Efficient, Non-Cooperative Sharing of Computing Resources

Orna Agmon Ben-Yehuda

Efficient, Non-Cooperative Sharing of Computing Resources

Research Thesis

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

Orna Agmon Ben-Yehuda

Submitted to the Senate of
the Technion — Israel Institute of Technology
Tamuz 5773 Haifa June 2013

The research thesis was done under the supervision of Prof. Assaf Schuster in the Computer Science Department.

The generous financial support of the Technion and the Hasso-Plattner Institute is gratefully acknowledged.

Publication List

(J-1) Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. *"Deconstructing Amazon EC2 Spot Instance Pricing."* ACM Transactions on Economics and Computation (TEAC). Accepted. ACM TEAC's impact factor is still undefined.

I did the research as well as most of the writing. The other authors helped with the structuring of the paper and the writing, as well as with obtaining data from IBM Research.

(C-1) Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. *"Deconstructing Amazon EC2 Spot Instance Pricing."* In Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom) 2011. Acceptance ratio: 24%. I did the research as well as most of the writing The other authors helped with the structuring of the paper and the writing.

(C-2) Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Alexandru Iosup, Assaf Schuster, Mark Silberstein, Artyom Sharov, and Dan Tsafrir. *"ExPERT: Pareto-Efficient Task Replication on Grids and a Cloud."* In Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2012. Acceptance ratio: 21%. I did the writing and the theoretical analysis, designed and wrote the ExPERT code, ran the simulated experiments and analyzed both the simulated experiments and the real ones. Prof. Iosup acted as a co-adviser. Mr. Sharov conducted the real experiments. Dr. Silberstein helped with the real experiments' system setup, and suggested the initial research problem.

(C-3) Orna Agmon Ben-Yehuda, Muli Ben Yehuda, Assaf Schuster, Dan Tsafrir. *"The Resource-as-a-Service (RaaS) Cloud."* In Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), June 2012. Acceptance ratio: 32%. The paper was solicited for publication in the Communications of the ACM (CACM). CACM had an ISI impact factor of 1.92 for 2011, and it was rated 2/18 in the field of software engineering. I did the research and most of the writing. This paper represents a vision that Mr. Ben-Yehuda and I share.

Contents

Abstract **1**

Abbreviations and Notations **2**

1 Introduction **8**

2 Related Work **11**

2.1 Pareto-Efficient Task Replication on Grids and a Cloud . . . 11

2.2 Amazon EC2 Spot Instances 13

2.3 Ginseng and Resource-as-a-Service 15

3 ExPERT: Pareto-Efficient Task Replication on Grids and a Cloud **19**

3.1 Abstract 19

3.2 Introduction 20

3.3 The Basic System Model 22

3.3.1 Terminology 22

3.3.2 Model and Assumptions 24

3.4 The Scheduling Strategy Space 25

3.5 The ExPERT Framework 27

3.6 The Experimental Setup 35

3.7 The Experimental Results 38

3.8 Conclusion 45

4 Deconstructing Amazon EC2 Spot Instance Pricing **46**

4.1 abstract 46

4.2 Introduction 46

4.3 Pricing Cloud Instances 48
4.4 Methodology 49
4.5 Evidence for Artificial Pricing Intervention 50
4.5.1 Market-Driven Auctions 50
4.5.2 Evidence: Availability as a Function of Price 50
4.5.3 Dynamic Random Reserve Price 54
4.6 Pricing Epochs 58
4.7 Spot Price Simulation 59
4.7.1 Simulator Event-Driven Loop 60
4.7.2 Workload Modeling 60
4.7.3 Customer Bid Modeling 61
4.7.4 Price Change Timing 62
4.7.5 Simulation Results 63
4.8 Dynamic Reserve Price Benefits 64
4.9 Reexamination of Prior Work 65
4.10 Conclusions 66
4.11 Epilogue 67

5 The Resource-as-a-Service (RaaS) Cloud 78
5.1 abstract 78
5.2 Introduction 79
5.3 Recent IaaS Trends 80
5.3.1 Duration of Rent and Pricing 80
5.3.2 Resource Granularity 81
5.3.3 Market-Driven Resource Pricing 82
5.3.4 Tiered Service Levels 84
5.4 Economic Dynamics 85
5.4.1 Forces Acting on Clients 85
5.4.2 Forces Acting on Providers 86
5.4.3 Implications of Increased Competition 87
5.5 The RaaS Cloud 88
5.5.1 Trading in Fine-Grained Resources 88
5.5.2 Prioritized Service Levels 90
5.6 Implications, Challenges, Opportunities 92

6 Ginseng: Market Driven Memory Allocation (Memory-as-a-Service) 96

6.1 Abstract . . . 96
6.2 Introduction . . . 97
6.3 System Architecture . . . 99
6.4 Memory Auctions . . . 99
6.4.1 Non-concave Valuation Functions . . . 101
6.4.2 Memory Waste . . . 103
6.5 Repeated Auction Protocol . . . 104
6.6 The Auction . . . 106
6.6.1 Allocation Rule . . . 107
6.6.2 Payment Rule . . . 107
6.6.3 Complexity . . . 107
6.6.4 Example of a Single Round . . . 108
6.7 Guest Strategy . . . 109
6.7.1 Choosing the bid price p . . . 109
6.7.2 Choosing q^m . . . 110
6.7.3 Evaluating Guest Utility . . . 111
6.8 Experimental Setup . . . 111
6.9 Performance Evaluation . . . 115
6.9.1 Comparing Social Welfare . . . 115
6.9.2 Reclaim Factor Analysis . . . 117
6.9.3 Simulated Experiments . . . 119
6.9.4 Impact of Off-Line Profiling . . . 121
6.10 Conclusions . . . 121

7 RaaS: Additional Research Directions 127

7.1 Single Resource . . . 127
7.1.1 Game-Theoretic Challenges . . . 127
7.1.2 Guest Logic . . . 128
7.1.3 Host Logic . . . 128
7.1.4 Provider Logic: Global Cloud View . . . 129
7.1.5 Minimal Price . . . 129
7.1.6 Memory Shedding . . . 129
7.2 Multi-Resource Allocation . . . 130
7.2.1 Bidding Language . . . 131

7.2.2 Allocation Rule 132
7.3 Side-Channel Attacks 134
7.3.1 Information Leakage 136
7.3.2 Disruptive Activity 138
7.3.3 Prevention of Disruptive Activity 138

8 Conclusion 140

Appendix:Software 141

Abstract in Hebrew א

List of Figures

3.1 Remaining tasks over time 23
3.2 A Pareto frontier 23
3.3 $NTDM_r$ task instance flow 26
3.4 ExPERT stages 27
3.5 CDF of single result turnaround time 32
3.6 Pareto frontier and sampled strategies 33
3.7 Pareto frontier and examples of best points 34
3.8 Static strategies compared 39
3.9 Pareto frontiers obtained for various M_r values 43
3.10 Reliable pool use by efficient strategies. 44

4.1 Availability of Windows-running spot instance types 51
4.2 Availability of Linux-running spot instance types 52
4.3 Availability of Windows types as a function of normalized price 53
4.4 Standard deviation of white noise 56
4.5 Periodogram 56
4.6 Autocorrelation of mean daily values 68
4.7 Time epochs of us-east.windows.m1.small 69
4.8 CDF of runtime on various systems 70
4.9 Inter-arrival time autocorrelation 71
4.10 CDF of time interval between price changes 71
4.11 Simulation results for various bidding models, with constant and $AR(1)$ reserve price, on the basis of a grid trace (LPC-EGEE) 72
4.12 Simulation results for various bidding models, with constant and $AR(1)$ reserve price, on the basis of a trace of cloud 1 . . 73

4.13 Simulation results for various bidding models, with constant and $AR(1)$ reserve price, on the basis of a trace of cloud 2 . . 74
4.14 Simulation results for various bidding models, with constant and $AR(1)$ reserve price, on the basis of a trace of cloud 3 . . 75
4.15 Availability in the second epoch 76
4.16 Running time truncation 76
4.17 The history of this paper and a price trace 77

5.1 Correlated cloud price reduction dates for three major cloud providers during 2012 86

6.1 Ginseng system architecture 100
6.2 Application performance as a function of guest physical memory 102
6.3 Strategies for choice of unit price for two maximal quantities 110
6.4 Valuation functions for different loads 117
6.5 Social welfare comparison 118
6.6 Performance comparison 123
6.7 Impact of reclaim factor on social welfare for a mixed workload 124
6.8 Two mixed-workload experiment traces of utility and memory allocation 124
6.9 Simulation results of Ginseng performance 125
6.10 Comparison of predicted performance with measured performance 126

7.1 Indivisible multi-resource allocation 134
7.2 Forbidden bounding box (bandwidth first) 135
7.3 Forbidden bounding box (memory first) 135
7.4 Indivisible multi-resource allocation branching 136

List of Tables

3.1 User-defined parameters 28
3.2 Values for user-defined parameters 35
3.3 Workloads 36
3.4 Real resource pools used in our experiments 37
3.5 Experimental parameters 41
3.6 Experimental results 42

6.1 Guest configuration 114

Abstract

The defining characteristic of cloud computing platforms is money. In clouds, non-cooperative clients pay their providers for the shared computing resources they use as they use them. The introduction of monetary compensation thus gives rise to a host of new possibilities for efficiently sharing computing resources. We investigate the economic foundations of cloud computing systems and propose new mechanisms for non-cooperative clients and providers to share cloud resources efficiently. We (1) demonstrate how clients can co-optimize both the run-time and costs of their workloads by running them on the right combination of cloud and grid resources; (2) analyze how the leading cloud provider, Amazon EC2, prices its spare capacity (”spot instances”) and show that contrary to popular belief, spot instance prices, supposedly based on supply and demand, were actually artificially generated by Amazon; (3) propose the Resource-as-a-Service (RaaS) economic model of cloud computing, where clients pay the right price for the resources they need as they need them; and (4) present a prototype RaaS cloud computing platform that efficiently rents physical memory to non-cooperative clients at a fine-grained time and resource granularity.

Abbreviations and Notations

Chapter-local symbols, which might be used differently in different chapters, are marked with $\sim$.

	Introduction and Related Work	
BoT	—	Bag of Tasks, a group of tasks that needs to be completed in full
IaaS	—	Infrastructure-as-a-Service, a cloud model in which full virtual machines are rented
EC2	—	Elastic Compute Cloud, Amazon's IaaS offering
VMM	—	Virtual Machine Manager, also referred to as host or hypervisor
VM	—	Virtual Machine, also referred to as guest
MB	—	MegaByte
CPU	—	Central Processing Unit
RaaS	—	Resource-as-a-Service, a cloud model in which resources are rented to virtual machines
Ginseng	—	Our RaaS prototype: a memory allocation framework
ExPERT	—	Our grids and cloud recommendation system
m1.small	—	An EC2 instance type
ap-southeast	—	An EC2 region
SLA	—	Service Level Agreement
I/O	—	Input Output
PSP	—	Progressive Second Price, a divisible good auction
MPSP	—	Memory Progressive Second Price, a divisible memory auction
GSP	—	Generalized Second Price, Google's auction
VCG	—	Vickrey-Clarke-Groves, the original second price auction [37, 55, 138]
	ExPERT (Chapter 3)	

T_{tail}	—	The tail phase start time
GridBoT	—	A user scheduling system for grid and cloud BoTs
BOINC	—	Berkeley Open Infrastructure for Network Computing
T_{ur}	—	Mean CPU time of a successful task instance on an unreliable machine [second]
$NTDM_r$	—	Our replication strategy model, which is composed of the parameters N, T, D, and M_r
N	$\sim$	The maximal number of instances sent for each task to the unreliable system since the start of the tail phase.
D	$\sim$	A deadline for an instance, measured from its submission to the system [second]
T	$\sim$	A timeout, the minimal waiting time before submitting another instance of the same task. [second]
M_r	$\sim$	The ratio of the effective sizes of reliable and unreliable pools
T_r	$\sim$	Task CPU time on a reliable machine [second]
C_{ur}	$\sim$	Cents-per-second cost of unreliable machine
C_r	$\sim$	Cents-per-second cost of reliable machine
M_r^{max}	$\sim$	Maximal ratio of reliable machines to unreliable machines
CDF	—	Cumulative Distribution Function
$F(\cdot)$	$\sim$	The CDF of result turnaround time
$\sharp ur$	$\sim$	The effective size of the unreliable pool [number of nodes]
t	$\sim$	Instance turnaround time [second]
t'	$\sim$	Instance sending time [second]
$F_s(t)$	$\sim$	The CDF of successful task instances
$\gamma(t')$	$\sim$	The unreliable pool's reliability at time t'
$F_{s_2}(t)$	$\sim$	The CDF of successful instances during the second epoch
$F_{s_1}(t)$	$\sim$	The CDF of successful instances during the first epoch
$\hat{F}(t,t')$	$\sim$	$F(t,t')$ as was computed for instances sent at time t'
FCFS	—	First Come First Served
x	$\sim$	A random variable
WL	—	Workload
WM	—	University of Wisconsin Madison
ACP	—	AMD's Average CPU Power metric
Tech	—	Technion
OSG	—	Open Science Grid

W	—	Watt
AR	∼	All to Reliable (a user strategy)
TR	∼	all Tail to Reliable (a user strategy)
AUR	∼	All to UnReliable (a user strategy)
$B = 7.5$	∼	Budget of \$7.5 for a BoT of 150 tasks (a user strategy)
CN∞	∼	Combine resources, no replication (a user strategy)
CN1T0	∼	Combine resources, replicate at tail with $N = 1$, $T = 0$
TMS	∼	Tail phase MakeSpan
C	∼	Cost per task
Δ	∼	Deviation of simulated values from real ones [same units as values]
RI	∼	The number of task instances sent to the reliable pool.
Avg.	—	Average
		Spot Instance Pricing (Chapter 4)
D	∼	Declared bid price
H	∼	A spot price trace file, a history
T_b	∼	the Beginning of a time interval within a history [hour]
T_b	∼	The End of a time interval within a history [hour]
$T^H_{b \to e}$	∼	The time between T_b and T_e [hour]
N	∼	The number of goods sold in an auction
F	∼	Floor price [\$/hour]
C	∼	Ceiling price[\$/hour]
Δ	∼	price changes in a price trace [\$/hour]
AR(1)	—	An auto-regressive process of the first order
a_1	∼	A coefficient of the AR(1) process
epsilon	∼	White noise [\$/hour]
sigma	∼	White noise standard deviation [\$/hour]
P_i	∼	A price in the list, whose index is i[\$/hour]
x	∼	band width [\$/hour]
y	∼	Matched white noise of AR(1) process [\$][\$/hour]
PSD	—	Power Spectral Density [dB/rad/sample]
LPC-EGEE	—	LHC Physics Center-Enabling Grids for E-sciencE
RC2	—	Research Compute Cloud
LANL-CM5	—	Los Alamos National Laboratory Connection Machine 5
SDSC-Paragon	—	San-Diego Supercomputer Center Paragon
$\mathcal{N}$	—	Normal distribution

The Resource-as-a-Service (RaaS) Cloud (Chapter 5)		
SaaS	—	Software-as-a-Service
PaaS	—	Platform-as-a-Service
X	∼	A minimal unavailability period that is considered a breach of contract
Y	∼	A minimal fraction of the service period that is considered a breach of contract
Z	∼	A service period
HP	—	Hewlett-Packard
S3	—	Amazon Simple Storage Service
API	—	application programming interfaces
QoS	—	Quality of Service
GB	—	GigaByte
RAM	—	Random Access Memory
GPGPU	—	General-Purpose Graphics Processing Unit
FPGA	—	Field-Programmable Gate Array
SR-IOV	—	Single-Root Input/Output (I/O) Virtualization
Gbps	—	Giga bit per second

