Market Driven Multi-Resource Allocation

Liran Funaro
Market Driven Multi-Resource Allocation

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Liran Funaro

Submitted to the Senate of the Technion — Israel Institute of Technology
Tevet 5780 Haifa January 2020
This research was carried out under the supervision of Prof. Assaf Schuster and Dr. Orna Agmon Ben-Yehuda, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


**Acknowledgements**

I am grateful to my advisors, Orna Agmon Ben-Yehuda and Assaf Schuster, for the countless hours of invaluable guidance, insights, and support. With your unending patience, you paved the way for me to grow as a researcher, and for that I am indebted. You have been mentors and friends.


I thank my extended family and friends for their encouragement, understanding and, especially, for their faith in me.

Most of all, I thank my wife, Tamar Camus. I dedicate this work to you. You provide me with comfort, motivation, advice, endless love and support, which helps put everything into perspective. Thank you for patiently listening to my talks, reading my papers, giving me invaluable editorial notes, enduring alongside me every deadline, and reminding me to celebrate every success. I could not have done this without you.

The generous financial support of the Technion, the Hasso-Plattner Institute, and the Professor A. Pazi Joint Research Foundation is gratefully acknowledged.
# Contents

List of Figures  
Abstract  
1 Introduction  
2 Ginseng: Market-Driven LLC Allocation  
  2.1 Introduction  
  2.2 System Architecture  
  2.3 Cache Architecture  
    2.3.1 Restricting LLC Access: The Pit  
  2.4 Cache Auction  
  2.5 Auction Protocol  
  2.6 Auction Rules  
    2.6.1 Allocation Rule  
    2.6.2 Payment Rule  
    2.6.3 Complexity  
  2.7 Experimental Setup  
    2.7.1 Machine Setup  
    2.7.2 Workloads  
    2.7.3 Alternative Cache Allocation Methods  
    2.7.4 Time Scales  
  2.8 Evaluation  
    2.8.1 Comparing Social Welfare  
    2.8.2 Influence of Off-Line Profiling  
    2.8.3 To Share or Not To Share  
    2.8.4 Dynamic Allocation Overhead  
  2.9 Related Work  
  2.10 Conclusions and Future Work  
3 Efficient Multi-Resource, Multi-Unit VCG Auction  
  3.1 Introduction  
  3.2 The Non-Linear Optimization Problem
3.3 Joint Valuation Algorithm .................................................. 35
  3.3.1 Finding the Optimal Allocation ......................................... 35
  3.3.2 Payment Computation .................................................. 36
  3.3.3 Joining Two Valuation Functions ...................................... 36
  3.3.4 Upper-Bound Limit .................................................... 38
3.4 Correctness Proof ........................................................ 42
  3.4.1 Notations ............................................................... 42
  3.4.2 Supporting Lemma ..................................................... 42
  3.4.3 Proof by Induction .................................................... 43
3.5 Complexity Analysis of Joining Two Valuations ......................... 44
  3.5.1 Worst Case ............................................................ 44
  3.5.2 Single Resource ...................................................... 45
  3.5.3 Multiple Resources ................................................... 45
3.6 Evaluation ........................................................................ 46
  3.6.1 Implementation Details ................................................. 46
  3.6.2 Benchmark Dataset ..................................................... 47
  3.6.3 Experimental Setup .................................................... 48
3.7 Results ............................................................................. 48
  3.7.1 Naïve Joining of Valuation Functions .................................. 49
  3.7.2 Ideal Case Analysis ..................................................... 49
  3.7.3 Data Structure Analysis ................................................ 49
  3.7.4 False Positives .......................................................... 50
  3.7.5 Separate Single-Resource Auction ..................................... 51
  3.7.6 Verification ............................................................... 52
3.8 Related Work ...................................................................... 52
3.9 Conclusions and Future Work ............................................. 52
4 Memory Elasticity Benchmark .............................................. 55
  4.1 Introduction ...................................................................... 55
  4.2 Memory Elasticity Methods ................................................. 58
    4.2.1 Applications with Resource Trade-off ............................... 58
    4.2.2 Memory-Aware Applications ....................................... 59
    4.2.3 Multiple Short-Lived Jobs .......................................... 60
    4.2.4 Rigid Applications ................................................... 60
  4.3 Memory Elasticity Metrics .................................................. 60
    4.3.1 Static Metrics .......................................................... 61
    4.3.2 Dynamic Metrics ...................................................... 62
  4.4 Evaluation ......................................................................... 64
    4.4.1 Measuring the Elasticity Metrics .................................... 64
    4.4.2 Validating our Metrics ............................................... 66
    4.4.3 Applications and Benchmarks ...................................... 67
4.4.4 Implementation Details ................................. 68
4.4.5 Experimental Setup ................................. 69
4.5 Results ................................................. 69
  4.5.1 Memcached ........................................ 69
  4.5.2 Memory Consumer .................................. 70
  4.5.3 Validation ......................................... 71
4.6 Related Work ............................................. 73
4.7 Conclusions and Future Work .......................... 73

5 Stochastic Resource Allocation .......................... 75
  5.1 Introduction ........................................... 75
  5.2 Allocation Mechanisms and Incentives ................. 77
    5.2.1 Client Requirements ............................ 78
    5.2.2 Provider Goals .................................. 78
    5.2.3 Allocation Mechanisms .......................... 79
  5.3 StochasticAllocation ................................. 80
    5.3.1 Implementing Stochastic Allocation via Shares ... 81
  5.4 Realistic Workload Modeling ......................... 82
    5.4.1 Performance ........................................ 82
    5.4.2 Load ................................................. 82
    5.4.3 Valuation ........................................... 84
  5.5 Evaluation Methodology ............................... 87
  5.6 Compared Mechanisms ................................ 89
  5.7 Results ............................................... 91
    5.7.1 Provider’s Economic Benefit ...................... 91
    5.7.2 Server Utilization ................................ 92
    5.7.3 Clients’ Preferences .............................. 93
    5.7.4 Clients’ Attainment Ratio ....................... 93
    5.7.5 Validation ........................................... 94
  5.8 Related Work .......................................... 96
  5.9 Conclusions and Future Work ........................ 97

6 Conclusion ................................................. 99

A Software .................................................. 101
  A.1 Cache Set Finder (AKA Plumber) .................... 101
  A.2 VCG Algorithm Implementation ..................... 101
  A.3 Elastic Java-Virtual-Machine (JVM) ................. 102
  A.4 Stochastic Allocation Simulator (SAS) ............. 102

Hebrew Abstract ............................................ i
List of Figures

2.1 Ginseng system architecture ........................................ 10
2.2 Performance of various applications, normalized by their performance in
the *pit*. Measured with 11 other guests assigned to the *pit*. All of the
applications were allocated two hardware threads. ................. 16
2.3 *Composite-Scimark* performance when sharing the cache with cache-
polluter vs. non-cache-polluter applications. The performance was nor-
malized to the minimum measurement in all the experiments. ....... 17
2.4 Example of performance profiling: *Composite-Scimark* under different
server loads (i.e., active VMs) and with different numbers of allocated
hardware threads. .................................................. 19
2.5 *Composite-Scimark* valuation function for different server-loads (i.e., ac-
tive VMs) and when allocated 2 hardware threads. Note the different
scale in the vertical axes. ........................................... 19
2.6 Maximum improvement factor of *Ginseng* compared to the shared-cache
and ideal-static methods with different assumptions on the number of
high, medium, and low valuation customers. The maximum is over any
number of guests with the application, or mixture of applications. .... 22
2.7 Social welfare under different cache allocation methods as a function of
the number of guests. The dashed lines indicate an experiment where
the clients perform identically to the profiler (artificial clients). Cache-
utilizer applications can greatly benefit from *Ginseng*. Cache-neutral
applications can still enjoy the benefits of *Ginseng*, albeit to a lesser
extent. Applications with a small memory working set will prefer sharing
the cache with others like it. ........................................ 23
2.8 Expected performance deviation for all applications in all of our ex-
periments. The *pit* measurements are excluded as the performance is
expected to fluctuate when sharing a small number of cache ways. .... 24
2.9 Social welfare under hard vs. soft-partitioning. Striped columns indicate
better social welfare under soft-partitioning. Black indicates the opposite. 26
2.10 Social welfare improvement under soft-partitioning compared with hard-partitioning for various application pairs. Boxes show the middle 50% of the values over different valuation scale factor ratios. Whiskers mark extreme values.

3.1 Example of a four-dimensional binary search tree. Each letter represents a four-dimensional vector. Each array in a rectangle marked with $dX$ is sorted by dimension X.

3.2 The performance of our algorithm in each of our datasets (concave, increasing and mostly-increasing).

3.3 The performance of the naïve approach for joining two valuation functions. Markers depict the performance with different numbers of resources. The lines are the complexity function described in Section 3.3.3, scaled to fit the markers.

3.4 The average number of matching allocations for each query for each dataset and number of resources.

3.5 The runtime portion of each function of data structures (DS) when used by our algorithm in an auction. The whiskers represent the 25th and 75th percentiles.

3.6 False-positive ratio for the combination of data structures for each number of resources. The translucent bands represent the confidence intervals.

3.7 The social welfare when using a separate single-resource auction normalized to the optimal social welfare. The whiskers represent the standard deviation.

4.1 Off-the-shelf memcached performance ("get" hits per second) as a function of allocated memory when using 500MB internal cache size and memslap as its workload. Data is according to Agmon Ben-Yehuda et al. [ABYPBY+14].

4.2 PostgreSQL tested with Pgbench [Ish14] workload, with 40 clients, while changing the memory allocation on the fly.

4.3 Lusearch benchmark from the DaCapo benchmark suite [BGH+06] with different limits on the maximal heap size in an off-the-shelf JVM. These measurements were produced by restarting the JVM for each maximal heap size.

4.4 Illustrating performance measurements of an application.

4.5 The definition of $T_{mem}$ given an application’s performance under dynamic conditions.

4.6 Illustration of two applications with the same $T_{mem}$ and $P_{mem}$, but with a different performance loss during the transient period.
4.7 Memcached performance ("get" hits per second) as a function of allocated memory. ........................................ 69
4.8 Elastic and off-the-shelf memcached oscillation test results. .......... 70
4.9 Memory consumer performance as a function of allocated memory. ... 71
4.10 Memory consumer oscillation test results. .............................. 71
4.11 Memcached profiler prediction accuracy distribution. Each rectangle shows the results for different parameters according to its row and column. The joined row shows all the experiments with any memory amplitude, and the joined column shows all the experiments with any change rate. ................................................................. 72
4.12 Memory consumer profiler prediction accuracy distribution with a rate of 300 memory changes per hour and a maximal amplitude of 2 GB. . 72

5.1 Generated performance function examples. .............................. 83
5.2 A client’s generated load. This client’s required performance was 42% on average. ................................................................. 85
5.3 A probability density function (PDF) of the portion between the two valuation types. The horizontal axis describes the long-term requirements portion, and the immediate requirements portion completes it to 1. . . 86
5.4 Creating the valuation function for various maximal available CPU usage values. This client’s immediate requirements constitute 77% of its budget. 87
5.5 Iterative states in our evaluation methodology. .......................... 87
5.6 A typical iterative process and its convergence. The number of clients per server increases over time until it peaks and starts to drop due to the high utilization and the increasing number of shares per server, which reduce the value of a share. The provider’s revenue decreases over time as more clients switch to cheaper bundles (with shares). ......................... 88
5.7 CDF of the potential resource use for a number of shares. ............... 90
5.8 Comparison of allocated clients per server and provider’s revenue. The number following the mechanism’s name is the share price of each tested case. The revenues are normalized by the FP revenue. The error bars indicate the variance in CPS between the servers. ......................... 92
5.9 The provider’s profit and profit margin. Given a server cost, the plot shows the expected profit and profit margin of the provider if it were to offer each tested mechanism. The profits are normalized by the FP revenues. ................................................................. 93
5.10 Comparison of the social welfare. ......................................... 94
5.11 CPU utilization distribution for each tested case, differentiated by the total utilized CPU (left) and the portion of the CPU consumed only by clients that reserved it (right). ................................. 94
5.12 The distribution of clients’ selected bundles. Each color represents a different tested unit price for the shares. .......................... 95
5.13 The selected number of virtual cores in our fixed-performance experiment and in the Azure dataset. ............................... 95
5.14 The distribution of clients’ revenue normalized by their expected value in each tested case. .............................................. 96
Abstract

Suboptimal resource utilization among public and private cloud providers prevents them from maximizing their economic potential. Long-term allocated resources are often idle when they might have been subleased for a short period. Providers could address this problem by overcommitment of their resources, but this may lead to unpredictable client performance if all clients try to use the resources simultaneously. A better alternative would be for cloud providers to allocate their physical resources to their clients dynamically, as needed, thereby maximizing the benefit that the former get out of given hardware, and maintaining the latter’s satisfaction.

In this work we focus on economic resource allocation mechanisms for maximizing the provider’s profits and the aggregated value all clients draw from the cloud. We provide two approaches to achieving this objective: an auction-based mechanism, and a stochastic resource allocation coupled with a smart pricing scheme.

In developing these mechanisms, we had to overcome a number of challenges. One challenge involved the high computational complexity of the optimization problem required by the auction mechanism. To solve it, we developed an efficient auction algorithm.

The other challenge was the lack of benchmarks to evaluate a memory allocation scheme. We needed applications whose performance is proportional to the memory availability, i.e., memory elastic applications. Accordingly, we compiled a set of memory elastic benchmarks, as well as a language and a method to evaluate their elasticity.
Chapter 1

Introduction

Resource allocation is one of the main challenges faced by public and private cloud providers. One main purpose of a resource allocation scheme is to make it possible to serve all clients on each server according to their service-level-agreement (SLA). The allocation scheme design is important because, among other things, it can directly affect the resource utilization of the provider’s servers. This, in turn, will affect the number of clients the provider can accommodate per server without violating the clients’ SLA. Hence, a well-designed resource allocation scheme can reduce the provider’s operation costs. Nonetheless, the provider’s profits may not increase if the income from clients is reduced compared with existing allocation schemes. Thus, the allocation scheme needs to be coupled with a compatible pricing scheme that ensures commensurate income from clients.

This work explores designs for market-driven resource allocation schemes that will increase resource utilization on servers while taking into account the provider’s and clients’ financial needs. To this end, we need to find the gap between current resource utilization and an optimal one. We also need to understand the origins of the gap. In addition, we need to consider the different financial requirements of private and public cloud providers, as well as their clients’ financial requirements.

The main cause of non-optimal resource utilization is fixed, reserved, resource bundles. Such bundling is still the most prevalent resource allocation providers offer to clients. In these services, clients rent a bundle with a fixed resource allocation that is reserved for them for a long period (e.g., a month). They choose a bundle that will satisfy their needs for the entire duration. Nevertheless, they do not use their resources all the time. The provider guarantees with good probability that the clients will be able to use their rented resources at any given time, thus it is forced to reserve these resources and cannot resell them or use them for other purposes. Consequently, at any given time, a large portion of the resources in the servers might be unutilized.

Since clients are selfish, they will always choose the superior bundle of resources when it costs them the same as an inferior bundle. A superior bundle of resources will require the provider to reserve more resources to the client, and thus will reduce
the server’s resource utilization. Our approach to the problem is to incentivize clients to favor a bundle with minimal reserved resources. We give them an option to add resources on-the-fly on demand instead of fixed, reserved, resource bundles. We achieve this goal by designing allocation mechanisms that incorporate a smart pricing scheme.

We adopted two different mechanisms to achieve our objective, each designed to accomplish different goals. One mechanism is an auction-based one that optimizes the clients’ economic benefit. The other is a new approach in which we allocate a stochastic amount of resources alongside a fixed, reserved amount. The stochastic allocation mechanism is coupled with a smart pricing scheme, designed to maintain the provider’s income from client’s payments and increase the provider’s profits. Both mechanisms improve hardware utilization by using some kind of economic mechanism that incentivizes clients to reduce the fixed, reserved portion of their bundle.

**Auction-Based Mechanism**

We use an auction mechanism that optimizes the *social welfare*: a game-theoretic concept, which means the aggregated value all the clients draw from the cloud. In this mechanism, each client rents a base resource bundle that is reserved exclusively for it. Thereafter, every short period, each client bids according to its valuation for an additional resource allocation amount. Then, the provider actuates the auction mechanism, which solves the optimization problem: finding the allocation that maximizes the social welfare. This approach ensures that the resources are allocated in proportion to the clients’ valuation of the resources. Thus, clients with higher valuations will have a higher quality of service (QoS). The mechanism is designed such that if the demand is low, clients can rent additional resources at a very low price, which is financially beneficial to them. Moreover, it incentivizes clients to rent a smaller reserved bundle because they can bid for additional resources at a lower price on average compared to the initial reservation price they paid.

This mechanism was first introduced by Agmon Ben-Yehuda et al. [ABYPBY+14] for memory (RAM) allocation. We extended this mechanism to last-level-cache (LLC) allocation. In Chapter 2, we show that using an auction to allocate LLC can improve the social welfare by up to 42.8 times compared with the state-of-the-art static and dynamic allocation schemes. This mechanism allows more clients per server without hindering the clients QoS, because it takes the clients’ requirements into account at every given time period.

The work on memory and LLC allocation presented two new challenges.

The first challenge arises from the high computational complexity of the auction mechanism, which forces a long time period between auctions—more than an hour—when auctioning multiple resources (e.g., memory, LLC, CPU and bandwidth). In Chapter 3 we introduced a new efficient multi-resource auction algorithm with a
pseudo near-linear complexity. Using this algorithm, a provider can deploy a multi-
resource auction for allocation of additional resources in an existing virtual machine
(VM) every two minutes, for up to 256 clients in a single physical machine.

The second challenge arises from the need for applications to be elastic in their
memory requirements so that dynamic memory allocation will be possible. In other
words, their performance should be proportional to the memory availability. We note
that—to date—such applications are scarce, which makes it difficult to evaluate the
contribution of a dynamic memory allocation scheme. In Chapter 4, we constructed
a benchmark suite of memory-elastic applications, and formulated a language and a
method to evaluate their memory elasticity.

**Stochastic Allocation Mechanism**

The auction approach has two main drawbacks. To find the optimal resource allocation,
the auction mechanism requires a large and constant amount of computational power
that could have been allocated to the clients. In addition, the auction mechanism we
used does not take the provider’s profits into consideration. To mitigate this, we offer
an alternative approach, called stochastic allocation, that does not require the provider
to solve an optimization problem. Under the stochastic allocation mechanism, the
provider offers clients a combination of a quantity of reserved resources with a choice
of a number of shares, which represents a stochastic allocation of the resources. The
provider posts fixed unit-prices for both goods and publishes statistics periodically on
resource availability for each number of stochastic allocation shares. Each client may
choose to rent reserved and/or stochastic resources.

In Chapter 5, we investigate the optimal pricing for stochastic allocation shares for a
CPU in which the number of clients per server is maximal—5.6 times higher compared
with selling only fixed, reserved resources, and 1.7 times higher compared with burstable
instances, which is the most prevalent flexible allocation method used today. We also
show that using our mechanism, the provider’s profits improve significantly, by 28%–
44% compared to burstable instances, and the social welfare is over 98% of the optimum.
Chapter 2

Ginseng: Market-Driven LLC Allocation

Abstract

Cloud providers must dynamically allocate their physical resources to the right client to maximize the benefit that they can get out of given hardware. Cache Allocation Technology (CAT) makes it possible for the provider to allocate last level cache to virtual machines to prevent cache pollution. The provider can also allocate the cache to optimize client benefit. But how should it optimize client benefit, when it does not even know what the client plans to do?

We present an auction-based mechanism that dynamically allocates cache while optimizing client benefit and improving hardware utilization. We evaluate our mechanism on benchmarks from the Phoronix Test Suite. Experimental results show that Ginseng for cache allocation improved clients’ aggregated benefit by up to $42.8\times$ compared with state-of-the-art static and dynamic algorithms.

2.1 Introduction

Infrastructure-as-a-Service (IaaS) cloud computing providers rent computing resources wrapped as an infrastructure, i.e., a guest virtual machine (VM), to their clients. To compete in the tough market of cloud computing, providers must improve their clients’ quality-of-service (QoS) while maintaining competitive pricing and reducing per-client management cost. Thus, better hardware utilization is necessary. New Intel technology that supports last level cache (LLC) allocation allows better cache utilization via cache partitioning.

Providers can utilize this new technology to guarantee clients’ performance requirements by preventing applications from polluting each other’s cache [KW13], as Intel
intended [Int15]. Moreover, they can accommodate more clients’ performance requirements by granting more LLC to those who benefit from it and preventing access to those who do not. This increases the provider’s ability to consolidate the physical host.

Without any client performance information, the provider can only optimize guest performance according to host-known metrics, such as instructions-per-second (IPC), LLC hit-rate, LLC reads-count, and so forth [GLJ+15, YBDJ15, LCG+15]. The host does not know what the client’s real benefit from more cache is, nor can it compare benefits that different clients draw from cache. For example, higher IPC does not necessarily indicate better performance, as the guest VM might just be polling on a spin-lock more quickly.

Moreover, a client may be willing to settle for poorer performance in exchange for a lower payment [ABYSS+12]. This might be the case, for example, when the guest VM of a performance-demanding client is running maintenance work in between important workloads every few seconds. Nevertheless, lacking the guests’ current workload information, the host will try to improve its performance despite the lack of benefit to the client. This, in turn, may hinder the performance of other guests. It is therefore in the interest of both provider and client that clients pay only for the fine grained LLC they need, when they need it [ABYBYST14, ABYBYST12, AFG+10, GHDS+11, OZN+12, MABYS16]. Client satisfaction will thus be improved, as clients can pay less for the same performance but only when it is really needed, while providers will be able to improve hardware utilization.

However, real-world public cloud clients are selfish, rational economic entities. They will not let the provider know precisely how much benefit each quantity of cache ways would bring to it. They are black-boxes, and as such, unlike white-box clients [GHDS+11, HZPW09, HGS+11, NKG10], they will not share their true private information with their provider unless it is in their own best interest to do so. For example, if the host allocates cache ways to guests who will derive the greatest benefit from it, each guest will claim that it has the most to gain from additional cache ways. Likewise, if the host allocates cache ways to guests who perform poorly, each guest will claim poor performance.

Even passive black-box measurements taken by the provider can be manipulated [GLJ+15, YBDJ15, LCG+15]. For example, a guest can fake cache misses by adding random instructions in the code that access random—non-cached—memory addresses. Such instructions will not delay the out-of-order-execution (OOOE) CPU as they are independent of the other instructions, and they will induce many cache misses.

In this chapter of the thesis we address the problem of how cloud providers should allocate cache among selfish black-box clients in light of the new cache allocation technology.

Our contribution towards a solution to this problem is Ginseng [ABYPBY+14] for cache allocation, a market-driven auction system that maximizes the aggregated benefit of the guests in terms of the economic value they attribute to the desired
allocation, using game-theoretic principles. This approach encourages even a selfish guest to bid for cache according to its true benefit. Ginseng was first introduced for memory allocation [ABYPBY+14], and a similar, auction-based approach was used before for bandwidth allocation [LS99]. Furthermore, Amazon has been auctioning virtual machines since 2009 [ABYBYST13].

We evaluate Ginseng on benchmarks from the Phoronix Test Suite [LT11], which we classify according to their benefit from the cache. We show that Ginseng improves the aggregated economic benefit of guests from cache by up to 42.8× compared to the prevalent method used by today’s cloud providers.

Our second contribution is an evaluation of the attributes which differentiate dynamic cache allocation from other resources: (1) As opposed to memory, cache does not have to be exclusively allocated and can be shared effectively. However, mutual trust is required to allow benefit for the sharing participants. (2) Unlike bandwidth—but much like memory—cache has to be warmed up before the guest can benefit from it. However—unlike memory pages—cache must be allocated consecutively, which induces more cache way transfers when the allocation is changed. Furthermore, Intel’s allocation mechanism might fail to enforce frequent transfers of cache ownership. Thus, dynamic cache allocation might incur performance overhead. We address the question of when it is beneficial to share cache and when exclusive allocation is preferable, and we measure and analyze the overhead of frequently changing the allocation.

2.2 System Architecture

Ginseng is a market-driven cloud mechanism that allocates resources to guest virtual-machines by means of an auction. It is implemented for cloud hosts running the KVM hypervisor [KKL+07] but can work seamlessly on any other hypervisor. Ginseng for cache allocation controls the cache ways allocated to each guest using the cache-driver described in §2.3.

