing (SA-OVMP) problem from the perspective of a cloud service provider. The notation used in the chapter is presented in Table 4.7.

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 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 4.3 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 [101] 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 [101] 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 (SA-OVMP) problem as follows,

SA-OVMP problem: We consider a cloud provider having a set of servers, $\mathcal{S} = \{S_1, S_2, \ldots, S_{|\mathcal{S}|}\}$, 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 $\{V_1, V_2 \ldots, V_j, \ldots\}$, 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:

$$C_k^m - q_j^m + s_j^k \geq 0 \tag{4.1}$$

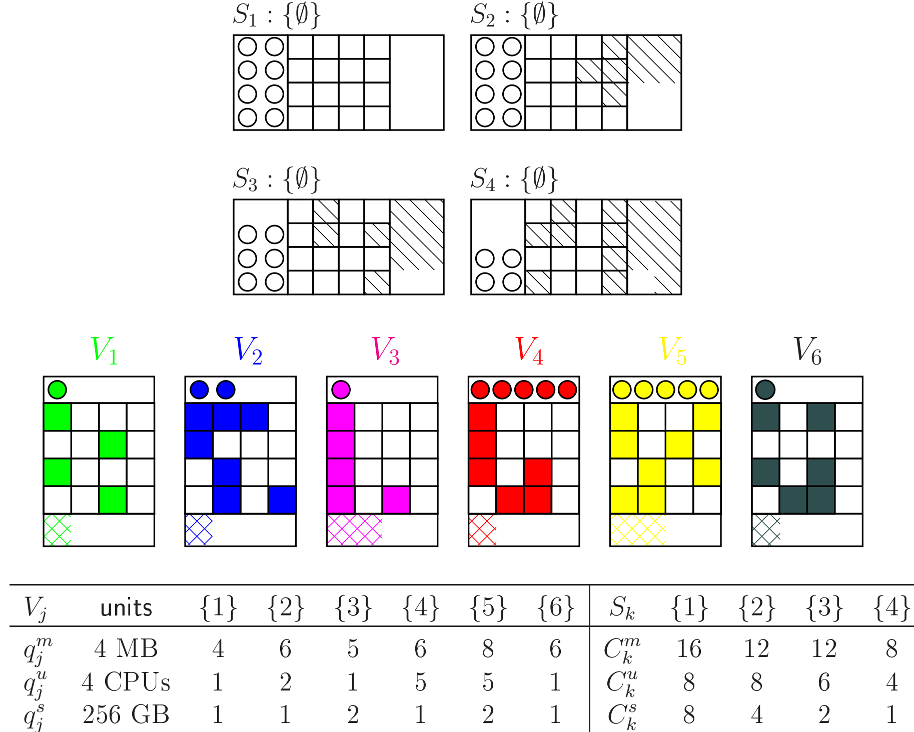
$$C_k^r - q_j^r \geq 0,\ \forall r \in \mathcal{R} \tag{4.2}$$

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 4.1 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 4.2 guarantee that the capacities of the other types of resources of server S_k are also not exceeded.

4.3 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



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

Figure 4.1: SA-OVMP: VM Requests and Resource Configuration.

are inactive servers, i.e., $\overline{\mathcal{S}} = \mathcal{S}$. All the sharing-aware algorithms presented in the chapter 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 4.2.

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 Figure 4.1. Each server in Figure 4.1, $\mathcal{S}_1$ 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 memory, 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 above each server image. Each VM in Figure 4.1, 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 512 GB of storage identified by the two mesh blocks 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 Figure 4.2 through Figure 4.12 for each of the proposed algorithms.

4.3.1 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 5 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

Algorithm 5 NFS

1: **Input:** VM instance arrival (V_j)
2: {$S_{\tilde{k}}$*: currently active server.*}
3: **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**
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** ($[C^m_{\tilde{k}}, C^u_{\tilde{k}}, C^s_{\tilde{k}}] - [q^m_j - s^{\tilde{k}}_j, q^u_j, q^s_j] \geq [0, 0, 0]$) **then**
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: $[C^m_{\tilde{k}}, C^u_{\tilde{k}}, C^s_{\tilde{k}}] \leftarrow [C^m_{\tilde{k}}, C^u_{\tilde{k}}, C^s_{\tilde{k}}] - [q^m_j - s^{\tilde{k}}_j, q^u_j, q^s_j]$

case in our SA-OVMP problem), a search must ensue to find a server which meets the V_j's resource demand.

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-

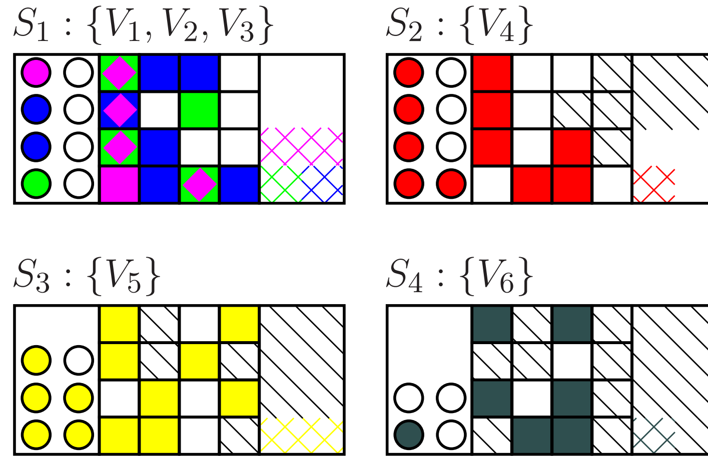


Figure 4.2: NFS: VM Assignment

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 4.2, thus resulting in a total run time of $\mathcal{O}(NM)$ for NFS.

Figure 4.2 illustrates the assignment of VMs to servers according to NFS for the SA-OVMP instance presented in Figure 4.1. 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

Algorithm 6 FFS

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

sharing-oblivious NF implementation would also require all four servers to assign the VMs; albeit, more memory would be consumed on server S_1.

4.3.2 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 6 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

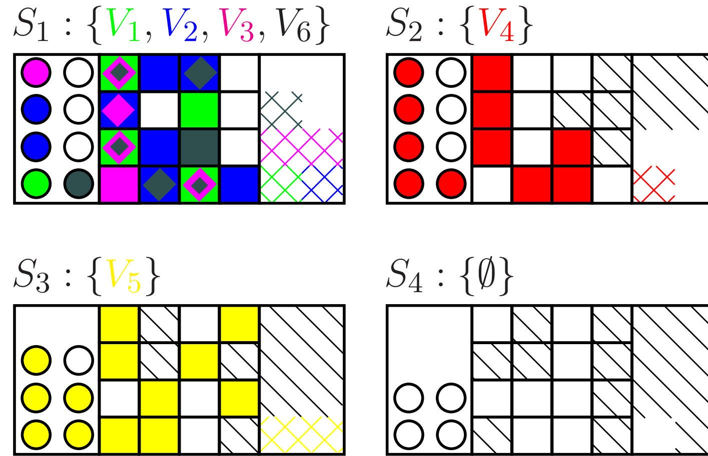


Figure 4.3: FFS: VM Assignment

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 5, 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).

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 4.2 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 Figure 4.3, we present the assignment of VMs using FFS for the SA-OVMP instance from Figure 4.1. 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.