Ginseng: Market Driven Memory Allocation (Memory-as-a-Service) (Chapter 6)		
KVM	—	Kernel Virtual Machine
TCP/IP	—	Transmission Control Protocol / Internet Protocol
$V(mem, load)$	∼	The memory valuation function (for a given load)
$V(mem)$	∼	The memory valuation function (for the load the guest is currently experiencing)
$perf(mem, load)$	—	The performance the guest can achieve given certain load and memory quantity. Measured as performance rate, e.g., [Khit/second]
$V_p(perf)$	∼	The guest's owner's (i.e., the client's) valuation of performance function [$/second]
p	—	Bid unit-price ([$/second/MB])
q	—	Bid maximal required quantity [MB]
i	∼	Guest index
$bare_i$	∼	The bare minimal physical memory a guest requires to operate
α	—	Reclaim factor
t	∼	Auction round index

$base_i$	$\sim$	Base memory for guest i, a round's reference point [MB]
$final_i(t)$	—	The total memory allocated to guest i in round t [MB]
m_i	$\sim$	The number of desired ranges in guest i's bid
$[r_i^j, q_i^j]$	$\sim$	Desired range j in guest i's bid $[MB^2]$
q_i'	$\sim$	Memory quantity won by guest i [MB]
p_i'	—	Unit-price paid by guest i[\$/second/MB]
SW	—	Social welfare[\$/second]
N	$\sim$	The number of guests
g	$\sim$	A guest
R	$\sim$	A forbidden range of guest g $[MB^2]$
s_i	$\sim$	The strategy (bid) used by guest i
s_{-i}	$\sim$	The strategies (bids) used by the rest of the guests, excluding i
M	$\sim$	The total number of forbidden ranges in all the guests' bids
U_{est}	$\sim$	Guest utility, as estimated by it[\$/second]
p_{min}	$\sim$	The lowest price the guest can offer and still have a chance of getting any memory at all[\$/second/MB]
q_{est}	$\sim$	The memory amount that the guest will get, as estimated by it[MB]
p_{est}	$\sim$	The unit-price that the guest will get, as estimated by it[\$/second/MB]
Y	$\sim$	The known part of the cost
LLC	—	Last Level Cache
EIST	—	Enhanced Intel SpeedStep®Technology
NUMA	—	Non-Uniform Memory Access
C-STATE	—	CPU state, a power mode
BIOS	—	Basic Input/Output System
KSM	$\sim$	Kernel Samepage Merging
T_{memory}	$\sim$	A typical time that passes before the change in physical memory begins to affect performance [second]
$T_{auction}$	$\sim$	The time between auction rounds[second]
T_{load}	$\sim$	A typical time scale in which conditions (e.g., load) change[second]
f_i	$\sim$	A guest specific coefficient, weight [\$/Khit]
$Waste$	$\sim$	An upper bound on memory waste [MB]
SW_{max}	$\sim$	The social welfare that originates from the optimal allocation

Additional Directions of RaaS (Chapter 7)		
U_i	$\sim$	Utility of guest i
U_{host}	$\sim$	The utility of the host
A,B	$\sim$	Guest indices
D	$\sim$	Number of resources (also number of problem dimensions)
R_+^D	$\sim$	The multi-resource allocation space
$\vec{d}$	$\sim$	A multi-resource allocation vector
$V(\vec{d})$	$\sim$	A multi resource valuation function
v	$\sim$	A vector holding either 0 or 1 for each of the resources
$\vec{a}$	$\sim$	Multi-resource valuation function local approximation coefficients
$\vec{r}$	$\sim$	Start point of a desired range (in all the resources)
$\vec{q}$	$\sim$	End point of a desired range (in all the resources)
A_k	$\sim$	The amount of resource k that is available for auction
MPI	—	Message Passing Interface

Chapter 1

Introduction

Shared computers are liable to be inefficiently utilized due to conflicts of interest: the hardware and electricity bill are paid for by one economic entity, while the workloads that make use of the computers benefit other economic entities: the clients. Examples of such shared computers are clouds and grids. Grids are a way for privileged clients to barter surplus computing resources. The grid privileges of a client are defined by a complex system that involves the client's historical use of the grid, the amount of computing resources that the client's employer shared, and the client's type of business (with a preference to academic researchers, or specific scientific domains). The basic compute unit in the grid is called a *job* or a *task*. It is a process group on the shared machine that is controlled by the client. Tasks can be combined in larger dependency structures such as Bags of Tasks (BoTs): groups of tasks that need to be completed in full. To ensure a timely completion of such BoTs, clients often use replication: they execute the same task several times.

Clouds, on the other hand, rent computing resources to clients for a fee, in a simple exchange system. Unlike grids, Clouds allow for many models of resource sharing. When we analyze clouds in this work we focus on the model that supplies the closest environment to the one in grids: sharing of full operating systems, denoted Infrastructure-as-a-Service (IaaS). A major IaaS provider nowadays is Amazon, offering the Elastic Compute Cloud (EC2) at three commitment levels: reserved (where an instance is partially paid for in advance, and the client is guaranteed its availability for a low price); on-demand (where the client gets an instance if the provider has one to let);

and spot (where the provider may terminate the instance if it needs it back).

We examine resource utilization strategies available to clients in a traditional environment of grids, assisted by an on-demand IaaS cloud. Due to the unreliable nature of the grids, which suffer from deliberate task preeemption, these strategies involve task replication. The less reliable the grids are, the more wasteful the strategies are. The role of the cloud in these replication strategies is to limit BoT makespan by verifying that the task is performed within a deadline. We propose a model for task replication on grids and a cloud, and develop a recommendation system that finds a Pareto-efficient option within the model, which we demonstrate is extensive enough to include the required efficient solutions. Strategic client behavior is also required, for example, in Amazon EC2's spot instances, which are unreliable due to rapid price change. Spot instance prices are supposed to reflect changes in supply and demand, but we discovered that most of the perceived unreliability was artificially generated, masking resource under-utilization.

IaaS clouds are implemented using virtualization techniques. In a virtual system, the provider runs a basic operating system called a hypervisor, virtual machine manager (VMM), or *host* on a bare metal machine. On top of this host run guests, which are also called virtual machines (VMs).The host replaces the hardware functionality for the guests. It does so using various methods: by changing drivers, by changing the guest's code on-the-fly, and even using hardware assistance. It also controls the amount of resources that are exposed to the guest (on a sub-machine level). The most general form of virtualization is that of unmodified operating systems, which allows any operating system to be a guest without any need for further adaptations. It allows the client the freedom of choice of its own working environment.

Amazon was, for a while, the spearhead of several trends in the public cloud industry. These trends include refining the resource rental-time granularity, refining the resource quantity granularity, and offering more flexible service level agreements. Other providers soon followed suit, pushing these trends further by reducing the rental-time granularity to minutes and the resource granularity to hundreds of MBs and CPU fractions. They also freed the sub-machine resources (cores, memory and bandwidth) from bundling. Due to the need for non-cooperative users to efficiently share resources, these trends will likely culminate in the rise of a new economic model that we term the Resource-as-a-Service (RaaS) cloud. Instead of fixed bundles, cloud providers will increasingly sell resources individually, reprice them, and adjust their quantity every few seconds in accordance with market-driven supply-and-demand conditions.

We propose Ginseng, a RaaS prototype that efficiently rents physical memory to non-cooperative virtual machines at a fine time and resource granularity, at personally adapted prices, and with flexible service level agreements. Ginseng achieves a $\times 6.2$–$\times 15.8$ improvement in aggregate client satisfaction when compared with state-of-the-art approaches for cloud memory allocation. It achieves 83%–100% of the optimal aggregate client satisfaction.

Thus, in this work we demonstrate how the development of the shared computing resource models from a bartering economy to a simple exchange economy is already making the use of shared computing resources more efficient. On top of this shift, we predict the rise of a new cloud model, that will enable an even more efficient cloud. We lay out the design of such a prototype and demonstrate its efficiency.

Chapter 2

Related Work

2.1 Pareto-Efficient Task Replication on Grids and a Cloud

Much research on replication algorithms has relied on the assumption that computation is free of charge [29, 36, 42, 77, 127, 150] and limited only by its effect on load and makespan, whereas we explicitly consider execution costs. Dobber, van der Mei, and Koole [42] created an on-the-fly criterion for choosing between immediate replication and dynamic load balancing. Casanova [29] showed the impact of simple replication policies on resource waste and fairness. Kondo, Chien, and Casanova [77] combined replication with resource exclusion. Resource exclusion can also be combined with our work. Cirne et al. [36] and Silva et al. [127] analyzed immediate replication with no deadline for perfectly reliable heterogeneous machines. Borst et al. [24] used a slotted machine approach, which produced a geometric distribution for the turnaround time. Wingstrom and Casanova [150] assumed a specific distribution (generalized doubly folded normal distribution) probability of task failures and used it to maximize the probability of a whole BoT to finish executing, by choosing replication candidates. In contrast, we optimize cost and time simultaneously.

Bi-objective time-related problems were also analyzed in task scheduling. Vydyanathan et al. [143] aimed to minimize latency while meeting strict throughput requirements using replication, subject to a certain amount of resource waste, in terms of the number of occupied processors. They [144] also aimed to maximize the throughput while meeting latency constraints, as did Agrawal et al. [4] for linear task graphs. Our work optimizes one time-related and one monetary objective for BoTs.

The concept of utility functions as the target of the optimization process has also received attention. Buyya et al. [27] researched economic mechanisms for setting grid computation costs, for several utility functions. One of their estimation methods is Pareto-efficiency. Cirne et al.'s workqueue with replication strategy instructs the user to replicate a task [36], to get the fastest result. Our replication model language allows for improvement of this strategy with regard to energy saving by sending the replicas after the first result has failed to return for some time. Ding et al. [41] aimed to minimize the utility function of the energy-delay product on a multi-CPU machine, by using a helper thread which collects statistics and determines a deployment strategy. Lee, Subrata and Zomaya [82] aimed to minimize both grid resource use and makespan for a workflow application, by giving them an equal weight. Benoit et al. [22] assumed a linear risk model for machine unavailability on homogeneous remote machines, and considered overhead and operational costs. Our work allows for both a general user function and a general probability distribution of task success. Andrzejak, Kondo, and Anderson [14] controlled reliable and unreliable pool sizes in a combined pool to Pareto-optimize cost and availability for Web services.

Pareto frontier approximations were previously used in scheduling for the makespan and reliability objectives, but not for cost, by Dongarra et al. [45], who scheduled task graphs, by Saule and Trystram [119], and by Jeannot et al. [70].

Ramírez-Alcaraz et al. [113] evaluated scheduling heuristics and optimized a combined objective for parallel jobs, because they believed that computing a Pareto frontier in a grid environment is too slow. However, approximating the Pareto frontier for the cases we demonstrate in this work using `ExPERT` takes only minutes—hardly "too slow" for a BoT that runs for hours.

Oprescu and Kielmann [104] learned the run-time CDF on-line from the execution of the same BoT, as we do. However, they did not deal with reliability, since they used only clouds, and they utilized a heuristic to minimize makespan for a given budget. In contrast, our approach provides the client with full flexibility of choice, without forcing the choice of budget first, and is valid for grids, too, where reliability is an issue.

Pareto frontiers were also used to concurrently optimize the same objective for different users, to achieve socially efficient scheduling and resource management [33,38]. Zhao et al. [158] designed a market for BoTs, aiming to efficiently optimize social welfare under agent budget constraints. Our work focuses on multiple objectives of the same user.

2.2 Amazon EC2 Spot Instances

Pay-as-you-go IaaS cloud workload traces are quite rare, with the main data originating from spot prices. These traces were used as characteristic IaaS traces by many researchers, both to design user strategies and to learn more from the provider's point of view. Despite their name, spot instances are not markets, and in particular not Spot and Future markets. Works on such markets of computational resources are another trace source. In our work we claim that Amazon is using a certain random reserve price, which is a hidden minimal price. In this section we review therefor works that compare the reserve price with the minimal price method. In section 4.9 we discuss the trace-analyzing literature further in view of our results.

Cloud Traces IaaS pay-as-you-go cloud workload traces and models are so hard to come by that researchers like Toosi et al. [132] resorted to a grid and parallel systems model [90] with adapted runtime parameters to describe cloud workloads. Google [59] released two backend workload traces, the longest of which lasts 29 days. Liu [87] measured week-long traces of CPU utilization of EC2 machines, showing a strong daily pattern of the guest machines on the measured host. This pattern indicates that clients prefer to keep instances running idle rather than shut them off for the night. Such client behavior weakens the daily cycle of demand for EC2 machines in general (not necessarily spot instances).

Reserve Prices Li and Tan [84] showed that a (hidden) reserve price improves revenues of first price, sealed bid auctions for risk-averse clients. Li and Perrigne [85] showed that for first price sealed bid auctions, an optimal announced minimal price increases the seller's revenue compared with an arbitrary reserve price. They used data of timber sales in Canada. Katkar and

Reiley [74] found that for low-priced eBay sales of up to $20, (hidden) reserve prices deter good clients and yield lower revenues than minimal (published) prices. However, none of these works relate to an auction with a random reserve price, in which the price is set by the highest bidder that does not win the good. Ramberg [111] says that "the existence of a hidden reserve price is to a great extent similar to the situation where the invitor is bidding." She recommends that when the auction is run by the invitor (as is the case with Amazon's spot instances), "...it should not be a second price auction, or otherwise there should be some assurance that the invitor/operator will not submit bids."

Analyzing Spot Price Traces Concurrently with our work, Wee [148] also analyzed price-availability graphs of early EC2 traces, noted the knees and the different behavior of `m1.small`, and that the average price does not change over time. Wee only analyzed epochs in which the timing of price changes always included a quiet hour and assumed that Amazon does not have an incentive to change prices more often than once an hour. However, as we show in Section 4.7.4, Amazon's early price change timing was a vulnerability, incentivizing it to change prices more frequently than once an hour, as it later did. Wee [148] and Javadi and Buyya [69] also checked EC2 price traces for cycles. Javadi and Buyya, who computed various price trace statistics, claimed spot prices have daily and weekly cycles, but Wee found that cycles are statistically insignificant. Our findings agree with Wee's.

Using Spot Price Traces for Client Strategy Evaluation Most studies that use price traces use them to evaluate client strategies. Andrzejak, Kondo and Yi used spot price histories to advise the client how to minimize monetary costs while meeting a Service Level Agreement (SLA) [15], and to schedule checkpoints [154] and migrations [153]. Voorsluys et al. [141] created a spot instance broker.

Mattess, Vecchiola, and Buyya [95] examined client strategies for using spot instances to manage peak loads on scientific workloads. They identified a price band, noted that bidding just above the band is almost as good as bidding very high, and recommended bidding right under the on-demand price.

Chohan et al. [34] processed price histories to compute the probability that an instance with a certain bid price would last a certain time. They also identified a price-band and noted the cost-effectiveness of bidding at its top.