Ginseng has a host component and a guest component, as depicted in Figure 2.1. The host’s Auctioneer uses the Vickrey-Clarke-Groves (VCG) auction [Cla71, Gro73, Vic61], to be described in §2.4. The host’s communicators communicate with the guests using the auction protocol, detailed in §2.5. It also instructs the cache controller how to allocate cache ways among the guests. The guest’s economic agent bids on behalf of the client by stating a valuation for each number of cache ways. The guest component we implemented bids with a true valuation, as that is the best strategy for the guest [Cla71, Gro73, Vic61], but Ginseng does not enforce any restrictions on the implementation and strategy of the guest’s economic agent.
2.3 Cache Architecture

Intel’s cache, and LLC in particular, stores data in granularity of cache lines that typically vary from 64 to 256 bytes, depending on the machine. The cache is organized in ways, each of which is a hash table, where the key is a hash value of the line’s memory address and the value is the content of the cache line. Way locations that are designated to be filled by lines with the same keys are called a set.

When reading from a memory address or writing to it, the CPU first computes the address’s set by using the hash function. Then the line is stored in this set on one of the cache ways. If the entire set is full, the least recently used (LRU) line in the set will be evicted and replaced.

When an application uses the cache exclusively, it will evict its own least recently used data. However, when several applications use the same cache, one might evict the other’s cache lines and influence its performance. To prevent this, Intel’s new cache allocation technology (CAT) allows cache partitioning. The API defines the notion of classes-of-service (COS), which determine a set of cache ways. When a hardware thread is assigned to a COS, it is only allowed to store new cache lines in the ways determined by the COS. However, the COS does not limit reading from any of the ways in the cache.

Intel’s API requires that the selected ways in each COS be consecutive. The API does not impose exclusivity, so a cache way can be used by more than one COS.

We experimented with new hardware (Haswell) that supports the new cache allocation technology. It supports only four COSes and requires a minimum allocation of two cache ways for each COS. However, more advanced architectures such as Broadwell will support a minimum of one cache-way allocation and 16 COSes [Int].

Intel has already added support for CAT to the Linux kernel (tip) via the cgroups interface. However, at the time we experimented with the hardware, the modification was only available as a patch and was not stable enough. Therefore, we implemented our own user-level driver that allows control over the COSes and their assignment.
to CPUs. We implemented it in Python by writing directly to the model-specific registers (MSR) using rdmsr/wrmsr utilities for Linux. The driver code is available at https://github.com/liran-funaro/cat-driver.

2.3.1 Restricting LLC Access: The Pit

Preventing LLC access to some guests will allow us to allocate more cache ways to others who might benefit more from it. However, the cache allocation API does not allow LLC access to be restricted to specific guests. Instead, we assign them to a single COS we denote the pit. We allocate the pit the minimum number of cache ways allowed by the hardware (two for our hardware).

2.4 Cache Auction

Ginseng allocates cache efficiently because it uses a game-theoretic mechanism to elicit the guests’ true benefit from cache. The host conducts rounds of cache auctions to adapt the allocation according to the guests’ changing needs.

In Ginseng, each guest has a different, changing, private (secret) RL of cache, which is expressed in dollars per second for each cache way allocation. Each way will be allocated exclusively. The guest derives its RL by combining two private functions: performance as a function of cache ways (in performance units per second) and valuation of performance (in dollars per performance unit). By taking into account resource allocation and monetary worth, Ginseng is able to compare valuations, while the actual performance requirements are defined and controlled by the client [GSZI10].

As in Ginseng for memory allocation [ABYPBY+14], we define the aggregate benefit from a cache allocation to all guests—their satisfaction from the auction results—using the game-theoretic measure of social welfare. The social welfare of an allocation is defined as the sum of all the guests’ RLs of the cache they receive in the allocation.

Ginseng for cache allocation uses the VCG auction, which maximizes social welfare by encouraging even selfish participants with conflicting economic interests to inform the auctioneer of their true valuation of the auctioned goods. In VCG auctions, this is done by charging each participant for the damage it inflicts on other participants’ social welfare, rather than directly for the goods it wins. VCG auctions are used, for example, in Facebook’s repeated auctions [LLT12], as well as in other settings.

The guest’s RL for each allocation of cache can be affected by its expected performance given its current state. However, it can also be affected by variables unrelated to performance. For example, if the guest is a service provider without any traffic, it may value any number of cache ways as contributing zero to its utility. We denote the guest’s valuation by

$$V_{\text{perf}}(\text{perf}(\text{cache, state})),$$

$$V(\text{cache, state}) = V_{\text{perf}}(\text{perf}(\text{cache, state})),$$
where $V_{\text{perf}}(\text{perf})$ describes the value derived by the client for a given level of performance for a given guest, and $\text{perf}(\text{cache, state})$ describes the performance the guest can achieve given its current state and a certain number of cache ways. $V_{\text{perf}}(\text{perf})$ is private for each client; it is based on economic considerations and business logic.

For example, two clients run a market forecasting algorithm and need to evaluate 1,000 stocks on average to find a group of stocks that are expected to yield 10% profit. They can measure their performance in evaluated stocks per hour. The first client is willing to invest $10K. For this client, $V_{\text{perf}}(\text{perf}) = \frac{11}{\text{stock}} \cdot \text{perf}$. The second client, however, is only willing to invest $1K. For this client, $V_{\text{perf}}(\text{perf}) = \frac{11}{\text{stock}} \cdot \text{perf}$. Both clients will need to know $\text{perf}(\text{cache, state})$: how many stocks they can evaluate per hour when given various numbers of cache ways and under the current conditions (e.g., server load).

In our experiments, we use an offline mapping of performance as a function of cache and the current server load. We found this to be sufficiently accurate, as we demonstrate in §2.8.2. But performance can also be measured online, as demonstrated in a number of works [STS08, ZDS09, Alb99, LCG+15, GLJ+15, YBDJ15], and as might be required in real-world scenarios.

### 2.5 Auction Protocol

In *Ginseng*, each client pays a constant hourly fee for its guest VM while it is assigned to the *pit*. In each auction round, each guest can bid for exclusive cache ways. After each round, *Ginseng* calculates a new cache allocation, and guests exclusively rent the cache ways they won until the next round ends.

The constant fee is not affected by the auction results. It guarantees the lion’s share of the host’s revenues, so that the host can utilize the auction to maximize social welfare, thereby attracting more guests.

Clients with hard performance requirements can verify the availability of exclusive cache ways by prepaying for them (and thereby removing them from the cache ways that are up for rent). Supporting these clients will be easier in future hardware with more COSes. These clients are not included in our experiments, which were performed on Haswell. Clients with very low performance requirements are expected to pay in advance only the constant fee and bid with low valuations or not at all, so that they rarely pay for cache ways and manage to stay within their budget. Clients in between those extremes are expected to choose a flexible payment scheme that meets their needs.

Here follows the description of an auction round, along with a numeric example. In the example, as well as in the experiments that follow, the host’s clients are service providers with their own customers.

**Initialization.** Each guest is assigned to the *pit* as it enters the system.

**Auction Announcement.** The host informs each guest of the number of available cache ways, the server load (i.e., the number of active VMs) and the auction’s closing
time, after which bids are not accepted. In our example, the physical machine has 20 cache ways, two of which are dedicated to the *pit*, so the host announces an auction for 18 cache ways.

**Bidding.** Interested guests bid for cache ways. A bid is composed of a price per hour for each number of exclusive cache ways that the guest is willing to rent. In our example, 10 guests choose not to bid in this round, and 2 guests have strict performance requirements: Guest 1 is willing to pay $1 per hour when allocated 10 or more cache ways and $0 per hour for fewer cache ways. Guest 2 is willing to pay $5 per hour for allocation of 14 or more cache ways and $0 per hour for fewer cache ways.

**Bid Collection.** The host asynchronously collects guest bids as soon as the auction is announced. It considers the most recent bid from each guest, dismissing earlier bids. Guests that do not bid lose the auction automatically, and are assigned to the *pit*.

**Allocation and Payments.** The host computes the allocation and payments according to the VCG auction rules, using a specially designed algorithm described in §2.6. For each guest, it computes how much cache it won and at what price. The payment rule guarantees that the guest will not pay a price that exceeds its bid. The guest’s account is charged accordingly (and accurately, by the second). In the example, guest 1 loses, is assigned to the *pit* and pays nothing; guest 2 wins all of the cache ways and pays $1 per hour.

**Informing Guests and Assigning Cache Ways.** The host informs each guest of the auction results that are relevant to it: its cache allocation and payment. Then, the host takes cache ways from those who lost them and gives them to those who won, by updating their COSes as necessary.

### 2.6 Auction Rules

Every auction has an allocation rule—who gets the goods?—and a payment rule—how much do they pay? To determine who gets the goods, the VCG algorithm calculates the optimal allocation of cache ways: the one that maximizes social welfare—client satisfaction—as described in §2.4. To determine the optimal allocation, the VCG auction solves a constrained multi-unit allocation problem, as detailed in §2.6.1. To determine how each client pays, the VCG auction computes the damage it inflicts on other guests, as detailed in §2.6.2. After explaining the auction rules, we discuss their runtime complexity and provide an example showing how they are executed. A correctness proof is provided in Section 3.4.

#### 2.6.1 Allocation Rule

To find the optimal allocation—the one that maximizes the social welfare—*Ginseng* must consider all the allocations for the number of guests, the number of cache ways available, the size of the *pit*, and the maximum number of classes-of-service (COS)
available. Since the number of possible allocations is exponential in the number of guests and cache ways, iterating over them is impractical. Therefore, we introduce a simple algorithm that finds the optimal allocation in polynomial time.

First, the algorithm combines two guests into an effective guest with a joint valuation function. For any number of cache ways that the two guests will get, the joint function stores the optimal division of cache ways between the two guests, and returns the sum of the valuations of these guests for that cache way division. Then, in each step, it continues to combine the guests and the effective guests until a single effective guest remains. Its valuation function returns the maximal aggregated valuation of all the guests, which is the social welfare. The optimal allocation is then reconstructed from stored division data of the joint valuation functions.

### 2.6.2 Payment Rule

The payments follow the VCG exclusion compensation principle, as formulated in [ABYPBY+14]. Let \( a_k \) denote player’s \( k \) cache allocation, and let \( a_k' \) denote the number of cache ways that would have been allocated to guest \( k \) in an auction in which guest \( i \) did not participate and the rest of the guests bid as they bid in the current auction. Then guest \( i \) is charged a price \( p_i \), computed as follows:

\[
p_i = \sum_{k \neq i} V_k(a'_k) - V_k(a_k).
\]

The payment reflects the damage that guest \( i \)’s bid inflicted on other guests.

### 2.6.3 Complexity

Let \( N \) denote the number of bidding guests. Let \( W \) denote the total number of cache ways and let \( C \) denote the total number of COSes.

\( V_{\text{combined}}(w, c) \) is obtained by comparing \( O(w \cdot c) \) allocations and summing the two valuations for each allocation. That is, for \( W \cdot C \) values, the time complexity is \( O(W^2C^2) \). After \( N - 1 \) reductions we will have one combined valuation. So the total time complexity of the allocation algorithm is \( O(W^2 \cdot C^2 \cdot N) \).

To compute the payment for a guest that is allocated any cache ways, the allocation algorithm needs to be computed again without this guest. Since the number of winning guests is bounded by \( C \), in each auction round the allocation procedure is called up to \( \min(C, N) + 1 \) times, and the time complexity of the total allocation and payment calculation is \( O(W^2 \cdot C^2 \cdot N \cdot \min(C, N)) \).

The algorithm runtime was reasonable: less than one second using a single hardware thread, even when tested with thousands of cache ways and guests, and an unlimited number of COSes, in preparation for future architectures.
2.7 Experimental Setup

In this section we describe the experimental setup in which we evaluate Ginseng.

2.7.1 Machine Setup

We used a machine with two Intel(R) Xeon(R) E5-2658 v3 @ 2.20GHz CPUs with a 30MB, 20-way LLC that supports CAT. Each CPU had 12 cores with hyperthreading enabled, for a total of 48 hardware threads. One CPU was dedicated to the host and the other to the guests. As many guests as possible were each pinned to two exclusive hardware threads that resided on the same core. In experiments with more than 12 guests, some were pinned to one hardware thread each. This let us manage cache allocation per hardware thread and not per VM process. The machine had 32GB of RAM per socket. Each VM got 1GB of RAM, pinned to memory from the same node. The host ran Ubuntu Linux with kernel 4.0.9-040009-generic #201507212131, and the guests ran 3.2.0-29-generic-#46-Ubuntu.

Each application ran exclusively on a virtual machine (VM); hence we refer from now on to an application and the guest VM running it interchangeably. However, this is not compulsory; in real scenarios, the VM’s valuation can change in each bid to cater to changing conditions or changing applications, as is customary in the cattle model of cloud computing.

2.7.2 Workloads

The Phoronix Test Suite [LT11] includes over 100 benchmarks for a variety of applications. We chose a sample of 10 applications with varying cache utilization, along with their associated benchmarks: BZIP2 (1.5.0) uses parallel compression on a 256MB file. H.264 (2.0.0) encodes a video to H.264 format on the CPU. HMMer (1.1.0) searches the Pfam database for profile hidden Markov models. Gcrypt (1.0.3) uses the CAMELLIA256-ECB cipher. OpenSSL (1.9.0) uses an open-source SSL implementation with 4096-bit RSA. Five of the applications were taken from the SciMark 2.0 suite [PM00] (1.2.0), which is included in the Phoronix suite but exists also as a stand-alone: Fast Fourier Transform performs a one-dimensional forward transform of complex numbers. Dense LU Matrix Factorization computes the LU factorization of a dense matrix using partial pivoting. Monte-Carlo approximates the value of pi by using random point selection on a circle. Jacobi Successive Over-Relaxation performs Jacobi successive over-relaxation on a grid. Composite-Scimark is comprised of several SciMark 2.0 benchmarks. The following subsection shows the performance measurements of these benchmarks.

We also tested some larger, commonly used applications such as PostgreSQL and Memcached. However, we eventually decided not to use them in the experimental section as both require long warm-up periods and would reduce the number of experiments.
(a) Cache-utilizer application performance increases with more ways.

(b) Cache-neutral application performance is indifferent to cache ways.

Figure 2.2: Performance of various applications, normalized by their performance in the pit. Measured with 11 other guests assigned to the pit. All of the applications were allocated two hardware threads.

we were able to perform. The performance measurements of both these applications are also shown in the following subsection. We used the TPC-B benchmark with 10 clients to test PostgreSQL, which ran on a VM with 4GM of RAM. To test Memcached we used memslap with 64-byte values and 90% reads, and configured Memcached to use 64MB of RAM.

Classifying the Applications

We used the benchmarks to classify the above applications and demonstrate how they perform under different cache allotments and partitioning.

**Cache-utilizer** applications perform better when allocated more cache. The performance of such applications is depicted in Figure 2.2a.

**Cache-neutral** applications cannot utilize the cache to obtain better performance. The performance of such applications is depicted in Figure 2.2b. However, they might experience minor improvement as compared to being assigned to the pit.

**Cache-polluter** applications are cache-neutral applications that pollute the cache in a way that will harm the cache-utilizer’s performance when cache is shared with the
Figure 2.3: Composite-Scimark performance when sharing the cache with cache-polluter vs. non-cache-polluter applications. The performance was normalized to the minimum measurement in all the experiments.

polluter. To demonstrate this, we ran several experiments, in each of which we ran one cache-utilizer and 7 cache-neutral applications simultaneously. We assigned all of the cache to all of the guests in the shared scenario. In the partitioned scenario, we assigned 2 cache ways to all of the cache-neutral applications together and the rest of the available cache was allocated to the cache-utilizer. Figure 2.3 shows that the cache-utilizer’s performance drops when sharing cache with a cache-polluter application. We classified Monte-Carlo as a cache-polluter, whereas the rest of the cache-neutral applications were non-polluters.

It is likely that partitioning the cache can benefit cache-utilizer applications by protecting them from cache polluters without affecting cache-neutral applications. Furthermore, the provider may need to decide how to allocate the cache between several cache utilizers.
Living with Offline Profiling

Offline profiling is error-prone due to the dynamic nature of the cloud. For example, a cache-utilizer may depend on memory bandwidth. That is, if an application can benefit from faster access to the memory via cache, it will likely suffer when memory access time increases due to low memory bandwidth. Memory bandwidth isolation mechanisms have been studied [IZG+07, MSM+11, JESP12, NALS06], but are not yet available in commercial hardware [LCG+15]. Thus, we are compelled to accept the available memory bandwidth as dependent on the number of guests in the cloud. In a real cloud, the client might want to receive information from the host about its available (or expected) memory bandwidth and take it into account when deriving its valuation. In our Ginseng experiments, we consider the number of guests in the system to be the only factor influencing memory bandwidth and report it to each guest. The guest uses this information from the host as a factor in its valuation, employing its offline performance profiling for environments with various numbers of guests (Figure 2.4).

Valuations

The experimental scenario consists of cloud guests who are themselves service providers. Each guest serves one of its customers at a time. Each guest’s customer shares performance metrics but has different performance requirements. Thus, when customers change, this implies a change in the guest’s valuation function. The valuation function is formulated as the profiled performance function, normalized to the range [0..1], and multiplied by a scale factor that represents the amount the guest’s customer is willing to pay for the performance. The scale factor depends on the performance: if it is below the customer’s required performance, then the scale factor will be lower. Formally, we can express this as:

$$\text{valuation}(a) = s(\text{perf}(a)) \cdot \frac{\text{perf}(a) - \min_{\text{perf}}}{\max_{\text{perf}} - \min_{\text{perf}}},$$

where $a$ denotes the cache way allocation and $s$ denotes the scale factor. The pit is free of charge, and therefore $\text{valuation}(0) = 0$.

We characterize three customer types by their scale factors: A low-valuation customer has a constant scale factor $s = 0.05$. Such a client is unconcerned with performance or unwilling to pay to improve it. An medium-valuation customer has a scale factor $s = 1$ when meeting its performance requirements, and $s = 0.05$ otherwise. A high-valuation customer has a scale factor $s = 3$ when meeting its performance requirements, and $s = 0.05$ otherwise. The performance requirements of this type of customer are higher. See, for example, the valuation functions of a customer running Composite-Scimark (Figure 2.5).

In each experiment, each guest serves 10 customers with different valuations, one after the other. We emulated that by giving each guest a pool of valuations with four
Figure 2.4: Example of performance profiling: *Composite-Scimark* under different server loads (i.e., active VMs) and with different numbers of allocated hardware threads.

![Graph](image1.png)

Figure 2.5: *Composite-Scimark* valuation function for different server-loads (i.e., active VMs) and when allocated 2 hardware threads. Note the different scale in the vertical axes.

![Graph](image2.png)

customer-type distributions. The distributions are denoted as triplets of high-valuation, medium-valuation and low-valuation customers. We experimented with the following distributions: (1,1,8), (1,2,7), (0,5,5), and (3,3,4). For each guest we employed a different, randomly shuffled and unique order on those valuation sets. Hence, when
we repeated an experiment but with more guests, a guest that participated in both experiments had the same valuation order in both. This gives us an idea of what we could achieve if we consolidate more guests on the same physical host.

2.7.3 Alternative Cache Allocation Methods

We compared *Ginseng* with the following cache allocation methods:

- **Shared-cache** allocation, where all of the guests share the entire LLC. This was the prevalent method prior to the introduction of CAT.

- **Uniform-static** allocation, where each guest is allocated a fixed and equal number of cache ways, as many as the hardware allows. In our hardware there are 4 COSes, so for 4 clients or fewer the cache was divided equally. For more clients, three clients received six cache ways each, and the rest of the clients were assigned to the *pit*.

- **Performance-maximizing** allocation, where the guests’ allocation maximizes the overall performance of all of the applications. To this end, we employed *Ginseng*’s optimization algorithm to maximize the aggregate performance by using a constant scale factor \( s = 1 \) for all the guests’ valuations. We did not compare to this method when the experiment had more than one type of application, as the aggregated performance of different applications is meaningless. This allocation is in practice a static allocation, as there is no provider-observable difference in the application’s behavior during the experiment.

- **Ideal-static** allocation, where all the future client valuations are known in advance, and the static allocation that maximizes the social welfare is chosen. It serves as an upper bound for all the static allocations.

2.7.4 Time Scales

*Ginseng*’s responsiveness to guest valuation changes improves with more frequent auctions. Hence, an auction round is conducted every 10 seconds. In each round, the host collects guest bids for 3 seconds, and computes the optimal allocation and payments for at most 3 seconds (in practice it takes well under one second). Then the host notifies the guests of their new allocation and payments and applies the new allocation. However, to gather enough performance measurements for our experiments, we changed the guest’s valuation every 5 minutes in the dynamic allocation experiments. In the static allocation experiments, where valuation changes did not affect the guests’ state, 30 seconds were enough.

2.8 Evaluation

Our experiments were designed to answer the following questions: (1) Which cache allocation method results in the highest aggregated benefit of the guests, thus making
them most satisfied? (2) How accurate is off-line profiling of guest performance? (3) What are the limitations of a Ginseng-based cloud?

The data presented in this chapter of the thesis is based on 4,287 experiments, each lasting 10-50 minutes.

2.8.1 Comparing Social Welfare

We evaluated the social welfare achieved by Ginseng vs. each of the four other methods listed in §2.7.3, for all of the workloads and for workload mixtures (neutrals, utilizers, and a mixture of both). We varied the number of guests running the relevant applications. In the mixed workload experiments we cyclically chose the new workload from the set.

The social welfare was calculated from the measured performance of each application using its guest’s valuation function. Ginseng achieves much better social welfare than the other allocation methods for the tested workloads, as seen in Figure 2.6. It improves social welfare for Dense LU Matrix Factorization by up to $42.8 \times$ compared to shared-cache and by up to $26.3 \times$ compared to ideal-static. For Fast Fourier Transform and Composite-Scimark, Ginseng improves social welfare by $1.7 \times$ to $17.1 \times$ compared to shared-cache and ideal-static. For a heterogeneous cloud with cache-utilizers, Ginseng improves social welfare by up to $13.7 \times$ compared to other allocation methods.

As seen in Figures 2.7a, 2.7b, Ginseng increases the social welfare for an increasing number of up to 12 guests, because more high-valuation and medium-valuation customers can be served simultaneously. However, other methods, including performance-maximizing, improve the social welfare very little or not at all with more guests because they disregard client valuation changes.

For more than 12 guests, hardware threads become a bottleneck, and some guests only get one hardware thread; hence the social welfare gradually declines (Figure 2.7b). However, under Ginseng, some applications can compensate for fewer hardware threads with additional ways, so that Ginseng can maintain high social welfare while increasing server consolidation (Figure 2.7a).

Nevertheless, other allocation methods can still produce results closer to Ginseng for some specific scenarios. For example, when all guests run cache-neutral applications (Figure 2.7c), the applications are less likely to suffer from being consigned to the pit than when some guests run cache-utilizer applications. Although their performance does not depend strongly on cache allocation, their performance in the pit deteriorates when more guests are assigned to it. Thus, as the number of guests in the cloud increases, it becomes increasingly important to allocate cache ways to the right guests, as opposed to assigning them to the pit.

Shared-cache can produce better results than Ginseng when all guests use applications with a small memory working-set (Figure 2.7d). In such a case, cache misses are rare (e.g., a solid 80% hit ratio for 12 H.264 with shared-cache). Thus, because
none of the applications access the memory frequently, an application is expected to consume the maximum memory bandwidth when it does. Hence, memory bandwidth will not be a bottleneck in this case. However, when memory bandwidth is low due to frequent memory access by other applications, even a rare memory access can dramatically affect performance. This is illustrated in Figure 2.9a, where two applications are H.264 and 10 applications are Monte-Carlo. H.264 uses a small memory working-set but relies on prefetching to improve performance. When the cache is not shared, all the prefetched data remains in the cache, resulting in better performance. When the two H.264 applications share the cache, and the 10 Monte-Carlo applications are assigned to the pit, some of the H.264 data might be evicted from the cache. Because

Figure 2.6: Maximum improvement factor of Ginseng compared to the shared-cache and ideal-static methods with different assumptions on the number of high, medium, and low valuation customers. The maximum is over any number of guests with the application, or mixture of applications.