4.3.3 Best-Fit-Sharing (BFS) Algorithm

In order to design BFS, we introduce the *server resource scarcity* metric which characterizes the scarcity of aggregate 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:

$$e_j^k = \begin{cases} \max\left\{\frac{q_j^m - \sqrt{s_j^k}}{C_k^m}, \frac{q_j^u}{C_k^u}, \frac{q_j^s}{C_k^s}\right\} & \text{if } C_k^m - q_j^m + s_j^k \geq 0 \ \ \& \\ & C_k^u - q_j^u \geq 0 \ \ \& \\ & C_k^s - q_j^s \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

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 ratio in Equation 4.3 relative to the available server capacity type, if V_j were to

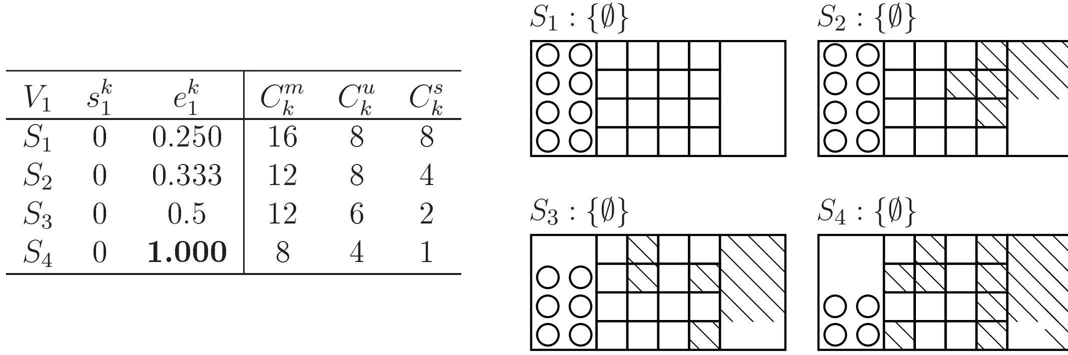
be instantiated on S_k. These ratios are only relevant if the V_j's resource requests do not over-commit any of the resource capacities on server S_k. The maximum remaining resource ratio among the three resource types reflects the scarcest remaining resource after server S_k instantiates VM V_j. In Equation 4.3, sharing influences the memory request by $\sqrt{s_j^k}$ instead of s_j^k in the numerator. This way we 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. 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.

BFS is given in Algorithm 7 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 violates 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 (lines 14 - 21) exactly as in FFS (Algorithm 6, 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).

Algorithm 7 BFS

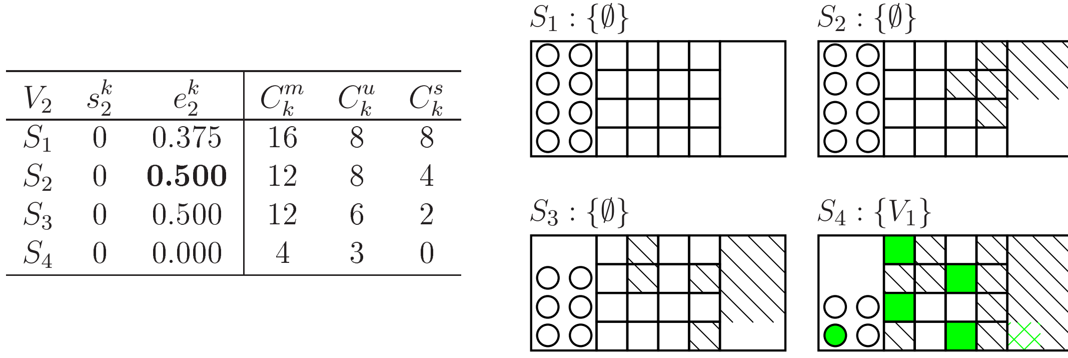
1: **Input:** VM instance arrival (V_j)
2: $\tilde{k} \leftarrow 0$
3: $flag \leftarrow 0$
4: **if** ($[C_k^m, C_k^u, C_k^s] - [q_j^m - s_j^k, q_j^u, q_j^s] \geq [0, 0, 0]$) **then**
5: $\quad flag \leftarrow 1$
6: $\quad e_j^k \leftarrow \max \left\{ \frac{q_j^m - \sqrt{s_j^k}}{C_k^m}, \frac{q_j^u}{C_k^u},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

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 4.3. 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.



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

Figure 4.4: BFS: Init

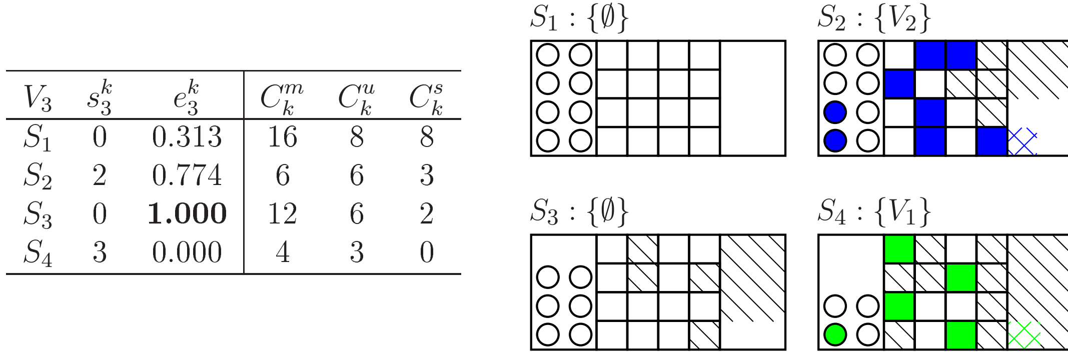


V_2	s_2^k	e_2^k	C_k^m	C_k^u	C_k^s
S_1	0	0.375	16	8	8
S_2	0	**0.500**	12	8	4
S_3	0	0.500	12	6	2
S_4	0	0.000	4	3	0

Figure 4.5: BFS: VM 1 Assignment

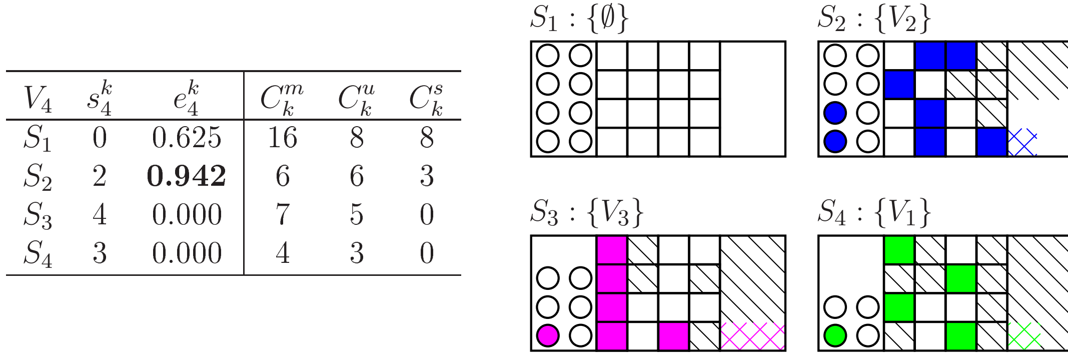
We now illustrate the assignment process of BFS using the SA-OVMP instance from Figure 4.1. Figures 4.4, 4.5, and 4.6 illustrate the process for VMs V_1 through V_3. The amount of sharing, s_1^k, and the server resource scarcity metric, e_1^k, are calculated relative to V_1 and the servers within the 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



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

Figure 4.6: BFS: VM 2 Assignment

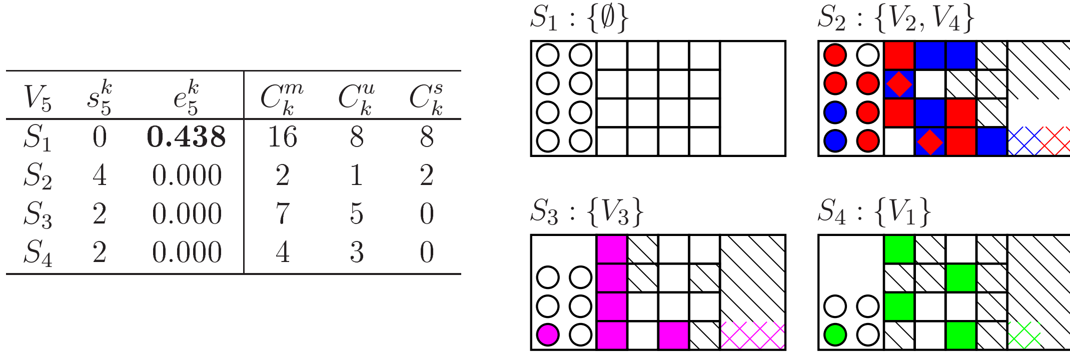


V_4	s_4^k	e_4^k	C_k^m	C_k^u	C_k^s
S_1	0	0.625	16	8	8
S_2	2	**0.942**	6	6	3
S_3	4	0.000	7	5	0
S_4	3	0.000	4	3	0