Wieder et al. [149] described a model for optimizing map-reduce on clouds using a utility function that depends on execution time, data transfer costs, and computation costs, which they assumed can be predicted for spot instances. Brebner and Liu [26] assessed cost and performance of various clouds, including spot instances. Vermeersch [137] analyzed spot price histories with the goal of optimizing the client's choice of deals on EC2.

Free Spot and Futures Markets While Amazon is currently the only provider offering "spot instances," free computing resource markets have already been analyzed. Ortuno and Harder [105] modeled a free market for computing power. Altmann et al. [7] described GridEcon, a foundation for a free spot and futures market. Vanmechelen, Depoorter, and Broeckhove [134] modeled a free market for computing power using spot and futures deals.

2.3 Ginseng and Resource-as-a-Service

Resource allocation can be done either in a white-box model, where the host knows what the guest is doing and has full visibility into it, or in a black-box model, where the host has no visibility into the guest. The latter is a reasonable assumption for a public cloud. Guest hinting is a method in which the guest passes specific information to the host, to make the allocation process more efficient. Most of the literature on divisible good allocation assumed that the client valuation of a good is monotonically rising and concave, meaning that the law of diminishing returns applies to it. In our work we show that for memory allocation this assumption is unrealistic.

White-Box Memory Overcommitment. Heo et al. [60] balanced memory allocations according to desired performance levels. Like us, they avoided quick changes, but for reasons of stability of the feedback loop. Under memory pressure they divided the memory according to a fair share policy. In Q-clouds, Nathuji, Kansal and Ghaffarkhah followed a concept of both measuring and selling performance [102]: The guest specifies several performance and payment levels and the host chooses which level to fulfill. This approach is convenient to the host, which is guaranteed a demand for any excess production power it has. Our approach is guest oriented, leaving the designation of the current required resource amount in the hands of the guest. In Ginkgo, Hines et al. [61] and Gordon et al. [54] used optimization with constraint satisfaction to optimize a general social welfare function of the guests' performance. These works assume guest cooperation, while we analyze the guest as a non-cooperative, selfish agent. Our work is the first work on memory allocation which assumes non-cooperative guests.

Black-Box Techniques. Magenheimer [92] used the guests' own performance statistics to guide overcommitment. Jones, Arpaci-Dusseau, and Arpaci-Dusseau [72] inferred information about the unified buffer cache and virtual memory by monitoring IO and inferring major page faults. Zhao and Wang [160] monitored use of physical pages. Waldspurger [145] randomly sampled pages to find unused pages to reclaim, and introduced the "idle memory tax," which resembles our reclaim factor. These methods can be fooled by a selfish guest, and like white-box methods, ignore the client's valuation of performance. Gupta et al. [56] did not require any guest cooperation for their content based page sharing. Wood et al. [152] allocated guests to physical hosts according to their memory contents. Gong, Gu and Wilkes [52] and Shen et al. [122] used learning algorithms to predict guest resource requirements.

Sekar and Maniatis [121] argued that all resource use must be accurately attributed to the guests who use it so that it can be billed. In contrast, Ginseng lays the burden of metering on the client: the client can measure its current performance and decide how much it is willing to pay for memory.

Concurrently with our work, Vorontsov [142] proposed the mempressure control group, which includes an interface for requesting that applications release memory.

Guest Hint Techniques. Schwidefsky et al. [120] used guest hints to improve host swapping. Miłoś et al. [101] incentivized guests to supply sharing hints by counting a shared page as a fraction of a non-shared page. Like Ginseng, their method can be applied to non-cooperative guests.

General Resource Allocation For Monotonically Rising, Concave Valuations. Kelly [75] used a proportionally fair allocation: clients bid prices, pay them, and get bandwidth in proportion to their prices. His allocation is optimal for *price taking* clients (who do not anticipate their impact on the price they pay). Popa et al. [109] traded off proportional fairness with starvation prevention. Johari and Tsitsiklis [71] computed the price of anarchy of Kelly's auction, and Sanghavi and Hajek [118] improved the auction in this respect.

Maillé and Tuffin [93] extended the PSP to multi-bids, thus saving the auction rounds needed to reach equilibrium. Their guests disclosed a sampling of their resource valuation functions to the host, which computed the optimal allocation according to these approximated valuation functions. One such single auction has the complexity of a single PSP auction, times the number of sampling points. Though a multi-bid auction is more efficient for static problems, it loses its appeal in dynamic problems which require repeated auction rounds anyhow. Other drawbacks of the multi-bid auction are that the guest needs to know the memory valuation function for the full range; that frequent guest updates pose a burden to the host; and that the guest cannot directly explore working points which currently seem less than optimal. (It can do so indirectly by faking its valuation function.) In contrast, our memory progressive second price (MPSP) auction leaves the control over the currently desired resource allocation to the guest, who best knows its own current and future needs. Maillé and Tuffin also showed that the PSP's social welfare converges to theirs [94].

Chase et al. [31] allocated CPU time assuming client valuations of the resource are fully known, concave, and monotonically increasing.

Google's generalized second price (GSP) auction uses a limited bidding language and is not a VCG auction [46].

Urgaonkar, Shenoy, and Roscoe [133] overbooked bandwidth and CPU cycles given full profiling information but did not address memory.

Unlike bandwidth and CPU auctions, our memory auction is oriented toward minimizing transfer of ownership. Unlike divisible good auctions, it supports non-concave valuation functions.

Ghodsi et al. [51], Dolev et al. [44] and Gutman and Nisan [57] considered allocating multiple resources to strategic guests whose private information is the relative quantities they require of the resources. In contrast, Ginseng compares valuations of different strategic clients.

Auctions With Non-concave Valuations. Bae et al. [19] supported a single bidder with a non-concave valuation function. Dobzinski and Nisan [43] presented truthful polynomial time approximation algorithms for multi-unit auctions with k-minded valuations. They only assumed that the valuations are non-decreasing (because they allow *free disposal*—shedding of unneeded goods), and did not require them to be concave, but allowed the guests to make queries before bidding. Our bidding language of forbidden ranges is more efficient than free disposal, because it allows immediate auctioning of the undesired memory.

Chapter 3

ExPERT: Pareto-Efficient Task Replication on Grids and a Cloud

3.1 Abstract

Many scientists perform extensive computations by executing large bags of similar tasks (BoTs) in mixtures of computational environments, such as grids and clouds. Although the reliability and cost may vary considerably across these environments, no tool exists to assist scientists in the selection of environments that can both fulfill deadlines and fit budgets. To address this situation, we introduce the `ExPERT` BoT scheduling framework. Our framework systematically selects from a large search space the Pareto-efficient scheduling strategies, that is, the strategies that deliver the best results for both makespan and cost. `ExPERT` chooses from them the best strategy according to a general, user-specified utility function. Through simulations and experiments in real production environments, we demonstrate that `ExPERT` can substantially reduce both makespan and cost in comparison to common scheduling strategies. For bioinformatics BoTs executed in a real mixed *grid+cloud* environment, we show how the scheduling strategy selected by `ExPERT` reduces both makespan and cost by 30%-70%, in comparison to commonly-used scheduling strategies.

3.2 Introduction

The emergence of cloud computing creates a new opportunity for many scientists: using thousands of computational resources assembled from both grids and clouds to run their large-scale applications. This opportunity, however, also adds complexity, as the shared grid systems and the pay-per-use public clouds differ with regard to performance, reliability, and cost. How can scientists optimize the trade-offs between these three factors and thus efficiently use the mixture of resources available to them? To answer this question, we introduce `ExPERT`, a general scheduling framework which finds Pareto-efficient job execution strategies in environments with mixtures of *unreliable and reliable* resources.

Today's grids and clouds reside in two extremes of the reliability and cost spectrum. Grid resources are often regarded as unreliable. Studies [64,78,79] and empirical data collected in the Failure Trace Archive [79] give strong evidence of the low long-term resource availability in traditional and desktop grids, with yearly resource availability averages of 70% or less. The constrained resource availability in grids is often a result of the sharing policy employed by each resource provider—for example, the grid at UW-Madison [151] employs preemptive fair-share policies [131], which vacate running tasks of external users when local users submit tasks. Commercial clouds, in contrast, have service-level agreements that guarantee resource availability averages of over 99%. Cost-wise, scientists often perceive grids as being free of charge, whereas clouds are pay-per-use. Accordingly, many grid users are now exploring the opportunity to migrate their scientific applications to commercial clouds for increased reliability [66, 68, 128], which could prove prohibitively expensive [128].

Scientific grid applications are often executed as Bags of Tasks (BoTs)—large-scale jobs comprised of hundreds to thousands of asynchronous tasks that must be completed to produce a single scientific result. Previous studies [63, 65] have shown that BoTs consistently account for over 90% of the multi-year workloads of some production grids. Thus, BoTs have been the *de facto* standard for executing jobs in unreliable grid environments over the past decade.

When executing BoTs in a grid environment, scientists *replicate* tasks. Replication increases the odds of timely task completion despite resource unreliability [13, 29, 77, 125, 155], but also wastes CPU cycles and energy, and incurs other system-wide costs [29] such as scheduler overload and delays to other users. It is difficult to select a *replication strategy* that yields the desired balance between the BoT response time (makespan) and the BoT execution cost. A wrong strategy can be expensive, increasing *both* makespan and cost. Although various heuristics were devised to pick a “good” replication strategy, our study is the first to focus on explicitly identifying Pareto-efficient strategies, that is, strategies that incur only the necessary cost and take no longer than necessary to execute a given task.

We envision a world in which BoTs are executed on whatever systems are best suited to the user’s preferences at that time, be they grids, clouds, dedicated self-owned machines, or any combination thereof. This vision presents many optimization opportunities; optimizing the structure of the reliable+unreliable environment is only one of many examples. These opportunities can be exploited only when taking into account the individual preferences of each scientist. One scientist might want to obtain results by completing a BoT as quickly as possible, regardless of cost. Another might choose to minimize the cost and complete the BoT only on grid resources. Yet another scientist might try to complete work as soon as possible but under strict budget constraints (e.g., [104]). What all users share is a desire for efficient scheduling strategies.

Our main research goal is to determine *which strategies are Pareto-efficient and which of them the user should pick*. The following four questions will guide us in helping the user choose the best possible strategy. *What mixture of reliable and unreliable resources should be used? How many times should tasks be replicated on unreliable resources? What deadline should be set for those replicas? What is the proper timeout between submitting task instances?* Although Pareto-efficient strategies have been investigated before in different contexts [1, 40, 96, 100], they are generally considered too computationally-intensive for online scheduling scenarios. However, we show here that even low-resolution searches for Pareto-efficient strategies benefit scheduling large numbers of tasks online.

Our first contribution is a model for task scheduling in mixed environments with varying reliability and cost (Sections 3.3 and 3.4). Our second contribution is `ExPERT`, a framework for dynamic online selection of a Pareto-efficient scheduling strategy, which offers a wide spectrum of efficient strategies for different user makespan-cost trade-offs, leading to substantial savings in both (Section 3.5). We evaluate `ExPERT` through both simulations and experiments in real environments (Section 3.6), and show (Section 3.7) that `ExPERT` can save substantial makespan and cost in comparison to scheduling strategies commonly used for workload scheduling in grids.

3.3 The Basic System Model

In this section we introduce the basic system model used throughout this work. We first build towards the concept of the Pareto frontier, then present the model for the system and the environment.

3.3.1 Terminology

A *t*ask is a small computational unit. A task *i*nstance is submitted to a resource. If the resource successfully performs the task, it returns a *r*esult. For a successful task instance, the *result turnaround time* is the time between submitting an instance and receiving a result. For a failed instance, this is ∞. A *BoT* is a set of asynchronous, independent tasks, forming a single logical computation. Users submit BoTs to be executed task-by-task. We divide BoT execution into the *throughput phase* and the *tail phase*, as depicted in Figure 3.1. The *remaining tasks* are tasks which have not yet returned a result. The *tail phase start time* (T_{tail}) occurs when there are fewer remaining tasks than available unreliable resources. A BoT is completed when each of its tasks has returned a result. The *makespan of a BoT* is the period elapsed from its submission to its completion. Similarly, the *tail phase makespan* is the period elapsed from T_{tail} until the completion of the BoT.

Replication is the submission of multiple *instances* of the same task, possibly overlapping in time. A task is complete when one of its instances returns a successful result. The *reliability* of a resource pool is the probability that an instance submitted to that pool will return a result.

Cost is a user-defined price tag for performing a task, and may reflect monetary payments (e.g., for a cloud), environmental damage, or depletion of grid-user credentials. We ignore the costs of failed instances since it is difficult to justify charging for unobtained results.

The *user's scheduling system* (user scheduler) sends and replicates the

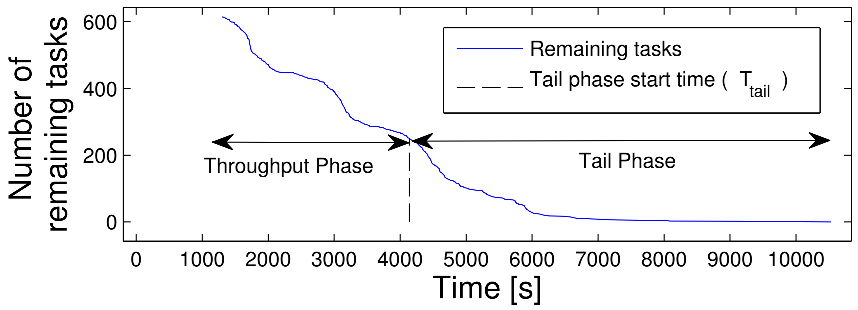


Figure 3.1: Remaining tasks over time during the throughput and tail phases. Input: Experiment 6 (Table 3.5).

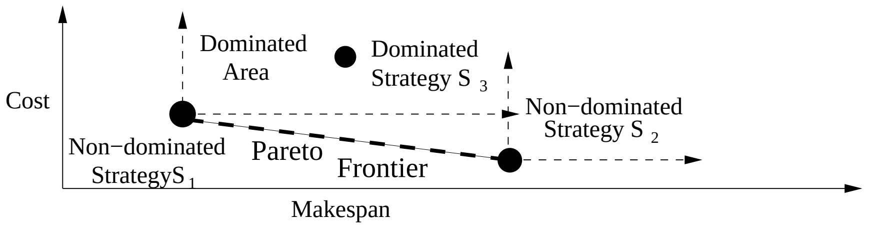


Figure 3.2: A Pareto frontier. Strategies S_1 and S_2 form the Pareto frontier. S_1 dominates S_3.

user's tasks to the available resource pools. A user *strategy* is a set of input parameters indicating when, where, and how the user wants to send and replicate tasks.

The *performance metrics* are *cost per task* (the average cost of all BoT tasks) and makespan. A *user's utility function* is a function of the performance metrics of a strategy that quantifies the benefit perceived by the user when running the BoT. The user would like to optimize this function, for a given BoT and environment, when selecting a strategy. For example, a user who wants the cheapest strategy can use a utility function that only considers costs.

A strategy is *dominated* by another strategy if its performance is worse than or identical to the other for both metrics (cost and makespan) and strictly worse for at least one. A strategy that is not dominated by any other strategy is *Pareto-efficient*; the user cannot improve this strategy's makespan without paying more than its cost. As illustrated in Figure 3.2, several Pareto-efficient strategies may co-exist for a given *unreliable+reliable* system and workload (BoT). The *Pareto frontier* (or "Skyline operator" [25]) is the locus of all efficient strategies with respect to the searched strategy space. Any strategy that optimizes the user's utility function is Pareto-efficient. Furthermore, for any Pareto-efficient strategy, there exists a utility function that the strategy maximizes in the search space.