(a) Ginseng improvement factor over shared-cache allocation method.

(b) Ginseng improvement factor over ideal-static allocation method.
Ginseng improved SW by $4.6 \times$ over Perf. Maximizing
Ginseng
Uniform
Ginseng-Simulation
Perf. Maximizing
Shared Cache
Ideal Static

Figure 2.7: Social welfare under different cache allocation methods as a function of the number of VMs. The dashed lines indicate an experiment where the clients perform identically to the profiler (artificial clients). Cache-utilizer applications can greatly benefit from Ginseng. Cache-neutral applications can still enjoy the benefits of Ginseng, albeit to a lesser extent. Applications with a small memory working set will prefer sharing the cache with others like it.

Although our primary concern is improving the social welfare, it is interesting to monitor the commonly considered metric of aggregated performance. This metric is only applicable in the scenario where all the guests run the same application. In these experiments we conducted, the aggregated performance improves slightly with Ginseng in most cases. In some cases, the shared-cache or performance-maximizing methods improve the aggregated performance by up to 10% in comparison to Ginseng. The only exception is H.264, which in some cases yielded a 200% improvement in the aggregated performance with shared-cache, due to, as we mentioned above, its small
memory working-set. This does not diminish the above because the applications are not considered equal, in contrast to what standard performance improvement methods assume.

### 2.8.2 Influence of Off-Line Profiling

We experimented with off-line performance profiling data (e.g., Figure 2.4) that was measured in a controlled environment. However, in a live environment, profiling data should be collected on-line, so that it remains fresh under changing conditions. To retrospectively justify the use of off-line profiling in our experiments, we measured the deviation of actual performance from performance predicted by the off-line profiling (for the conditions at the time).

In Figure 2.8, we see that the deviation from the expected performance was under 10% in most cases. Moreover, the median deviation for all the applications was under 1%, and 95% of the measurements deviate from the predicted performance by less than 12%. The accuracy of the profiling is reflected in the small difference between *Ginseng* and the simulation (Figure 2.7), and shows that a more accurate profiler would achieve only a minor improvement.

![Performance Deviation Distribution](image)

**Figure 2.8:** Expected performance deviation for all applications in all of our experiments. The *pit* measurements are excluded as the performance is expected to fluctuate when sharing a small number of cache ways.

### 2.8.3 To Share or Not To Share

We have already seen cases where partitioning the cache can benefit cache-utilizer applications without affecting cache-neutral ones. However, in some cases, a partitioning that includes limited sharing could greatly improve overall performance.

We consider two possible simple partitioning schemes where we reserve two cache ways for the *pit*. **Hard-partitioning** allocates a set of exclusive cache ways to each guest. A guest that values cache more than others will be allocated more cache ways. **Soft-partitioning** allocates all the cache ways to the guest that values cache the most.
The guest that values cache second-most gets a subset of the previous guest’s ways, and so forth. For simplicity, we only let guests bid for the right to use fixed COSes (for example: $COS_1 = [1..2]$, $COS_2 = [3..20]$, $COS_3 = [3..15]$, $COS_4 = [3..10]$). Guests will need to consider how they value these COSes, assuming other COSes may be occupied by at most one application per COS.

As we have seen, guests can successfully estimate their expected performance for a given allocation of exclusive cache (i.e., hard-partitioning). However, it is harder to valuate a given soft-partitioning allocation when the cache is shared with an unknown guest, as is common in the cloud. Even if the neighbor guest is known, the performance and valuation still depend on additional dimensions (quantity and share level) that further complicate the bidding and optimization process for guest and host alike.

*Ginseng* uses hard-partitioning due to its simplicity and accuracy of estimation. In this section we assess the benefit guests might have achieved from soft-partitioning. To simplify, we tested several pairs of cache-utilizer applications, and the pit contained 10 *Monte-Carlo* applications that served as cache-polluters. We measured the performance of each pair for all possible cache allocations in the hard-partitioning and soft-partitioning allocation schemes. Then, we compared each pair’s performance in these settings. We used each application’s measured performance, normalized to its performance when assigned to the pit, as its RL function, and experimented with different ratios of scale factor between each pair’s RLs.

Although soft-partitioning sometimes yields better social welfare than hard partitioning (Figure 2.9b), it usually improves it by no more than 10% (Figures 2.10 and 2.9a), or even degrades it.

### 2.8.4 Dynamic Allocation Overhead

Transferred cache ways require a warm-up period. Moreover, they are likely to contain the previous application’s data. If the previous application is a cache-utilizer, it is likely to access this data soon, and have this data marked as most-recently-used (MRU). This creates competition for the other application. If it accesses its own data too slowly, it may end up evicting that data from its previously owned ways to store new data in the cache. It will thus take longer (possibly forever) for the second application to benefit from additional cache ways. We refer to such a scenario as *cache leakage*.

Furthermore, any allocation change is constrained by the need to preserve the consecutiveness of ways in a COS. For example, let the initial allocation be $COS_1 = [1..4]$, $COS_2 = [5..6]$ and $COS_3 = [7..10]$. To transfer a way from $COS_3$ to $COS_1$, $COS_2$ must also change. The least disruptive transfer moves two ways: way 7 to $COS_2$ and way 5 to $COS_1$. Compared with the required transfer of a single way, this consecutiveness-constrained transfer doubles the *cache leakage* effect.

We measured how dynamic allocation changes affect application performance. In each experiment, a guest machine ran one of the workloads listed in §2.7.2.
(a) Both applications are $H.264$. Hard-partitioning yields better social welfare than soft-partitioning for all ratios.

(b) Both applications are *Fast Fourier Transform*. The maximal improvement for soft-partitioning over hard-partitioning is achieved when the applications' scale factors are equal.

Figure 2.9: Social welfare under hard vs. soft-partitioning. Striped columns indicate better social welfare under soft-partitioning. Black indicates the opposite.

Figure 2.10: Social welfare improvement under soft-partitioning compared with hard-partitioning for various application pairs. Boxes show the middle 50% of the values over different valuation scale factor ratios. Whiskers mark extreme values.
same time, the host natively ran an application that repeatedly touches all its data, in parallel, using 8 hardware threads and by utilizing the CPU’s out-of-order-execution (OOOE) mechanism. We designed this application to ensure that its data fits perfectly in its allocated cache ways, by detecting cache lines that reside on the same cache set [HWH13, YGL+15]. When an application keeps its cache lines marked as MRU, the cache leakage effect is amplified, and thus represents a worst-case scenario.

Each experiment ran for 10 minutes. In each experiment both applications were allocated a basic set of ways. Another set was transferred between the applications every [10..60] seconds. The numbers of basic and transferred ways were in the range [2..10].

In the baseline experiments the cache ways were transferred once, from the application, to ensure that the application’s performance was not affected by the cache leakage. Half the performance measurements in these experiments were high and half were low.

In the experiments with the frequent transfer intervals, there is a similar performance distribution, whose values varied by up to 4% from the baseline values (high values were lower, low values were higher). The mean performance over the duration of the experiment varied from the baseline by up to 1.1% for all of the workloads. Mean performance values did not depend on the transfer frequency: the effect of a single transfer is negligible, and when there are many intermittent leaks, those that benefit an application will compensate for those that harm it.

2.9 Related Work

Market Driven Resource Allocation. Lazar and Semret auctioned bandwidth [LS99]. Agmon Ben-Yehuda et al. introduced Ginseng as a memory auctioning platform [ABYPBY+14]. Drexler and Miller [DM88] and Maillé and Tuffin [MT04] suggested an auction to compute a market clearing price for memory and bandwidth, respectively. Waldspurger et al. auctioned processor time slices [WHH+92].

Cache Partitioning [SSSS14]. Many hardware solutions detect cache pollution by non-reused data and prevent its future insertion, or apply partitioning to prevent the application from interfering with other applications [DS06, GAV14, JDR01, JCMH99, LR02, PRS07, QJP+07]; others rely on the user or OS to allocate the cache, like CAT does [CJRD00, KSJ09, SK11, LCC11]. However, CAT is the first hardware implementation of such a mechanism in commodity hardware.

A cache-polluter can prevent caching of specific data by using Intel’s non-temporal store instruction. Cache can be partitioned in software using page-coloring [TDF90] to prevent cache pollution: by the program [BAM+96, STS08], by the OS [LLD+09, ZDS09, GLJ+15, YWCL14], and under virtualized environments [WWL+12, JCW+09, RNSE09]. Some works proposed to guarantee the applications’ performance demands via LLC management [GSZI10, HAJ+12, SSM+11, RLT06, GCGV06, NKG10, LCG+15];
these works require the guests to reveal their performance requirements without any incentive to do so.

Although page-coloring allows a finer granularity in the cache allocation, it will not be as effective as CAT for this work as it requires that memory be moved in order to change the cache allocation, which will place a heavy burden on both the clients and the provider.

This work can be leveraged by vCAT [XTP+17], a new method for allocating cache ways using CAT per virtual machine without having to pin a VM to a specific CPU core.

**Shared Cache Performance Interference.** VM *Performance interference* when sharing LLC [KKB+07, GLKS11, KS14] was analyzed and predicted. Such methods can help guests estimate their performance on shared cache and allow a biddable soft-partitioning scheme.

### 2.10 Conclusions and Future Work

*Ginseng* efficiently allocates cache to selfish black-box guests while maximizing their aggregate benefit. *Ginseng* can also benefit private clouds, where it distinguishes between guests that perform the same function for different purposes, such as a test server vs. a production server. *Cache-Ginseng* is the first economically-based cache allocation method, and cache is the second resource implemented in the *Ginseng* framework. *Cache-Ginseng* works by hard-partitioning the cache in short intervals according to a VCG auction in which the guests have an incentive to bid their true valuation of the cache.

The guests utilize their cache fast enough to allow such rapid changes in the allocation without any substantial effect on their performance. *Ginseng* achieves up to $42.8\times$ improvement in social welfare when compared with alternative cache allocation methods. Shared cache allocation may improve on these results. Formulating a bidding and valuation language for shared cache remains as future work.

Although the VCG auction has a high computational complexity, the coarse cache allocation granularity makes it suitable for cache auction. Similarly, it can be efficiently used to allocate other small numbered multi-unit resources whose valuation functions are monotonically rising: CPUs, for example. Thus, *Ginseng* is not only a platform for auctioning cache and memory, but also a concrete step toward the Resource-as-a-Service (RaaS) cloud [ABYBST12, ABYBST14], in which all resources, not just cache and memory, will be bought and sold on-the-fly. Extending *Ginseng* to additional resources and to their concurrent allocation remains as future work.

For *Ginseng* to be applicable in real public or private clouds, further work is required to create tools for clients to evaluate their expected performance with different resource allocations, where the parameters of the cloud are dynamic (e.g., online profiling), and to assist the clients in valuating their performance in economic terms. Furthermore, to
maximize the social welfare over an entire cloud to prevent overcrowded machines, VM migration support should be implemented in *Ginseng* in a way that takes into account the economic benefit and cost to the client and the provider.
Chapter 3

Efficient Multi-Resource, Multi-Unit VCG Auction

Abstract

We consider the optimization problem of a multi-resource, multi-unit VCG auction that produces an exact, i.e., non-approximated, social welfare. We present an algorithm that solves this optimization problem with pseudo-polynomial complexity and demonstrate its efficiency via our implementation. Our implementation is efficient enough to be deployed in real systems to allocate computing resources in fine time-granularity. Our algorithm has a pseudo-near-linear time complexity on average (over all possible realistic inputs) with respect to the number of clients and the number of possible unit allocations. In the worst case, it is quadratic with respect to the number of possible allocations. Our experiments validate our analysis and show near-linear complexity. This is in contrast to the unbounded, nonpolynomial complexity of known solutions, which do not scale well for a large number of agents.

For a single resource and concave valuations, our algorithm reproduces the results of a well-known algorithm. It does so, however, without subjecting the valuations to any restrictions and supports a multiple resource auction, which improves the social welfare over a combination of single-resource auctions by a factor of 2.5-50. This makes our algorithm applicable to real clients in a real system.

3.1 Introduction

Infrastructure-as-a-Service (IaaS) providers have been using auctions to control congestion via preemptible virtual-machine (VM) instances for nearly a decade [ABYPBY+14, Ama18b, Ali18, Pac18]. A natural extension of this idea is to auction additional individual resources in an existing VM. VCG\textsuperscript{1} [Cla71, Gro73, Vic61] are appealing for this purpose, as they are truthful: they incentivize clients to reveal their true valuation of

\textsuperscript{1}The auction is named after William Vickrey, Edward H. Clarke, and Theodore Groves.
the resources, which helps cloud providers accurately price their services. Moreover, VCG maximizes the social welfare—the aggregate valuation the clients assign to the chosen resource allocation. For private (corporate) cloud providers, maximizing the social welfare maximizes the aggregate value the in-house clients generate for the corporation [ABYSS+12]. Cloud clients compete for multiple resources (e.g., RAM, CPU, bandwidth), and these need to be combined in a single auction. A single resource VCG auction is computationally hard to solve [MT07], and a multi-resource auction is more difficult.

Other solutions, besides auctions, were proposed for mitigating congestion. Posted prices [Kov18] and burstable performance [Ama18a, Mic18b, Goo18a, Rac18, Clo18] incentivize clients to reduce their requirements and hence reduce the congestion. Spot instances are based on the uniform price auction [ABYBYST13]. VCG (or generally affine-maximizer) mechanisms, however, are the only known truthful mechanisms that maximize social welfare [Rob79, LMN03].

The optimization problem for a single-resource VCG auction can be reduced to a multiple-choice knapsack problem (MCK), which is NP-hard but can be solved in pseudo-polynomial time via dynamic programming [KPP04]. Many approximated, sub-optimal solutions have been proposed for the MCK problem [Law79, CK05]. However, for VCG to be truthful, an exact, optimal social welfare must be found [NR07]. To obtain a more efficiently, exact solution for a single resource VCG auction, researchers relax the problem by requiring all the functions that describe client valuations of a resource allocation (henceforth valuation functions) to be monotonically increasing and concave [LS99, MT04] or usually concave [ABYPBY+14]. Others solve the problem for a single resource when only one function is not concave but is monotonically increasing [BBB+08]. Concave valuation functions are an unrealistic requirement for cloud clients as their valuation functions have multiple inflection points [FAS19, YBDJ15, CS14, Wil09, ZWS06, LS07].

In a real computational system, we may need to allocate multiple resources [DFH+12]. To auction multiple resources, we must consider the relationship between them. Usually, computing resources are complementary goods: a client who is willing to pay one dollar for an additional single unit of CPU time and RAM is unwilling to pay anything for each resource individually. Alternatively, the resources might be substitute goods: a client who is willing to pay one dollar for an additional single unit of each resource is unwilling to pay two dollars for both resources together. Thus, in both cases, the client cannot bid in an individual auction for each resource. If this client partitions its budget between two resources, it may win only one or both. A client pays for a worthless bundle if it wins only one of two complementary resources, or if it wins both substitute resources. Such a scenario will also decrease the utilization. Only a multiple resource auction that considers the clients’ value for each combination of resources can both optimize the social welfare and be truthful.

Unfortunately, single resource solutions do not apply for multiple resources. The
multiple resource VCG auction can be reduced to a multiple-choice, multidimensional
knapsack problem (MCMK or d-MCK), which to the best of our knowledge has no
pseudo-polynomial solutions. Similarly to MCK, MCMK also has many approxi-
mated solutions [GLYL17, ARK+06, MJS97, KLMA02, HMS04]. Such solutions pro-
vide near-optimal results: the best of them yields results within 6% of the optimal
value, which does not guarantee the auction will be truthful and maximize the social
welfare. Exact solutions for MCMK have been proposed via branch-and-bound algo-
rithms (B&B) [GTHH18, HSS04, Sbi07, RG08, GL00]; however, their results indicate
an implicit nonpolynomial increase in runtime with respect to the number of possible
allocations. These solutions were only tested empirically with small datasets and did
not scale well for many clients and large, complete valuation functions.

Moreover, MCMK solutions were not designed for a VCG auction and thus do
not allow efficient calculation of payments according to the VCG payment rule. To
compute a winning client’s payment in a VCG auction, the auctioneer must find the
social welfare that could be achieved when that winning client is excluded from the
auction. Solutions not tailored to VCG must compute the payments by repeatedly
finding the optimal allocation for each winning client if that client had not partici-
pated in the auction. This implies a worst-case quadratic complexity with respect to the
number of clients.

In this work, we implement an efficient, exact, multi-unit, multidimensional resource
VCG auction. Two approaches can be considered for this problem. The resources may
be treated as infinitely divisible (continuous), as Lazar and Semret [LS99], Maillé and
Tuffin [MT04], and Agmon Ben-Yehuda et al. [ABYPBY+14] do for a single resource.
The other approach, which we adopt, divides each resource into identical units of a pre-
defined size (e.g., a single CPU second can be time-shared as 1000 millisecond units).
The smaller the units are, the closer the auction’s result is to the continuous solu-
tion, and the higher the complexity of finding the allocation that maximizes the social
welfare.

In the multi-unit, multi-resource auction, agents, representing the clients, can bid
using a multidimensional valuation function, which attaches a monetary value to each
number of units of each resource. To find the exact solution, the auctioneer must con-
sider all the allocations for the number of agents and the number of resource units
available. Since the number of possible divisions of resources between agents is expon-
ential in the number of agents and resource units, iterating over them is impractical.

We present a method for solving a multi-unit, multi-resource auction without any
restrictions on the valuation functions, in pseudo-near-linear time on average, over all
possible realistic valuation functions, with respect to the number of clients \( n \) and the
number of possible unit allocations for each client \( N \). Our algorithm’s worst-case time
complexity is \( O(n \cdot N^2) \), as opposed to the worst-case nonpolynomial complexity of the
known MCMK algorithms. Furthermore, our algorithm computes the VCG auction
payments without repeating the full auction for each winning client. The payment
calculation complexity is a function of $N$ and the number of winning clients. It does not depend on the number of clients in the auction ($n$). Our solution is also applicable to a single resource auction and has a better average complexity than the dynamic programming solution, which is $O(n \cdot N^2)$ [KPP04]. All of the above makes it feasible to choose a VCG auction as a resource allocation mechanism in a real system.

**Our contributions** are an optimization algorithm for the multi-unit, multi-resource allocation problem and an implementation of this algorithm with a choice of data structures to support it. We prove the correctness of the algorithm in Section 3.4 and numerically analyze its complexity in Section 3.5. We evaluate the performance of our implementation in Section 3.7 using each data structure and verify the correctness of the results. We validate our results for a single resource with concave valuation functions, by comparing to Maillé and Tuffin’s results, and show that separate single-resource auctions produce sub-optimal results, in contrast to multi-resource auctions, which produce optimal results. The implementation can be extended using other data structures. We analyze the algorithm’s best possible performance independently of the choice of a data structure.

### 3.2 The Non-Linear Optimization Problem

In this chapter of the thesis, vectorized arithmetic operators are defined element-wise. For example, $\vec{a} + \vec{b} = (a_0 + b_0, ..., a_n + b_n)$, and $\vec{a} \leq \vec{b} \iff \forall i \in 1..R : a_i \leq b_i$. The symbols used in this chapter of the thesis are listed in Table 3.1.

In an ideal VCG auction, the auctioneer computes the exact allocation that maximizes the social welfare. Each winning client pays the auctioneer according to the damage it caused the rest of the clients—i.e., the exclusion compensation principle. This payment rule makes the auction truthful: the best client strategy is to bid with its true valuation of the resources. Thus, VCG optimizes the social welfare according to true data about client valuations.

The VCG optimization problem can be described as a non-linear optimization problem (NLP) that is separable, non-convex, and linearly and discretely constrained, as follows:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>number of agents</td>
</tr>
<tr>
<td>$R$</td>
<td>number of resources</td>
</tr>
<tr>
<td>$\vec{m}$</td>
<td>number of units for each resource: $(m_1, ..., m_R)$</td>
</tr>
<tr>
<td>$\vec{a}_i$</td>
<td>allocation of agent $i$ for each resource: $(a_{i,1}, ..., a_{i,R})$</td>
</tr>
<tr>
<td>$A$</td>
<td>set of allocations ${\vec{a}<em>i}</em>{i=1}^n$</td>
</tr>
<tr>
<td>$V_i$</td>
<td>valuation function of agent $i \in 1..n$</td>
</tr>
<tr>
<td>$N$</td>
<td>the number of possible allocations on which a valuation function is defined $N = \prod_{r=1}^R (m_r + 1)$.</td>
</tr>
</tbody>
</table>

34
Separable: The sum of \( n \) separable valuation functions is maximized.

\[
\text{Maximize: } \sum_{i=1}^{n} V_i(\vec{a}_i). \quad (3.1)
\]

Such valuation functions can be represented as a multidimensional vector.

Non-Convex: None of the separable functions (\( V_i \)) are required to be convex, concave, or even monotonic.

Linearly Constrained:

\[
\sum_{i=1}^{n} \vec{a}_i \leq \vec{m}. \quad (3.2)
\]

Discretely Constrained: The resource is not continuous and is divided into units. Each \( a_{i,r} \) is a natural number (or zero) that represents the number of allocated units. Only a whole unit can be allocated. Hence, the \( V_i \) functions should be defined only on an even-spaced grid of the natural numbers.

3.3 Joint Valuation Algorithm

In the previous chapter (Chapter 2), we developed the joint valuation algorithm for finding the optimal allocation of resources in a single dimension, for monotonically increasing functions with \( O(n \cdot N^2) \) time complexity. In this work, we extend this algorithm to multidimensional non-monotonic valuation functions, such that it fulfills all the constraints delineated in Section 3.2. While the complexity of a naïve extension is proportional to the square of the number of possible unit-allocation combinations (Section 3.3.3), our extension has a pseudo-near-linear time complexity on average over all possible realistic valuation functions (numerically analyzed in Section 3.5). We prove that the algorithm produces the correct optimal allocation and the correct payments in Section 3.4.

3.3.1 Finding the Optimal Allocation

To find the optimal allocation, two agents are first combined into one effective agent with a joint valuation function (Section 3.3.3). For any number and combination of goods that the two agents will obtain together, the joint function stores the optimal division of goods between them, and the sum of the valuations of these agents for this optimal division. Then another agent is joined to the effective agent, and then another, etc. This process produces a new joint valuation function at each stage, until the final effective agent’s valuation function is the maximal aggregated valuation of all the agents. Its maximal value is the maximal social welfare. The optimal allocation is then reconstructed from the stored division data of the joint valuation functions.
3.3.2 Payment Computation

Our algorithm is efficient in the number of times that the optimal allocation must be computed. To compute a winning agent’s payment according to the exclusion compensation principle, the auctioneer must determine the social welfare that could be achieved when that winning agent is excluded from the auction. This can be naively computed by repeatedly finding the optimal allocation for each winning agent, without its participation in the auction. Our algorithm, however, reduces the number of repetitions by using a preliminary step. It re-computes the joint valuation function by joining the agents in reverse order to that taken when first finding the optimal allocation. For each winning agent \( j \), the joint valuation function of the rest of the agents is computed by joining the intermediate effective valuation function right before adding agent \( j \), which includes agents \( 1, \ldots, j - 1 \), and the one right before adding \( j \) in the reverse order, which includes agents \( j + 1, \ldots, n \). The maximal value of this function is the maximal social welfare achievable without this agent, as required for the calculation of that agent’s payment according to the exclusion compensation principle.

3.3.3 Joining Two Valuation Functions

To naively join two valuation functions, we need to find, for each possible allocation, how to best divide the resources between the two clients. For each possible allocation of the joint agents \( \vec{a}_j \), there are \( \prod_{r=1}^{R} (a_{j,r} + 1) \) possible divisions of the resource. To compute the full joint valuation function of two clients, each with \( N \) possible allocations, the number of possible resource divisions to compare is

\[
\sum_{\vec{a}_j \text{ s.t. } \vec{a}_j \leq \vec{m}} \left( \prod_{r=1}^{R} (a_{j,r} + 1) \right) = \prod_{r=1}^{R} \frac{m_r(m_r + 1)}{2} = O(N^2), \tag{3.3}
\]

for four resources, each with 15 units, \( N^2 = 2^{16} \). This number of comparisons will take a few seconds to compute on a standard CPU for each joining of two valuation functions. For many clients, however, this can add up to a full hour.