Figure 4.7: BFS: VM 3 Assignment

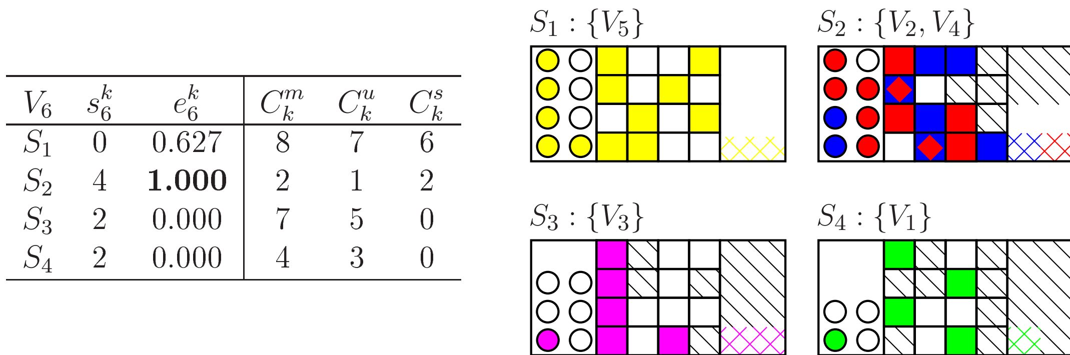
highest server resource scarcity metric, e.g., S_2, to host V_2 and the resource capacities of S_2 are updated. Relative to server S_4, $e_2^4 = 0$ since there is not enough memory available. Next, V_3 is ready for instantiation. With V_1 assigned to S_4 and V_2 assigned to S_2, V_3 has two opportunities to share pages, leading to $s_3^2 = 2$ and $s_2^3 = 3$. 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 are illustrated in Figures 4.7, 4.8 and 4.9. 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 requests, the resource scarcity metric has a value of 1.0



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

Figure 4.8: BFS: VM 4 Assignment



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

Figure 4.9: BFS: VM 5 Assignment

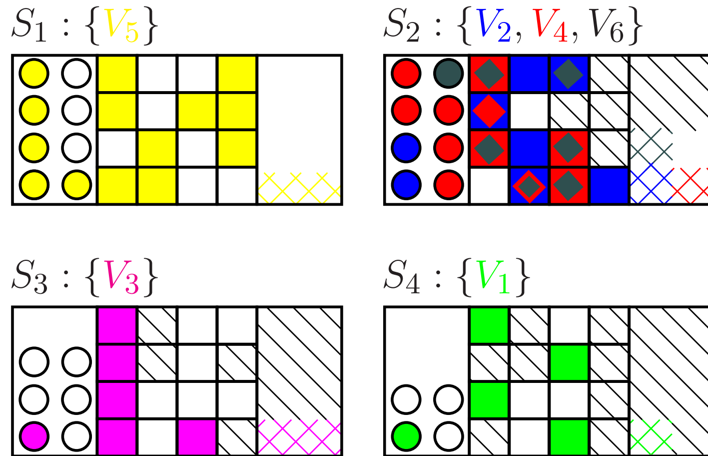


Figure 4.10: BFS: VM Final Assignment

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 Figure 4.10.

4.3.4 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 4.3 is changed to the minimum operator. Due to the similarity to BFS, the run time complexity of WFS is also $\mathcal{O}(NM)$.

4.4 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 4.2 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 in Equations 4.4 through 4.10

A boolean decision vector $y \in \{0,1\}^M$ is the solution to our program from Equation (4.4); 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 (4.6) 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 (4.7) 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 (4.8) 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. Equations (4.9) and (4.10) ensure decision variables y_k and

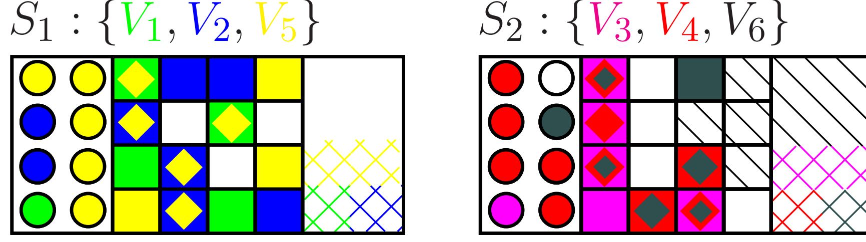


Figure 4.11: Optimal VM Assignment

x_{jk} are boolean.

$$\text{minimize: } B = \sum_{k:S_k \in \mathcal{S}} y_k \tag{4.4}$$

$$\text{subject to:} \tag{4.5}$$

$$\sum_{k:S_k \in \mathcal{S}} x_{jk} = 1, \ \forall j : V_j \in \mathcal{V} \tag{4.6}$$

$$\sum_{j:V_j \in \mathcal{V}} q_j^r x_{jk} \leq y_k C_k^r, \ \forall k : S_k \in \mathcal{S}, \ \forall r \in \mathcal{R} \tag{4.7}$$

$$\sum_{\mathcal{J} \in \mathcal{P}(\mathcal{V})} (-1)^{(|\mathcal{J}|+1)} \sigma_{\mathcal{J}} \prod_{\hat{j} \in \mathcal{J}} x_{\hat{j}k} \leq y_k C_k^m, \ \forall k : S_k \in \mathcal{S} \tag{4.8}$$

$$\forall \ y_k \in \{0, \ 1\} \tag{4.9}$$

$$\forall \ x_{jk} \in \{0, \ 1\} \tag{4.10}$$

Figure 4.11 shows the solution of our multilinear program for the SA-OVMP instance from Figure 4.1. 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 4.3. 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.

In Equation (4.8), 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 $\sigma_{346} = 3$, i.e., all VMs in $\mathcal{J}$ which include V_3, V_4 and

V_6 share 3 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 Figure 4.12. 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 (4.8) 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 Figure 4.12, 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,

$$(+1)(\sigma_3 + \sigma_4 + \sigma_6) + (-1)(\sigma_{34} + \sigma_{36} + \sigma_{46}) \ + \ (+1)(\sigma_{346}) \leq C_2^m \tag{4.11}$$

By substituting the values for $\sigma_{\mathcal{J}}$ from Figure 4.12 and performing the calculation above in Equation 4.11, 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 Figure 4.12. 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 (4.8) 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 $(-1)^{(|\mathcal{J}|+1)}$.

In order to optimally solve the "offline" Sharing-Aware VM Packing problem, we use the AMPL [30] mathematical programming framework and an open-source solver, Couenne [8], which employs a branch & bound algorithm for solving mixed integer nonlinear programs in general; which is applicable to solving our multilinear program. The "offline" Sharing-Aware VM Packing problem is a new and more complex variant of the bin packing and extends characteristics from the set-union bin packing problem initially considered in Tang

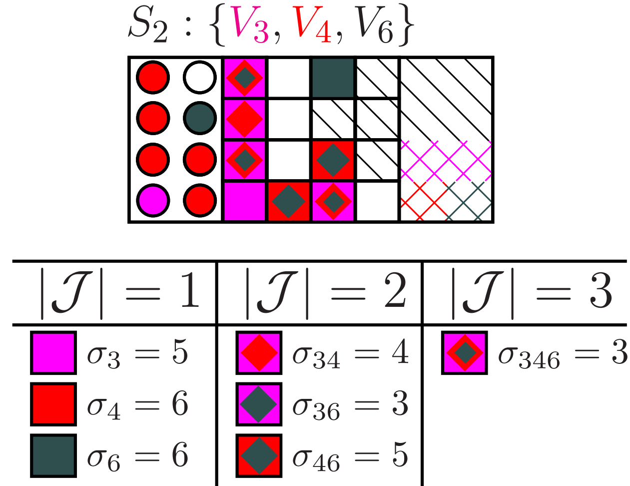


Figure 4.12: Sharing parameter values among V_3, V_4 and V_6

and Denardo [93]. Since bin packing and its variants are strongly $\mathcal{NP}$-hard, we infer that our "offline" Sharing-Aware VM Packing problem is also strongly $\mathcal{NP}$-hard. Therefore, solving the "offline" Sharing-Aware VM Packing problem is only practical for small problems. Solving the "offline" version of the SA-OVMP problem instance in Figure 4.1 only takes a few seconds; although, when we increased the number of VMs to 15 and the number of servers to 8, the time required to solve the problem was approximately 22 minutes. Therefore, heuristic methods, such as those described in Section 4.3, are required in order to efficiently solve problem instances with a large number of VMs and servers considered in real-world applications.

4.5 Experimental Results

In this section, we describe the experimental setup including our strategy for generating VM streams, simulating server configurations, and modeling page sharing. We perform extensive experiments with our sharing-aware online algorithms and their sharing-oblivious counterparts and then analyze the results.

4.5.1 Experimental Setup

All software used for the experiments is implemented in C++ and is run on 2.93 GHz Intel hexa-core dual-processor 64-bit systems within the Wayne State University HPC Grid [102].