3.3.2 Model and Assumptions

We outline now the model and the assumptions for this work, first the environment, then the execution infrastructure. The assumptions are inspired by real-world user schedulers such as GridBoT [125], which are designed for CPU-bound BoTs that are not data bound.

Our model of the environment consists of two task queues. One queue is serviced by the *unreliable pool*, and the other is serviced by the *reliable pool*.

We characterize the reliable and unreliable pools in terms of speed, reliability, and effective size. Unreliable machines operate at various speeds; reliable machines are homogeneous. (We assume they are of the same cloud instance type or belong to a homogeneous self-owned cluster. Thus, they are far more homogeneous than the unreliable machines.) Failures in the unreliable pool are abundant and unrelated across different domains [64]; reliable machines never fail (we justify the approximation by the large reliability difference between the unreliable and reliable pools). The reliable and unreliable pools have different *effective sizes* (number of resources that the user can concurrently use). We assume that effectively there are many more unreliable than reliable machines (typical effective sizes are hundreds of unreliable nodes and tens of reliable nodes), and thus we do not consider using only the reliable resources. Resources are charged as used, per charging period (one hour on EC2, one second on grids and self-owned machines).

We make no assumptions on task waiting time or on the unreliable system's scheduling policy, other than that both can be modeled statistically. Since we allow for loose connectivity between the scheduler and the hosts [13], it may be impossible to abort tasks, and the exact time of a task failure may not be known. A task which did not return its result by its deadline is considered failed. We assume the user has an overlay middleware that replaces malfunctioning hosts with new ones from the same pool. Our experiments show that such middleware can maintain an approximately constant number of unreliable resources when requesting up to 200 machines from a larger infrastructure.

3.4 The Scheduling Strategy Space

In this section we introduce our model for scheduling tasks with replication in an environment with mixed reliability, cost, and speed. The model generalizes state-of-the-art user strategies, e.g., of GridBoT users [125]. We focus on optimizing the tail phase makespan and cost by controlling the tail phase scheduling strategy, for three reasons. First, in naive BOINC executions [13], the tail phase is an opportunity for improvement [124], as seen in Figure 3.1: the task return rate in the tail phase is low, while many resources are idle. Second, replication is inefficient during the throughput phase [50]. Third, setting the decision point after the throughput phase lets us base the optimization on the highly-relevant statistical data (e.g., of task turnaround times) collected during the throughput phase.

During the throughput phase we use a "no replication" strategy, with a deadline of several times the *average task CPU time on the unreliable resource* (denoted by T_{ur} and estimated according to several random tasks). This deadline length is a compromise between the time it takes to identify dysfunctional machines and the probability of task completion. A long deadline allows results to be accepted after a long time, but leads to long turnaround times. For the tail phase, we can consider strategies with deadlines set to the measured turnaround times. Deadlines much longer than T_{ur} are not interesting, because strategies with such deadlines are inefficient.

When the tail phase starts, all unreliable resources are occupied by instances of different tasks, and the queues are empty. From that point on, additional instances are enqueued by a scheduling process: first to the unreliable pool, then to the reliable one, as illustrated in Figure 3.3. This scheduling process, which we name $NTDM_r$, is controlled by four user parameters, N, T, D and M_r. Different strategies have different $NTDM_r$ values:

N is the maximal number of instances sent for each task to the unreliable system since the start of the tail phase. A last, $(N+1)^{th}$ instance is sent

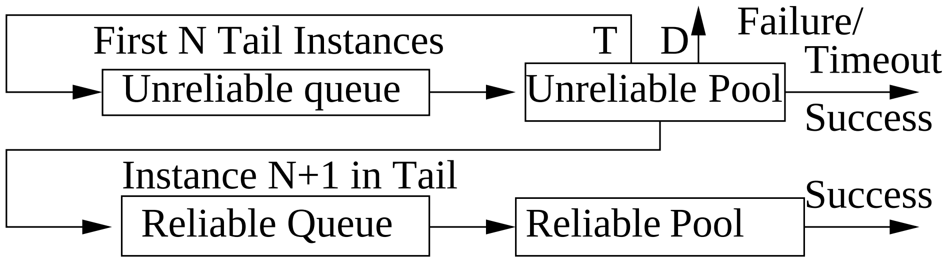


Figure 3.3: $NTDM_r$ task instance flow during throughput phase and tail phase. Reliable machines serve only instance $N+1$ during the tail phase (throughput phase instances are not counted). During the throughput phase, $T = D$, so there is no replication

to the reliable system without a deadline, to ensure task completion. A user without access to a reliable environment is restricted to $N = \infty$ strategies. Increasing N improves the chance that the reliable instance will not be required, but increases the load on the unreliable pool. It also increases the probability of receiving and paying for more than one result per task.

D is a deadline for an instance, measured from its submission to the system. Setting a large value for D improves the instance's chances to complete on time, but increases the time that elapses before the user becomes aware of failures. Short deadlines enable quick resubmission of failed tasks.

T is a timeout: the minimal waiting time before submitting another instance of the same task. Rather than having all instances submitted at the same time, each is submitted after a period T has passed from the previous instance submission, provided that no result has yet been returned. T restricts resource consumption.

$\mathbf{M_r}$ is the ratio of the effective sizes of reliable and unreliable pools. It provides a user-defined upper bound on the number of concurrently used reliable resources. Small M_r values create long queues for the reliable pool. A long reliable queue may indirectly reduce costs by allowing unreliable instances to return a result and cancel the reliable instance before it is sent. We demonstrate M_r's contribution to the cost reduction of efficient strategies in Section 3.7.

The user's main goal is to choose values for N, T, D, and M_r, such that the resulting makespan and cost optimize a specific utility function. How-

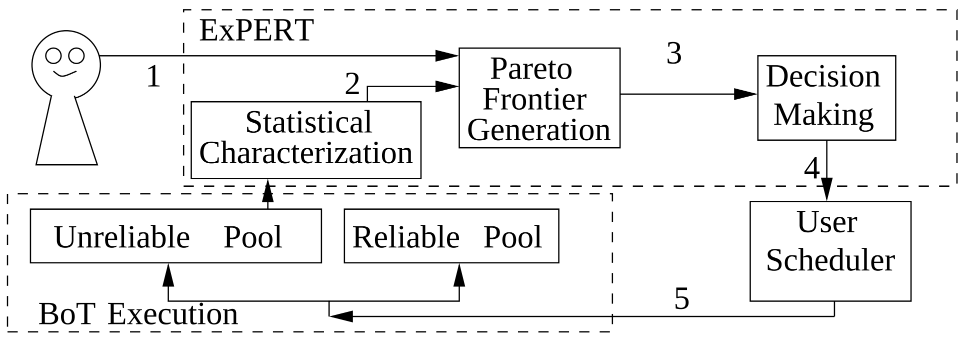


Figure 3.4: Flow of the ExPERT stages, with user intervention points. Numbered arrows indicate process steps.

ever, the user does not know the cost-makespan trade-off, or what parameter values would lead to a specific makespan or cost. To help the user choose these values, we introduce in the next section a framework for the selection of an efficient replication strategy.

3.5 The ExPERT Framework

In this section we explain the design and use of the ExPERT scheduling framework. Our main design goal is to restrict the $NTDM_r$ space to Pareto-efficient strategies, from among which the user can then make an educated choice. To achieve this goal, ExPERT defines a scheduling process, which includes building a Pareto frontier of $NTDM_r$ strategies, out of which the best strategy for the user is chosen.

The ExPERT Scheduling Process: The $NTDM_r$ task instance flow is depicted in Figure 3.4. The user provides her parameters and, optionally, a utility function. ExPERT then statistically characterizes the workload and the unreliable system on the basis of historical data, analyzes a range of strategies, generates the Pareto frontier, and presents the user with makespan- and cost-efficient strategies. After either the user or ExPERT decides which strategy in the frontier to use, ExPERT passes the N, T, D, M_r input parameters of the chosen strategy to the user's scheduler, which then replicates tasks and submits them to the two resource queues.

The ExPERT framework is extensible in three ways. First, in Step 2 it

Table 3.1: User-defined parameters

Item	Definition
T_{ur}	Mean CPU time of a successful task instance on an unreliable machine
T_r	Task CPU time on a reliable machine
C_{ur}	Cents-per-second cost of unreliable machine
C_r	Cents-per-second cost of reliable machine
M_r^{max}	Maximal ratio of reliable machines to unreliable machines

allows for alternative methods of gathering and analyzing the system properties. Second, in Step 3 it allows for alternative algorithms for construction of the Pareto frontier. Third, in Step 4 it allows the user to employ any utility function which prefers lower makespans and costs: using the Pareto frontier allows freedom of choice with regard to the utility function.

Traditionally, BoTs are executed through schedulers such as GridBoT [125], BOINC or Condor using a pre-set strategy, defined when the BoT is submitted. Though historical performance data has been used by others for resource exclusion [77] and for resource allocation adaptation [104], `ExPERT` is the first to use it to optimize general makespan and cost preferences. In addition, once the Pareto frontier is computed, it supplies the user with an understanding of the trade-offs available in the system, to be utilized in the future, possibly with different utility functions.

User Input: The user supplies `ExPERT` with data about mean CPU times (denoted T_r, T_{ur}), runtime costs in cents per second (denoted C_r, C_{ur}), and the reliable resource pool's effective size relative to the unreliable one (Table 3.1). M_r^{max}, the upper bound of M_r, is derived from the unreliable pool's effective size, as well as from the number of self-owned machines, or from a restriction on the number of concurrent on-demand cloud instances (e.g., at most 20 concurrent instances for Amazon EC2 first-time users). Runtime costs might reflect monetary payments, energy waste, environmental damage, or other costs. For example, a user might set unreliable costs as zero, representing the grid as free of charge, or set it to account for power consumption. `ExPERT` uses this data to estimate the BoT's cost and makespan under different strategies, when it searches the strategy space.

Statistical Characterization: `ExPERT` statistically characterizes the workload and the unreliable system using $F(\cdot)$, the Cumulative Distribution Function (CDF) of *result turnaround time*. It also estimates the *effective size of the unreliable pool*, denoted as $\sharp ur$, by running iterations of the *`ExPERT` estimator* (described below) over the throughput phase until the estimated result rate matches the real result rate. The estimated $\sharp ur$ and $F(\cdot)$ are used to predict the makespan and cost of a given strategy and BoT. We describe here the estimation of $F(\cdot)$. The estimated $\sharp ur$ and $F(\cdot)$ are later used in step 3 to statistically predict the makespan and cost of applying a scheduling strategy to the execution of a given BoT.

$F(\cdot)$ effectively models many environmental, workload, and user-dependent factors. It is used to predict result turnaround time during the tail phase, so it is best estimated in conditions that resemble those prevailing during this phase. The throughput phase supplies us with such data, but it can also be obtained from other sources. If the throughput phase is too short to collect enough data before the tail phase starts, public grid traces can be combined with statistical data about the workload to estimate the CDF.

The CDF is computed as follows:

$$F(t, t') = F_s(t)\gamma(t'). \tag{3.1}$$

Here t denotes instance turnaround time, and t' denotes instance sending time. $F_s(t)$ denotes the *CDF of successful task instances* (i.e., those which returned results). It can be directly computed from the turnaround times of results. $\gamma(t')$ denotes the *unreliable pool's reliability at time t'*: the probability that an instance sent at time t' to the unreliable pool returns a result at all. $\gamma(t')$ is computed for disjoint sets of consecutively sent instances as the number of results received by the deadline, divided by the number of instances.

Because $F(\cdot)$ depends on t' through $\gamma(t')$, the CDF might change over time, necessitating a prediction model. `ExPERT` can either compute $\gamma(t')$ *offline* or estimate it *online*. The accuracy of the two models is compared in Section 3.7. In the offline model, $\gamma(t')$ is fully known (it is computed after all the results have returned). In the online model, $\gamma(t')$ is predicted according to information available at the decision making time T_{tail}. Depending on when the instance was sent, at time T_{tail} we might have full knowledge, partial knowledge, or no knowledge whether the instance will have returned a result by the time its deadline arrives. The time-line of the instance sending time t' is divided into three epochs as follows.

1. Full Knowledge Epoch: the instance was sent at time t' such that $t' < T_{tail} - D$. Instances sent during this first epoch that have not yet returned will not return anymore, so all the information about these tasks is known at time T_{tail}, in which the online reliability is evaluated. The online reliability model is identical to offline reliability during this epoch.

2. Partial Knowledge Epoch: $T_{tail} - D \leq t' < T_{tail}$. Instances sent during this second epoch that have not yet returned may still return. We use Equation 3.1 to approximate the probability that an instance sent at time t' will eventually finish. That is, we try to compute $\gamma(t')$ on the basis of the observable task success rate ($F_s(t)$). According to our model in Equation 3.1, $F(t, t')$ is separable. Hence, instead of computing $F_s(t)$ according to data of this second epoch to evaluate $F_{s_2}(t)$, we use $F_{s_1}(t)$, that is, the CDF of successful instances during the first epoch.

   Let $\hat{F}(t, t')$ denote $F(t, t')$ as was computed for instances sent at time t'. With the information known at time T_{tail}, the CDF is fully known ($F(t, t') = \hat{F}(t, t')$) for small values of t ($t \leq T_{tail} - t'$). However, for larger values of t, no information exists. As t' approaches T_{tail}, $\hat{F}(T_{tail} - t', t')$ becomes less accurate, because it relies on less data.

   We substitute the approximations $F_{s_1}(t)$ and $\hat{F}(t, t')$ in Equation 3.1 for the time t for which we have the most data ($t = T_{tail} - t'$):

   $$\gamma(t') = \frac{F(T_{tail} - t', t')}{F_s(T_{tail} - t')} \approx \frac{\hat{F}(T_{tail} - t', t')}{F_{s_1}(T_{tail} - t')}. \tag{3.2}$$

   Due to the diminishing accuracy of the computation of $\hat{F}(T_{tail} - t', t')$, Equation 3.2 may result in fluctuating, unreasonable values, which need to be truncated. From below, we limit by the minimal historical value during the first epoch. From above we only limit it by 1 because resource exclusion [77] (that is, the mechanism of avoiding faulty hosts) might raise the reliability values above their maximal historical values.

3. Zero Knowledge Epoch: $t' \geq T_{tail}$, the instances have not yet been sent at the decision making time, and no result has yet returned. We use an average of the mean reliabilities during the Full Knowledge and the Partial Knowledge Epochs, thus incorporating old accurate data as well as updated, possibly inaccurate data. Our experiments indicate that an average of equal weights produces a good prediction for $\gamma(t')$ during this epoch.

Pareto Frontier Generation: ExPERT generates the Pareto frontier using data from the previous steps in two moves. First it samples the strategy space and analyzes the sampled strategies. Then it computes the Pareto frontier of the sampled strategies, from which the best strategy can be chosen. The sampling resolution is configurable, limited in range by the deadline used in the throughput phase. We found that focusing the resolution in the lower end of the range is more beneficial, as it accounts for the knee of the Pareto frontier, which improves with resolution.