The complexity of finding the optimal allocation and the payments depends on the complexity of joining two valuation functions. Let \( J(N) \) denote the complexity of joining two valuation functions with \( N \) possible allocations. Then the algorithm’s time complexity is \( O(n \cdot J(N)) \).

We can reduce the complexity of \( J(N) \) by reducing the number of compared allocations. To do so, we filter out allocations that cannot maximize the social welfare. If an allocation globally maximizes the social welfare, then (1) it is Pareto efficient: one agent’s allocation cannot be improved without hindering another’s, and (2) it is also a local optimum: the aggregated valuation cannot be increased by taking a resource from one agent and giving it to another.
Formally, the Pareto efficiency property means that if the allocation is optimal, any left partial derivative of any single agent’s valuation function is positive: \( \partial_r V_i(\vec{a}_i) > 0 \). The local optimum property means that for an optimal allocation, any right partial derivative of any single agent’s valuation function is no greater than any of the other agents’ left partial derivatives: \( \partial_r V_i(\vec{a}_i) \leq \partial_r V_j(\vec{a}_j) \). Both are true element-wise for each resource \( (r) \) dimension. Since our domain is discrete, partial derivatives are not defined. We will define the left/right partial derivatives as the difference in the values between adjacent points in the allocation space \( (dr = 1 \text{ for all the resources}) \).

Using these properties, we restrict the search during the joining of two valuation functions. We first eliminate client allocations in which the left partial derivative of their valuation function in one of the resource dimensions is non-positive. Second, for each possible allocation of the first valuation function, we only consider allocations of the second function in which the condition on the partial derivative is maintained. To accommodate boundary allocations (allocations that reside on the valuation function’s domain boundary), where the left or right partial derivative is not well defined, we assign the minimal allocation (zero) a left partial derivative of infinity, and assign the maximal allocation \( (m_r \text{ for each resource } r) \) a right partial derivative of zero. We do this because we cannot assign an agent with less than zero or more than the maximal quantity.

These two restrictions will eliminate most of the allocation comparisons to an average-case complexity of \( O(N) \) comparisons instead of \( O(N^2) \). This complexity holds under the assumption of a uniform distribution over the valuation function’s inflection points locations. We analyze the average-case complexity under this assumption in Section 3.5, and validate this analysis empirically in Section 3.7. Algorithm 3.1 describes the joining of two valuation functions.

Algorithm 3.1 Joining two valuation functions.

**Data:** \( V_i, V_j \): valuation functions

**Result:** \( V_r \): joint valuation function, \( A_r \): the allocation that produces \( V_r \)

1. Initialize \( V_r \) and \( A_r \) to zeros
2. Calculate \( V_i \)’s and \( V_j \)’s gradients and store them into an array of vectors
3. Remove allocations such that \( \partial_r V_i(\vec{a}_i) \leq 0 \) (for each \( r \))
4. Remove allocations such that \( \partial_r V_j(\vec{a}_j) \leq 0 \) (for each \( r \))
5. foreach \( \vec{a}_i \) do
6. \hspace{1em} foreach \( \vec{a}_j \) such that for each \( r \): \( \partial_r V_i(\vec{a}_i) \leq \partial_r V_j(\vec{a}_j) \) and \( \partial_r V_i(\vec{a}_i) \) and \( \vec{a}_i + \vec{a}_j \leq \vec{m} \) do
7. \hspace{2em} \( v_r \leftarrow V_i(\vec{a}_i) + V_j(\vec{a}_j) \) \( \vec{a}_r \leftarrow \vec{a}_i + \vec{a}_j \)
8. \hspace{2em} if \( V_r(\vec{a}_r) < v_r \) then
9. \hspace{3em} \( V_r(\vec{a}_r) \leftarrow v_r \) \( A_r(\vec{a}_r) \leftarrow \vec{a}_i, \vec{a}_j \)
10. \hspace{2em} end
11. \hspace{1em} end
12. end
3.3.4 Upper-Bound Limit

Eliminating allocations that cannot be Pareto efficient (Lines 3 and 4 in Algorithm 3.1) requires verifying a simple lower limit condition on the left partial derivative in the initialization of the algorithm. The local optimum property (Line 6 in Algorithm 3.1), however, requires repeated elimination for each loop iteration (Line 5 in Algorithm 3.1) with different multi-dimensional conditions each time.

When joining two valuation functions of agents \( i \) and \( j \), for each possible allocation \( \vec{a}_i \) of agent \( i \), we seek all the allocations \( \vec{a}_j \) of agent \( j \) for which the local optimum property is maintained. Formally, we seek all \( \vec{a}_j \) such that:

\[
\nabla^+ V_j(\vec{a}_j) \leq \nabla^- V_i(\vec{a}_i) \tag{3.4}
\]

\[
-\nabla^- V_j(\vec{a}_j) \leq -\nabla^+ V_i(\vec{a}_i) \tag{3.5}
\]

\[
\vec{a}_j \leq \vec{m} - \vec{a}_i \tag{3.6}
\]

where we define

\[
\nabla^+ V_i(\vec{a}_i) = (\partial_{1^+} V_i(\vec{a}_i), ..., \partial_{R^+} V_i(\vec{a}_i)) \tag{3.7}
\]

\[
\nabla^- V_i(\vec{a}_i) = (\partial_{1^-} V_i(\vec{a}_i), ..., \partial_{R^-} V_i(\vec{a}_i)) \tag{3.8}
\]

as the right and left gradients, respectively.

Each of these inequalities defines \( R \) upper-bound requirements on agent \( j \)'s allocation, for a total of \( 3R \) requirements. For each of agent \( i \)'s possible allocations, we need to efficiently find agent \( j \)'s allocations that match these requirements. To do so, we preprocess agent \( j \)'s valuation function using a dedicated upper-bound data structure that allows efficient retrieval of allocations that match these requirements. We map each possible allocation of agent \( j \) (\( \vec{a}_j \)) to a new \( 3R \)-dimensional vector:

\[
(\nabla^+ V_j(\vec{a}_j), -\nabla^- V_j(\vec{a}_j), \vec{a}_j). \tag{3.9}
\]

We store these vectors in a \textit{k-dimensional upper-bound data structure}, where \( k = 3R \). The data structure will contain a total of \( N \) vectors and thus its complexity will depend on \( k \) and \( N \). Then, for each possible allocation of agent \( i \) (\( \vec{a}_i \)), we query all the vectors (defined in Equation 3.9) that are smaller than or equal to the following vector:

\[
(\nabla^- V_i(\vec{a}_i), -\nabla^+ V_i(\vec{a}_i), \vec{m} - \vec{a}_i). \tag{3.10}
\]

The \( k \)-dimensional upper-bound data structure must support the following methods:

- **construct(all vectors)**: create the data structure.
- **query(vector)**: find all the vectors that are smaller than or equal to a certain vector (element-wise).
fetch(): return all the vectors that match the last query.

We consider the $k$-dimensional ($k$-d) binary search trees that are listed in Table 3.2 along with their space and time complexities. The complexity of result fetching is linear with the number of returned vectors and not with the number of matching vectors, because some data structures trade accuracy for efficiency, returning false positives.

Table 3.2: Upper-bound data structure comparison.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Complexity ($O(...)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$-d Tree [Lue78]</td>
<td>$N \log^{k-1} N$</td>
</tr>
<tr>
<td>Simultaneous 1-d Bin. Searches</td>
<td>$kN \log N$</td>
</tr>
<tr>
<td>Simultaneous 2-d Trees</td>
<td>$kN \log N$</td>
</tr>
</tbody>
</table>

$k$-d Tree

Algorithm 3.2 describes the construction of this tree.

Algorithm 3.2 $k$-d Tree Construction

Data: $v$: an array of $N$ vectors

Result: a $k$-d Tree

1 call RecursiveBuild(1, $N$, $v$)

2 Function RecursiveBuild(d: sort dimension, $M$: array size, $v$: array of $M$ vectors):

3 Sort the array of vectors ($v$) by dimension $d$

4 if $d < k$ then

5     Create $\log M$ copies of the input array

6     Partition each copied array $t = 1..\log M$ into $2^{t-1}$ even parts

7     foreach $v_i$: array partition do

8         call RecursiveBuild($d + 1$, $v_i$, $v_i$ size)

9     end

10 end

11 end

Figure 3.1 shows an example of a four-dimensional binary search tree. Each letter in the example represents a four-dimensional vector. The initial array (d1) is sorted by the first dimension. Each of the following blocks (d2, d3, d4) is built from a sorted array created on the previous block. In this example, we partition the array up to partitions of the size of two, as creating a sub-tree of one vector is not useful.

To query, we do a binary search by the first dimension on the first sorted array. Each time the binary search continues to the upper half of the array—i.e., all the vectors in the lower half are smaller in that dimension than the query—the vectors in the lower half are filtered by the next dimension, by recursively running the query on the sub-tree created from the lower half of the array. Then, the search continues to the upper half. Finally, in the deepest sub-tree, we simply return all the vectors that are lower than the position returned by the binary search. For example, starting from array d1 in
Figure 3.1: Example of a four-dimensional binary search tree. Each letter represents a four-dimensional vector. Each array in a rectangle marked with $dX$ is sorted by dimension $X$.

Figure 3.1, if the query is larger than vector $h$, the binary search will continue to the upper half of the array ($i$ to $p$) and recursively run the query on the array with the bold frame in block $d2$.

This data structure never returns false positives but has prohibitive time and space complexity. For example, four resources require a 12-dimensional data structure with $O(N \log^{11}(N))$ memory and time complexity. Even for a small $N$, e.g., $N = 1024$, this can consume an entire machine. Thus, we did not test the performance of this data structure. Following are more efficient methods that reduce the complexity by reducing the accuracy of the results.

**Simultaneous 1-d Binary Searches**

We store $k$ arrays, each sorted according to another dimension. For each upper-bound query, we perform a simultaneous binary search on all the arrays. That is, instead of searching one array at a time, we perform each step on all the sorted arrays simultaneously. In each step, some array searches continue to the lower half and some to the upper half. We continue searching only with the array searches that continue to the lower half. If all of the searches continue to the upper half, we continue with all of them. When the search finishes, we have found the dimension that filters the most vectors independently of the other dimensions. We will return all the vectors that are lower than the position the search found. This is time and space efficient, but yields many false positives, because we only filter by one dimension.

**Simultaneous 2-d Trees**

A multidimensional binary search tree problem can be relaxed by constructing many two-dimensional binary search trees, each sorted according to a different combination
of two dimensions. Then, all of them can be queried, and the vectors fetched from the tree whose query returned the least vectors.

To make the multiple queries more efficient, we use only a subset of the combinations that we believe will filter the most vectors: all the combinations of two dimensions that originate from the same resource $r$:

$$\left(\partial_{r+} V_i(\vec{a}_i), -\partial_{r-} V_i(\vec{a}_i), a_{i,r}\right).$$

(3.11)

We also construct the trees in a way that reduces the number of repeated queries: we first build $3R$ arrays, each sorted by a different dimension. Then, from each array, we build two trees, each for the other two dimensions that originated from the same resource.

Following this, an upper-bound query is implemented: (1) First, a simultaneous 1-d binary search is performed on each sorted main dimension array. The partitions that had to be searched when the binary search continued to the upper half are stored for later. When the search is finished, the main dimension that found the lowest upper bound is chosen (or one of them is chosen if more than one remained). (2) Next, each stored partition is searched simultaneously in the two sub-trees of the chosen main dimension. For each simultaneous search in the stored partition, we will return the results from the one that yielded the fewest vectors.

The query, as described above, will require one simultaneous binary search on $3R$ arrays, then at most $\log N$ simultaneous searches on two arrays. This results in considerably fewer searches than when searching each combination of two dimensions individually. Consequently, the time and space complexity of simultaneous 2-d trees are not much higher than they are for simultaneous 1-d binary searches, but the former yields considerably fewer false positives.

Combination

Many of the vectors were created from a boundary allocation (having maximal or zero allocation in one of the resources). Boundary allocations have a minimal partial derivative in the direction of the boundary; hence boundary allocations are never filtered by the dimensions that correspond to the partial derivative in that direction. We can classify vectors according to their boundary type (which domain boundaries the vector’s allocation resides on), and filter each class only by the vital dimensions: those with a higher value than the minimal. For vectors with only one vital dimension, we use simultaneous 1-d binary searches, for those with two we use a single 2-d tree and, for those with more, we use simultaneous 2-d trees. This reduces both the construction time and the query time, as the trees are smaller and each is filtered only by vital dimensions.
3.4 Correctness Proof

We prove that our algorithm produces correct results, i.e., an allocation that maximizes the social welfare.

3.4.1 Notations

We use the notations from Table 3.1. Let \( P = \{i\}_{i=1}^{n} \) denote the set of all agents. We define an allocation for any subset of agents \( G \subseteq P \) and for maximal quantities of allocatable goods \( \bar{m} \) as follows:

\[
A_{\bar{m}}^{G} = \{\bar{a}_i\}_{i \in G}.
\]

We denote agent \( i \)'s valuation for an allocation \( A_{\bar{m}}^{G} \) as

\[
V_i(A_{\bar{m}}^{G}) = \begin{cases} 
V_i(A_{\bar{m}}^{G[i]}), & \text{if } i \in G \\
0, & \text{otherwise}
\end{cases}
\]

(3.13)

where \( A_{\bar{m}}^{G[i]} \) is the allocation of agent \( i \).

For any subset \( H \subseteq G \subseteq P \) under the allocation \( A_{\bar{m}}^{G} \), we denote by \( V_H(A_{\bar{m}}^{G}) \) the aggregated valuation of the agents in \( H \) under this allocation, by \( S_H(A_{\bar{m}}^{G}) \) the sum of resources allocated to the agents in \( H \) under this allocation (element wise), and by \( E_H(A_{\bar{m}}^{G}) \) the subset of allocations of the agents in \( H \) under this allocation. Formally,

\[
V_H(A_{\bar{m}}^{G}) = \sum_{i \in H} V_i(A_{\bar{m}}^{G[i]})
\]

(3.14)

\[
S_H(A_{\bar{m}}^{G}) = \sum_{i \in H} \bar{a}_i
\]

(3.15)

\[
E_H(A_{\bar{m}}^{G}) = \{\bar{a}_i\}_{i \in H}
\]

(3.16)

The social welfare of an allocation is defined as the aggregated sum of all the agents’ valuations for that allocation, i.e., \( \text{SW}(A_{\bar{m}}^{G}) = V_P(A_{\bar{m}}^{G}) \).

An allocation \( A_{\bar{m}}^{G} \) is valid if \( S(A_{\bar{m}}^{G}) \leq \bar{m} \). A valid allocation \( A_{\bar{m}}^{G} \) is optimal if it maximizes the aggregated valuation:

\[
\forall B_{\bar{m}}^{G}: V_G(B_{\bar{m}}^{G}) \leq V_G(A_{\bar{m}}^{G}),
\]

(3.17)

where \( B_{\bar{m}}^{G} \) is a valid allocation.

3.4.2 Supporting Lemma

In this subsection we will prove Lemma 1, which supports the use of the additive process of joining the valuations one by one. Following (Section 3.4.3) is a proof by induction that uses Lemma 1 to prove the optimality of the results.
Lemma 1. For any optimal allocation \( \hat{A}_P \) and any subset of agents \( G \subseteq P \), the allocations of the agents in \( G \) are also optimal for the case where the agents in \( G \) are the only agents and the number of allocatable units is exactly the sum of their allocations. That is, \( V_G(\hat{A}_P) = V_G(\hat{A}_{G \in G}) \), where \( \bar{m}_G = S_G(\hat{A}_P) \).

Proof. Assume the claim is false. Then, there exists an optimal allocation \( \hat{A}_P \) and a subset \( G \subseteq P \), such that the allocations of the agents in \( G \) are not optimal for the case where these agents are the only agents and the number of allocatable units is exactly the sum of their allocations. That is, \( V_G(\hat{A}_P) \neq V_G(\hat{A}_{G \in G}) \), where \( \bar{m}_G = S_G(\hat{A}_P) \).

There are two cases:

1. \( V_G(\hat{A}_P) < V_G(\hat{A}_{G \in G}) \).

   Combine \( \hat{A}_P \) and \( \hat{A}_{G \in G} \) to create a new allocation \( \tilde{A} \) such that the agents in \( G \) get the resources they get under \( \hat{A}_{G \in G} \), and the rest of the agents get the resources they get under \( \hat{A}_P \). The new allocation \( \tilde{A} \) is valid because \( \hat{A}_{G \in G} \) is valid, and \( S_G(\hat{A}_P) = \bar{m}_G \geq S_G(\hat{A}_{G \in G}) \), so \( S_P(\tilde{A}) \leq \bar{m} \). According to the assumption, \( V_G(\hat{A}_P) < V_G(\hat{A}_{G \in G}) \),

   \[
   V_G(\hat{A}_P) < V_G(\hat{A}_{G \in G}), \tag{3.18}
   \]

   and thus

   \[
   V_P(\hat{A}_P) = V_G(\hat{A}_P) + V_{P \setminus G}(\hat{A}_P), \tag{3.19}
   \]

   which according to (3.18) is smaller than

   \[
   V_P(\hat{A}_{G \in G}) + V_{P \setminus G}(\hat{A}_P) = V(\tilde{A}), \tag{3.20}
   \]

   in contradiction to the optimality of allocation \( \hat{A}_P \).

2. \( V_G(\hat{A}_P) > V(\hat{A}_{G \in G}) \).

   Since \( S_G(\hat{A}_P) = \bar{m}_G \), then \( E_G(\hat{A}_P) \) is a valid allocation for the subset of agents \( G \) with maximal allocatable resources of \( \bar{m}_G \), and it yields a higher aggregated value than \( \hat{A}_{G \in G} \), in contradiction to the optimality of the allocation \( \hat{A}_{G \in G} \).

3.4.3 Proof by Induction

Our algorithm joins valuations into an accumulated valuation one by one. At each step, for each number of resources \( \bar{m}_i \), the algorithm iterates over all possible combination of resources \( \bar{m}_i, \bar{m}_j \) such that \( \bar{m}_i + \bar{m}_j \leq \bar{m} \). Then, for each \( \bar{m} \), the algorithm chooses the \( \bar{m}_i, \bar{m}_j \) that yielded the maximal aggregated value. Finally, we choose \( \bar{m} \) that yields the maximal value in the final joint valuation function.

We prove by induction that the above algorithm finds an optimal allocation. For generality, we do not assume that the joining of the valuations is done in any particular order. Instead, at each step, any two valuation functions might be joined to form a single effective one.

Theorem 2. For a subset of agents \( G \) and allocatable quantities \( \bar{m} \) of goods, the algorithm finds an optimal allocation \( \hat{A}_{G \in G} \).
Proof. $|G| = 1$. For one agent, no joining of two valuations is needed. The algorithm simply chooses the maximal valuation for any allocation up to $\vec{m}$. This is the maximum social welfare by definition.

Inductive hypothesis. Suppose the theorem holds when $|G| \leq k$, for some $k \geq 1$. Let $|G| = k + 1$.

Consider any two non-empty, disjoint subsets: $X$ and $Y$, where $X \cup Y = G$. By the pigeonhole principle, $|X| \leq k$ and $|Y| \leq k$, and

$$\tilde{S}_G(\hat{A}_G) = \tilde{S}_X(\hat{A}_G) + \tilde{S}_Y(\hat{A}_G) \leq \vec{m}$$

since the optimality of allocation $\hat{A}_G$ implies it is valid.

Let us denote $\vec{m}_X = \tilde{S}_X(\hat{A}_G)$, $\vec{m}_Y = \tilde{S}_Y(\hat{A}_G)$. Then

$$\vec{m}_X + \vec{m}_Y \leq \vec{m}.$$  \hfill (3.22)

According to Lemma 1,

$$V_X(\hat{A}_G) = V_X(\hat{A}_X)$$  \hfill (3.23)

$$V_Y(\hat{A}_G) = V_Y(\hat{A}_Y)$$  \hfill (3.24)

$$\implies V_X(\hat{A}_X) + V_Y(\hat{A}_Y) = V_G(\hat{A}_G).$$

Hence, since we search all the options where $\vec{m}_X + \vec{m}_Y \leq \vec{m}$ and find optimal allocations $\hat{A}_X$, $\hat{A}_Y$ for each of them, we must encounter an allocation with the above aggregated valuation. Because it is the maximal value, our algorithm will prefer this allocation to the alternatives. So, the theorem holds for $|G| = k + 1$.

By induction, the theorem holds for every size of $G$. \hfill \blacksquare

3.5 Complexity Analysis of Joining Two Valuations

We first show the worst-case time complexity of $O(N^2)$, which may be relevant only in unrealistic scenarios. Then, we analyze the worst-case complexity of a single resource over realistic valuation functions, and find it equal $O(N \log N)$. Finally, we show that multiple resources yield the same time complexity, but on average over all possible realistic valuation functions.

3.5.1 Worst Case

The worst case complexity of joining two valuation functions is $O(N^2)$, when for every query, the number of matching allocations is proportionate to $N$. This can happen, for example, when both valuation functions are linear, with an identical slope. Any of the $N$ queries on one of the functions will return every allocation ($O(N)$), as the upper-bound limit is inclusive. This adversarial example, however, is unlikely on a real cloud,
with a mixture of clients and valuation functions, and where precise linear scaling is rare. We will thus consider in the following only strictly convex/concave functions, i.e., without any precise linear parts.

3.5.2 Single Resource

To analyze the complexity we will assume $N \to \infty$, which approximates a smooth continuous function were the left partial derivative is equal to the right. This reduces the local optimum property to a single rule: for an optimal allocation, all the agents’ valuation functions have identical gradients.

For a single resource with concave/convex valuation functions, each derivative value is obtained at most once. Hence, each query will match at most one allocation. For a function with one or more inflection points, each query will match a number of allocations up to the number of inflection points in the function. The number of inflection points is related to the number of hierarchies in the resource. For example, a CPU might have two inflection points: when switching from a single-core to multiple-cores, and then to multiple-chips. Memory might also have two inflection points when switching between cache, RAM and storage. Five inflection points, however, might be considered unrealistically high for computing resource valuation functions. Thus, we consider the number of possible inflection points for each resource to be a constant as it is independent on the parameters ($n$, $N$ and $R$) and is generally small. This yields a maximal complexity of $O(N)$.

The time complexity of joining two valuation functions is at least $O(N \log N)$, the data structure construction complexity. Hence, the complexity of joining two valuation functions is $O(N \log N)$.

3.5.3 Multiple Resources

Similarly to a single resource, for multiple resources with concave/convex valuation functions, each gradient vector is obtained at most once. We can consider each resource to have inflection points independently of the other resources, e.g., it is possible to switch from a single processor to a multi-processor algorithm regardless of the RAM usage. Thus, if each resource has $t$ inflection points, we can divide the valuation function domain into $(t + 1)^R$ sections, each being convex or concave. That is, each gradient vector might be obtained at most once in each of these sections. The actual number of matches is much lower than $(t + 1)^R$, and is constant as shown in Section 3.7.2.

We reconcile these differences by showing that the average case, over all possible realistic valuation functions yields a constant number of matching allocations. To do this, we will assume without loss of generality that the partial derivatives on each of the inflection points and in the function boundaries distribute uniformly from zero to the maximal derivative. The partial derivatives of the required gradient will also distribute uniformly with the same boundaries. Then, for exactly two inflection points
per resource, we will have three sections, each with different uniformly distributed boundaries. The probability of a single derivative that is uniformly distributed to be in these boundaries is $\frac{1}{3}$, and thus, for each resource, exactly one section is expected to have this gradient. Thus, regardless of the number of resources $R$, exactly one section is expected to have the required gradient (out of the total $(t+1)^R$). Since only a single matching allocation exists in that section, the expected number of matching allocations is exactly one.

Furthermore, if we assume that the required gradient has different derivative boundaries, as we would expect in the real world, then a higher number of inflection points will yield a single matching section as well. If the first client’s valuation function has a maximal derivative $d$ times higher than the second, then $\lfloor 3 \cdot d - 1 \rfloor$ number of inflection points per resource will yield at most one matching allocation per query. Since the joint valuation function is expected to have higher derivatives with each joining, we would expect $d$ to grow in each step, and thus reduce the number of matching allocations. This yields an average complexity of $O(N)$ over realistic valuation functions.

Hence, similarly to a single resource, the complexity of joining two multi-resource valuation functions is $O(N \log N)$.

### 3.6 Evaluation

Here we empirically evaluate the algorithm’s complexity, and verify that our implementation is efficient enough to be applicable in a real system.