Resource	Low Resource Request VMs in Experiments						High Resource Request VMs in Experiments					
	{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

Table 4.8: SA-OVMP Experiment: VM Instance Types.

VM Streams

Fairly recently, Google has made workload usage traces from Google compute cells [83] available to the public. Researchers have thoroughly investigated various components of the usage traces, such as applications [26] and workloads [67] [81] [59]. 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 [81] [17], 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 [81], requests reflecting degrees of fractal self-similarity [17], and the proportion of lower memory and CPU requests significantly outweigh higher memory and CPU requests within the trace [82]. 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 4.8 and are available online [38]. 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 1000 VMs requests which exhibit various percentages of mixture between low and high VM resource requests. We design a set of VM streams accounting for

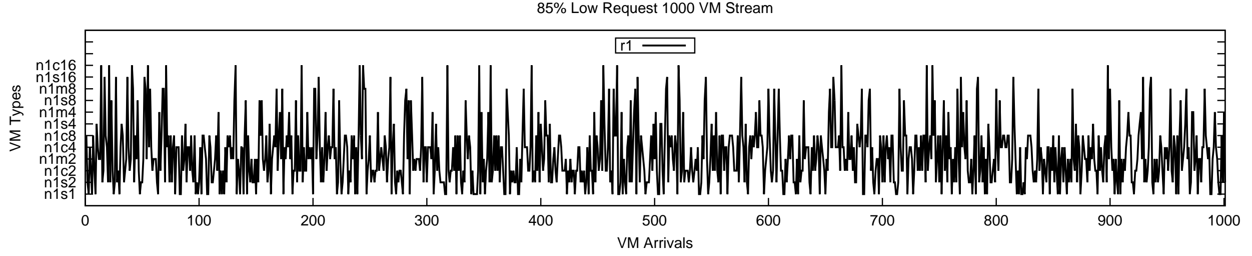


Figure 4.13: 85% Low Request 1000 VM Stream.

various VM type mixtures in increments of 5%, ranging from 5% low (and 95% high) to 95% low (and 5% 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% low request 1000 VM stream, we select 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 5 different orderings for each mixture per 500 and 1000 VM streams; totaling 190 unique VM streams used in our experiments. Figure 4.13 illustrates a 85% low requesting resource 1000 VM r1 stream while Figure 4.14 illustrates a 15% low requesting resource 1000 VM r2 stream. We show the different VM types on the vertical axis and the arrival sequence of the 1000 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% of the VM stream). Stream r2 plot shows that the majority of the VM types correspond to our low resource requests (approximately 15% of the VM stream).

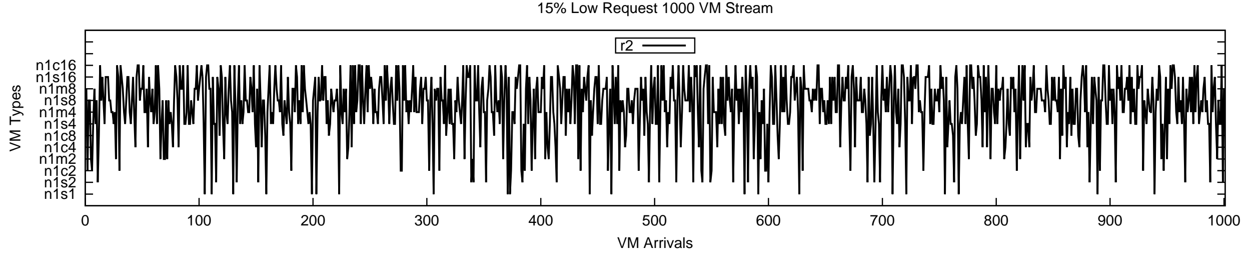


Figure 4.14: 15% Low Request 1000 VM Stream.

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 [81] [59]. 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 [84] 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 *et al.* [59] 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% of the CPU resource, and 25% of the memory resource of the largest machine, and claims 31% of the 12,477 servers, or approximately 3,835 servers. For

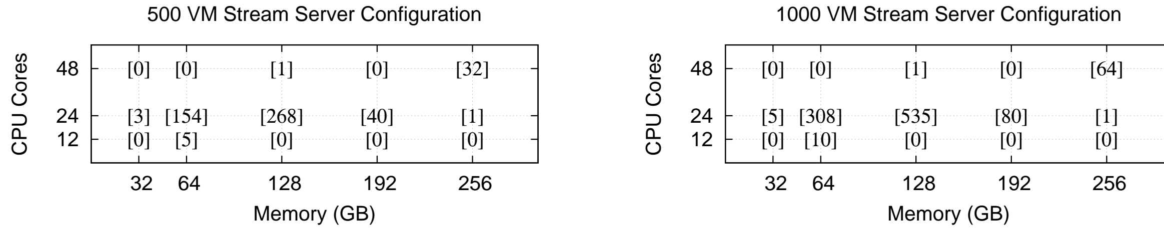


Figure 4.15: Server Configurations.

our experiments, we use the server capacity groups and percentage of group population from Liu *et al.* [59], 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 1000 servers for the 1000 VM streams, where their grouping and percentage of population is consistent with the results from Liu *et al.* [59]. Figure 4.15 illustrates the number of servers per group for the 500 and 1000 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 1000 servers as the maximum number of servers that can be activated.

Modeling Page Sharing

For our experiments, we abstract a subset of the available software from Google Cloud Launcher [36] 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 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 [86]. 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 [86]. 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.* [86] and Bazarbayev *et al.* [7]. 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.

4.5.2 Analysis of Results

We now compare the performance of our proposed sharing-aware online algorithms from Section 4.3 against their sharing-oblivious counterparts. Specifically, we show that by

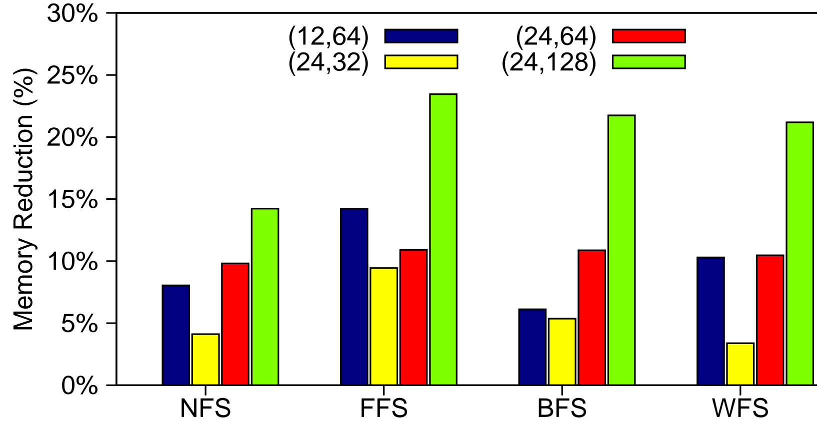


Figure 4.16: Average Memory Reduction: 500 VM Stream.

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 Figure 4.16 and Figure 4.17, 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 1000 VM streams, respectively. We compare our sharing-aware algorithms, NFS, FFS, BFS, and WFS with sharing-oblivious algorithms, Next-Fit (NF), First-Fit (FF), Best-Fit (BF), 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 com-

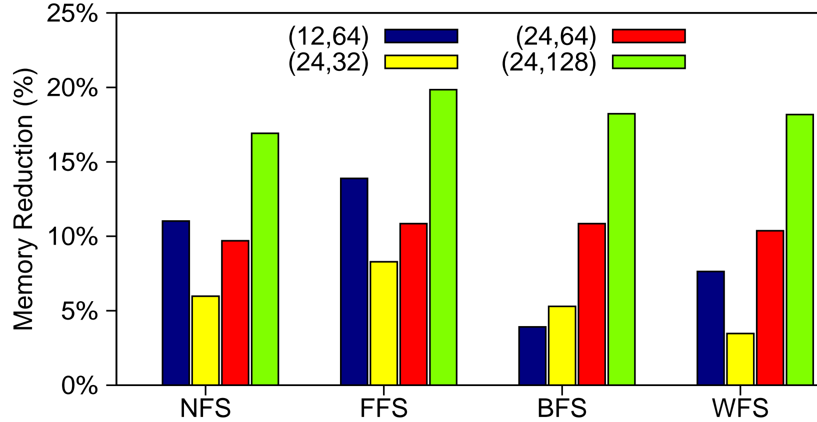


Figure 4.17: Average Memory Reduction: 1000 VM Stream.

pare the overall memory utilization between each sharing-aware algorithm and its sharing-oblivious counterpart. For both 500 and 1000 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 1000 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% in the best case for the largest server capacity category, i.e., (24, 128), and can reduce the required memory by approximately 5% for the worst case in the smallest server capacity category, i.e., (12, 64).