The *ExPERT Estimator* estimates the mean makespan and cost of each sampled strategy through simulation. The *ExPERT Estimator* models $\sharp ur$ unreliable and $\lceil M_r \sharp ur \rceil$ reliable resources, each resource pool having a separate, infinite queue. For simplicity we assume the queues are First Come First Served (FCFS): from each queue, tasks are submitted according to the order in which they entered the queue, unless they are canceled before they are submitted. If one instance of a task succeeds after another is enqueued but before it is sent, the other instance is canceled. If the other instance was already sent, it is *not* aborted. For each instance sent to the unreliable pool, a random number $x \in [0, 1]$ is uniformly drawn. The instance turnaround time t solves the equation $F(t, t') = x$. If $t \geq D$, the instance is considered timed-out.

At each time-step the *ExPERT Estimator* first checks each running instance for success or timeout. Then, if a task has not yet returned a result, time T has already passed since its last instance was sent, and no instance of this task is currently enqueued, the *Estimator* enqueues one instance for this task. Finally, instances are allocated to machines. ExPERT uses the average cost and makespan of several such estimations as expectation values of the real cost and makespan.

Once all the sampled strategies are analyzed, ExPERT produces the Pareto frontier by eliminating dominated strategies from the set of sampled strate-

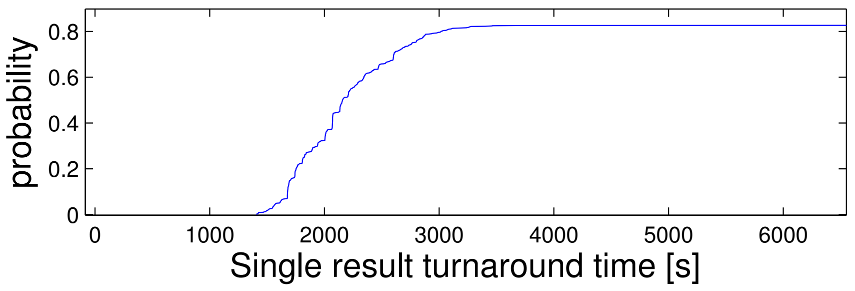


Figure 3.5: CDF of single result turnaround time. Input: Experiment 11 (Table 3.5).

gies, such that only non-dominated points remain, as illustrated in Figure 3.2. Each point on the Pareto frontier represents a Pareto-efficient strategy. Under the rational assumption of monotonicity of the utility function, all strategies that may be the best within the sampled space for *any* utility function are included in the frontier. `ExPERT` uses a hierarchical approach, which resembles the s-Pareto frontier [96]: the strategies are first divided according to their N values, since different N values account for distinct separate conceptual solutions. Then `ExPERT` merges the different frontiers. The user's utility function is not explicitly required for frontier generation—the user may withhold information about his or her utility function, and only choose a strategy from the Pareto frontier after it is presented. Furthermore, once created, the same frontier can be used by different users with different utility functions.

Decision Making: After `ExPERT` generates the Pareto frontier, `ExPERT` chooses the best strategy for the user according to her utility function; otherwise, the user programs any other algorithm to choose the best strategy for her needs. We present an example of decision making for a scientific BoT, with a task turnaround time CDF as given in Figure 3.5 and user supplied parameters as listed in Table 3.2.

We begin by showcasing the difficulty of selecting an appropriate scheduling strategy. Using an inefficient strategy (such as an $NTDM_r$ strategy that is *not* on the Pareto frontier) might waste a lot of time and money. For our example, Figure 3.6 displays only some of the sampled strategies and the resulting Pareto frontier (the depiction of the explored strategy space was

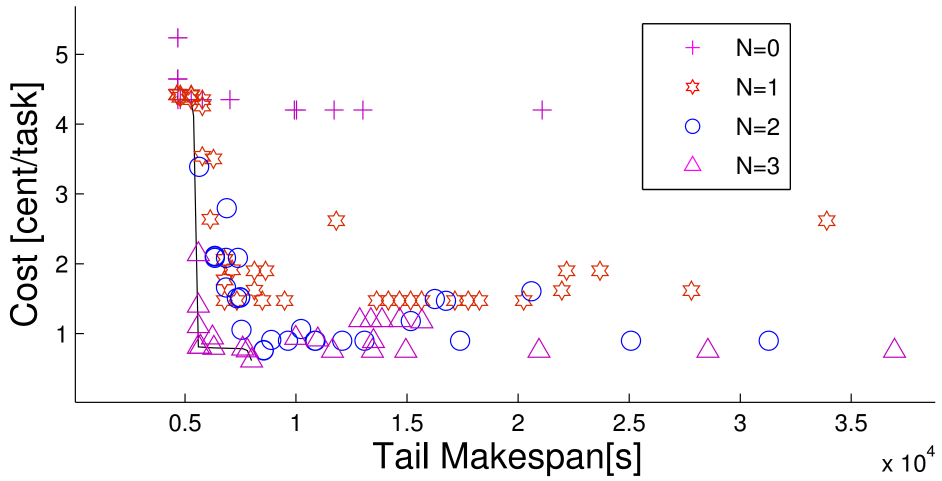


Figure 3.6: Pareto frontier and sampled strategies. Input: Experiment 11 (Table 3.5).

diluted for clarity.) Here, using the Pareto frontier can save the user from paying an inefficient cost of $4\frac{cent}{task}$ using $N = 0$ (no replication), instead of an efficient cost of under $1\frac{cent}{task}$ (4 times better) when using $N = 3$. Furthermore, a user who chooses $N = 1$ and is willing to pay $2\frac{cent}{task}$ may obtain a poor makespan of over 25,000s (the top right-most hexagram symbol in Figure 3.6). In contrast, `ExPERT` recommends a strategy based on using $N = 3$, which leads to a makespan around 5,000s (5 times better) and a cost of under $1\frac{cent}{task}$ (the triangle symbol at the "knee" of the continuous curve in Figure 3.6).

We next illustrate how `ExPERT` assists the user's decision process. Figure 3.7 depicts the Pareto frontier in terms of cost and makespan. `ExPERT` marks the frontier for several strategies, which are best for some simple user preferences such as 'minimize tail phase makespan', 'minimize cost', 'minimize tail-phase-makespan × cost', and 'work within a budget' or 'finish in time'. If the user supplies `ExPERT` with a different utility function, `ExPERT` also finds the best strategy for it. A user who does not provide a utility function can choose one of the Pareto-efficient strategies presented at this stage. The Pareto frontier is discrete (we draw the connecting line for vi-

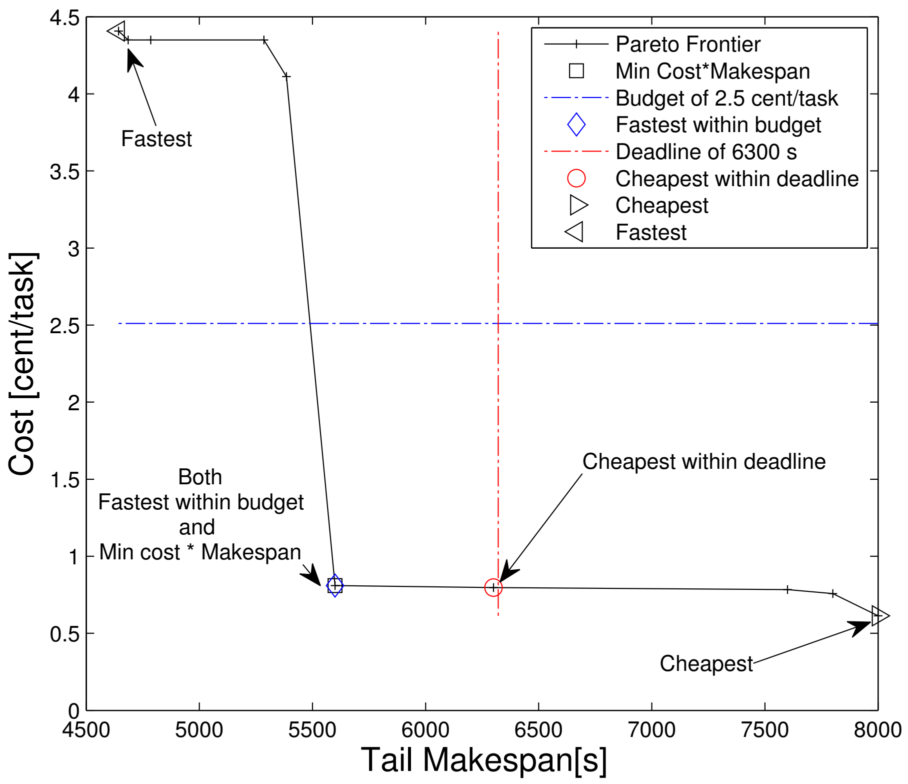


Figure 3.7: Pareto frontier and examples of best points for various user utility functions. Input: Experiment 11 (Table 3.5).

sual purposes only), so only the discrete points on it have attached input parameters. For a higher-density frontier, that is, a frontier that renders the connecting line in Figure 3.7, a higher-density sampling of the search space is required. However, even a low sampling resolution closely approaches the extreme strategies (the cheapest and the fastest).

The strategy is now chosen in terms of cost and makespan. To finalize the process, `ExPERT` presents the user with the parameters N, T, D and M_r, which define the chosen strategy. Those parameters are passed to the user's scheduler and are used to run the user's tasks.

Table 3.2: Values for user-defined parameters

Item	Value
T_{ur}	Mean CPU time of successful instances on unreliable pool (2,066 seconds for Experiment 11)
T_r	For real/simulated experiment comparison: mean CPU time over reliable instances. Otherwise: T_{ur}.
C_{ur}	$\frac{1}{3600}\frac{cent}{second} = 10\frac{cent}{KWH} \cdot 100W$
C_r	$\frac{34}{3600}\frac{cent}{second}$: EC2's m1.large on-demand rate

3.6 The Experimental Setup

In this section we present our experimental setup. To evaluate `ExPERT` in a variety of scenarios yet within our budget, we ran a series of real-world experiments and augmented the results with simulated experiments. The simulator was created by re-using a prototype implementation of the ***ExPERT** Estimator*; our simulations can be seen therefore as emulations of the `ExPERT` process. We validated the simulator's accuracy by comparing simulation results with results obtained through real-world experiments performed on different combinations of unreliable and reliable pools, including grids, self-owned machines, and Amazon EC2. To validate the simulator, we used various BoTs which perform genetic linkage analysis, a statistical method used by geneticists to determine the location of disease-related mutations on the chromosome. The BoTs, which are a characteristic workload (real full applications) for the superlink-online system [126], are characterized in Table 3.3. In pure simulation experiments we used the CDF shown in Figure 3.5.

Experimental Environments: The real-world experiments were conducted using GridBoT [125], which provides a unified front-end to multiple grids and clouds. GridBoT interprets a language for encoding scheduling and replication strategies on the basis of run-time data, to simultaneously execute the BoTs in multiple pools. GridBoT relies on BOINC, so it is based on weak connectivity.

To implement the limit to the CPU time consumed by a task instance, we used the BOINC parameter `rsc_fpops_bound`, which poses a limitation on the number of flops a host may dedicate to any a task instance. Since

Table 3.3: Workloads with T, D strategy parameters and throughput phase statistics. WL denotes Workload index. WM is an execution environment from Table 3.4.

WL	♯Tasks	T[s]	D[s]	CPU time on WM[s]		
				Average	Min.	Max.
WL1	820	2,500	4,000	1,597	1,019	3,558
WL2	820	1,700	4,000	1,597	1,019	3,558
WL3	3276	5,000	8,000	1,911	1,484	6,435
WL4	3276	3,000	5,000	2,232	1,643	4,517
WL5	615	4,000	6,000	878	1,571	4,947
WL6	615	4,000	4,000	729	1,512	3,534
WL7	615	2,500	4,000	987	1,542	3,250

this parameter only approximates the limit, we manually verified that task instances never continued beyond D.

The simulation-based experiments used the same discrete event-based *ExPERT Estimator* we developed for building the Pareto frontier. Although we considered using a grid simulator [28, 30, 67], ultimately we decided to build our own simulation environment. Our simulations are specifically tailored for running ExPERT and have a simple, trace-based setup. More importantly, as far as we know, no other simulator has been validated for the scheduling strategies and environments investigated in this work. For comparison, we augmented the $NTDM_r$ strategies already implemented in the *Estimator* with several static strategies described below.

The user-specified parameters used in our experiments are summarized in Table 3.2. To estimate C_{ur} we used the characteristic power difference between an active and idle state according to AMD's ACP metric [11]. We multiplied those power differences for Opteron processors [11] by two, to allow for cooling system power, reaching a range of 52W-157W; hence we use 100W here.

The resource pools are detailed in Table 3.4. Each experiment used one unreliable resource combination (one row) and at most one reliable resource. Experiments 1-6 used old resource exclusion data, thus choosing more reliable machines from the unreliable pools. In experiments 7-13 this data was deleted at the beginning of each experiment, thus allowing any machine in

Table 3.4: Real resource pools used in our experiments

Reliable	Properties
Tech	20 self-owned CPUs in the Technion
EC2	20 m1.large Amazon EC2 cloud instances

Unreliable	Properties
WM	UW-Madison Condor pool. Utilizes preemption. `http://www.cs.wisc.edu/condor/uwcs`
OSG	Open Science Grid. Does not preempt. `http://www.opensciencegrid.org`
OSG+WM	Combined pool, half $\sharp ur$ from each
WM+EC2	Combined pool, 20 EC2 + 200 WM
WM+Tech	Combined pool, 20 Tech + 200 WM

the unreliable pool to serve the BoT.

Static Scheduling Strategies: Without a tool such as `ExPERT`, users (e.g., GridBoT users) have resorted to static strategies. A static strategy is pre-set before the BoT starts, and does not require further computations during the BoT's run-time. Unless otherwise stated, during the throughput phase these strategies are "no replication" ($N = \infty, T = D = 4T_{ur}$) and the reliable pool is idle. Although some of these strategies are $NTDM_r$ strategies, they are not necessarily Pareto-efficient. We compare them to Pareto efficient strategies found by `ExPERT` in the next section. The static strategies are:

AR: All to Reliable: use only reliable machines for the duration of the BoT. This is a fast strategy when there are many fast reliable machines and the reliability of the unreliable machines is low.

TRR: all Tail Replicated to Reliable: at T_{tail}, replicate all remaining tasks to the reliable pool. This is an $NTDM_r$ strategy ($N = 0$,$T = 0$,$Mr = M_r^{max}$).

TR: all Tail to Reliable: at T_{tail}, enqueue every *timed out* tail task to the reliable pool. This is an $NTDM_r$ strategy ($N = 0, T = D, Mr = M_r^{max}$).

AUR: All to UnReliable, no replication: use the default throughput phase strategy during the tail phase. This is the cheapest option for a cheap unreliable system. This is an $NTDM_r$ strategy ($N = \infty, T = D$).

B=7.5: Budget of \$7.5 for a BoT of 150 tasks ($\frac{2}{3}\frac{cent}{task}$): replicate all remaining tasks on the reliable pool once the estimated cost of the replication is within the remaining budget. Until then, use the default throughput phase strategy.

CN∞: Combine resources, no replication: deploy tasks from the unreliable queue on the reliable pool if the unreliable pool is fully utilized. This is a common way of using the cloud, supplementing self-owned machines with cloud machines when the regular machines are busy.

CN1T0: Combine resources, replicate at tail with $N = 1$, $T = 0$: utilize all resources only during the throughput phase. At T_{tail}, replicate: for each remaining task, enqueue a reliable instance.