#### 3.6.1 Implementation Details

We implemented the joint function algorithm and Maillé and Tuffin’s [MT04] algorithm in C++ and Python. The code is available as open source\footnote{Available from: \url{https://github.com/liran-funaro/vecfunc-vcg}.}.

The joining of two valuation functions and the upper-bound data structures were implemented in C++. The algorithm can accept any upper-bound data structure as a template parameter. We implemented the naïve joining in C++ as well. Both implementations accept two $R$-dimensional tensors, which represent the clients’ valuation functions (or effective joint valuation functions), and return an $R$-dimensional tensor, which is the joint valuation function. The C++ library is called (via a Python wrapper) to join the functions one by one, and the allocation and payment calculations are implemented in Python.

Our C++ implementation of Maillé and Tuffin’s [MT04] algorithm accepts all the clients’ bids and returns the optimal allocation. This C++ implementation is called once (via a Python wrapper) to compute the optimal allocations, and then again for each winning client to compute the payments.
3.6.2 Benchmark Dataset

We considered three different types of datasets: concave, increasing, and mostly-increasing. We produced 10 datasets of each type, each with 256 clients that participate in the VCG auction. The concave datasets contain concave, strictly increasing valuation functions. These datasets are used to compare our results to Maillé and Tuffin’s method, where the types of valuation functions are very restricted [MT04]. The increasing datasets include weakly increasing valuation functions that might not be concave. This is our main test case as real-life valuation functions may have multiple inflection points [FAS19, YBDJ15, CS14, Wil09, ZWS06, LS07]. Valuation functions, however, are not expected to decrease when more resources are offered, if these resources can be freely discarded. The mostly-increasing datasets include valuation functions with multiple maximum points (non-monotonic). Such functions will increase for a large part of their input, but may occasionally decrease. They are realistic when the hindering resources are not disposed of, as is the case, for example, when allocating more RAM lengthens garbage collection time and performance drops [ABYPBY+14, YBKE06]. We use these datasets to show that our algorithm performs well even with non-monotonic functions. We did not test strictly convex valuation functions as they are not realistic.

For each client, we produced an $R$-dimensional valuation function ($V_i : [0, 1]^R \in \mathbb{R}^R \rightarrow [0, \infty) \in \mathbb{R}$), which it uses as its bid. We generated $R$ intermediate single-dimensional functions ($v_r^i : [0, 1] \in \mathbb{R} \mapsto [0, 1] \in \mathbb{R}$) without loss of generality, where an input value of 1 represents the entire available resource $r$, and an output of 1 represents the client’s maximal valuation of the resource.

To compute a client’s valuation function—i.e., its bid for each bundle of units—for each single-dimensional function, we sampled a vector sized according to the number of available units for each resource and computed the vectors’ tensor product: $V_i = v_1^i \otimes ... \otimes v_R^i$. This yielded an $R$-dimensional tensor with values in the range of $[0, 1] \in \mathbb{R}$. To produce a valuation function of fewer than $R$ dimensions ($0 < r < R$), we used the same dataset but only with the first $r$ intermediate single-dimensional functions.

We modeled the clients’ maximal valuations using data from Azure’s public dataset [CBM+17], which includes information on Azure’s cloud clients, such as the bundle rented by each client. Assuming the client is rational, the cost of the bundle is a lower bound on the client’s valuation of this bundle. We modeled the clients’ expected revenue using a Pareto distribution (standard in economics) with an index of 1.1. A Pareto distribution with this parameter translates to the 80-20 rule: 20% of the population has 80% of the valuation, which is reasonable for income distributions [Sou01].

For each client, we drew a value from this Pareto distribution, with the condition that the value is higher than the client’s bundle cost (i.e., a conditional probability distribution). We then multiplied each client’s $R$-dimensional tensor with the maximal value drawn from the Pareto distribution, to produce the client’s valuation function.
3.6.3 Experimental Setup

We evaluated our algorithm on a machine with 16GB of RAM and two Intel(R) Xeon(R) E5-2420 CPUs @ 1.90GHz with 15MB LLC. Each CPU had six cores with hyper-threading enabled, for a total of 24 hardware threads. The host ran Linux with kernel 4.8.0-58-generic #63 16.04.1-Ubuntu. To reduce measurement noise, we tested using a single core, leaving the rest idle.

3.7 Results

The combination of data structures was chosen for the purpose of the evaluation as it performed the best. This is shown in the data structure comparison in Section 3.7.3.

Our algorithm scales linearly to the number of possible allocations ($N$), for any number of resources, as depicted in Figure 3.2. Although the performance differences between the concave, increasing and mostly-increasing datasets were insignificant, we can see that our algorithm performs better on the mostly-increasing dataset. This is because more allocations were eliminated in the preprocessing phase due to their negative left partial derivative. This preprocessing was included in the algorithm’s runtime.

![Figure 3.2: The performance of our algorithm in each of our datasets (concave, increasing and mostly-increasing).](image)

Adding resources results in larger vectors and thus higher complexity; at the same time, more vectors are eliminated in the preprocessing phase. This is why we see an increase in runtime for up to four resources, after which the performance begins to improve.

Figure 3.2b shows that the multi-resource auction is feasible even in the worst case: for concave/increasing valuation functions, and for three and four resources with 256 clients, a full auction takes less than two minutes for over 60,000 possible allocations.
3.7.1 Naïve Joining of Valuation Functions

The results show (Figure 3.3) that the performance of the naïve approach for joining two valuation functions fits the expected curve, as shown in Section 3.3.3, for any number of resources. Figure 3.3 depicts the performance for the increasing dataset. The naïve joining is not affected by valuation function properties such as monotonicity. The complexity function, described in Section 3.3.3, passes through all the markers, i.e., fits the actual performance perfectly. Each line, however, had to be scaled by a different factor to fit the markers. This might be an effect of the cache prefetching combined with the C-style multidimensional array representation. The naïve joining compares each allocation \( \vec{a}_i \) to all allocations \( \vec{a}_j \) s.t. \( \vec{a}_j \leq \vec{m} - \vec{a}_i \). For multidimensional valuation functions that are represented as C arrays, we will read the array non-continuously when \( \vec{a}_i > 0 \). This will reduce the effectiveness of the cache prefetching as it relies on the continuity of the reading.

![Figure 3.3: The performance of the naïve approach for joining two valuation functions. Markers depict the performance with different numbers of resources. The lines are the complexity function described in Section 3.3.3, scaled to fit the markers.](image)

3.7.2 Ideal Case Analysis

We ran another set of experiments on each dataset, where we counted, in each joining of two valuation functions, the number of allocations that matched the queries of the one valuation function, for each allocation of the other. Figure 3.4 shows the results. The number of matching allocations converges to a constant number. Thus, were we to have an ideal data structure that does not return false positives and with reasonable query and construction time, the complexity of joining two valuation functions would be \( O(N) \).

3.7.3 Data Structure Analysis

We timed each step of the algorithm: creating the data structure, performing the query and fetching the allocations. The results are shown in Figure 3.5.
Simultaneous 1-d binary searches have the fastest construction, but their false-positive ratio grows quickly with $N$, as indicated by the longer fetching time. Hence they are not scalable.

Simultaneous 2-d binary search trees performed similarly to the combination of trees. The construction time of the latter is better as some trees contain fewer vectors. We, therefore, recommend this solution.

In all the tested cases and for all the data structures, the construction time is more than 30% of the total runtime, and over 70% of the total runtime in some cases. Further improvement could be obtained by finding a data structure with a smaller construction time. If we consider simultaneous 1-d binary searches as a lower bound on the construction time—an upper-bound data structure must at least sort the vectors by each dimension—then improving the construction time will improve the algorithm by 10%-20% at most.

For similar reasons, the query time could not be lower than for 1-d binary searches, which is nearly identical to the combination of data structures. Hence, no further improvement in the query time is expected.

The fetching time is 10%-25% of the total time for the combination of data structures. Reducing the number of false positives may reduce this phase’s time. Thus, any improvement of the algorithm by increasing the accuracy of the data structures is bounded by 10%-25%.

Fetching the allocations includes an additional filtering (one by one) to remove all the false positives. Thus, the performance of the final step—i.e., comparing the allocations—is independent of the data structure. From the above, we conclude that any speedup of the algorithm via an improved data structure is limited by 20%-50%.

### 3.7.4 False Positives

We measured the ratio of false positive results when applying our algorithm on all the datasets (Figure 3.6). For any of the resources, the false-positive ratio grows linearly with the number of possible allocations. Using an ideal data structure could reduce the number of false positives by up to a factor of 60. Nonetheless, such an improvement could only speed up the optimization by 10%-25%, as shown in Section 3.7.3.
Figure 3.5: The runtime portion of each function of data structures (DS) when used by our algorithm in an auction. The whiskers represent the 25\textsuperscript{th} and 75\textsuperscript{th} percentiles.

Figure 3.6: False-positive ratio for the combination of data structures for each number of resources. The translucent bands represent the confidence intervals.

### 3.7.5 Separate Single-Resource Auction

We compared our multi-resource VCG auction implementation to the alternative of performing an auction for each resource separately. We used Maillé and Tuffin’s method for a single-resource auction with the concave valuation functions dataset. For each resource $r$, each client bid its intermediate single-dimensional valuation functions $v_r^i$ (see Section 3.6.2). Each client’s maximal valuation was treated as a budget, which was partitioned equally among its valuation functions for each resource. For example, for two resources, a client with a maximal valuation of 10 would have a maximal valuation of 5 for each of its resources.

Such an approach reduces the social welfare by over 60% on average compared to the optimum for two resources (Figure 3.7). When more resources are auctioned, the social welfare decreases even further.
Figure 3.7: The social welfare when using a separate single-resource auction normalized to the optimal social welfare. The whiskers represent the standard deviation.

### 3.7.6 Verification

To verify our implementation, we compared our algorithm’s results with those of Maillé and Tuffin [MT04] using the concave dataset and a single resource. For all the tested numbers of units ($N$), our algorithm produced the same allocation and payments as Maillé and Tuffin’s method.

We also compared our algorithm’s results for two and more resources to those of the naïve implementation. For all the tested numbers of units ($N$) and resources ($R$), our algorithm produced identical results to the naïve implementation.

### 3.8 Related Work

Many solutions were suggested for allocating multiple resources in the cloud. Non-economic solutions may optimize fairness according to clients’ requirements [GZH+11, DFH+12, GN12b, SR13, HGS+11] or consider the clients as a black box and use host measurements instead [XSC+13]. Hadi et al. [GP11] aim to maximize the profit of the providers by meeting client’s SLA. Some achieve truthfulness under restrictive conditions on the types of clients allowed to participate in the auction [LS99, MT04, ABYBPB+14, BBB+08, MN08], or by restrictions on the bidding language [YH07, CDW12, JZZL09, FSV17]. Other solutions offer only near-optimal auction results [SH04, NR07, ZLW14, Fuk11, APTT04].

### 3.9 Conclusions and Future Work

We introduced a new efficient algorithm to allocate multiple divisible resources via a VCG auction, without any restrictions on the valuation functions. We proved the algorithm’s correctness, verified it experimentally, and showed its efficiency on a large number of resources and its scalability when increasing the number of units per resource.

We analyzed how the different properties of the valuation functions affect the algorithm’s performance. We showed that using only concave valuation functions negligibly decreases the complexity compared to increasing valuation functions, and that mostly-increasing ones perform the best.
We combined data structures, tailoring them to our input data to create a data structure that produces fewer false positives and has faster construction time. We analyzed different data structures and showed a potential speedup of up to \(2\times\). Finding a better upper-bound data structure is left for future work.

Our algorithm allows cloud providers to implement the RaaS [ABYBYST12] model. They can deploy a multi-resource auction for allocation of additional resources in an existing VM every two minutes for up to 256 clients in a single physical machine. Our implementation can be adapted simply to use succinct valuation functions that are only defined on a small subset of the allocations. This will eliminate the exponential factor of \(N\) in \(R\), the number of resources, which may greatly improve the performance and might allow a sub-second auction granularity for a large number of clients. A succinct implementation might also support continuous valuation functions with good performance but unbounded complexity. Adapting the implementation for continuous succinct valuation functions is left for future work.
Chapter 4

Memory Elasticity Benchmark

Abstract

Cloud computing handles a vast share of the world’s computing, but it is not as efficient as it could be due to its lack of support for memory elasticity. An environment that supports memory elasticity can dynamically change the size of the application’s memory while it’s running, thereby optimizing the entire system’s use of memory. However, this means at least some of the applications must be memory-elastic. A memory elastic application can deal with memory size changes enforced on it, making the most out of all of the memory it has available at any one time. The performance of an ideal memory-elastic application would not be hindered by frequent memory changes. Instead, it would depend on global values, such as the sum of memory it receives over time.

Memory elasticity has not been achieved thus far due to a circular dependency problem. On the one hand, it is difficult to develop computer systems for memory elasticity without proper benchmarking, driven by actual applications. On the other, application developers do not have an incentive to make their applications memory-elastic, when real-world systems do not support this property nor do they incentivize it economically.

To overcome this challenge, we propose a system of memory-elastic benchmarks and an evaluation methodology for an application’s memory elasticity characteristics. We validate this methodology by using it to accurately predict the performance of an application, with a maximal deviation of 8% on average. The proposed benchmarks and methodology have the potential to help bootstrap computer systems and applications towards memory elasticity.

4.1 Introduction

Today’s cloud providers make every effort to improve their resource utilization and thereby make more money off the same hardware. Rigid allocation prevents them from
utilizing the hardware efficiently, so they offer clients various options for resource elasticity [FAS19]. These elastic options allow clients to change their resource consumption on the fly by exploiting resources that are momentarily unused by other clients.

Resource elasticity is seamless in services such as Application-as-a-Service (AaaS) and serverless computing. Here, clients rent a black-box execution environment that exposes a limited application programming interface (API) they can use. The environment’s resource consumption and workload distribution are controlled by the provider. Thus, the client shares resources with other clients who occupy the same environment. In such services, the provider handles the client’s resource elasticity, relieving the client of this burden.

However, these environments are not suitable for all applications. Some clients need a broader API, have a proprietary application, or simply use a less common application that is not supported by the provider. Other clients may have specific performance requirements that the provider is unable to guarantee. For example, clients might need to be physically closer to their data or maintain some continuity between runs.

Clients who require more than what is provided by AaaS and serverless computing will deploy their applications using Infrastructure-as-a-Service (IaaS) and Container-as-a-Service (CaaS). In such services, clients rent a bundle of rigid, exclusive, resources in the form of a single virtual machine (VM) or an OS container. Many IaaS and CaaS providers offer CPU elasticity in the form of burstable performance, which offers a basic level of CPU performance but can ‘burst’ to a higher level when required. Under certain conditions, this lets clients use more CPU than their initial allocation, in the same VM/container. These providers include Google [Goo18a], Amazon [Ama18a], Azure [Mic18b], CloudSigma [Clo18], and RackSpace [Rac18].

With the current proliferation of CPU elasticity schemes, CPU utilization is adequately optimized and more clients can be allocated to the same physical servers [FAS19]. This leaves memory as the bottleneck resource: it is an expensive resource that limits machine occupancy. Memory elasticity schemes should be a natural extension to CPU elasticity, allowing clients to use more memory in the same VM/container than their initial memory allocation. Clients who can tolerate a temporary memory shortage could benefit from these schemes by lowering the amount of memory they rent exclusively. They could then compensate for the reduced requirement by bursting when they really need more memory. For example, clients who can postpone memory-intensive phases in their operations, can reserve a small initial amount of memory and make use of more memory whenever it is available (e.g., for maintenance operations). This enables clients to time-share memory and the provider can squeeze more applications onto the same hardware [FAS19].

To achieve this, clients need memory-elastic applications that can change their maximal memory usage on the fly and whose performance is proportional to their memory usage. Unfortunately, memory-elastic applications are scarce. Although developers usually strive to make their application’s performance proportionate to its CPU and
bandwidth availability, most applications are not designed with memory elasticity in mind. Developers generally address only the maximal memory footprint of their application. They treat it as constant or a value dictated by the current application workload. The operating system’s swapping mechanism allows seamless application operation when the available memory is insufficient, but this results in a graceless performance degradation; even a minor memory loss may degrade the performance significantly. Such is the case, for example, with memcached\(^1\), as shown in Figure 4.1.

![Figure 4.1](image)

**Figure 4.1:** Off-the-shelf memcached performance (“get” hits per second) as a function of allocated memory when using 500MB internal cache size and memslap as its workload. Data is according to Agmon Ben-Yehuda et al. [ABYPBY\(^+\)14].

Why do developers toil towards making performance scale nicely with the CPU and bandwidth, but neglect doing this for memory? Developing memory-elastic applications requires more work. With a proliferation of memory-elastic systems, developers could be incentivized to make this effort, as they did with CPU elastic applications.

Research has been done into systems that allow frequent memory allocation changes [ABYPBY\(^+\)14, GHDS\(^+\)11]. However, without memory-elastic applications, such systems cannot be used to their full potential and will not be accepted by the commercial community. As early as 2010, cloud provider CloudSigma allowed clients to change their memory allocation and billing during runtime. Unfortunately, other commercial cloud providers did not follow in their footsteps and this option is no longer promoted on the CloudSigma web-page.

We’re seeing a circular dependency problem between memory-elastic applications and systems that require and incentivize such properties. When one of these elements is scarce, there is no incentive to develop the other because real benefit only comes when both elements exist.

A proof that memory-elastic applications exist or can be created is essential to break this circular dependency. **Our first contribution** is a set of memory-elastic applications with a memory to performance trade-off (Section 4.4.3).

\(^1\)Memcached is a popular, open-source, in-memory data store used to reduce the number of times an external data source must be read.
Once the circular dependency problem is solved, and elastic memory systems and applications exist commercially, a language and method for quantifying application elasticity will help clients choose a resource bundle that best suits their application. In addition, quantifying an application’s elasticity will help cloud providers test and optimize their systems. **Our second contribution** comprises a methodology and terminology for evaluating memory elasticity (Section 4.3). These specify how to determine a memory elasticity score for each application that can be used as a *memory elasticity benchmark*.

The methodology is implemented as an open-source *memory elasticity evaluation framework* (Section 4.4). We validated the evaluation process using our framework (Section 4.5) over a set of memory-elastic applications. The results show that our memory elasticity score can accurately predict an application’s performance, with an average deviation of 8%.

## 4.2 Memory Elasticity Methods

This section presents several application properties that can be used to allow memory elasticity.

### 4.2.1 Applications with Resource Trade-off

Mechanisms that were designed to allow trade-off between memory and other resources can be used to provide memory elasticity.

**Memory as cache:** Some applications use the RAM to cache computation results, network traffic, and so on (e.g., using memcached). They can increase their memory footprint when memory is cheap or more available to the application. For example, an application might switch to caching mode for network requests when memory is abundant and avoid caching when high bandwidth is more available or cheaper than memory. Since caches are designed to drop data frequently, cache-enabled applications are already designed to withstand data loss when the memory footprint decreases.

Similarly, applications that rely on the operating system’s page cache to reduce the storage latency (e.g., PostgreSQL\(^2\)) may also be affected by how much memory is available to the operating system. They can seamlessly improve their performance when more memory is available to the operating system. For example, PostgreSQL exhibits performance proportional to the memory availability, as presented in Figure 4.2.

**Intermediate calculations:** Applications that use huge amounts of on-disk data (e.g., databases, Hadoop) can use larger memory buffers to reduce disk access and speed up temporarily data-heavy operations, such as sorting and large matrix multiplication.

**Garbage collected memory:** Applications with automatic memory management (e.g., Java applications) may need fewer garbage-collection cycles with a larger heap.

\(^2\)PostgreSQL is a database application.
and improve their performance as depicted in Figure 4.3. On the other hand, when the memory is too large, the garbage collection might take longer, as shown by Soman et al. [SKB04].

4.2.2 Memory-Aware Applications

Memory-aware applications adjust their memory consumption according to the available memory observed during their initiation period, but cannot adjust it during runtime. Specifically, most of the commonly used memory trade-offs we mentioned (Subsection 4.2.1) are predefined and implemented as memory-aware applications. Memcached only allows the cache size to be set at startup, PostgreSQL’s temporary buffers are defined using a static configuration file, and the Java-Virtual-Machine (JVM) allows setting the minimum and maximum heap size only using the command-line parameters at startup.

These applications can be made memory-elastic by restarting them when the memory changes, but this solution is not suitable when the application needs to be contin-
uously available. With a small effort, as shown in Subsection 4.4.3, these applications can be tweaked to become memory-elastic.

4.2.3 Multiple Short-Lived Jobs

Some applications have multiple short-lived jobs, each with different memory requirements. For example, web servers might require a certain memory to handle each session. They may be able to handle more concurrent sessions when more memory is available. To deal with lack of memory, they can cap the number of concurrent sessions; thus, they trade off memory for latency and throughput.

Another example is batch workload schedulers, such as SLURM or Sun Grid Engine, which execute many short-lived jobs, each with a predefined resource requirement. The scheduler can adapt the concurrency according to the actual memory allocation, running more concurrent jobs when more memory is available. Alternatively, if the jobs are memory-aware, it can keep the concurrency constant and adapt the memory requirements of each job such that the combined memory requirements of all running jobs will match the allocation. Moreover, it can combine these two strategies.

4.2.4 Rigid Applications

Applications that cannot use any of the above techniques will resort to memory-swapping once the available memory is not sufficient for their memory required footprint. This option is usually not advisable as it suffers from inadequate performance.

4.3 Memory Elasticity Metrics

Any developer who implements a mechanism from the previous section will naturally want to measure its effect on the application’s memory elasticity. Developers are used to measuring and comparing metrics such as throughput, goodput, latency, jitter, and load capacity. Such metrics quantify the application’s performance and enable its comparison to similar applications or to other versions of the same application. But can we use them to quantify the application’s elasticity?

We could compare the performance of two applications under the same dynamic memory conditions and consider the one with the better results as more memory-elastic. However, the results may be sensitive to the order or frequency of memory allocations. A single scenario or even several scenarios do not necessarily indicate how the applications behave under untested scenarios. This is because we try to infer memory elasticity from observations of metrics that only hint about elasticity, but do not measure it directly.

Our goal is to quantify an application’s behavior in a dynamic memory scenario and compare it to other applications, using metrics that directly relate to memory
elasticity. We target metrics that capture the characteristics that make an application more memory-elastic.

In this section, we present our novel memory elasticity metrics, which predict how well an application can utilize momentarily available memory, assuming it has the necessary load that requires the memory. First we define a set of static metrics to describe the application’s elasticity, without considering the implications of changing the memory allocation during runtime. This part determines the memory domain in which the application has the potential to be memory-elastic. If the application has such a domain, a second set of metrics can then be defined within this domain. These dynamic metrics quantify how well the application responds within the elasticity domain to dynamic memory changes—changes made during runtime. Finally, in Section 4.4, we describe the experiments we designed to compute these metrics for each application.

In addition to the elasticity metrics, which are comparable across applications, we define elasticity characteristics that can be used by clients to configure their VM and their application.

4.3.1 Static Metrics

First, we define a static memory→performance function \( P_{\text{mem}} \) that describes the performance of the application given a static memory allocation. Then, we define the application’s elasticity domain. We denote by \( \text{mem}_{L} \) the memory allocation that is sufficient for the application to yield the minimum required performance. This might be the memory below which thrashing occurs or it might be defined by a service level agreement (SLA). We denote by \( \text{mem}_{H} \) the maximal memory allocation that yields any performance improvement over a smaller memory allocation. If \( \text{mem}_{L} \) is identical to \( \text{mem}_{H} \), the application is simply inelastic, in which case any other elasticity metric is irrelevant. If the application might be elastic, we define its elasticity domain as \([\text{mem}_{L}, \text{mem}_{H}]\) and its elasticity range as \( \text{mem}_{H} - \text{mem}_{L} \). An application with a larger elasticity range can withstand more dynamic scenarios and thus is considered more elastic. Therefore, our first elasticity metric is the application’s elasticity range.

We also define the improvement factor per memory unit (IFMU) in the elasticity domain as:

\[
IFMU = \frac{P_{\text{mem}}(\text{mem}_{H})}{P_{\text{mem}}(\text{mem}_{L})} \cdot \frac{\text{mem}_{H} - \text{mem}_{L}}{\text{mem}_{H} - \text{mem}_{L}}.
\] (4.1)

An application with greater IFMU has the potential to gain more from a dynamic memory scenario. Hence, the application’s IFMU is our second elasticity metric.