In Figure 4.18 and Figure 4.19, 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% in increments of 5%. 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

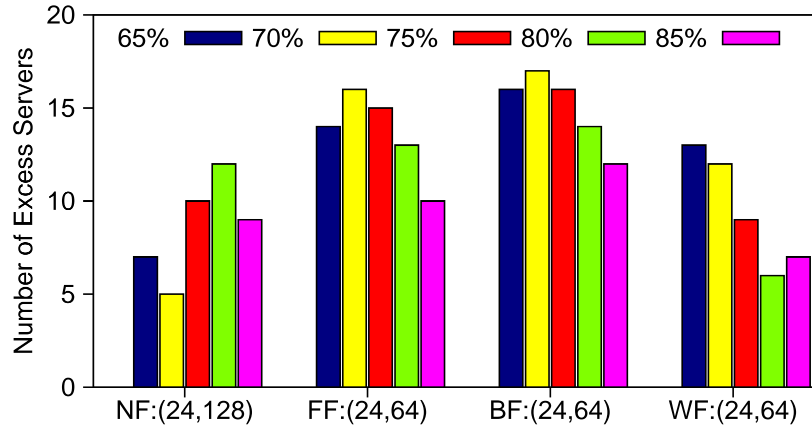


Figure 4.18: Excess Active Servers: 500 VM Stream.

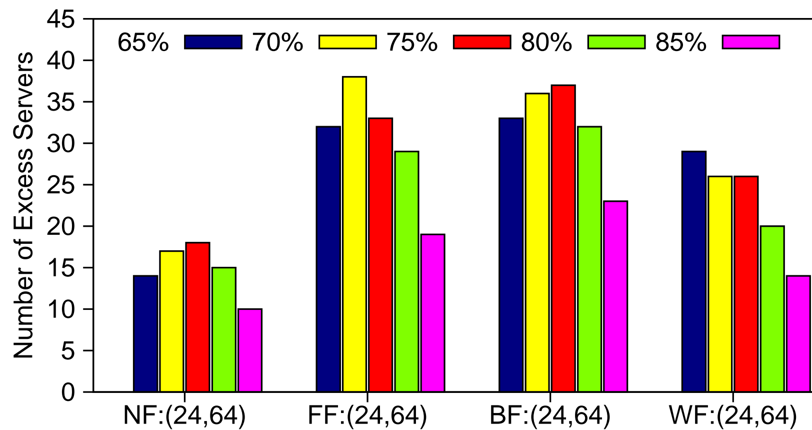


Figure 4.19: Excess Active Servers: 1000 VM Stream.

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 Figure 4.18, 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%, required 16 to 17 extra servers

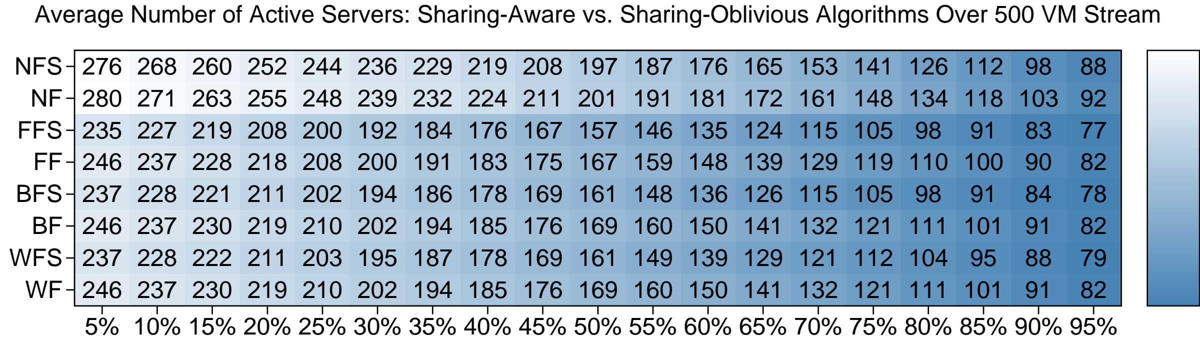


Figure 4.20: Average Active Servers Over All 500 VM Streams.

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 1000 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%; 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.

In Figure 4.20 and Figure 4.21, we compare the average number of servers required to host the VMs for the 500 and 1000 VM streams, respectively, 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 the horizontal axis are the percentages of low resource requesting VMs in the VM stream. The heat map representation has the darkest shade of gray when the highest number of servers are used, e.g., for the 500 VM stream the maximum value is 280 by NF, and has the lightest shade of gray when the lowest number of bins are used, e.g., a minimum value of 77 by FFS also for the 500 VM stream. 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

Average Number of Active Servers: Sharing-Aware vs. Sharing-Oblivious Algorithms Over 1000 VM Stream

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
NFS	553	537	520	504	485	467	448	430	409	389	369	348	326	300	276	250	220	195	172
NF	560	542	526	512	492	472	455	436	418	399	381	360	338	318	294	267	233	208	180
FFS	470	451	434	417	399	381	366	348	330	309	287	263	241	220	205	190	177	163	151
FF	491	472	453	436	417	399	380	363	346	330	312	292	273	253	235	217	197	178	160
BFS	472	454	438	422	403	386	369	353	335	315	291	267	245	224	204	190	178	165	151
BF	491	473	455	440	422	403	386	369	350	333	317	297	276	258	238	218	200	179	161
WFS	472	454	438	422	404	386	370	353	335	315	292	272	252	235	217	202	186	170	154
WF	491	473	455	440	422	403	386	369	350	333	317	297	276	258	238	218	200	179	161

Figure 4.21: Average Active Servers Over All 1000 VM Streams.

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 activates slightly 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 both the 500 and 1000 VM streams occur around the 60% to 85% 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.

4.6 Summary

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 in-

frastructure, and to streaming frameworks. Determining the theoretical performance bounds for the sharing-aware online algorithms is another open avenue for future research.

CHAPTER 5: CONCLUSION

In this Ph.D. dissertation, we presented our research accomplishments in the design and analysis of sharing-aware resource management algorithms for virtual computing environments. We conclude the dissertation by summarizing our contributions and describing possible future research directions.

5.1 Summary of Contributions

In Chapter 1, we detailed the concepts which serve as the foundation for understanding sharing-aware resource management by including an introduction to virtualization, an explanation of how page sharing operates, a motivation for formulating page sharing relationships, and a review of relevant approximation algorithm concepts and models. In Chapter 2, we addressed the problem of sharing-aware VM maximization, SAVMM, in a general sharing model by designing a greedy approximation algorithm, G-SAVMM, based on a new efficiency metric and characterized its worst case performance. We then performed extensive experiments to evaluate the performance of G-SAVMM against other knapsack-like VM allocation algorithms. Our results show that G-SAVMM generates higher revenue and is efficient when compared to the other knapsack-like VM allocation algorithms in our experiments. In Chapter 3, we have addressed the problem of multi-resource sharing-aware VM maximization, MSAVMM, in a general sharing model. We formulated MSAVMM as a new multilinear binary program, BMP-MSAVMM, inspired by the 0-1 knapsack formulation and solved it optimally using small MSAVMM instances. For larger, more realistic MSAVMM instances, we proposed and designed a greedy approximation algorithm, G-MSAVMM, based on a new efficiency metric and characterized its worst case performance. In order to evaluate G-MSAVMM, we detailed unique experiment design strategies through filtering and synthesizing Google cluster workload traces while modeling page sharing behavior using existing results from the literature. To demonstrate the increase in performance by G-MSAVMM, we compared it with the performance of several other knapsack-like VM allocation algorithms using the filtered and synthesized cluster Google workload traces. Our results show that

G-MSAVMM generates much higher revenue and is extremely efficient when compared to the other algorithms in our experiments. In Chapter 4, we addressed the problem of sharing-aware online VM packing with multiple resource requirements and heterogeneous server capacities, SA-OVMP, in a general sharing model. We proposed and designed a family of new sharing-aware online algorithms which solves SA-OVMP; namely, NFS, FFS, BFS, and WFS. We introduced a new server resource scarcity metric necessary for designing BFS and WFS which established cloud server priorities for instantiating online VM requests. We then formulated SA-OVMP as a new multilinear binary program inspired by the 0-1 bin-packing formulation and have optimally solved it using small SA-OVMP instances. Lastly, we performed extensive experiments to compare the performance of our sharing-aware online VM packing algorithms to that of their sharing-oblivious counterparts using the Google cluster workload traces and the PM configurations on which they are derived. Our results show that the proposed family of sharing-aware online algorithms drastically reduces the number of required PMs to instantiate the VM streams when compared to their sharing-oblivious counterparts.