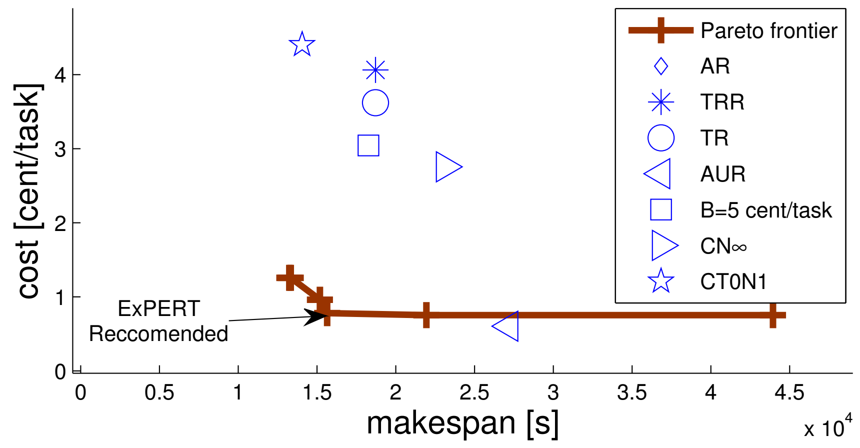
3.7 The Experimental Results

We begin by evaluating $NTDM_r$ Pareto frontiers by comparing them to the static strategies introduced in Section 3.6. We proceed to demonstrate the importance of M_r as a strategy parameter in Section 3.7. We then validate the ***ExPERT** Estimator* logic in Section 3.7 and discuss the time it takes to run ExPERT in Section 3.7.

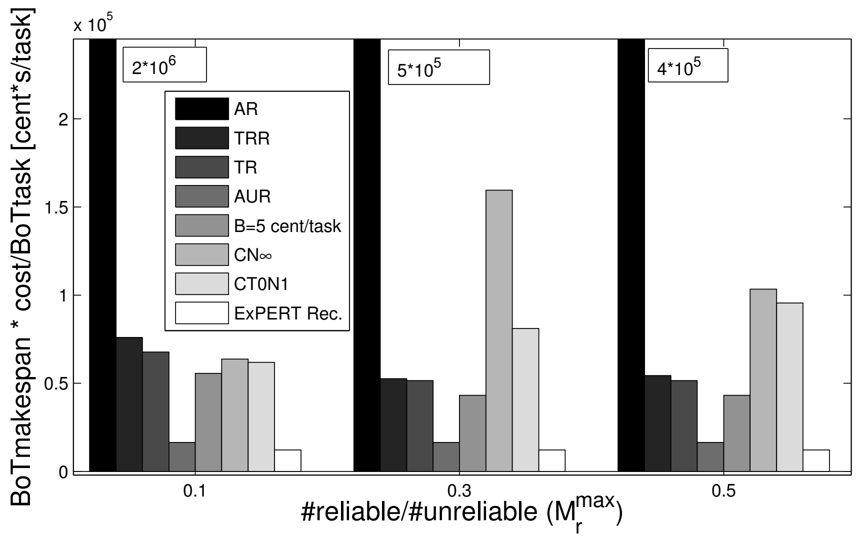
ExPERT vs. Static Scheduling Strategies: To evaluate the benefits of using $NTDM_r$ Pareto-efficient strategies, we compare them with the seven static scheduling strategies. The comparison is performed for a BoT of 150 tasks, with 50 machines in the unreliable resource pool. The Pareto frontier is obtained by sampling the strategy space in the range $N = 0 \ldots 3$, $M_r = 0.02 \ldots M_r^{max}$, and $0 \leq T \leq D \leq 4T_{ur}$. T, D were evenly sampled within their range at 5 different values each. M_r was sampled by at most 7 values, listed in Figure 3.9.

We first compare the makespan and cost of the static strategies to the Pareto frontier on a system where $M_r^{max} = 0.1$, and depict the results in Figure 3.8(a). **The Pareto frontier found by ExPERT dominates all the tested static strategies except AUR**; that is, for *any* utility function, for each tested static strategy except AUR, ExPERT recommends at least one $NTDM_r$ strategy that improves both metrics. For example, ExPERT finds *several* strategies that dominate the commonly-used $CN\infty$ strategy. One such strategy is:

ExPERT recommended ($N = 3, T = T_{ur}, D = 2T_{ur}, M_r = 0.02$): send $N = 3$ instances to the unreliable pool during the tail phase, with timeout set



(a) Performance of strategies on the Pareto frontier vs. that of static strategies, for $M_r^{max} = 0.1$. Strategy AR at (makespan around 70,000s, cost=22$\frac{cent}{task}$) is not shown.



(b) Makespan-cost product for static and *ExPERT recommended* strategies, for $M_r^{max} = 0.1, 0.3, 0.5$. Bars for strategy AR are truncated; their height appears beside them. Smaller values are better.

Figure 3.8: Static strategies compared to Pareto-efficient $NTDM_r$ strategies. Input: Experiment 11.

to occur after twice the average task time ($D = 2T_{ur}$). Send each subsequent instance after the average task time ($T = T_{ur}$) from the sending of the prior instance had passed. Use only one ($\sharp ur = 50$, $50 \times M_r = 1$) reliable machine at a time.

This strategy, which is located in Figure 3.8(a) at the "knee" of the Pareto frontier, yields a makespan of 15,640s for the cost of $0.78\frac{cent}{task}$, **cutting 72% of $CN\infty$'s cost and 33% of its makespan.** This strategy does not dominate AUR, by definition the cheapest strategy. Nonetheless, several strategies found by `ExPERT` on the Pareto frontier lead to much better makespan than AUR, with only a small increase in cost.

The dominance of the $NTDM_r$ Pareto frontier demonstrates the power of Pareto-efficient scheduling over static strategies. The frontier's dominance is not a direct consequence of the way it is built, which only guarantees that it will dominate the $NTDM_r$ strategies in the sampled space. **The fact that the $NTDM_r$ Pareto frontier dominates the static strategies implies that $NTDM_r$ is a good scheduling model: the efficient strategies the user looks for can be expressed as points in the sampled $NTDM_r$ space.**

Next, we focus on the performance of the strategies in terms of a specific utility function: *minimize tail-phase-makespan* $\times$ *cost per task*. We compare the utility obtained by the user when the scheduling strategy is *`ExPERT` recommended* or one of the seven static scheduling strategies. Figure 3.8(b) depicts the results of this comparison. *`ExPERT` recommended* is 25% better than the second-best performer, AUR, 72%-78% better than the third-best performer, and several orders of magnitude better than the worst performer, AR. We conclude that *`ExPERT` recommended* delivers significantly better utility than *all* the tested static strategies and outperforms (dominates) all these strategies except AUR.

Each static strategy might be tailored for a special scenario and a utility function. However, as Figure 3.8(b) demonstrates, using `ExPERT` to search the strategy space for that special scenario will provide the user with the best strategy in the search space, for a small computational cost (see below).

Impact of M_r `ExPERT`'s Performance: M_r provides a bound on the number of concurrently used reliable resources (see Section 3.4). We now demonstrate the benefit of elasticity, justifying the model decision which allows M_r to be a scheduling strategy parameter rather than a system con-

Table 3.5: Experimental parameters. WL denotes workload according to Table 3.3. N is the $NTDM_r$ parameter. $\sharp ur$ is an estimate for the effective size of the unreliable pool. ur and r denote choice of pools according to Table 3.4. The strategy in Experiment 5 is *CN∞: Combine resources, no replication* (which is not an $NTDM_r$ strategy).

Experiment Parameters					
No.	WL	N	$\sharp ur$	ur	r
1	WL1	0	202	WM	Tech
2	WL1	2	199	WM	Tech
3	WL6	∞	200+20	WM+Tech	-
4	WL3	0	206	WM	Tech
5	WL6	∞	200+20	WM+EC2	-
6	WL5	∞	201	WM	-
7	WL1	0	208	WM	Tech
8	WL2	1	208	WM	Tech
9	WL1	0	251	OSG+WM	Tech
10	WL7	0	208	WM	EC2
11	WL1	0	200	OSG	Tech
12	WL1	0	200	WM	Tech
13	WL4	0	204	WM	Tech

stant. We consider $M_r = 0.02 \ldots 0.50$, which means that reliable resources are less than 50% of the resources available to the user.

First we demonstrate why users need to be able to set M_r as a parameter of their scheduling strategy. To this end, we compare the Pareto frontiers created by fixing M_r; we depict in Figure 3.9 seven such frontiers. As shown by the figure, high M_r values allow a wide range of makespan values overall, but low M_r values can only lead to relatively longer makespans. For example, the Pareto frontier for $M_r = 0.02$ starts at a tail makespan of over 5,500s, which is 25% larger than the makespans achievable when $M_r \geq 0.30$. We also observe that, for the same achieved makespan, lower M_r values lead in general to lower cost. We conclude that **to find Pareto-efficient $NTDM_r$ strategies, M_r should *not* be fixed in advance, but set in accordance with the desired makespan.**

We investigate next the impact of M_r in the execution of the BoT on the resources provided by the reliable pool. For each Pareto-efficient strategy

Table 3.6: Experimental results. γ denotes the average reliability of the unreliable pool. RI denotes the number of task instances sent to the reliable pool. TMS and C denote tail phase makespan and cost per task. ΔTMS and ΔC denote deviation of simulated values from real ones. Averages are computed over the absolute values of the results.

	Measured in Real Experiment				Simulated Experiment Deviation			
					Offline [%]		Online [%]	
No.	γ	RI	TMS[s]	C$\left[\frac{cent}{task}\right]$	$\frac{\Delta TMS}{TMS}$	$\frac{\Delta C}{C}$	$\frac{\Delta TMS}{TMS}$	$\frac{\Delta C}{C}$
1	0.995	50	6,908	1.60	8	3	35	33
2	0.983	0	3,704	39	21	-4	8	-4
3	0.981	0	6,005	41	1	-4	4	-4
4	0.974	49	10,487	1.10	2	2	-56	-32
5	0.970	41	6,113	1.48	37	-2	29	-2
6	0.942	0	6,394	0.42	3	-4	-40	-4
7	0.864	77	10,130	2.38	3	2	32	26
8	0.857	16	4,162	0.88	19	15	-37	-10
9	0.853	108	14,029	3.28	7	0	-1	-4
10	0.844	118	11,761	3.67	-14	-35	-7	-28
11	0.827	89	11,656	2.86	8	1	-7	-7
12	0.788	107	12,869	3.09	-9	-13	-2	-5
13	0.746	100	20,239	1.54	-3	-7	-7	-10
Avg.	0.894	58	9,574	1.78	10	7	20	13

operating in this environment, we compare three operational metrics: the strategy parameter M_r, the maximal number of reliable resources used during the BoT's run (denoted *used* M_r), and the maximal size of the reliable queue built during the run. Figure 3.10 depicts the results of this comparison. We find that for most Pareto-efficient strategies, the number of used resources from the reliable pool, *used* M_r, is equal to the number of resources set through the strategy parameter, M_r. This is because, during the BoT's tail phase, tasks sometimes wait in the queue to the reliable pool, as seen in Figure 3.10: the maximal length of the reliable queue is almost never zero; that is, the queue is almost always used. The right-most point on the M_r and *used* M_r curves, for which the values of M_r and *used* M_r are different, is the exception. We explain this by an intrinsic load-balancing property of the $NDTM_r$ systems: when the reliable pool queue is long, slow unreliable

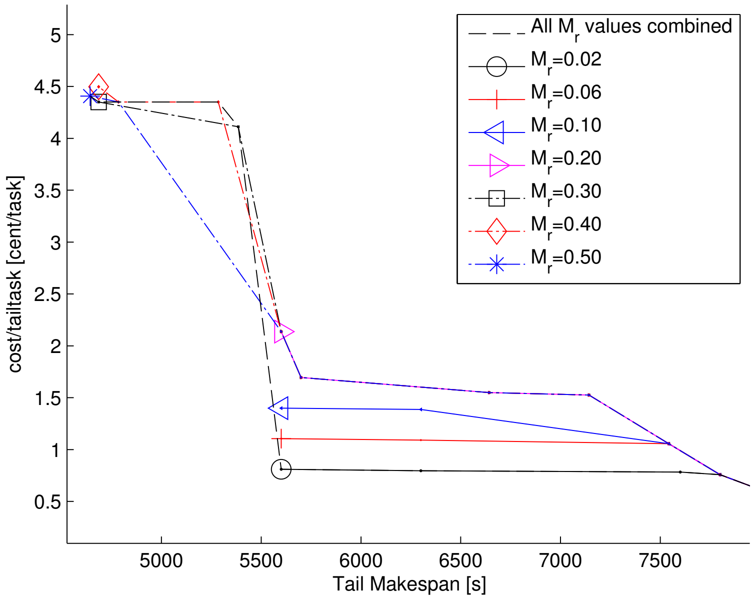


Figure 3.9: Pareto frontiers obtained for various M_r values. The topmost efficient point of each Pareto frontier is highlighted. Pareto frontiers of high M_r values have a wider makespan range. Low M_r values yield lower costs.

instances return results before the reliable instance is sent, which leads to the reliable instance being canceled and its cost being spared.

Simulator Validation: We conducted 13 large-scale experiments to validate the simulator and the ***ExPERT** Estimator*. In each experiment, we applied a single strategy to specific workload and resource pools. Since the simulations include a random component, we ensured statistical confidence by comparing the performance metrics (tail phase makespan, cost per task) of each real experiment with mean values of 10 simulated experiments. We compared real and simulated performance metrics for both the offline and the online models (defined in Section 3.5). The experiments are listed by decreasing order of average reliability in Tables 3.5 and 3.6.

On average, performance metrics of the offline simulations, which use full knowledge of the unreliable pool's reliability $\gamma(t')$, deviate from real experimental values by 7% and 10% for cost and tail phase makespan, respectively. The on-line simulations, which extrapolate $\gamma(t')$ during the tail phase, devi-

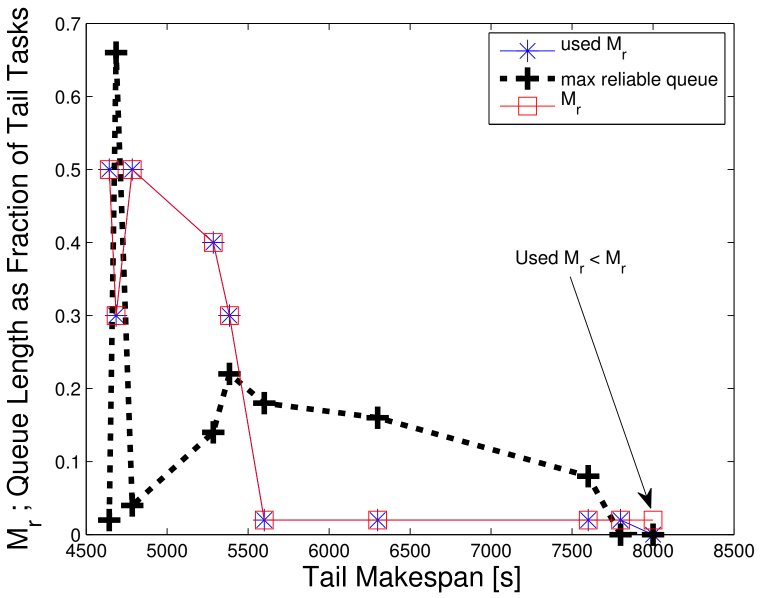


Figure 3.10: Reliable pool use by efficient strategies.

ated from real experimental values by twice as much.