An example of a performance function \( P_{\text{mem}} \) is illustrated in Figure 4.4. In this example, the application needs at least 1 GB of RAM \( (\text{mem}_{L} = 1\text{GB}) \), and gains no performance improvement beyond 4.5 GB of RAM \( (\text{mem}_{H} = 4.5\text{GB}) \). Thus, its elasticity domain is 1 GB to 4.5 GB, its elasticity range is 3.5 GB, and its IFMU is about 3 per GB.
4.3.2 Dynamic Metrics

When the memory changes during runtime, the performance is not necessarily affected immediately. An application might require some time to utilize the added memory. For example, it takes time to fill the cache and it takes even longer to notice an improved hit-rate due to the re-use of cached items. Upon memory reduction, an application might need to prepare for eviction a few seconds ahead of the change, to avoid memory swapping.

Agmon Ben-Yehuda et al. [ABYPBY+14] defined $T_{\text{mem}}$ as an upper bound on the time of the transient performance before stabilization. We extend this scalar definition of $T_{\text{mem}}$ to a function with two variables $T_{\text{mem}}(s, d)$, where $s$ denotes a source memory allocation and $d$ denotes a target (destination) memory allocation.

Figure 4.5 illustrates $T_{\text{mem}}$ in two scenarios: memory is increased in the first and decreased in the second. In phase A (starts at time $t_0$) the memory allocation is $\alpha$, with an average performance of $P_{\text{mem}}(\alpha)$. As phase B begins (time $= t_1$), the memory allocation changes to $\beta$, but the performance stabilizes in $P_{\text{mem}}(\beta)$ only after $t_2 - t_1$ seconds (time $= t_2$). This time is defined as $T_{\text{mem}}(s = \alpha, d = \beta) = t_2 - t_1$. To generalize, when $s < d$ and the memory is increasing, $T_{\text{mem}}(s, d)$ indicates the period starting with the allocation change and ending with the application reaching the statically measured performance ($P_{\text{mem}}(d)$).

To prepare for the decreased memory allocation in phase C (time $= t_3$), the application starts releasing memory ahead of the memory change (time $= t_4$). This time is defined as $T_{\text{mem}}(s = \beta, d = \alpha) = t_4 - t_3$. To generalize, when $s > d$ and the memory is decreasing, $T_{\text{mem}}(s, d)$ indicates how far ahead of the memory change the application started to modify its state to accommodate the updated memory allocation and reduce its performance accordingly. We assume here that the application properly prepares for the memory reduction and manages to release its memory before the allocation is...
applied. If this is not the case, and the application fails to release its memory, it is considered a bug or a misconfiguration.

Figure 4.5: The definition of $T_{\text{mem}}$ given an application’s performance under dynamic conditions.

$T_{\text{mem}}$ is not a good enough metric to compare different applications. During the transient period, application A may reach 90% of the maximal performance after a short period; after this period, it slowly increases to $P_{\text{mem}}(d)$. Application B may have the exact same $T_{\text{mem}}$ and $P_{\text{mem}}$ as A, but during the transient time its performance is low for most of the time. It only increases near the end of the transient, as illustrated in Figure 4.6a.

Figure 4.6: Illustration of two applications with the same $T_{\text{mem}}$ and $P_{\text{mem}}$, but with a different performance loss during the transient period.

The clients using these applications pay for memory that does not immediately translate to their expected performance for the same duration ($T_{\text{mem}}$). However, a client using application A loses more performance over time than a client using application

---

Technion - Computer Science Department - Ph.D. Thesis  PHD-2020-01 - 2020
B, as illustrated by the filled areas above the performance curves in Figure 4.6a. These filled areas represent the aggregate *performance loss* and are formally defined by

\[ L_{\text{mem}}(s, d) = \int_0^{T_{\text{mem}}(s, d)} (P_{\text{mem}}(\max\{s, d\}) - p(x))dx , \]  

(4.2)

where \( p(x) \) is the performance of the application at time \( x \in [0, T_{\text{mem}}(s, d)] \) during the memory transition.

To account for this misrepresentation of actual performance, we define the effective \( T_{\text{mem}} \), denoted by \( E_{\text{mem}}(s, d) \), which is comparable across applications. Consider a fictional scenario, in which the performance changes abruptly between the two performance levels with a time delay, as illustrated in Figure 4.6b. In this fictional scenario, the time delay is chosen so the performance loss in the fictional scenario is identical to the measured performance loss (Equation 4.2). \( E_{\text{mem}}(s, d) \) is defined as that time delay. It is the aggregate performance loss for the memory change, divided by the performance difference between the levels. Formally,

\[ E_{\text{mem}}(s, d) = \frac{L_{\text{mem}}(s, d)}{|P_{\text{mem}}(d) - P_{\text{mem}}(s)|} . \]  

(4.3)

Similarly, upon a decrease in memory allocation, the performance drops in the fictional scenario to \( P_{\text{mem}}(d) \) seconds before the memory allocation actually changes.

Application A has a shorter \( E_{\text{mem}}(s, d) \) compared with B, as illustrated in Figure 4.6b. Because their \( P_{\text{mem}} \) functions are identical, we can directly infer that application A has less performance loss, and thus it is more memory-elastic than application B. Therefore, we consider an application with a shorter \( E_{\text{mem}}(s, d) \) as being more memory-elastic. The application’s \( E_{\text{mem}} \) function is our third and final memory elasticity metric.

### 4.4 Evaluation

In this section we explain how developers can measure their own application’s memory elasticity characteristics and metrics. Then, we design a validation method for these metrics. Finally, we discuss the applications we evaluated using these metrics.

#### 4.4.1 Measuring the Elasticity Metrics

In Section 4.2, we defined three key elasticity metrics that are comparable across applications: elasticity range, IFMU, and \( E_{\text{mem}} \). In addition, we defined different characteristics such as elasticity domain, \( P_{\text{mem}} \) and \( T_{\text{mem}} \), which can be used by clients to configure their VM and their application. The following are the experiments we designed to measure these metrics and characteristics for each application.
Static Metrics

To find each application’s characteristics ($\text{mem}_L$, $\text{mem}_H$ and $P_{\text{mem}}$), we perform a few incremental tests. First, we roughly estimate $\text{mem}_L$ and $\text{mem}_H$, and then we determine their exact values.

We perform static tests in which we choose a high-enough static memory allocation as an initial guess for $\text{mem}_H$, and test the application with this allocation. This high-enough memory allocation can be estimated on the basis of preliminary knowledge (e.g., the application’s working set size in the chosen workload). Otherwise, this test can be repeated with different higher memory allocations, until the performance is similar in at least two different memory allocations.

In this test we also measure the warm-up time: how long it takes the application to reach its maximal performance and maximal memory usage. The warm-up time is required for future experiments.

We need additional static tests in order to choose a low static memory allocation that will serve as an initial guess for $\text{mem}_L$. This is chosen such that the application can still function properly and the OS swapping mechanism is not activated. Avoiding swapping is a strict requirement for this test, since guest swapping may swap out other applications’ memory, or even that of the operating system. This would increase the memory available to the tested application and possibly lead to unexpected results.

Next, we perform a pyramid test, in which an application starts working on a guest virtual machine (VM) with a maximal memory allocation (i.e., our initial guess of $\text{mem}_H$). We allow the application to warm up for the warm-up duration we found in the static tests. Then we gradually decrease the memory allocation, in steps, until we reach our initial guess of $\text{mem}_L$. Throughout each step, the memory allocation remains constant. To avoid measuring transient effects, each step includes enough time to measure the application’s performance in a reliable manner after the warm-up time. The time that is considered to be sufficient is usually dictated by the workload. For example, PostgreSQL benchmark (pgbench) recommends running a test for at least a few minutes to get reproducible results.

To validate that the application’s performance in a certain memory allocation step is not affected by the step from which it descended, we repeat the process in reverse—gradually increasing the allocation by steps to the maximum. We verify that the performance in the increasing phase is similar to the performance in the decreasing phase for each tested memory allocation.

We also record the application’s average performance and its standard deviation for each memory allocation. We then use these measurements to determine the exact $\text{mem}_L$ and $\text{mem}_H$ values, and the $P_{\text{mem}}$ function for the application.

For the results presented in this chapter of the thesis, we tested the guest memory allocations in steps of 512 MB: 1024 MB, 1536 MB, 2048 MB, and so forth, until there
was no performance improvement. The lowest memory allocation we could measure without swapping was 896 MB.

**Dynamic Metrics**

First, for each application we determine its *safe retreat time*: how much time ahead of the memory drop the application must start reducing its memory consumption to avoid swapping. To this end, we conduct a *drop-test* in which the memory allocation drops from \( \text{mem}_H \) to \( \text{mem}_L \). Here, the application starts changing its state as soon as it is notified of the memory allocation drop. Our default settings gave the application a prior notice of 30 seconds. We then measure how long it takes the application to reach the lower memory state. This duration is used as the application’s safe retreat time for the subsequent tests.

To generate \( T_{\text{mem}}(s, d) \) and \( E_{\text{mem}}(s, d) \) for each application, we perform an *oscillation test*. In such a test, the memory allocation oscillates between two values for five cycles, and the performance is recorded. We performed an oscillation test for each application, for any pair of values taken from the values tested in the pyramid test.

As previously defined (Subsection 4.3.2), when we increase the memory, the transition period lasts from the application of the new memory allocation to the stabilization of the performance. We define that the performance stabilizes when the application’s average performance over a predefined time-window reaches \( P_{\text{mem}}(d) \) for the first time after the allocation change. The time-window size is different for each application and is chosen according to the application’s characteristics (e.g., PostgreSQL requires a window of a few minutes to mask the measurement noise).

When we decrease the memory, the transition period lasts from the time the application proactively prepares for the lower memory allocation until the new memory is allocated. This definition relies on the valid measurement of the safe retreat time. Indeed, we validated in our experiments that the safe retreat time was sufficient. We also log the application’s internal memory state during the experiment for this purpose.

For each application, for each transition in each test, we compute \( T_{\text{mem}}(s, d) \) and \( E_{\text{mem}}(s, d) \) from the performance measurements. Then, for each application, we compute the average values of \( T_{\text{mem}}(s, d) \) and \( E_{\text{mem}}(s, d) \) for each pair of source and target \((s, d)\), to be used as the \( T_{\text{mem}}(s, d) \) and \( E_{\text{mem}}(s, d) \) for that application.

### 4.4.2 Validating our Metrics

To validate our memory elasticity metrics, we needed to show they are able to predict the application’s performance in any dynamic memory scenario. To start, we randomly generated multiple benchmark traces of differences in memory allocations; these indicated the histories of how much memory was added or taken from the previous allocation. The traces represented different scenarios, characterized by two properties:

1. *Rate*: the number of memory changes per hour.
2. Amplitude: the maximal difference between two consecutive memory allocations.

Since the purpose of this set of experiments was to validate the dynamic properties against trace results and not compare applications, we tested each application on several of these traces in which its elasticity could be expressed. The rates were limited by the application’s $T_{mem}$. That is, the allocation cycle time $\frac{360}{\text{rate}}$ seconds had to be greater than the maximal $T_{mem}$. The amplitudes were limited by the application’s elasticity range.

Then, we calculated the average actual performance of the application over the entire period of the experiment, excluding the warm-up time at the beginning of the experiment. We calculated the expected “ideal” average performance, using the static profiler by applying the static memory performance function ($P_{mem}$) to the application’s memory allocation. We also calculated the expected “realistic” average performance, using the elastic profiler, which calculates the performance in the fictional scenario, inferred by the application’s $E_{mem}$ function.

To calculate how accurate each of our (ideal and realistic) predictions were compared with the actual performance, we used the following norm:

$$100 \cdot \left| \frac{\text{predicated performance}}{\text{actual performance}} - 1 \right|,$$

which is the maximal deviation (in percentage) from the actual performance.

Our memory elasticity metrics are meaningful if they can be used to accurately predict the application’s performance with good probability. If these metrics are valid, they will allow us to evaluate and quantify an application’s elasticity by performing four simple tests: static-test, pyramid-test, drop-test, and oscillation-test, without the need to repeat this validation process for future applications.

4.4.3 Applications and Benchmarks

We wanted to identify applications that could be used as elastic benchmarks. Our preference was for memory-elastic off-the-shelf applications, but we also modified applications or tweaked their settings to enable memory-elasticity.

Most benchmarks suites are composed of a benchmarking utility that runs different applications. These benchmarking utilities generally execute an application for a short period and measure the average performance over that period (e.g., DaCappo [BGH+06], SPEC CPU [BL+18]). This operation is repeated to produce statistically significant results. However, this standard mode of operation is not fitting for the measurement of performance over different runtime phases, especially while changing memory allocations. To this end, the application needs to continuously run over a period of time that is significantly longer than the transient effects of a memory allocation change.
To evaluate elasticity we require a benchmarking utility that executes an application for any time period with a constant load and reports frequent performance statistics during the benchmark runtime. This could work well, for example, to test an elastic application that is an always-on, always-available service, responding to client requests. Ideally, the application being tested should be able to change its mode or adapt its memory utilization given notification of an upcoming memory allocation, or do so seamlessly without requiring a hint.

We tested the following applications.

Memcached [Fit04] is a memory cache service that runs in the background alongside another application. It can be used to cache computation results, network responses, and so forth. The performance function \( P_{mem} \) of off-the-shelf memcached, shown in Figure 4.1, resembles a step function and is typical of the operating system’s efforts to handle memory pressure through swapping [ABYPBY+14]. We used the elastic adaptation of memcached \(^3\) [ABYPBY+14, AAS18, MFA+18], which supports memory elasticity by changing its memory footprint upon receiving a command via a socket. Memcached has its memory arranged in linked lists of slabs, for which it maintains metadata, so that it can tell which slab to overwrite. The elastic memcached used this mechanism to choose which memory slabs to lose when the memory footprint needed to be decreased, along with the malloc_trim() function, which forces libc to release memory from the heap back to the operating system. As its workload driver, we used memaslap with concurrency of 20, a window size of 100K, and 90% get requests (the rest are set requests).

MemoryConsumer [ABYPBY+14] is a dedicated dynamic memory benchmark that accesses random pages in a predefined memory region. When allocated less memory, it is informed of the change and only attempts to access memory pages it can reach (without the risk of touching out-of-boundary pages). The application initiates a ‘sleep’ operation to artificially prevent access to any pages beyond the memory it knows it has; this reduces the throughput without causing any additional issues resulting from actually touching swapped out pages. Hence, its hit rate increases linearly with the available memory. The benefit of this artificial benchmark is that it is designed to have an almost zero \( T_{mem} \), with highly reproducible performance measurements. We tested it with a constant load of 10 threads accessing the memory simultaneously.

4.4.4 Implementation Details

We implemented the benchmarking framework in Python 3.7. Some of the code is based on the evaluation framework for Ginseng [FAS16, ABYPBY+14], which is based on the memory overcommitment manager (MOM) by Litke [Lit10]. The code is available as open source\(^4\).

\(^3\)The elastic memcached is available from: https://github.com/ladypine/memcached.

\(^4\)The source code is available from: https://github.com/liran-funaro/elastic-benchmarks.
4.4.5 Experimental Setup

We evaluated our framework on a machine with 16 GB of RAM and 2 Intel(R) Xeon(R) E5-2420 CPUs @ 1.90 GHz with 15 MB LLC. Each CPU had 6 hyper-threaded cores, for a total of 24 hardware threads. Each application was set up in advance in a qcow2 image of a guest virtual machine, running on a QEMU/KVM instance. Each guest VM was allocated with 4 cores, and was pinned to cores on a single NUMA node. We controlled the guest memory using the memory balloon module [Wal02]. The host ran Ubuntu 16.04.1 with kernel 4.8.0-58-generic #63, and the guests ran Ubuntu 18.04.2 with kernel 4.15.0-50-generic #54. To reduce measurement noise, we disabled hyper-threading, pstate, and ksm in the host, and tested one benchmark at a time.

4.5 Results

In this section, we analyze each of the applications by evaluating the application’s static and dynamic metrics, and validating them.

4.5.1 Memcached

The static evaluation of the elastic memcached is presented in Figure 4.7. This application’s memory domain is from 896 MB to 3584 MB, making its memory range 2688 MB. Our static evaluation suggested that memcached requires 7 minutes of warm-up time and its improvement factor (IFMU) is 1.8 per GB.

![Figure 4.7: Memcached performance (“get” hits per second) as a function of allocated memory.](image)

The dynamic evaluation of memcached is depicted in Figure 4.8. The drop-test suggests that memcached needs about 6 seconds of safe retreat time. We can see that although $T_{mem}(s, d) = 6$ when $d < s$, in some cases $E_{mem}(s, d) > 6$ for these parameters. This is because memcached has to do extra work in order to choose the least used items to release. This reduces its performance below the average performance for the target memory and makes the effective transient period longer than the actual one. When $d > s$, larger memory changes induce longer $T_{mem}$ and $E_{mem}$, as expected.
Figure 4.8: Elastic and off-the-shelf memcached oscillation test results.

Off-the-shelf memcached had the exact same range, IFMU, and $P_{mem}$ as the elastic one. However, its dynamic evaluation showed that it suffers far more from transitions, as depicted in Figure 4.8c. The measured $E_{mem}$ of the elastic memcached is at least 33% shorter than that of the off-the-shelf one, and for most of the transitions, it is at least 90% shorter. Given that the static properties of the application did not change, we can clearly determine that the effort to modify memcached to be memory-elastic has been useful.

4.5.2 Memory Consumer

The static evaluation of memory consumer is depicted in Figure 4.9. This application’s memory domain is 896 MB to 2048 MB, making its memory range 1152 MB. The static evaluation suggests that memory consumer needs less than 1 minute of warm-up and it has an improvement factor (IFMU) of 2.2 per GB.

The dynamic evaluation of memory consumer is depicted in Figure 4.10. The drop-test suggested that memory consumer needs about 3 seconds of safe retreat time. Memory consumer has minor $E_{mem}$ values (compared with any memcached version), because it immediately allocates and releases memory. Memory consumer was designed specifically to have such low $E_{mem}$ values.
Figure 4.9: Memory consumer performance as a function of allocated memory.

The $T_{\text{mem}}$ drop values ($d < s$) are longer than the increase values ($d > s$), which is counterintuitive. In a real application, we expect that it takes more time to occupy the memory than to release it. This application, however, allocates the memory immediately, but releases the memory ahead of the memory allocation, to be on the safe side.

Figure 4.10: Memory consumer oscillation test results.

4.5.3 Validation

In addition to the above tests, for each application, we validated the dynamic metrics according to the method described in Subsection 4.4.2.

Memcached

The accuracy distribution of the static and the elastic profiler is presented in Figure 4.11. The static profiler’s accuracy has an average deviation of 13% from the actual performance. Its accuracy has a high variation of up to 40% as seen in Figure 4.11. The elastic profiler, however, is twice as accurate. It predicts the performance of memcached with an average deviation of 8% from the actual performance over all verification parameters, and with no deviation greater than 20%.
Figure 4.11: Memcached profiler prediction accuracy distribution. Each rectangle shows the results for different parameters according to its row and column. The joined row shows all the experiments with any memory amplitude, and the joined column shows all the experiments with any change rate.

Memory Consumer

The accuracy distribution of the static and the elastic profiler of memory consumer is depicted in Figure 4.12. The static and the elastic profilers can predict the performance with an average deviation of less than 1% from the actual performance. The elastic profiler cannot improve much over the static one because the transient period for memory consumer is negligible. We only validated this application with one combination of parameters because it produced sufficient accuracy. Thus, we would not gain more insight from less dynamic scenarios.

Figure 4.12: Memory consumer profiler prediction accuracy distribution with a rate of 300 memory changes per hour and a maximal amplitude of 2 GB.
4.6 Related Work

Since the time memory balloons were developed by Waldspurger [Wal02], several systems and concepts have been built to enable system administrators to dynamically reallocate RAM among virtual machines (e.g., Litke [Lit10], Shrawankar and Eram Shaikh [SS15], Gordon et al. [GHDS+11, HGS+11], Nathuji et al. [NKG10], Dolev et al. [DFH+12], and Heo et al. [HZPW09]).

Many researchers have addressed the issue of evaluating a system’s elasticity [HKR13, HDYW16, GHD13, HKWG15, WHGK14]. However, little work has been done on the evaluation and development of memory elasticity for applications. This is in contrast with the application CPU elasticity, which was studied extensively, as reviewed by Kumar [Kum02].

The non profit maximizing operating system has an interactive, on-the-fly configurable network stack, which adapts to different monetary conditions according to its service level agreement [BYABYT16]. All these systems could be better evaluated given a solid benchmark suite for elastic memory.

Salomie et al. [SARE13] implemented a kernel module that supports ballooning pages right out of the application’s memory to the host (i.e., application level ballooning) and demonstrated it on a Java-virtual-machine (JVM). Such a solution may be helpful for transferring memory faster, but it depends on a specific operating system and requires the installation of a kernel module. This approach increases the coupling between software layers, which complicates the adoption of the application.

The Automatically Tuned Linear Algebra Software (ATLAS) [WD98] is memory aware: it configures itself by benchmarking to optimally use the hardware it runs on, considering mainly the size of the cache. It does so upon installation and does not change the configuration on-the-fly.

4.7 Conclusions and Future Work

We developed a method and a framework for evaluating memory-elastic applications in a comparable manner, and demonstrated its usage. For example, our evaluations demonstrated that the $E_{mem}$ values of memory consumer are significantly lower than memcached values; this indicated that memory consumer has a higher elasticity with regard to the transient period. In addition, the elastic memcached had significantly shorter $E_{mem}$ values compared to the non-elastic version.

We verified our framework by showing it can predict the performance of an application under dynamic memory changes with high accuracy. Our evaluations showed an average deviation of 8% and 1% of the actual performance for memcached and memory consumer, respectively.

Additional applications will allow more clients to use this framework to evaluate their memory elasticity bundle offerings, and allow providers to easily evaluate new
memory-elastic systems. We note that PostgreSQL and the Java virtual machine are good candidates for these efforts. Adding these applications and more to the framework is left for future work.
Chapter 5

Stochastic Resource Allocation

Abstract

Suboptimal resource utilization among public and private cloud providers prevents them from maximizing their economic potential. Long-term allocated resources are often idle when they might have been subleased for a short period. Alternatively, arbitrary resource overcommitment may lead to unpredictable client performance.

We propose a mechanism for fixed availability (traditional) resource allocation alongside stochastic resource allocation in the form of shares. We show its benefit for private and public cloud providers and for a wide range of clients. Our simulations show that our mechanism can increase server consolidation by 5.6 times on average compared with selling only fixed performance resources, and by 1.7 times compared with burstable instances, which is the most prevalent flexible allocation method. Our mechanism also yields better performance (i.e., higher revenues) or a lower cost than burstable instances for a wide range of clients, making it more profitable for them.

5.1 Introduction

Most cloud provider costs result from purchased servers, power requirements, and infrastructure (power and cooling systems) [Ham18, PMZ+10, WDG+16]. Hence, most of these costs are proportional to the number of servers. Although the CPUs on active servers are underutilized [Liu12, AFG+10], these servers still draw most of the power they would draw if their CPUs were fully utilized [SJSS16]. Further client consolidation would increase the revenues per server without increasing the costs [BCH13].

CPU underutilization originates from the provider’s obligation to provide its clients with their contracted quality of service (QoS) according to their service-level agreement (SLA) [ABYBYST12]. Infrastructure as a Service (IaaS) and Container as a Service (CaaS) clients rent a bundle of resources in the form of a virtual machine (VM) or an OS container. Even though clients may choose a fixed instance contract (e.g., a fixed performance instance on Amazon EC2), with a bundle that meets their load needs,
they will not use their resources all the time. Moreover, because the proportion of resources in the bundle, e.g. the ratio of RAM to CPUs, is determined by the provider, it is not always optimal for the client. Therefore, most resources will generally have unused margins \([RTG^+12]\). Suboptimal utilization might still be a problem even with less rigid services such as Application as a Service (AaaS) and serverless computing, where the client does not necessarily rent a bundle of resources but rather a black-box execution environment. Under such models, providers still set resources aside to handle unexpected loads \([CBLS13]\) or cater to preferred clients requiring resources on short notice \([LKHRE15]\). Providers lose money on these contingency plans because they do not maximize resource utilization. Maximizing resource utilization will enable providers to consolidate more clients on each machine, increase the income per machine, and reduce the pressure to expand their infrastructure.