5.2 Future Research Directions

We believe our work will encourage new research in the area of resource management within virtual computing environments. The possible future directions are presented in the next subsections.

5.2.1 Analyzing Sharing-Aware Online VM Packing Performance

Our previous work in VM Packing was focused on the design of online sharing-aware resource management algorithms, investigated their run time complexities and performed extensive experiments measuring their performance. To extend the work therein, deriving performance bounds for the proposed algorithms using metrics suitable for online environments, e.g., competitive and relative worst order ratios, remain open problems in the literature.

Competitive ratios have been studied in the research literature and have been used to characterize the performance of online algorithms in various areas: VM resource manage-

ment [3] [57] [88], packet transmission [94], caching [51], paging [2] [87] and in generalized bin packing settings [19] [31]; yet, to the best of our knowledge, *no study has focused on determining competitive ratios for sharing-aware online resource management algorithms*.

While the competitive ratio has been used in the research literature to characterize the behavior of online performance against offline performance, other metrics [12], e.g., Max / Max ratio [9], random order ratio [53], etc., have evolved which also gauge performance. The *relative worst order ratio* [10] establishes a metric for comparing online algorithms directly by measuring the performance of two comparable online algorithms on their respective worst case input sequence. Relative worst order ratios have been studied in the research literature and have been used to characterize the performance of newly developed bin packing algorithms [10] [28], applied to the seat reservation [13] and paging problems [11]; yet, to the best of our knowledge, *no study has focused on determining relative worst order ratios for online resource management algorithms in a virtual computing environment*. In some cases, the relative worst order ratio is a better quality of measure for online algorithms than the competitive ratio [28].

5.2.2 Sharing-Aware Algorithms for Container Management

Future trends in virtual resource management must consider new provisioning techniques as enterprises are operating at unprecedented scales and experimenting with next-generation technologies. While VMs are the dominant medium for machine instantiation and operating system hosting in clouds, *containers* are making a popular comeback from their inception decades ago. Containers are a lightweight alternative to hypervisor-based virtualization where, unlike hypervisors, containers do not have the overhead of abstracting the PM hardware to virtualize resources. Instead, containers abstract the operating system kernel, where the kernel can then be split into multiple, nested containers. As a result, recent studies have shown the efficiency of utilizing containers over standard VM hypervisor-virtualization [29] [68] [103]. Open source scheduling systems such as Google's Kubernetes and Apache's Brooklyn orchestration framework lead the way for enterprises to

reveal new and efficient means of service virtualization. When institutions such as Google manage 2 billion virtual images weekly, the venue for engineering new algorithms at scale and for next-generation virtual environments while further conserving resources and meeting user demand appear to be wide open.

Google's Kubernetes engineering team has completed *pod* [40]; a dynamic container placement procedure within a cluster inspired by knapsack heuristics. Studying the approximability properties of the knapsack heuristic algorithms through *pods* is an open opportunity of research for both an online and offline setting. Furthermore, investigating the online container to *pod* packing on compute nodes may be studied to address the unique development of systems for dynamic cluster management. Lastly, discovering the approximability properties of bin packing algorithms specific to containers is an open avenue of research. Given the current industry appeal of containers, extentions of our research to sharing-aware algorithms in container-based virtualization environments would be a fruitful endeavor.

5.2.3 Sharing-Aware Streaming Resource Management

We envision an opportunity to extend our sharing-aware algorithms onto systems which consider real-time distributed stream processing. Real-time distributed stream processing is increasingly popular due to responding to events as they occur in areas such as social media, real-time analytics, fraud detection, etc. Apache Storm [90] is an example of a popular open source real-time distributed stream processing framework suitable for these tasks. Therefore, minimizing resource consumption therein would be advantageous to systems which manage these frameworks. In particular, sharing memory resources among multiple, duplicate data streams would reduce overall system memory utilization. This is especially useful for applications consisting of streams which have to be pre-allocated with a specific amount of memory to ensure processing consistency. Very recently, *resource-aware* scheduling for real-time distributed stream processing systems have been proposed in the literature [74]. Therefore, we believe our research can be translated to real-time distributed stream processing frameworks in order to improve their efficiency.

APPENDIX

Journal Publications

J1. S. Rampersaud, L. Mashayekhy, and D. Grosu, "Computing Nash Equilibria in Bimatrix Games: GPU-Based Parallel Support Enumeration." *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, Dec. 2014, pp. 3111-3123.

Paper selected as the Featured Article for the IEEE TPDS December 2014 Issue.

Journal Publications Under Review

R1. S. Rampersaud and D. Grosu, "Sharing-Aware Online Virtual Machine Packing for Heterogeneous Environments." *IEEE Transactions on Parallel and Distributed Systems*, 2016.

R2. S. Rampersaud and D. Grosu, "An Approximation Algorithm for Sharing-Aware Virtual Machine Maximization." *IEEE Transactions on Computers*, 2016.

Conference Publications

C1. S. Rampersaud and D. Grosu, "Sharing-Aware Online Algorithms for Virtual Machine Packing in Cloud Environments." *Proceedings of the IEEE 8th International Conference on Cloud Computing (CLOUD'15)*, NYC, USA, July 2015, pp. 718-725.

C2. S. Rampersaud and D. Grosu, "Sharing-Aware Resource Management Algorithms for Virtual Computing Environments." *Proceedings of the IEEE 3rd International Conference on Cloud Engineering (IC2E'15)*, Tempe, USA, March 2015, pp. 493-495.

Best PhD Symposium Presentation Runner-Up Award

C3. S. Rampersaud and D. Grosu, "A Multi-Resource Sharing-Aware Approximation Algorithm for Virtual Machine Maximization." *Proceedings of the IEEE 3rd International Conference on Cloud Engineering (IC2E'15)*, Tempe, USA, March 2015, pp. 266-274.

C4. S. Rampersaud and D. Grosu, "A Sharing-Aware Greedy Algorithm for Virtual Machine Maximization." *Proceedings of the IEEE 14th International Symposium on Network Computing and Applications (NCA'14)*, Cambridge, USA, August 2014, pp. 113-120.

C5. M. Ahmed, S. Rampersaud, N. Fisher, D. Grosu and L. Schwiebert, “GPU-Based EDF-Schedulability Analysis for Multi-Modal Real-Time Systems.” *Proceedings of the IEEE 15th International Conference on High Performance Computing and Communications (HPCC’13)*, Zhangjiajie, China, November 2013, pp. 254-263.

C6. S. Rampersaud and D. Grosu, “Digital Cancellation Event Options in Limit Order Markets with Automated Liquidity Self-Provisioning.” *Proceedings of the IEEE 10th International Conference on e-Business Engineering (ICEBE’13)*, Coventry, UK, September 2013, pp. 38-43.

C7. S. Rampersaud L. Mashayekhy and D. Grosu, “Computing Nash Equilibria in Bimatrix Games: GPU-based Parallel Support Enumeration.” *Proceedings of the IEEE 31st International Performance Computing and Communications Conference (IPCCC’12)*, Austin, USA, December 2012, pp. 332-341.

REFERENCES

[1] O. A. Abdul-Rahman and K. Aida, "Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon," in *Proc. IEEE Intl. Conf. on Cloud Computing Technology and Science*, 2014, pp. 272–277.

[2] D. Achlioptas, M. Chrobak, and J. Noga, "Competitive analysis of randomized paging algorithms," *Theoretical Computer Science*, vol. 234, no. 1, pp. 203–218, 2000.

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

[4] J. D. Bagley, E. R. Floto, S. C. Hsieh, and V. Watson, "Sharing data and services in a virtual machine system," in *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, ser. SOSP '75. New York, NY, USA: ACM, 1975, pp. 82–88.