We identify four main causes for these deviations. First, the simulator provides expectation values of performance metrics. In contrast, a real experiment is a single, unreproducible sample. When a large number of tasks are replicated during the tail phase, the performance metrics tend to be close to the mean values of the simulated experiments. When the opposite occurs, for example in Experiment 2, where only four very long instances were sent after T_{tail}, the makespan observed in the real environment is further from the offline simulation. Second, the simulator assumes $F_s(t)$ does not depend on t' and attributes all CDF changes to $\gamma(t')$. However, in real experiments $F_s(t)$ does depend on t', due to resource exclusion [77] policies and a varying task length distribution. Third, `ExPERT` assumes it is never informed of failures before the deadline D. In real experiments, some machines do inform about failures and are replaced. Fourth, in real experiments, the effective size of the unreliable pool is variable and hard to measure. Hence, T_{tail} is detected when there are more free hosts than remaining tasks. The tasks remaining at this time are denoted *tail tasks*. This may be a transient state, before the actual start of the tail phase. In simulated experiments, the number of machines is fixed. T_{tail} is detected when the number of remaining tasks equals the number of tail tasks in the real experiment.

`ExPERT` **Runtime** The computational cost of running our `ExPERT` prototype, in the resolution used throughout this paper, is several minutes to sample the strategy space and analyze it, on an Intel(R) Core(TM)2 Duo CPU P8400 @ 2.26GHz. The space sampling is composed of dozens of single strategy simulations, each lasting several seconds. We consider a runtime in the order of minutes, appropriate for BoTs of hundreds of tasks that are the focus of this work. `ExPERT`'s runtime may be further shortened at the expense of accuracy, by reducing the number of random repetitions from over 10 to just 1. Similarly, flexibility may be traded with time by changing the resolution in which the search space is sampled. Gradually building the Pareto frontier using evolutionary multi-objective optimization algorithms can also reduce `ExPERT`'s runtime.

3.8 Conclusion

We addressed one of the main problems facing scientists who rely on Bags-of-Tasks (BoTs) in mixtures of computational environments such as grids and clouds: the lack of tools for selecting Pareto-efficient scheduling strategies for general user-defined utility functions. For any user-provided utility function, `ExPERT` finds the best strategy in a large, sampled strategy space. `ExPERT` can achieve a 72% cost reduction and a 33% shorter makespan compared with commonly-used static scheduling strategies. For a utility function of *makespan* $\times$ *cost*, `ExPERT` provided a strategy which was 25% better than the second-best, and 72-78% better than the third best strategy. These improvements stem from `ExPERT`'s ability to explore a large strategy space under minimal user guidance, and to automatically adapt to the varying reliability, cost, and speed of resources. They also show that the $NTDM_r$ strategy space is large enough to provide considerable flexibility in both makespan and cost. `ExPERT`'s predictive accuracy has been verified through experiments on real grids and a real cloud. The Pareto frontier created by `ExPERT` provides users with an understanding of the cost-makespan trade-offs of executing their BoTs.

Chapter 4

Deconstructing Amazon EC2 Spot Instance Pricing

4.1 abstract

Cloud providers possessing large quantities of spare capacity must either incentivize clients to purchase it or suffer losses. Amazon is the first cloud provider to address this challenge, by allowing clients to bid on spare capacity and by granting resources to bidders while their bids exceed a periodically changing spot price. Amazon publicizes the spot price but does not disclose how it is determined.

By analyzing the spot price histories of Amazon's EC2 cloud, we reverse engineer how prices are set and construct a model that generates prices consistent with existing price traces. Our findings suggest that usually prices are not market-driven, as sometimes previously assumed. Rather, they are likely to be generated most of the time at random from within a tight price range via a dynamic hidden reserve price mechanism. Our model could help clients make informed bids, cloud providers design profitable systems, and researchers design pricing algorithms.

4.2 Introduction

Unsold cloud capacity is wasted capacity, so cloud providers would naturally like to sell it. They would especially like to sell the capacity of machines which cannot be turned off and have higher overhead expenses. Clients might be enticed to purchase this capacity if they are provided with enough incentive, notably, a cheaper price. In late 2009, Amazon was the first cloud provider to attempt to provide such an incentive by announcing its *spot instances* pricing system. "Spot Instances [...] allow customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current Spot Price. The Spot Price changes periodically based on supply and demand, and customers whose bids exceeds it gain access to the available Spot Instances" [9]. With this system, Amazon motivates purchasing cheaper capacity while ensuring it can continuously act in its best interest by maintaining control over the spot price. **Section 4.3** summarizes the publicly available information regarding Amazon's pricing system.

Amazon does not disclose its underlying pricing policies. Despite much interest from outside Amazon [34, 69, 95, 117, 148], its spot pricing scheme has not, until now, been deciphered. The only information Amazon does reveal is its temporal spot prices, which must be publicized to make the pricing system work. While Amazon provides only its most recent price history, interested parties record and accumulate all the data ever published by Amazon, making it available on the Web [88,136]. We leverage the resulting trace files for this study. The trace files, along with the methodology we employ to use them, are described in **Section 4.4**.

Knowing how a leading cloud provider like Amazon prices its unused capacity is of potential interest to both cloud providers and cloud clients. Understanding the considerations, policies, and mechanisms involved may allow other providers to better compete and to utilize their own unused capacity more effectively. Clients can likewise exploit this knowledge to optimize their bids, to predict how long their spot instances would be able to run, and to reason about when to purchase cheaper or costlier capacity.

Motivated by these benefits, we attempt in **Sections 4.5–4.6** to uncover how Amazon prices its unused EC2 capacity. We construct a spare capacity pricing model and present evidence suggesting that prices are typically *not* determined according to Amazon's public definition of the spot pricing system as quoted above. Rather, the evidence suggests that spot prices are usually drawn from a tight, fixed range of prices, reflecting a random reserve price that is not driven by supply and demand. (A *reserve price* is a hidden price below which bids are ignored.) Consequently, published spot prices reveal little about actual, real-life client bids; studies that assume otherwise (in particular [32, 157]) are, in our view, misguided.We speculate that Amazon utilizes such a price range because its spare capacity usually exceeds the demand.

In **Section 4.7** we put our model to the test by conducting pricing simulations (based on cloud and grid workloads) and by showing their results to be consistent with EC2 price traces. We then discuss the possible benefits of using dynamic reserve price systems (such as the one we believe is used by Amazon) in **Section 4.8**. Finally, we offer some concluding remarks in **Section 4.10**.

4.3 Pricing Cloud Instances

Amazon's EC2 clients rent virtual machines called *instances*, such that each instance has a *type* describing its computational resources as follows: m1.small, m1.large and m1.xlarge denote, respectively, small, large, and extra-large "-standard" instances; m2.xlarge, m2.2xlarge, and m2.4xlarge denote, respectively, extra-large, double extra-large, and quadruple extra-large "high memory" instances; and c1.medium and c1.xlarge denote, respectively, medium and extra-large "high CPU" instances.

An instance is rented within a geographical *region*. We use data from four EC2 regions: us-east, us-west, eu-west and ap-southeast, which correspond to Amazon's data centers in Virginia, California, Ireland, and Singapore.

Amazon offers three purchasing models, all requiring a fee from a few cents to a few dollars, per hour, per running instance. The models provide different assurances regarding when instances can be launched and terminated. Paying a yearly fee (of hundreds to thousands of dollars) buys clients the ability to launch one *reserved instance* whenever they wish. Clients may instead choose to forgo the yearly fee and attempt to purchase an *on-demand instance* when they need it, at a higher hourly fee and with no guarantee that launching will be possible at any given time. Both reserved and on-demand instances remain active until terminated by the client.

The third, cheapest purchasing model provides no guarantee regarding either launch or termination time. When placing a request for a *spot instance*, clients bid the maximum hourly price they are willing to pay for running it (called *declared price* or *bid*). The request is granted if the bid is higher than the spot price; otherwise it waits. Periodically, Amazon publishes a new *spot price* and launches all waiting instance requests with a maximum price exceeding this value; the instances will run until clients terminate them or the spot price increases above their maximum price. All running spot instances incur a uniform hourly charge, which is the current spot price. The charge is in full hours, unless the instance was terminated due to a spot price change, in which case the last fraction of an hour is free of charge.

In this work, we assume that instances with bids equal to the spot price are treated similarly to instances with bids higher than the spot price.

4.4 Methodology

Trace Files We analyze 64 ($= 8 \times 4 \times 2$) spot price trace files associated with the 8 aforementioned instance types, the 4 aforementioned regions, and 2 operating systems (Linux and Windows). The traces were collected by Lossen [88] and Vermeersch [136]. They start as early as 30 November 2009 (traces for region ap-southeast are only available from the end of April 2010). In this paper, unless otherwise stated, we use data accumulated until 13 July 2010.

Availability We define the availability of a declared price as the fraction of the time in which the spot price was equal to or lower than that declared price. Formally, a *persistent request* is a series of requests for an instance that is immediately re-requested every time it is terminated due to the spot price rising above its bid. Given a declared price D, we define D's *availability* to be the time fraction in which a persistently requested instance would run if D is its declared price. Formally, let H be a spot price trace file, and let T_b and T_e be the beginning and end of a time interval within H. The availability of D within H during $[T_b, T_e]$ is:

$$availability^H(D) \mid_{[T_b,T_e]} = \frac{T^H_{b \to e}(D)}{T_e - T_b}$$

, where $T^H_{b \to e}(D)$ denotes the time between T_b and T_e during which the spot price was lower than or equal to D. The availability of price D reflects the probability that spot instances with this bid would be immediately launched when requested at some uniformly random time within $[T_b, T_e]$.

4.5 Evidence for Artificial Pricing Intervention

4.5.1 Market-Driven Auctions

Amazon's description of "How Spot Instances Work" [9] gives the impression that spot prices are set through a uniform price, sealed-bid, market-driven auction. "Uniform price" means all bidders pay the same price. "Sealed-bid" means bids are unknown to other bidders. "Market-driven" means the spot price is set according to the clients' bids. Many auctions fit this description. One example of such an auction is an $(N+1)^{th}$ price auction of multiple goods, with retroactive supply limitation (after clients bid). Of course, Amazon could be using some other market-driven mechanism consistent with their description.

In an $(N+1)^{th}$ price auction of multiple goods, each client bids for a single good (i.e., a spot instance). The provider sorts the bids and chooses the top N bidders. The provider is free to set the number of sold goods N, as long as N does not exceed the available capacity. The provider may set N up-front as the available capacity, but it may also retroactively further restrict N after receiving the bids, to maximize revenue. The provider sets the uniform price to the price declared by the highest bidder who *did not* win the auction (bidder number $N+1$) and publishes it. The top N winning bidders pay the published price and their instances start running. In this case, the published price is a price bid by an actual client.

The provider may also decide to ignore bids below a hidden *reserve price* or below a publicly known *minimal price*, to prevent the goods from being sold cheaply, or to give the impression of increased demand.

We conjecture that usually, contrary to impressions conveyed by Amazon [9] and assumptions made by researchers [32, 157], the spot price is set according to a constantly changing reserve price, disregarding client bids. In other words, most of the time the spot price is *not* market-driven but is set by Amazon according to an undisclosed algorithm.

4.5.2 Evidence: Availability as a Function of Price

In support of this conjecture, we analyze the relationship between an instance's declared price (how much a client would be willing to pay for it) and the resulting availability between 20 January 2010 and 13 July 2010.

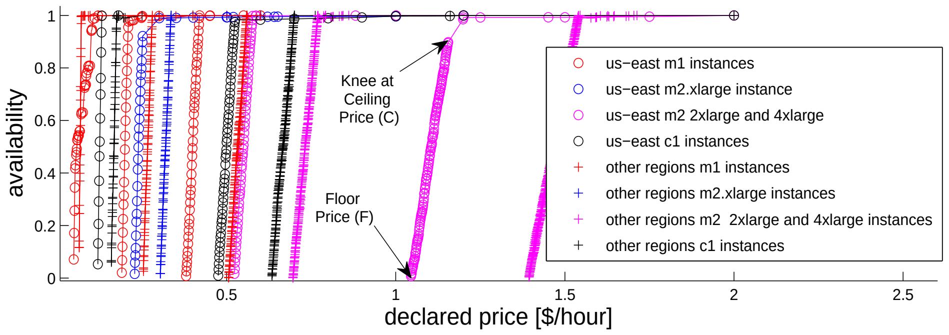


Figure 4.1: Availability of Windows-running spot instance types as a function of their declared price. The legend is multiplexed: us-west, eu-west, ap-southeast all appear in the legend as "other regions". m1.small, m1.large and m1.xlarge all appear as m1. c1.medium and c1.xlarge appear as c1.

Fig. 4.1 shows the availability of different spot instance types as a function of declared price (price-availability graphs), for all examined Windows spot instance types in all regions. Results for instances running Linux (not shown) are qualitatively similar. The prices of different resources are usually in different ranges (e.g., us-east.c1.medium's usual price range is a third of us-east.c1.xlarge's), but they all share the same functional shape: a sharp linear increase in availability, during which the price resolution is 0.1 cent. The increase may last until an availability of 1.0 is reached, or end with a *knee* at a high availability (usually above 0.95). A knee is a sharp change in the graph's slope; it is usually accompanied by a sharp decrease in the graph's resolution. Above the knee, the availability grows with declared price, but at a slower, varying rate.

Fig. 4.2 shows *normalized* price-availability graphs for Linux: each spot price is divided by the price of a similar on-demand instance. We see that Linux types can be classified by region. Each of the two region classes has a distinct normalized price range in which the availability's dependency on the price is linear. One class contains us-east, and the other class contains the other regions.

Fig. 4.3 shows the data presented in Fig. 4.1 as normalized price-availability graphs. As in Fig. 4.2, different types can be classified by region: us-east or

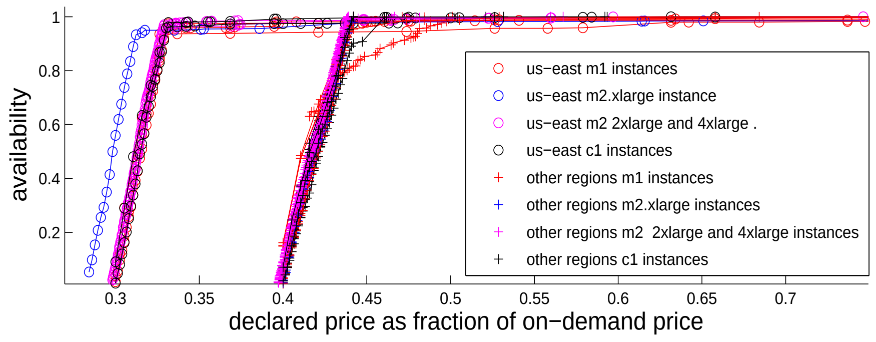


Figure 4.2: Availability of Linux-running spot instance types as a function of their normalized declared price. The declared price is divided by the price of a similar on-demand instance. The legend is multiplexed as in Fig. 4.1. All 32 curves are shown in full, but most of them overlap.

all other regions. Not as in Fig. 4.2, different types have different normalized prices within a class, and the relative price difference between any type pair is the same in each class. The `m1.small` type, indicated in Fig. 4.3 by an arrow, has a particularly low knee, with an availability of 0.45. The normalized ranges of the `us-east.windows.c1` instances, whose absolute prices so differed in Fig. 4.1, are now identical. Figs. 4.1–4.3 show that availability strongly depends on declared price for all regions and all instance types, and that this dependency has a typical recurring shape, which can be explained by assuming that Amazon uses the same mechanism to set the price in different regions. The particular shape of the dependency could be explained in one of two ways: either Amazon's spot prices reflect real client bids and the shaped dependency occurs naturally, or the spot prices are the result of a dynamic hidden reserve price algorithm, of which the shaped dependency is an artifact.