How should providers allocate unutilized resources residing in a single physical machine among their clients in a manner that will increase the revenues of the former and incentivize the latter to agree to this allocation scheme?

A simple method for utilizing momentarily available resources in a single physical machine is to divide them among the clients/services that reside on that server: either evenly or proportionally to the amount of fixed resources they rented. Without an additional billing mechanism, the provider has no direct benefit from this approach. Moreover, if this is an ongoing state of affairs, the clients might take the higher QoS for granted, and be disappointed when it decreases to the fixed (paid for) level.

The most popular approach for utilizing momentarily available resources is *burstable performance* \([Goo18a, Ama18a, Mic18b, Clo18, Rac18]\). On Amazon \([Ama18a]\) and Azure \([Mic18b]\), for example, a client gains credits periodically, at an even rate. The client either consumes credits by using the resource or hoards the credits and “bursts” later, using more resources than its periodic credits allow. If the client runs out of credits, it must wait for the next period to use the resource again. The credit mechanism limits the client to a certain average resource consumption according to its credit allocation rate.

We assume a client that consumes resources at an even rate, in line with its credits, is guaranteed never to be starved. This assumption requires the provider to prioritize clients that have not yet consumed the credits of the current period over clients that are currently “bursting”. To never starve a non-bursting client, the provider must reserve for each such client a resource amount that equals the client’s credit rate.

Both these quantities, the average resource consumption and the reserved resource quantity, are defined by the same number—the credit rate. This coupling is the main drawback of burstable instances. It induces two limitations: First, it forces a client that can function well without reserved resources to rent a bundle that offers them, just to get an average consumption rate. This requires that the provider reserve these resources, which in turn limits the number of clients per server. Second, this coupling limits each client’s average utilization (over a period) to its reserved allocation. The sum
of reserved resources in a server thereby serves as an upper limit on the server’s total resource utilization. Resources not reserved for clients (e.g., the provider’s reserves) cannot contribute to the total average utilization. Clients that did not reach their average utilization limit further reduce the total average utilization.

In unlimited burstable instances, the provider allows the client to exceed its average consumption rate, and charges a fixed (higher) price for the surplus average consumption during a billing period\(^1\). This overcomes the utilization limit but does not increase the number of clients per server.

In this chapter of the thesis, we show that decoupling the reserved resource quantity from the average consumption rate allows clients to explicitly reserve only the resources they truly require. In turn, this allows the provider to increase the number of clients per server.

Our contribution is twofold. First, we introduce a Stochastic Allocation (SA) mechanism that allows the provider to sell reserved resources alongside an additional stochastic allocation. We compare this mechanism to other mechanisms using simulations (Section 5.5) that cover a wide range of clients (Section 5.4). For over 56% of these clients, our mechanism is more profitable than the burstable performance mechanism. We show a 1.7 times increase in the number of clients per server compared with burstable instances (Section 5.7). We further show that such a mechanism can increase the provider’s overall profits by 28\%-44\%, depending on our assumptions about the provider’s profit margins. Moreover, we show that a private cloud provider can utilize this mechanism to increase its clients’ aggregated economic benefit, while reducing the provider’s costs.

Second, we present the Stochastic Allocation Simulator (SAS), a validated infrastructure for cloud simulations. This infrastructure can generate a large dataset of realistic clients with different behaviors and simulate their rational bundle selection given a known distribution of available resources. It then simulates the load on a server using these clients (using a completely fair scheduler (CFS) \([\text{Pab09]}\)) and yields highly detailed statistical information. SAS is published as an open-source project along with the code to replicate our simulations and their data\(^2\).

5.2 Allocation Mechanisms and Incentives

A resource allocation mechanism is useful only if it is incentive-compatible for clients and providers. In other words, they must all gain from participating. In this section we classify clients by their requirements, discuss providers’ goals, and then review a number of allocation mechanisms in view of the parties’ incentives.

\(^1\)A day in Amazon [Ama18a], 5 minutes in CloudSigma [Clo18].

5.2.1 Client Requirements

Most client requirements range between long-term requirements and immediate requirements. A long-term requirements client may have non-interactive workloads. It might value finishing the workload by or before a deadline [CBM+17], but it might not value getting partial results ahead of time. It only cares about long-term promises that guarantee meeting deadlines with high probability.

At the other end of the spectrum is the immediate requirements client. It runs brief independent workloads or an interactive workload, and sleeps the rest of the time. The failure or fulfillment of one workload does not affect the client’s future requirements. It only cares for instant gratification. Such a client may not wish to rent a full (usually underutilized) machine, which might guarantee each workload is finished on time. Rather, it may prefer to yield its resources when they are idle and be compensated accordingly.

The clients in between these two extremes have a combination of long-term and immediate requirements. They need a guarantee that their long-term requirements will be met, but might demand additional ad hoc resources to support an immediate load surge. They are mixed requirements clients.

Websites, for example, might partition their budget proportionately to the gain from satisfying these dual requirements. They would not like to miss an opportunity to show an advertisement to their visitors. Hence, the budget for their immediate requirements might be proportionate to the income from an ad. In addition, they would like to preserve their customers’ visit rate. Users are unlikely to abandon them because of a momentary slowdown, but regular low responsiveness might reduce user visits. Thus, their budget for long-term requirements might be proportionate to the estimated loss of revenues due to an expected abandon rate.

The Azure Public Dataset [CBM+17] offers insight into how real cloud users are distributed by category/type. Most clients (60%) that ran over three days in the dataset were classified as delay insensitive (i.e., long-term requirements clients), and 33% were classified as interactive (i.e., short-term requirements clients). The other 7% could not be classified. Cortez et al. [CBM+17], who classified the clients, suggested that clients with short workloads, each with a deadline, might be classified as either interactive or unknown.

5.2.2 Provider Goals

Public cloud providers that rent computing resources to paying clients would like to maximize their profit from renting their machines. However, prices are limited due to price wars among providers [ABYBYST14]. Consequently, to increase their profits, public cloud providers resort to higher consolidation and overcommitment [GN12a, TT13]: they sell the same resources to more clients, risking an SLA violation and having to pay client compensation [TKTHR14].
Private cloud providers would like to maximize the aggregated value all their clients draw from the cloud: the game-theoretic concept of \textit{social welfare}. Accordingly, they would like to prioritize the most financially valuable clients, because their workloads carry the maximal benefit to the organization. Additionally, they wish to maximize client consolidation in their existing infrastructure, similar to public cloud providers.

### 5.2.3 Allocation Mechanisms

In this section we survey allocation mechanisms and pricing schemes that increase the server consolidation by incentivizing clients to reduce their reserved requirements. Providers often offer their clients one or more of these mechanisms simultaneously.

**Fixed performance** instances consist of a bundle of reserved resources. They are guaranteed to be constantly available to the client throughout the rental period. A long-term requirements client, however, usually only fully utilizes one resource in the bundle—which is its bottleneck. If the bundle's size and shape are determined by the provider, a long-term requirements client is likely to have non-required, unutilized resource margins. Clients that also have immediate requirements need to compromise: over-provision according to the maximal load at high costs—or under-provision and save money, at the risk of being short on resources. Thus, most clients pay for resources they do not utilize, and which the provider cannot resell.

**Burstable performance** instances offer a baseline resource guarantee, which may be exceeded when necessary. These instances are suitable for clients with immediate requirements, which are mostly inactive until driven by an event. Long-term requirements clients might not require bursting, as they typically use the resources at an even, maximal rate.

We compared pricing of burstable and fixed performance instances using identical CPU models and optimization types. According to the regression analysis, using Amazon's and Azure's publicly available pricing data [Ama17, Mic18b, Ama18a], a burstable instance with similar price and characteristics to a fixed instance is limited to an average performance of 10\%–30\% of the fixed instance maximal performance, depending on the instance type.

Therefore, clients utilizing, on average, less than 10\%–30\% of their maximal resources will save money by renting burstable instances instead of fixed ones. Accordingly, pure long-term requirements clients will pay for burstable instances 3.3–10 times more than their fixed instance bill, to get the same performance.

**Preemptible instances**, deployed by many providers [Ama18b, Mic18a, Goo18b, Ali18, Pac18], offer a low-cost VM whose availability depends on the available resources in the cloud. The provider can shut down the instance at any time to reclaim the resources. An immediate requirements client can scale horizontally, i.e., expand the number of active VMs with an increasing load, at a low cost. Nevertheless, horizontal expansion incurs an overhead, for the provider and clients, when booting a machine and
gracefully shutting it down. A long-term requirements client might use these instances whenever available or fall back to higher cost, nonpreemptible, instances [LBMAL12].

This mechanism allows the provider to rent unallocated resources while waiting for higher paying clients to rent them. Unused reserved resources of other clients, however, cannot be used to create a new preemptible instance. Reserved resources must be supplied on demand, which is not possible due to the long notification\(^3\) required before shutting down the preemptible instance.

**Posted prices**, formerly deployed by CloudSigma [Kov18], are a mechanism for resource pressure management. In this mechanism, the provider periodically changes the resource unit-prices, which it posts publicly via an online API. Clients with immediate requirements can use the resource when the prices are low, while using the baseline for their long-term requirements.

If clients do not cap their resource utilization in response to price changes, posted prices might be ineffective in increasing client density. Clients might not reduce their consumption in response to price surges, as they might value steady performance as long as the average cost remains within their budget [LS07, Wil09]. Moreover, clients will agree to participate only if price surges are limited by the cost of a horizontal expansion—the clients’ alternative.

**Immediate resource auctions** allow clients to rent a baseline performance, and bid—every few seconds—for an immediate, temporary, resource allocation. Such a mechanism was implemented in Ginseng for RAM [ABYPBY+14] and last level cache (LLC) [FAS16]. It is suitable for clients with immediate requirements that need not plan ahead. Such clients can bid according to their momentary expected valuation of the resource. Clients with long-term requirements can also benefit from the mechanism by getting cheap resources when these are abundant. Nevertheless, it is hard for such clients to assign a momentary value to a resource with unknown future availability.

Similarly to posted prices, a horizontal expansion might be more profitable for some clients than costly, temporary but immediate, allocation.

### 5.3 Stochastic Allocation

Client performance can be quantified in terms of stochastic properties such as mean, standard deviation, minimum or maximum. The client can infer these on the basis of its experience [VdBVB15]. Different clients might assign different monetary values to these properties [OZN+12, VPB09, IOY+11, MH12, FJV+12, Sha18, CCBW14], as described in Section 5.2.1. When offered a choice of bundles, each differently priced and stochastically characterized, the client can estimate its valuation for the bundle and its expected profit if selected [CLN12].

\(^3\)30 seconds in Google Cloud [Goo18b], two minutes for Amazon [Ama18b].
Accordingly, to effectively utilize the resources, we propose the Stochastic Allocation (SA) mechanism. Under the SA mechanism, the provider offers clients a combination: a choice of a stochastic allocation class and an amount of reserved resources. The provider posts fixed unit-prices for both goods. Each client may choose to rent reserved and/or stochastic resources—the latter enable consumption of resources that are unused by other clients. The provider prioritizes clients when they consume their reserved resources. To allow clients to make an educated decision when renting such a stochastic bundle, the provider publishes statistics on resource availability, for each SA class.

SA supports an asymmetrical bundle of reserved resources and SA classes. It allows clients to reduce their reserved resource requirements, and thus increases the number of clients per server. The SA mechanism bridges the idle resource gap between the provider’s obligation to safeguard clients’ reserved resources and their dynamic demands.

5.3.1 Implementing Stochastic Allocation via Shares

To evaluate the SA mechanism for CPU using shares, we simulated it on the basis of the completely fair scheduler (CFS) [Pab09] algorithm, as was implemented in the Linux kernel. CFS combines a share-based resource allocation system with a hard rate limit. Each task is assigned a number of shares, which entitle it to a portion of the resources proportional to the number of allocated shares. A task accumulates virtual runtime according to its actual runtime divided by its number of shares. In each period, the task with the least accumulated virtual runtime is run. Thus, at any given time, the virtual accumulated time of all active clients is nearly identical.

CFS can easily implement a credit system, because having a portion of the shares is effectively the same as reserving the same portion of the resources. For example, in a machine with 64 CPUs, a process allocated 1 share out of a total of 64 is guaranteed at least 1 CPU. Nevertheless, CFS does not support a key feature of SA: defining a different consumption share for the leftover CPUs. Rather, the consumption rate is constrained to be identical to the reserved portion of the shares.

We added a degree of freedom to CFS, adapting it to support asymmetric reserved resources and share allocations. We duplicated the CFS logic, to have a second, alternative, virtual runtime clock and a second priority queue that is sorted according to the alternative clock. The existing knobs (share and limit) are associated with the original, main CFS, which is used as a reserve mechanism. The alternative CFS takes the newly introduced alt-share and alt-limit knobs. Once the limit rate of the main CFS is reached, the task is moved to the alternative one. The scheduler only pulls tasks from the alt-queue if the original queue is empty. For example, to reserve 1 CPU out of 64, allocate 10 shares (e.g., of 100) and set a total limit of 2 CPUs, the administrator would allocate a process with

- 1 main share such that \( \frac{\text{client main share}}{\text{total main shares}} = \frac{\text{reserved CPU}}{\text{total CPUs}} \).
• a main limit of 1 CPU to mark the point where excess resources start being consumed,

• 10 alt-shares (out of 100) for the stochastic part, and

• an alt-limit of 2 CPUs, for the actual capping.

Our adapted CFS allows the provider to implement our SA mechanism. Once the client chooses its bundle of reserved resources and number of shares, the provider assigns the client to the appropriate server while trying to maintain an even distribution of total shares across the servers.

5.4 Realistic Workload Modeling

How effective would the SA mechanism be on a cloud? How does it compare with other mechanisms such as fixed performance and burstable instances? How would it affect client density? How would it affect the provider’s profits?

Experimentation at this scale requires a full commercial cloud and thousands of real clients. To answer these questions, we resorted to simulations, showing the method’s potential on equal grounds with the simulated burstable-performance. We modeled client workload and then simulated a cloud with various mechanisms (Section 5.5).

Our realistic modeling is based on real data from the Azure Public Dataset [CBM+17], which was used to deduce real consumption and market demand. We generated 12 client-datasets, which we ran in parallel on our 12-core machine. Each dataset contained 1024 clients.

To create a single dataset, we sampled a random group of clients from Azure’s dataset. To model each client, we generated three functions that were consistent with its given statistics: performance, load, and valuation.

5.4.1 Performance

The performance function (resource → perf) indicates the maximal performance (in the range [0, 1]) that a resource allocation can yield, assuming the workload can utilize the resource. We generated a random monotonically rising function for each client. These functions are not necessarily always concave; they can have inflection points, as a real application utility function might have [TCGK12, ZWS06]. Examples of generated performance functions are shown in Fig. 5.1.

5.4.2 Load

The load function (time → perf) indicates the client’s required performance (in the range [0, 1]) at a given time. For each sampled client, we generated a realistic load function for a single day, in 12-second intervals (i.e., 7200 samples for each client for
each day). For IaaS and CaaS clients, this means that their VM/container was active for at least a day. For AaaS and serverless clients, this means that they had many small tasks which may span across a whole day and are considered as their day’s load.

To do this, we used the client’s sampled load from Azure’s data. Azure’s data contain statistics in 5-minute intervals per client, for up to 30 days. Each sample contains minimum, maximum and average CPU usage. To get enough statistical information, we chose only clients with at least a day’s worth of data (288 samples).

The simplest way to interpret the data would be to maintain the average CPU usage constant over each 5-minute period, but then the extremum values would not be reached. To remedy that, the usage must reach other values, in particular the minimum and maximum values, and yet maintain the average usage. To take all the values in the sample into account, we divided each sample time into multiple samples that adhere to the given minimum, maximum and average CPU usage: min and max are visited, usage values are only between these values, and the average value is according to the measured data. This mandatory enhancement of the data introduces several degrees of freedom: which values to visit and when.

A simple solution is to visit the minimum and maximum once, and then fill the rest of the time with a value that will correct the average. This solution is arbitrary: the minimum and maximum values can be visited more. Also, it is natural for more values in the min-max range to be visited as well. This simple but reality-consistent solution can be smoothly extended using a beta distribution, which can be defined by its average, bounds, and density—i.e., having more samples near the mean value or near the bounds. The density represents a degree of freedom in the function choice.

For each 5-minute sample, we generated a beta distribution with the sample’s characteristics (i.e., minimum, maximum and average) and a density of one, which yields a uniform distribution when the average is exactly in the middle of the bounds (see Algorithm 5.1). We then drew samples from this beta distribution to fill the 5-minute interval with 25 samples (a sample every 12 seconds during the five minutes). This preserved the stochastic characteristics of the data. Nonetheless, we did not assume any correlations between different client’s load variations in each 5-minute interval.
Algorithm 5.1 Generating a random sample.

Data: $b$: minimum, $t$: maximum, $m$: mean, $d$: density

1. $m_s \leftarrow \frac{m-b}{t-b}$;  // scale to beta’s domain: $[0, 1]$
2. if $m_s < 0.5$ then
3.     $\alpha \leftarrow \frac{d m_s}{1-m_s}$, $\beta \leftarrow d$
4. else
5.     $\alpha \leftarrow d$, $\beta \leftarrow d \frac{1-m_s}{m_s}$
6. end
7. return $\beta(a, \beta) \cdot (t-b) + b$;  // draw and scale

We assumed that some clients might choose a smaller bundle than their maximal potential consumption. Hence, we extrapolated the client’s consumption to what it would have been had it not been limited by its rented cores. To this end, if a sample’s maximum was near the client’s limit (i.e., within 90% of its number of virtual cores), we matched the sample with a beta function with a higher maximal value (64 cores), while maintaining the same average and minimum. That is, we created a modified beta function that allows for some over-the-top samples. Fig. 5.2a shows an example of a generated load from a real client.

Each client gets its realistic load samples and treats them as a load history. It models this history using a cumulative distribution function (CDF) (Fig. 5.2b), and uses it to statistically predict its load for the upcoming day, assuming that “That which hath been is that which shall be”. The client later uses its statistical load prediction to predict its expected revenue and profit from the various bundles.

5.4.3 Valuation

Real (human) clients may choose an offering in any way they like. They may choose randomly, take some time to make a decision, or go through a long iterative process of selection and improvement. In the simulation we needed to create realistic artificial intelligence agents which mimic the behavior of real clients. We did this using the valuation function tool. A valuation function ($\text{perf} \rightarrow \$)$ indicates the monetary value that a client attributes to the stochastic properties of the performance. It is based on business logic, such as the expected revenue from this performance.

The valuation of a client is the sum of two sub-functions; each takes different properties of the performance into account. The immediate valuation function, $V_{imm}$, represents the expected income from the immediate performance, defined in this work as the average performance over a short period of 12 seconds. The long-term valuation function, $V_{lt}$, represents the benefit from the long-term performance, i.e., the average performance over the entire day.

Using these two functions, a client can estimate the expected value of its revenue from a combined bundle of reserved resources and shares. Let us define two random variables: $X_{load}$ denotes the client’s load and $X_s$ denotes the resource availability given
(a) The load over a single day. The local mean shows the average load over a local window, and the heat map shows the density of required performance over that window.

(b) The cumulative distribution of the load over that day. For example, this client’s required performance will be less than 20% for 39% of the time.

Figure 5.2: A client’s generated load. This client’s required performance was 42% on average.

Let \( r \) denote the client’s reserved resources. The actual performance, \( P_{r,s} \), is the minimum value of the load and the performance that the resources allow:

\[
P_{r,s} = \min\{X_{\text{load}}, \text{perf}(r + X_s)\}.
\] (5.1)

The client calculates its expected revenue by adding two valuation functions: \( E(V_{\text{imm}}(P_{r,s})) \), its expected revenue from immediate performance, and \( V_{\text{lt}}(E(P_{r,s})) \), its expected revenue from long-term performance. Accordingly, the client estimates these random variables on the basis of its self-created load CDF and the CDF representing the potential use of a resource attributed to a share, supplied by the provider.

Some researchers modeled valuations using analytic functions (e.g., power law [SKZ+17]), which are easy to symbolically analyze. But real user valuations are sophisticated [LS07]. We tailored to each client an individual, piece-wise linear function, to model actual consumption. Such non-symbolic functions may be harder to manipulate; however, they cover a wider range of functions.

To produce these valuation functions, we first characterized each client using three values: rented number of cores (i.e., its bundle), the expected revenue and the portion of the revenue affiliated with each requirement.

The number of rented cores was obtained from Azure’s data. We used cores as the basic currency for the simulation: one core costs $1 a day. Assuming the client is rational, the cost of the cores the client rents is a lower bound on its valuation of these cores. We modeled the clients’ expected revenue using a Pareto distribution (standard
in economics) with an index of 1.1. A Pareto distribution with this parameter translates to the 80-20 rule: 20% of the population has 80% of the valuation, which is reasonable for income distributions [Sou01].

For each client, we drew a value from this Pareto distribution, with the condition that the value is higher than the client’s number of cores (i.e., a conditional probability distribution).

We used Azure’s strict client classification (interactive, delay insensitive and unknown) to infer for each client the portions of the revenue affiliated with each requirement. To this end, we assigned to each class a different truncated normal distribution function (in the range $[0, 1]$), which describes the division between the requirements, as depicted in Fig. 5.3. For each client, we chose a distribution function according to its class, and drew a sample from it to get the client’s budget portions.

![Figure 5.3: A probability density function (PDF) of the portion between the two valuation types. The horizontal axis describes the long-term requirements portion, and the immediate requirements portion completes it to 1.](image)

Using these three values (bundle, revenue and portion), we produced two monotonically rising functions, one for each valuation type. We engineered these functions such that when used to produce the valuation of each bundle, the client’s bundle yields the maximal profit ($value - cost$), and its expected revenue for this bundle will be the revenue we draw for this client. To this end, we used an iterative process: we assumed that the revenue from zero performance is always zero and thus started with the identity functions $V_{imm}(x) = x$ and $V_{lt}(x) = x$. Then, in each iteration, we

1. estimated the client’s expected value for different bundles using these functions;
2. adjusted each function, such that the value attributed to the client’s selected bundle matched its portion of the revenue; and
3. adjusted the values that the functions attributed to other bundles, such that they were less profitable for the client.

Fig. 5.4 depicts an example of a generated valuation, and a simple example of a choice of fixed resources, i.e., $X_s \equiv 0$. The choice of a combination of reserved resources and shares is of a higher dimension, as the valuation function is $(\text{shares} \times \text{reserved}) \rightarrow \text{money}$. 

86
5.5 Evaluation Methodology

We simulated a cloud with various mechanisms: fixed performance, stochastic allocation, and the most prevalent “burst” mechanism. We compared the provider’s revenue, resource utilization, and client density.

Our evaluation method was an iterative process. A single iteration simulated a day and took 2-3 real minutes to run. It is described in Fig. 5.5. An example of the progress of the full process is given in Fig. 5.6. The initial iteration simulated a cloud that offered only fixed performance (reserved) resources. The subsequent iterations simulated the introduction of another mechanism (e.g., stochastic allocation) alongside the reserved resources. Some clients were free to change their bundle choice at each step. We continued the process until the measurements were steady for at least 128 iterations (Fig. 5.6), and considered only the results of the last 60 iterations. Here we describe each step of our simulations, as depicted in Fig. 5.5.

**Selecting Fixed Performance and/or Share Allocation.** Each client computed, for each possible bundle, the valuation it will draw from it. It used its own load...
Figure 5.6: A typical iterative process and its convergence. The number of clients per server increases over time until it peaks and starts to drop due to the high utilization and the increasing number of shares per server, which reduce the value of a share. The provider’s revenue decreases over time as more clients switch to cheaper bundles (with shares).

statistics and the provider’s statistical description of the resources that every share amount yields. Because the client could not foresee its exact load for the upcoming day, it used the load statistics gathered over the entire recorded period. The client selected the most profitable bundle of fixed performance and/or share allocation for its load and resource requirement distribution. Formally, the client’s decision can be described as:

$$\arg\max_{r,s} \left\{ E(V_{imm}(P_{r,s})) + V_{lt}(E(P_{r,s})) - \text{Cost}_{r,s} \right\},$$

(5.2)

where $r$ is the number of reserved cores, $s$ is the number of shares, and $P_{r,s}$ is defined as in Section 5.4.3.