[5] I. Banerjee, P. Moltmann, K. Tati, and R. Venkatasubramanian. (2013) Esx memory resource management: Transparent page sharing. [Online]. Available: https://labs.vmware.com/academic/publications/vmware-white-papers

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

[7] 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 Intl. Conf. on Distributed Computing Systems*, 2013, pp. 93–101.

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

[9] S. Ben-David and A. Borodin, "A new measure for the study of on-line algorithms," *Algorithmica*, vol. 11, no. 1, pp. 73–91, 1994.

[10] J. Boyar and L. M. Favrholdt, "The relative worst order ratio for online algorithms," *ACM Trans. Algorithms*, vol. 3, no. 2, May 2007.

[11] J. Boyar, L. M. Favrholdt, and K. S. Larsen, “The relative worst-order ratio applied to paging,” *Journal of Computer and System Sciences*, vol. 73, no. 5, pp. 818 – 843, 2007.

[12] J. Boyar, S. Irani, and K. S. Larsen, “A comparison of performance measures for online algorithms.” in *WADS*, ser. Lecture Notes in Computer Science, vol. 5664. Springer, 2009, pp. 119–130.

[13] J. Boyar and P. Medvedev, “The relative worst order ratio applied to seat reservation,” *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 4, p. 48, 2008.

[14] 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.

[15] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, “Disco: Running commodity operating systems on scalable multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 412–447, Nov. 1997.

[16] T. Carli, S. Henriot, J. Cohen, and J. Tomasik, “A packing problem approach to energy-aware load distribution in clouds,” *CoRR*, vol. abs/1403.0493, 2014. [Online]. Available: http://arxiv.org/abs/1403.0493

[17] 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 Intl. Congress on Big Data*, June 2014, pp. 733–739.

[18] CISCO. Cisco Global Cloud Index: Forecast and Methodology. [Online]. Available: http://newsroom.cisco.com/press-release-content?articleId=1724918

[19] E. G. Coffman Jr., J. Csirik, G. Galambos, S. Martello, and D. Vigo, “Bin packing approximation algorithms: Survey and classification,” in *Handbook of Combinatorial Optimization*. Springer New York, 2013, pp. 455–531.

[20] E. Coffman Jr., M. Garey, and D. Johnson, "Approximation algorithms for bin-packing: An updated survey," in *Algorithm Design for Computer System Design*, ser. International Centre for Mechanical Sciences. Springer Vienna, 1984, vol. 284, pp. 49–106.

[21] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz, "Almost optimal virtual machine placement for traffic intense data centers," in *INFOCOM*, 2013, pp. 355–359.

[22] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling.* New York, NY: Addison-Wesley, 1967.

[23] M. Cordero, L. Correia, H. Lin, V. Thatikonda, R. Xavier, and S. Vetter, *IBM PowerVM Virtualization Introduction and Configuration.* IBM Redbooks, 2013.

[24] J. Czyzyk, M. Mesnier, and J. More, "The neos server," *IEEE J. Computational Sci. Eng.*, vol. 5, no. 3, pp. 68–75, Jul 1998.

[25] G. B. Dantzig, "Discrete-variable extremum problems," *Operations Research*, vol. 5, no. 2, pp. pp. 266–277, 1957.

[26] S. Di, D. Kondo, and C. Franck, "Characterizing cloud applications on a Google data center," in *Proc. 42nd Intl. Conf. on Parallel Processing*, Oct. 2013.

[27] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial, "No justified complaints: On fair sharing of multiple resources," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: ACM, 2012, pp. 68–75.

[28] L. Epstein, L. M. Favrholdt, and J. S. Kohrt, "Comparing online algorithms for bin packing problems," *Journal of Scheduling*, vol. 15, no. 1, pp. 13–21, 2012.

[29] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *IBM Research Report RC25482 (AUS1407-001)*, vol. 28, p. 32, 2014. [Online]. Available: http://www.research.ibm.com/

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

[31] M. R. Garey, R. L. Graham, and J. D. Ullman, "Worst-case analysis of memory allocation algorithms," in *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '72. New York, NY, USA: ACM, 1972, pp. 143–150.

[32] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1979.

[33] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. on Networked Systems Design and Implementation*, 2011, pp. 323–336.

[34] P. Gilmore and R. Gomory, "A linear programming approach to the cutting stock problem," *Operations Research*, vol. 9, pp. 849–859, 1961.

[35] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer*, vol. 7, no. 9, pp. 34–45, Sep. 1974.

[36] Google. (2015) Cloud laucher. [Online]. Available: https://cloud.google.com/launcher/explore

[37] ——. (2015) Google cloud storage. [Online]. Available: https://cloud.google.com/storage/docs/overview

[38] ——. (2015) Google compute engine pricing. [Online]. Available: https://cloud.google.com/compute/pricing

[39] ——. (2015) Kubernetes. [Online]. Available: http://kubernetes.io/

[40] ——. (2015) Pods. [Online]. Available: http://kubernetes.io/v1.1/docs/user-guide/pods.html

[41] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," *Commun. ACM*, vol. 53, no. 10, pp. 85–93, Oct. 2010.

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

[43] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[44] P. Hsieh. (2004) Hash functions. [Online]. Available: http://www.azillionmonkeys.com/qed/hash.html

[45] B. Jenkins. (1997) Jenkins hashing. [Online]. Available: http://www.burtleburtle.net/bob/hash/doobs.html

[46] M. Jeyakanthan and A. Nayak, "Policy management: leveraging the open virtualization format with contract and solution models," *IEEE Network*, vol. 26, no. 5, pp. 22–27, September 2012.

[47] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework," in *Proceedings of IEEE INFOCOM*, March 2012, pp. 1206–1214.

[48] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 299–325, 1974.

[49] D. S. Johnson, "Fast allocation algorithms," in *IEEE Conference Record of 13th Annual Symposium on Switching and Automata Theory, 1972.* IEEE, 1972, pp. 144–154.

[50] S. Kamali and A. López-Ortiz, "Efficient online strategies for renting servers in the cloud," in *SOFSEM 2015: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, vol. 8939, pp. 277–288.

[51] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*, Oct 1986, pp. 244–254.

[52] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems.* Springer, 2004.

[53] C. Kenyon, "Best-fit bin-packing with random order," in *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms.* Society for Industrial and Applied Mathematics, 1996.

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

[55] P. J. Kolesar, "A branch and bound algorithm for the knapsack problem," *Management Science*, vol. 13, no. 9, pp. 723–735, 1967.

[56] T.-C. Lai, "Worst-case analysis of greedy algorithms for the unbounded knapsack, subset-sum and partition problems," *Oper. Res. Lett.*, vol. 14, no. 4, pp. 215–220, Nov. 1993.

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

[58] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel, "An Algorithm for the Traveling Salesman Problem," *Operations Research*, vol. 11, pp. 972–989, 1963.

[59] Z. Liu and S. Cho, "Characterizing machines and workloads on a Google cluster," in *Proc. 8th Intl. Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, Sept 2012.

[60] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations.* New York, NY, USA: John Wiley & Sons, Inc., 1990.

[61] L. Mashayekhy, M. Nejad, D. Grosu, and A. Vasilakos, "An online mechanism for resource allocation and pricing in clouds," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2015.

[62] L. Mashayekhy, M. Nejad, and D. Grosu, "Cloud federations in the sky: Formation game and mechanism," *IEEE Transactions on Cloud Computing*, vol. 3, no. 1, pp. 14–27, Jan 2015.

[63] ——, “Physical machine resource management in clouds: A mechanism design approach,” *IEEE Transactions on Cloud Computing*, vol. 3, no. 3, pp. 247–260, July 2015.

[64] ——, “A ptas mechanism for provisioning and allocation of heterogeneous cloud resources,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 9, pp. 2386–2399, Sept 2015.

[65] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, “Satori: Enlightened page sharing,” in *Proc. of the 2009 Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2009.

[66] D. Minarolli and B. Freisleben, “Utility-driven allocation of multiple types of resources to virtual machines in clouds,” in *Proc. of the IEEE 13th Conference on Commerce and Enterprise Computing*, Sept 2011, pp. 137–144.

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

[68] R. Morabito, J. Kjallman, and M. Komu, “Hypervisors vs. lightweight virtualization: A performance comparison,” in *Proc. 3rd IEEE Intl. Conf. on Cloud Engineering*, March 2015.

[69] M. M. Nejad, L. Mashayekhy, and D. Grosu, “A family of truthful greedy mechanisms for dynamic virtual machine provisioning and allocation in clouds,” in *Proc. of IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 188–195.