Changing Choices. Initially, each client chose a number of reserved cores. In each subsequent iteration, 128 out of the 1024 clients in each dataset (12.5%) were allowed to switch their bundle to any offer available in that simulation. This is consistent with the behavior of a real market, in which clients are unlikely to update their bundles all at once. Numerically, the limitation on the number of clients changing bundles simultaneously makes the solution method more stable, reducing oscillation over iterations and enabling the solution to converge.

At this stage, the provider’s revenue was calculated by summing the prices of the clients’ bundles.

Allocating Clients to Machines. To allocate clients to 64-core servers, we randomly shuffled them. Then, one at a time, each client was assigned to the first server that could accommodate the reserved component of its bundle. To ensure an even distribution of shares among the servers, a client that rented only shares was assigned to the server with the least sum of shares at that point.

We then calculated the average number of clients per server by repeating the allocation process and taking the average over the active servers. For a single resource, the assignment algorithm achieves near-optimal allocation, except for the last active server, which may be partially full. Accordingly, to measure the average number of clients per
server, we disregarded the last active server. For a large cloud with more machines and clients, a single last server is negligible. Each dataset contains 1024 clients, so there are always enough full, representative active servers in the simulation.

**Simulating a Cloud.** Each client load for the current day (iteration) was selected cyclically from its data over multiple days. We applied the server allocation algorithm 16 times for each of the 12 datasets and simulated the actual resource allocation of the first server of each dataset each time. Due to our assignment algorithm, the first server is the busiest. This serves as a worst-case analysis as these clients experience the most resource stress, and would thus be more reluctant to reduce their reserve requirements. The server’s resource allocation was simulated using our modified CFS (Section 5.3.1).

At this stage, we collected client and server statistics: clients’ expected revenue (i.e., their valuation), clients’ effective revenue from their effective performance, and the server’s resource utilization distribution.

**Calculating the Statistical Potential of a Share.** To allow the clients to rationally select a bundle of reserved resources and shares, our simulated provider supplies statistical information regarding the shares, which represents their potential: the distribution of the maximal resource amount that a client might attain over a short period—12 seconds in our case—with the commensurate number of shares. To this end, we collected the utilization statistics of the machines and computed their distributions. This produced an effective two-dimensional probability density function ($PDF_{cpu}(t,r)$) for the total unconsumed CPU ($t$), and the CPU that was not exploited by clients that reserved it ($r$). A client can utilize an unused reserved CPU that adheres to its proportional share, or utilize the entire total unconsumed CPU. Thus, the CDF of the probability for a client with portion $p$ of shares to get $x$ of the resource is given by

$$CDF_p(x) = \int_{t=0}^{x} \int_{r=0}^{\frac{x}{p}} PDF_{cpu}(t,r) \cdot dt \cdot dr.$$ (5.3)

To deduce the portion ($p$) of each offered number of shares, the provider uses the average total share allocation per server. It publishes, accordingly, the corresponding distributions for each offered number of shares, in the form of a CDF ($CDF_p$, according to Eq. 5.3). Identical distributions were published to all of the clients, regardless of their actual server allocation. The provider’s utilization and average share allocation statistics remain concealed. Each client, then, uses the CDF of shares in the next bundle choice stage (Fig. 5.7).

5.6 **Compared Mechanisms**

Similarly to public cloud providers, we offer clients a choice of CPU performance units. In our simulations, these units are core portions, i.e., $\frac{1}{16}$, $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, 1, 2, 4, 8, 16, 32 core(s). In addition, the client can rent shares in the amount of $\frac{1}{16}$, $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, 1, 2, 4, 8,
16, 32 shares. A client that rents shares is not obligated to rent reserved resources. We evaluated each of the following mechanisms separately:

**Fixed Performance (FP).** Each initial iteration is a choice among fixed performance offerings. This is our baseline. When evaluating the rest of the mechanisms, FP was always offered to the clients as an alternative.

**Limited Stochastic Allocation (LSA).** In our mechanism, the client can rent shares alongside reserved resources, and utilize them only up to their absolute value. E.g., a client that rented \( \frac{1}{8} \) of a share can utilize up to \( \frac{1}{8} \) of a core in addition to its reserved allocation, even if the machine is underutilized.

**Unlimited Stochastic Allocation (USA).** For completeness, we also tested our SA mechanism in a scenario where the client can rent shares and use them without any capping. Its utilization is limited only by the server’s available resources, and is in proportion to the total number of actively used shares on the machine.

**Burstable Performance (BP).** We compared our mechanism to burstable performance, where instances are modeled assuming the allocated credit rate represents reserved resources. Moreover, the credit system limits the client to a certain average utilization per day; the provider will impose a fine for overutilization. Hence, we assumed rational clients will try to avoid exceeding the bundle’s average allocation. To adhere to the strict coupling of the burstable instance offerings, we only let clients rent bundles in which the number of reserved resources equaled the number of shares.

To select the most profitable bundle, a client using BP has to predict, for each bundle, the limit that will prevent it from exceeding the bundle’s average (i.e., its reserved allocation). To do this, the client takes into account its potential load and the statistical potential attributed to a share [WUNK17]. Once a bundle is selected, the client will not exceed its predicted limit so as to not incur penalties. However, overutilization was not fined by the provider.

According to our regression analysis (Section 5.2.3), a burstable instance is limited to an average performance of 10%–30% of the fixed instance maximal performance. Consequently, the cost of a bundle of matching reserved resources and shares should be 3.3–10 times the cost of renting only reserved resources. Given the fact that reserved...
resources cost $1 per core unit, the corresponding share should bear the rest of the cost—that is, a share cost of $2.3–$9 per share unit.

We tested BP using share unit prices of $2, $3 and $4. For LSA, we used share prices of $0.15, $0.5, $0.6, $0.7 and $0.9 per share unit. For USA, we used higher share prices of $3, $5 and $8, as they allow clients to use more of the resource and thus are more valuable to them.

**LSA or BP (LSA/BP).** We also tested the case where clients had a choice between these two mechanisms: LSA with a unit price of $0.5 or USA with a unit price of $3.

### 5.7 Results

First, we review our raw results by comparing the increase of clients per server while maintaining revenue. Then, using our simulation results, we make some assumptions to infer server costs, and use them to compare the provider’s profit from the various mechanisms.

As Fig. 5.8 shows, LSA .5 (Limited SA with a unit price of $0.5) can pack 1.7 times more clients into each server than BP 3 with the same revenue, and 5.7 times more than FP. USA has a similar number of clients per server (CPS) as LSA—albeit with significantly lower revenues.

LSA .15 allowed 233 CPS—the most among our tested cases, but reduced the provider’s revenue significantly (35%) compared with BP 3. LSA .5 matched the provider’s revenue when using BP 3, and allowed nearly as many CPS as LSA .15. LSA/BP allowed nearly as much CPS as LSA .5, however with lower revenues.

#### 5.7.1 Provider’s Economic Benefit

The public provider’s profit depends on its expenses on hardware, energy and infrastructure purchasing: data we do not have. This data can be used to estimate the daily server costs and derive the profit from the revenue. Although revenues for non-FP schemes are reduced, the providers’ profit may still grow due to lower daily server costs, which increase the profit margins—the profit divided by the revenue.

We estimated the server cost on the basis of our simulation results, and the assumption that the first provider to offer BP (Amazon) chose to offer this scheme to increase its profits. Amazon’s BP pricing matches BP 3. Hence, we assume that the FP profits are lower than those of BP 3. The profits are equal when the profit margin of FP is 38%, which implies a daily cost of $39 \(^4\) (in reserved core price units), assuming no new clients joined due to the new attractive track. A lower profit margin of 25% implies a server cost of $47, and a higher profit margin of 50% implies a server cost of $31.

In Fig. 5.9, we demonstrate the provider’s profits in the different tracks, using these three hypothesized values for server costs. If a server costs $39—the break-even case for

\(^4\) The daily server cost is calculated as follows: \(\frac{((1-\text{profit margin}) \cdot \text{profit})}{\text{number of servers}}\).
FP and BP—the provider can increase its profit by over 35% and its profit margins by at least 22% by offering LSA .6 instances instead of BP 3. The higher server cost ($47) leads to a 44% increase in profit. Moreover, in all of these cases, the profit margins grow when any stochastic allocation instances (excluding USA 3) are offered.

A private cloud provider is interested in maximizing the aggregated value of all its clients (social welfare). Fig. 5.10 shows that the BP track nearly maximized the social welfare (over 99% of the maximal social welfare, achieved when resources are abundant). LSA achieved over 97% of the maximal social welfare. It did so with fewer machines, meaning it produced 55% more value per machine than did BP. USA achieved a higher social welfare than LSA because it achieved higher resource utilization and thus created more value.

5.7.2 Server Utilization

The more expensive the stochastic allocation is, the lower the CPS will be (Fig. 5.8) and the lower the average utilization will be, as seen in Fig. 5.11. This is because fewer
clients prefer it over a reserved allocation. The reserved allocations, which grow larger, are generally less utilized, as also seen in Fig. 5.11.

For BP, the mean and median reserved utilization were higher than for LSA and USA, but the average total utilization was lower. This is because BP forces the clients to rent a reserved resource in order to rent a proportionate share, although they might want a smaller amount of reserved resources. This proves our claim: decoupling the reserved allocation from the average allocation will be preferred by clients, and will also yield higher server consolidation.

### 5.7.3 Clients’ Preferences

When offered a choice of FP and a flexible mechanism, how many would go for the flexible one? 92%–99% of the clients preferred LSA to FP (Fig. 5.12c), and 77%–96% preferred USA to FP (Fig. 5.12b). Only 65%–84% chose BP over FP (Fig. 5.12a). When offered BP 3 or LSA .5, 56% preferred LSA and 42% preferred BP. This indicates SA is more attractive to most clients than BP. Its flexibility enables it to cater to a wider set of client needs. Moreover, when offered any kind of stochastic allocation (either USA or LSA), 34%-46% of the clients avoided reserving resources at all.

### 5.7.4 Clients’ Attainment Ratio

Every 12 seconds, we calculated each client’s attained CPU utilization divided by its required CPU utilization—i.e., the client’s attainment ratio. In all the tested cases
(FP, BP, SA and USA), the average attainment ratio for all the clients was over 99.8%. This indicates that the clients were able to satisfy most of their load requirements.

5.7.5 Validation

To validate our simulation, we compared the distribution of the selected bundles in our FP simulation to the distribution in the entire Azure dataset (2,013,767 clients). Our FP response profile distribution matches Azure’s, with 10% less overprovisioning (Fig. 5.13). This is consistent with real clients being more risk averse than rational, simulated clients. This indicates that most of our simulated realistic clients have the same utility function distribution as real clients.
For further validation, we compared the utilization distribution in our FP simulation to real cloud data. Measurements taken before the introduction of burstable instances indicated average CPU utilization of 15%–20% [Liu12, AFG+10], which is consistent with the FP’s mean utilization in our simulation results, shown in Fig. 5.11.

We also confirmed that the clients in our simulations act rationally, that their load expectations are realistic, and that the published shares’ potential is accurate. To do this, we compared the clients’ expected value (before the server simulation) to their effective value (their actual revenues from the simulated performance). The average effective value tended to be slightly lower than the expected value, as seen in Fig. 5.14. The high variance is expected as the clients use statistics from up to a month to predict the load of a single day.

We validated that the iterations we chose—the last 60—indeed converged. Over the
Figure 5.14: The distribution of clients’ revenue normalized by their expected value in each tested case.

course of the last 60 iterations, up to 12% of the clients changed their selected bundle from the first iteration to the last, in all the tested cases. Moreover, the standard deviation of the selected bundles’ distribution over these iterations was under 0.6% and the standard deviation of the shares CDF was under 0.01%, in all the tested cases.

Finally, we analyzed the effect the different assumptions would have on the results. When we modified the \textbf{beta density} to be 0.5, 10 or 50, CPS was increased by up to 6% for SA, on the one hand, and reduced by up to 5% for BP, on the other, compared with the main value (1). When we avoided the \textbf{over-the-top} extrapolation of the generated load values, CPS was reduced by up to 7%. When we modified the \textbf{performance functions} so they were linear and concave, CPS was reduced by up to 3% compared with monotonically increasing ones. When we modified the \textbf{Pareto index}, CPS was reduced by up to 6% for a Pareto index of 0.8 and increased by up to 1% for an index of 1.3, compared with the main index (1.1). We also modified the number of clients that can \textbf{change their bundle}. The average CPS was not affected when 384 (or less) clients changed their bundle at once. When more than 256 clients changed their bundle, however, the results fluctuated. When more than 384 clients changed their bundle, the results failed to converge.

In all of the simulations, we compared the CPS ratio of LSA over BP. Throughout the above-mentioned modifications, this ratio turned out higher than in the main results presented earlier (Fig. 5.8). This indicates that the main results are numerically sound.

\section{5.8 Related Work}

Agmon Ben-Yehuda et al. [ABYBST14, ABYBST12] predicted that cloud providers will reduce the resource allocation intervals, as they currently do. They also predicted
the need for sophisticated economic mechanisms to efficiently allocate resources in the cloud. This work and the other mechanisms mentioned here follow this principle.

Many researchers have suggested ways to improve server consolidation and social welfare other than those used in the industry. Dynamic pricing schemes have been proposed to regulate demand \cite{ZJWT15, Kel97} or reduce interference \cite{ISBA12}. Shahrad et al. \cite{SKZ+17} also suggested incentivizing clients to limit their burstiness via an incentive compatible pricing scheme, in which clients profit from limiting themselves.

Other researchers suggested allowing clients to communicate information to the provider (the desired availability \cite{SW16, SRI16}, long-term (months) required service level objectives \cite{CCBW14} or short term requirements \cite{HGS+11}). This approach places the burden of placement and scheduling on the provider’s allocator, which must ensure that the client’s requirements are met with high probability. That is, it must solve an optimization problem. Our solution is simpler for the provider, who only needs to publish its statistics, and leaves the burden of making an informed choice to clients. Other researchers collected statistics about clients to improve utilization \cite{BGO+16, VPK+15, MP10, RSG17}, efficiency \cite{TGS14, TKU13}, consolidation \cite{WJLW11} or energy consumption \cite{LZDC14} via placement algorithms or resources reallocation.

Many solutions have been proposed for improving the utilization of dedicated large-scale clusters given a job scheduler \cite{VPK+15, Isa07, Apa18b, Apa18c, VMD+13, GAK+15, Apa18a, IPC+09, BEL+14, ZLT+14, SKAEMW13, Kub18, DHGR15, NSG+13, YBMT13, LK14, LCG+15, SL10, JMNY15, DK14, ABYSS+12, SSGS09}. Such schedulers are more flexible, and thus reach higher utilization than public clouds. They can do so because the data center provider is often the one that deploys this system and can control the fine-tuning of each task. Real clients’ utility functions were characterized on such systems \cite{LS07, Wil09}, but the characterization is inapplicable to public clouds or private clouds without a centralized scheduler.

5.9 Conclusions and Future Work

Stochastic CPU allocation via shares allows clients to reduce their reserved resource requirements. This allows the provider to increase the number of clients per server by more than 70% compared with burstable performance. As such, our method can increase the profits of the public cloud provider by over 28% compared with burstable instances. Furthermore, our method also benefits private cloud providers as it increases the client social welfare per server. It increases the value each server generates for the corporation by over 55%.

The private cloud’s social welfare can be improved further by allowing clients to bid for shares, just as clients bid for spot instances in horizontal scaling \cite{ABYBYST13, Ama18b, Mic18a, Goo18b}. Formulating a bidding and valuation language for stochastic allocation remains as future work. The provider might also charge the clients an
additional, fixed, price-per-use to discourage resource waste. Analyzing this is left for future work as well.

Our simulations show that almost all clients will prefer using our mechanisms versus the fixed performance track, and over 56% will prefer it over burstable performance. Our SA mechanisms offer a cheaper track than either fixed performance or burstable, and do not mandate reserving any resources. Hence, new clients, who were unable to afford other cloud services, might now join the cloud and further increase the provider’s revenues, without cannibalizing the market share of the existing offerings.

Our simulation infrastructure was validated using real data from a real cloud. Our methodology and the clients’ rationale were validated by the accuracy of the clients’ expectations, despite the reduced and compact data at their disposal. We hope that our infrastructure, published as an open source project, will allow more research on the applicability of novel allocation methods.

To implement our stochastic allocation method in a real cloud, our modified CFS should be implemented in the Linux kernel. It can also be implemented on other resources that can be allocated via a proportional share mechanism [Wal95]. Investigating the coexistence of this mechanism over multiple resources remains as future work as well.
Chapter 6

Conclusion

In this work we developed new market-driven resource allocation schemes. We showed how they can improve financial properties such as social welfare and profits, as well as technical properties such as resource utilization and the number of clients per server. We developed algorithms to support these mechanisms, designed rigorous evaluation methodologies, and implemented them as open source frameworks.

We believe that more research on resource allocation mechanisms in the cloud would help cloud providers immensely. To spur such efforts, more detailed data—such as the Azure Public Dataset [CBM+17]—should be published.

Our work has demonstrated how sophisticated allocation and pricing mechanisms can improve hardware—and thus energy—efficiency in the cloud significantly. Such improvements, however, can only take us so far. Even with the proliferation of new allocation mechanisms, resources are still being over-provisioned by clients and providers.

Clients suffer from nondeterministic performance due to shared, nonpartitioned resources. To protect themselves, they over-provision resources. To overcome this unnecessary holding of resources, allocation mechanisms need more control over hardware partitioning. For this to happen, vendors must design hardware that allows more resources to be partitioned, and by that reduce interference between clients and better meet clients’ needs. Notably missing is bus bandwidth control that will enable allocation of memory, disk and device bandwidth, which will greatly reduce interference.

Providers, on the other hand, over-provision to supply the required demand when needed, and waste energy on underutilized machines. Our allocation mechanisms may increase utilization in a single machine, which allows providers to put more clients on a single server and lower the power state of unused servers. Providers, however, refrain from lowering their servers’ power state to protect against unpredictable performance [TCB+19]. Reasons for unpredictable performance might be the high latency of the power state transitions [ILA+11], and because servers store shared data, which will have slower access time\(^1\). These barriers can be reduced by adapting hardware that is

---

\(^1\)This information was introduced to us by one of the knowledgeable reviewers of one of our papers.
designed for faster power state transition, and by decoupling storage from computation servers [Lig18].

Due to the above, we believe that allocation mechanisms can achieve their full potential only if they are developed and offered alongside hardware that is designed specifically to exploit these mechanisms.
Appendix A

Software

During the course of our research, we developed many pieces of useful code, ranging from full blown frameworks to small useful libraries and packages. We released all this code as open source under the GNU general public license v3.0 (GPL). Below we list the software that was directly related to the work in this thesis.

A.1 Cache Set Finder (AKA Plumber)


For the purpose of the work in Chapter 2, we built a tool that classifies memory addresses—in granularity of a cache line size—according to the cache set to which they are mapped. It is based on the work by Hund et al. [HWH13] and Yarom et al. [YGL+15].

Using this classification and information about how many cache sets the hardware has, this tool can verify that its data fits in its allocated cache perfectly without collisions. Then, the tool can repeatedly touch all its data, in parallel, to keep its cache lines marked as most-recently-used (MRU). We originally called it ‘Plumber’ because it is meant to test and deal with a possible cache leakage issue that was raised by the community. We later named it, formally, Cache Set Finder.

A.2 VCG Algorithm Implementation


For the purpose of the work in Chapter 3, we implemented our algorithm and Maillé and Tuffin’s [MT04] algorithm in C++ and Python. To do this, we used an auxiliary package we developed—called vecfunc—which can be used to represent continuous multidimensional functions as a discrete multidimensional vectors. It is also available as an open source from: https://github.com/liran-funaro/vecfunc.
A.3 Elastic Java-Virtual-Machine (JVM)


For the purpose of the work in Chapter 4, we supervised a student project by Yi Ren that modified OpenJDK JVM to allow the heap size to be changed during runtime.

OpenJDK and other java-virtual-machines (JVM) allow the maximum heap size to be changed only at boot time, using the command line. During runtime, a JVM resizes its heap automatically for the purpose of performance optimization, and the heap size limit is an upper bound on this optimization. Our modifications allow this upper bound limit to be changed during runtime, on demand. This means fewer garbage-collection (GC) cycles, when more memory is available. We achieved this with a relatively small modification to the OpenJDK garbage collection source code (hotspot).

Salomie et al. [SARE13] implemented a kernel module that supports ballooning pages right out of the application’s memory to the host—i.e., application level ballooning—and demonstrated it on a java-virtual-machine (JVM). Such a solution may be helpful for transferring memory faster, but it depends on a specific operating system and requires the installation of a kernel module. This approach increases the coupling between software layers, which complicates the adoption of the application.

A.4 Stochastic Allocation Simulator (SAS)


For the purpose of the work in Chapter 5, we developed a validated infrastructure for cloud simulations in Python and C++. We hope that this infrastructure will allow more research on the applicability of novel allocation methods.

This infrastructure can generate a large dataset of realistic clients with different behaviors and simulate their rational bundle selection, given a known distribution of available resources. The realistic modeling is based on real data from the Azure Public Dataset [CBM+17], which was used to deduce real consumption and market demand.

Our infrastructure can simulate the load on a server with these clients and yields highly detailed statistical information. It uses our modified completely fair scheduler (CFS) to allocate resources, but any other resource allocation mechanism can be used instead. SAS is published as an open-source project along with the code to replicate our simulations and their data.
Bibliography


מייצגות הקצאת משאבים באפור סטוטיסטית. הספק מפרס鲛יר Injury נבעת עדות לתחום הקצאת סטוטיסטית. כל קוקו Размер לכל יחידה קבלות של משאבים משוריינת/אסר סטוטיסטית אוסה או מועניין לעת ולעת הוא מפרס סטטיסטיקות לגב גובה במסחרבים לכל כמות של מדיה הקצאת סטוטיסטית. כל קוקו Размер לברון של שילוב עם משאבים משוריינת/אסר סטוטיסטית אוסה או מועניין לעת.

לאחר שהמוצר תופר מפורים עד מניין הקצאת סטוטיסטית של מעבר של מספים הלוקהות לשרט אנוי מוצאים תופור אפורים עד מניין הקצאת סטוטיסטית של מעבר של מספים הלוקהות לשרט או מוקסימים: בניה פי 5.6 לועות מכררח משאריות משוריינת בלבד, פי 1.7 לועות משאריות בחומש בתפר. שאחר מתפר帱 מתפר ธימת הקצאת הדינמית הרוחית יוצרת דע. אנוי מומזג מאריס כי באמצוות המונגין שלג, רוחי הספק משמטרים משמעויות בכת-28% לעבר לועות משאריות hombre מתפר Choi, ונכ הרוחית fark צהיה מעלי 98% מהאפורים.
תקציר

אחת האתגרים העיקרים של ספקיענןتمثل בניקוז משאבים. מנגנון הקצאת משאבים של הספק הוא התוכנן כך שיאפשר שירות שכולל כל הקהל בכל שירת באתא לתכוס את ה菲律 השונות של שירות (SLA).єחנק של יוזל המשאבים בשרת הספקים. דבר זה שיפיע על מספר הקהלות לשמש יוזל לכל חלוקה וכל התוכן של יוזל של הקהלות. ממא יוזל灯笼 את SLA של הקהלות. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ께 יוזל לשמש יוזל השיתוף לשמש יוזל. על יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ומ治理体系 משאבים יוזל של יוזל יוזל את SLA ו---


Danielle Movsowitz, Liran Funaro, Shunit Agmon, Orna Agmon Ben-Yehuda, and Orr Dunkelman. Why are repeated auctions in raas clouds risky? In *Proceedings of the 15th International Conference on Economics of Grids, Clouds, Systems, and Services (GECON ’18)*. Springer, 2018. *This paper is intended to be included in its first author’s thesis. My contributions for this paper are insights accompanied with client behavior simulations under resource deny attacks.*
חקלאת רב משאבمونעת מסחר ושוק

היבר על מחקר

לשם مليולי חלקי של הדרישות לכבכלת התואר
דוקטור לפילוסופיה

לירן פוגר

הוג_shutdown הטכניון — מכון טכנולוגי לישראל
סבת החשיפה חיפה
ינואר 2020
הказанת רב משאת מונע מסהר ושוך

לירן פנורה