[70] ——, “Truthful greedy mechanisms for dynamic virtual machine provisioning and allocation in clouds,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 26, no. 2, pp. 594 – 603, 2015.

[71] Y.-S. Pan, J.-H. Chiang, H.-L. Li, P.-J. Tsao, M.-F. Lin, and T. Chiueh, “Hypervisor support for efficient memory de-duplication,” in *Proc. of the IEEE 17th International Conference on Parallel and Distributed Systems*, Dec 2011, pp. 33–39.

[72] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," *Microsoft Research - Technical Report*, 2011.

[73] R. Parmelee, T. Peterson, C. Tillman, and D. Hatfield, "Virtual storage and virtual machine concepts," *IBM Systems Journal*, vol. 11, no. 2, pp. 99–130, 1972.

[74] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference.* ACM, 2015, pp. 149–161.

[75] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[76] X. Project. (2010) Xen 4.0 release notes. [Online]. Available: http://wiki.xen.org/wiki/Xen_4.0_Release_Notes

[77] S. Rampersaud and D. Grosu, "A sharing-aware greedy algorithm for virtual machine maximization," in *Proc. 13th IEEE Intl. Symp. on Network Computing and Applications*, Aug 2014, pp. 113–120.

[78] ——, "A sharing-aware greedy algorithm for virtual machine maximization," in *Proc. of 13th IEEE International Symposium on Network Computing and Applications*, Aug 2014, pp. 113–120.

[79] ——, "A multi-resource sharing-aware approximation algorithm for virtual machine maximization," in *Proc. 3rd IEEE Intl. Conf. on Cloud Engineering*, March 2015, pp. 266–274.

[80] ——, "Sharing-aware online algorithms for virtual machine packing in cloud environments," in *Proc. 8th IEEE Intl. Conf. on Cloud Computing*, July 2015, pp. 718–725.

[81] 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 Symposium on Cloud Computing*, Oct. 2012.

[82] ——, "Towards understanding heterogeneous clouds at scale: Google trace analysis," Intel Science and Technology Center For Cloud Computing, Carnegie Mellon University, Pittsburgh, PA., Tech. Rep. ISTC-CC-TR-12-101, April 2012.

[83] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2012.03.20.

[84] ——, "Obfuscatory obscanturism: making workload traces of commercially-sensitive systems safe to release," in *Proc. 3rd Intl. Workshop on Cloud Management*, Apr. 2012, pp. 1279–1286.

[85] S. Ross, *A First Course in Probability*, 8th ed. Upper Saddle River, NJ: Prentice Hall, 2010.

[86] M. Sindelar, R. Sitaraman, and P. Shenoy, "Sharing-aware algorithms for virtual machine colocation," in *Proc. of the 23rd ACM symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2011, pp. 367–378.

[87] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, no. 2, pp. 202–208, Feb. 1985.

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

[89] SPEC. (2015, May) Second quarter 2015 SPECvirt_sc2013 results. [Online]. Available: https://www.spec.org/virt_sc2013/

[90] A. Storm. (2015) Storm. [Online]. Available: https://storm.apache.org/

[91] M. I. Sviridenko, "Worst-case analysis of the greedy algorithm for a generalization of the maximum p-facility location problem," *Oper. Res. Lett.*, vol. 26, no. 4, pp. 193–197, May 2000.

[92] S. Takahashi, A. Takefusa, M. Shigeno, H. Nakada, T. Kudoh, and A. Yoshise, "Virtual machine packing algorithms for lower power consumption," in *Proc. of the IEEE 4th*

International Conference on Cloud Computing Technology and Science, Dec 2012, pp. 161–168.

[93] C. S. Tang and E. V. Denardo, "Models arising from a flexible manufacturing machine, part ii: Minimization of the number of switching instants," *Oper. Res.*, vol. 36, no. 5, pp. 778–784, Sep. 1988.

[94] R. Vaze, "Competitive ratio analysis of online algorithms to minimize packet transmission time in energy harvesting communication system," in *Proceedings of IEEE INFOCOM 2013*, April 2013, pp. 115–1123.

[95] V. V. Vazirani, *Approximation Algorithms.* New York, NY, USA: Springer-Verlag New York, Inc., 2001.

[96] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[97] VMWare. (2006) Virtualization overview. [Online]. Available: http://www.vmware.com/pdf/virtualization.pdf

[98] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002.

[99] W. Wang, B. Li, and B. Liang, "Dominant resource fairness in cloud computing systems with heterogeneous servers," in *2014 Proceedings of IEEE INFOCOM*, April 2014, pp. 583–591.

[100] L. Wei, B. He, and C. H. Foh, "Towards multi-resource physical machine provisioning for IaaS clouds," in *Proc. of IEEE International Conference on Communications*, June 2014, pp. 3469–3472.

[101] 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. of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2009, pp. 31–40.

[102] WSU. Wayne State University HPC Grid. [Online]. Available: http://www.grid.wayne.edu

[103] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb 2013, pp. 233–240.

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

[105] F. Xu, F. Liu, and H. Jin, "Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud," *IEEE Trans. on Computers*, vol. PP, no. 99, 2015.

[106] S. Zaman and D. Grosu, "Combinatorial auction-based mechanisms for vm provisioning and allocation in clouds," in *Proc. of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2012, pp. 729–734.

[107] ——, "A combinatorial auction-based mechanism for dynamic vm provisioning and allocation in clouds," *IEEE Trans. on Cloud Computing*, vol. 1, no. 2, pp. 129–141, 2013.

[108] ——, "Combinatorial auction-based allocation of virtual machine instances in clouds," *J. Parallel Distrib. Comput.*, vol. 73, no. 4, pp. 495–508, Apr. 2013.

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

ABSTRACT

SHARING-AWARE RESOURCE MANAGEMENT ALGORITHMS FOR VIRTUAL COMPUTING ENVIRONMENTS

by

SAFRAZ RAMPERSAUD

May 2016

Advisor: Dr. Daniel Grosu

Major: Computer Science

Degree: Doctor of Philosophy

Virtualization technologies in cloud computing are ubiquitous throughout data centers around the world where providers consider operational costs and fast delivery guarantees for a variety of profitable services. These providers should consistently invoke measures for increasing the efficiencies of their virtualized services in a competitive environment where fast entry to market, technology advancement, and service price differentials separate sustaining providers from antiquated ones. Therefore, providers seeking further efficiencies and revenue generating opportunities should consider how their resources are managed in virtual computing environments which leverage memory reclamation techniques, specifically *page-sharing*; motivating the design of new memory *sharing-aware* resource management algorithms. In this dissertation, we design families of offline and online sharing-aware algorithms for resource management in virtual computing environments and investigate their properties within a general sharing model. We evaluate our proposals by applying them to heterogeneous resource domains where large, re-engineered trace dataset inputs are developed in order to compare our algorithms. Lastly, we outline their applications to next-generation virtualization technologies and streaming architectures.

AUTOBIOGRAPHICAL STATEMENT

Safraz Rampersaud received his BSc degree in mathematics from Wayne State University in 2003. Following graduation, he worked with Detroit Public Schools as a mathematics Fellow for the NFS GK-12 program. During this time, he studied for his MSc degree in applied mathematics focusing on optimization as a student under Dr. Mordukhovich at Wayne State University. Following the completion of his MSc degree in 2005, he left for Baltimore, MD to work with Wells Fargo Structured Products Group, N.A. as a Securities Analyst and later worked as a Senior Operations Analyst with the Federal Home Loan Mortgage Corporation (Freddie Mac) in McLean, VA.

In Fall 2010, he entered the Ph.D. program in computer science at Wayne State University as a student under Dr. Grosu and was selected to participate as an NSF IGERT Fellow for the Socio-Technical Infrastructure for Electronic Transactions (STIET) program. During this time, he has published eight peer-reviewed papers in venues such as IEEE CLOUD, IEEE HPCC, IEEE IC2E, and IEEE ICEBE, has received awards for research (Stephen P. Hepler Award, Best PhD Symposium Presentation Runner Up Award), teaching (Outstanding Faculty Award, Tau Beta Phi Outstanding Teaching Service Award), and has had an article selected as the featured article for the December 2014 issue of IEEE Transactions on Parallel and Distributed Systems. His research interests include applied mathematics, approximation algorithms, distributed systems, e-commerce and virtualization.