Let us first consider the assumption that the shaped dependency occurs naturally due to real client bids. The differences between absolute price ranges of the same type in different regions (Fig. 4.1) show that different regions experience different supply and demand conditions. This means that uncoordinated client bids for different types and regions would have to naturally and independently create *all* of the following macro-economic phenom-

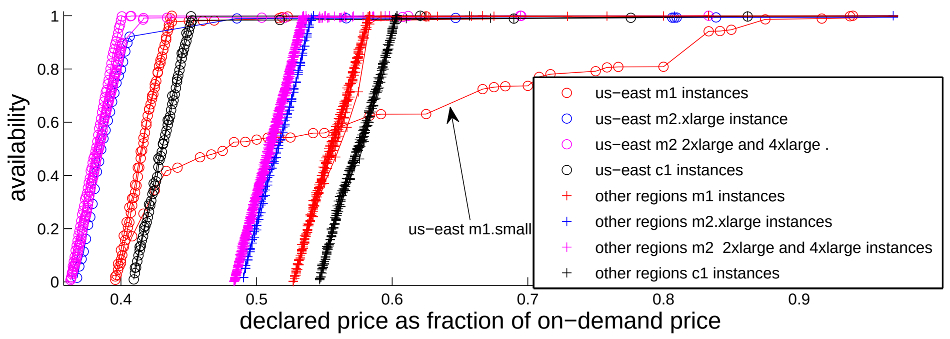


Figure 4.3: Availability of Windows-running spot instance types as a function of their normalized declared price. The declared price is divided by the price of a similar on-demand instance. The legend is multiplexed as in Fig. 4.1. All the data is shown in full, but many of the curves overlap. us-east.windows.m1.small is indicated by an arrow.

ena: (1) normalized prices turning out identical for various Linux types but different for Windows types; (2) a rigid linear connection between availability and price that turns out to be identical for different types and regions; (3) a distinct region having a normalized price range different than all the rest (which turn out to have identical ranges); and (4) normalized prices for Windows instances which differ from one another by identical amounts in each of the two region classes, creating the same pattern for both.

If real client bids shape these dependencies, then real clients bid below the knee. If that is indeed the case, then many spot instance clients present irrational micro-economic behavior. As many researchers working from client perspectives have found [34, 95, 117, 148], bidding below the knee is not cost-effective because it will subject the instance to frequent unavailability events. Slightly raising the bid, however, will result in the instance being almost completely protected. Bidding below the knee is not only irrational in light of low availability and a long waiting time for the price to drop below the bid, but also in light of the short continuous intervals in which the low prices are valid, as noted especially by Chohan et al. [34]. Such short intervals might prohibit the successful completion of a task, forcing the client to repeat it (and possibly pay for some of the useless compute time).

For the sake of argument, let us also consider the possibility that causing the macro-economic phenomena described above is the declared goal of a conspiring group of clients. They have already reverse-engineered Amazon's algorithm and submit coordinated bids that cause the aforementioned phenomena. Since the phenomena we describe can be seen in all 64 analyzed traces, these clients would have to consume a sizable share of the spot instance supply in all 64 resources, bidding low bids (which would then eventually become the spot price). This would systematically limit the supply available to the many different legitimate clients known to use EC2 spot instances. If the legitimate clients then bid higher than the spot price (which is usually below the knee), the spot price would rise, terminating the conspiring clients' instances. From this point on, the conspiring clients' effect on the spot price would be limited. Furthermore, customers must have Amazon's approval for the purchase of spot instances beyond the first one hundred. Hence, we consider this explanation highly unlikely.

Our hypothesis: We consider it unlikely that all four phenomena could have resulted from Amazon setting the price solely on the basis of client bids. We therefore lean towards the hypothesis that Amazon uses a dynamic algorithm, independent of client bids, to set a reserve price for the auction, that the auction's result is usually identical to the reserve price, and that the prices Amazon announces are therefore usually not market-driven. Both the simulation results presented in Section 4.7 and Occam's razor—preferring the simplest explanation—support this hypothesis.

If our hypothesis is correct, then all four phenomena listed above are easily explained by a dynamic reserve price algorithm which gets as input prices normalized by respective on-demand prices. This input is different for the us-east region, for different sets of types, and for different operating systems.

4.5.3 Dynamic Random Reserve Price

First we will characterize the requirements for a dynamic reserve price algorithm that will be consistent with the published EC2 price traces. Then we will construct such an algorithm, and propose it as a candidate for the algorithm behind the EC2 pricing.

We contend that the dynamic reserve price algorithm gets as input a *floor* *price* F and a *ceiling price* C for each spot instance type, with the floor and ceiling prices expressed as fractions of the on-demand price. The floor price is the minimal price, exemplified in Fig. 4.1 for the us-east.m2.2xlarge and us-east.m2.4xlarge types. The ceiling price is the price corresponding to the knee in the graph (shown in the same figure), or the maximal price if no knee exists. We refer to this price range, in which availability is a linear function of the price, as the pricing *band*. The algorithm dynamically changes the reserve price such that there is a linear relation between availability and prices in the floor–ceiling range. It guarantees that the reserve price never drops below the floor, which reflects Amazon's minimal-reserve price for spot instances, nor rises above the ceiling.

We deconstruct the reserve price algorithm using traces from April–July 2010, when the spot price in eight ap-southeast.windows instance types almost always stayed within the artificial band. We matched the price changes in those traces (denoted by Δ) with an $AR(1)$ (auto-regressive) process. We found a good match (i.e., negligible coefficients of higher orders a_i for $i > 1$) to the following process:

$$\Delta_i = -a_1 \Delta_{i-1} + \epsilon(\sigma), \tag{4.1}$$

where $a_1 = 0.7$ and $\epsilon(\sigma)$ is white noise with a standard deviation σ. Let F, C denote the floor and ceiling of the artificial band, respectively. We matched σ with a value of $0.39(C - F)$. These parameters fit all the analyzed types well, except for m1.small, which matched different values ($a_1 = 0.5, \sigma = 0.5(C-F)$). The standard deviations are given in Fig. 4.4. This close fit—the same parameters characterizing the randomness of several different traces—is consistent with our hypothesis that the prices are usually set by an artificial algorithm. The reason for m1.small's deviation is yet to be found.

On the basis of this analysis, we construct the $AR(1)$ *reserve price algorithm*: The process is initialized with a reserve price of $P_0 = F$ and a price change of $\Delta_0 = 0.1(F-C)$. The following prices are defined as $P_i = P_{i-1}+\Delta_i$, where $\Delta_i = -0.7 \cdot \Delta_{i-1} + \epsilon(0.39 \cdot (C - F))$. The process is truncated to the $[F, C]$ range by regenerating the white noise component while P_i is outside the $[F, C]$ range or identical to P_{i-1}. All prices are rounded to one-tenth of a cent, as done by Amazon during 2010.

To evaluate whether the trace produced by the truncated $AR(1)$ process

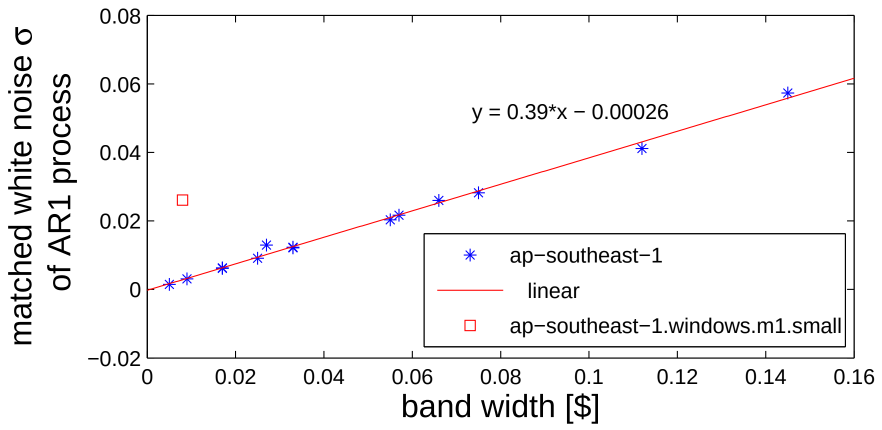


Figure 4.4: Standard deviation of the white noise of the matched $AR(1)$ process as a function of artificial price-band width

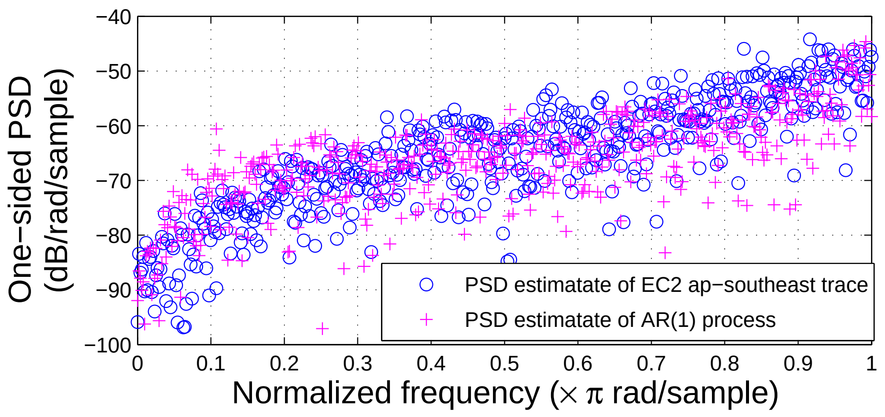


Figure 4.5: Power spectral density (periodogram) estimate of an EC2 price trace, and our derived $AR(1)$ price trace

matches the original EC2 trace, we compare their periodograms (normalized Fourier transforms) in Fig. 4.5. The periodogram comparison verifies that we captured the original signal's frequencies correctly, and not just the average time in each price. The match shows that our reverse-engineered reserve price algorithm is consistent with Amazon's.

The consistency of an $AR(1)$ process with the EC2 traces does not indicate the dynamics which create it. If this consistency can be explained mostly by natural fluctuations, then we would expect to see at least a weekly cycle. A daily cycle is unlikely, since clients all over the world use the same machines.

To search for a weekly cycle, we analyzed the utilization of memory in three IaaS pay-as-you-go cloud traces (described in detail in Section 4.7.2) and the price in the ap-southeast.linux traces. We computed each day's mean value (price or utilization for spot trace or cloud, respectively), taking into consideration the duration for which the value was valid. Each day's mean value was normalized by the mean value over the week to which it belongs. This local normalization is especially important when computing mean utilization, since over the years of the trace, both the capacity and the utilization increased. The autocorrelation of cloud utilization for three cloud workloads is depicted in Fig. 4.6(a). All three clouds have a significant weekly cycle, sometimes with a pattern lasting for several weeks. The weekly cycle is expressed by strong, positive autocorrelation coefficients for lags of 7, 14, 21 and even 28 days. In addition, there is strong positive autocorrelation with the previous day, meaning today's utilization is a good prediction for tomorrow. The confidence bounds are low (0.081, 0.084, 0.068) and slightly different from one.

Knowing autocorrelation can be expected in a cloud, let us turn to analyze the spot price autocorrelation that is depicted in Fig. 4.6(b). The confidence bounds are larger than in the cloud load graphs, and are identical to the fifth digit (0.2097). None of the eight price traces has any weekly cycle or any significant long range correlation. This finding agrees with Wee [148], who shows that none of the 64 EC2 traces we used exhibit notable weekly or daily patterns. Moreover, the one-day autocorrelation coefficients are negative for all the traces, meaning today's price is a bad prediction for tomorrow. Thus, the process generating the traces cannot be explained mostly by natural fluctuations.

Let us consider the hypothesis that natural dynamics account for a small part of the trace: usually the spot price is the dynamic reserve price, but sometimes the spot price rises above the reserve price due to market considerations. This would mean that usually the price traces reflect the reserve price only, but sometimes the prices are bids declared by real clients. This scenario is unlikely because, as discussed earlier, bidding inside the band is not cost-effective. Nonetheless, we check this hypothesis by analyzing mean trace prices, with the alternate hypothesis that natural dynamics account for no part of the trace. If the alternate hypothesis is true, the mean trace price should be the mean of the truncated $AR(1)$ process, which is a symmetric process: the middle of the band. If natural dynamics sometimes raise the price above the reserve price, the mean price should be higher than the middle of the band. However, for the 8 ap-southeast.windows traces we tested here, the mean price was *lower* than the middle of the band by up to 2%.

We conclude that the impact of natural dynamics on the price traces in the band range is statistically insignificant. The spot price within the band is almost always determined solely by the $AR(1)$ process, i.e., is equal to the reserve price. Since we assume prices above the band usually result from natural dynamics, we need to estimate how frequently the prices are above the band. On average, over the 64 traces we analyzed, prices were above the band 2% of the time. We conclude that during the other 98% of the time, prices are mainly determined by an artificial $AR(1)$ reserve price algorithm and hardly ever represent real client bids.

4.6 Pricing Epochs

To statistically analyze spot price histories, it would be erroneous to assume that the same pricing model applies to all the data in the history trace. Rather, each trace is divided to contiguous epochs associated with different pricing policies. We show here that our main traces are divided into three epochs as depicted in Fig. 4.7. Since the pricing mechanism changes notably and qualitatively between epochs, data regarding these epochs should be separated if an associated statistical analysis is to be sound. Accordingly, for the purpose of evaluating the effectiveness of client algorithms, strategies, and predictions, the data from a (single) epoch of interest should be used.

The *first epoch* starts, according to our analysis, as early as 30 November 2009 and ends on 14 December 2009, the date on which Amazon announced the availability of spot instances. During this time, instances were unknown to the general public. Hence, the population which undertook any bidding during the first epoch was smaller than the general public, of nearly constant size, and possibly had additional information regarding the internals of the pricing mechanism at that time.

The *second epoch* begins with the public announcement on 14 December 2009. It ends with a pricing mechanism change around 8 January 2010, when minimal spot prices abruptly change in most instances (usually decrease, though Fig. 4.7 demonstrates an increase). It is characterized by long intervals of constant low prices.

The *third epoch* begins on 20 January 2010. Instance types and regions began to change minimal price around January 8th, but we define the beginning of the epoch as the date in which the last one (eu-west.linux.m2.2xlarge) reached a new minimal price. Due to (1) the gradual move to the new minimal values and to (2) a bug in the pricing mechanism that was fixed in mid-January 2010 [10], we choose to disregard data from the transition period between the second and third epochs.

Additional epoch-defining dates are dates when the price-change timing algorithm was changed, e.g., 20 July 2010 and 9 February 2011 for the us-east region (see Section 4.7).

These abrupt time-correlated changes in many regions and instance types further support our hypothesis, since prices are likely to undergo abrupt changes at exactly the same time either when the market is efficient (which is not the case here, since absolute prices in Fig. 4.1 are not leveled) or when the prices are artificial.

4.7 Spot Price Simulation

To get a better feel for the validity of our hypothesis, we simulated two spot pricing systems, representing the dynamic hidden reserve price hypothesis and the alternate hypothesis of a constant reserve price. Both systems are based on a sealed-bid $(N+1)^{th}$ price auction with a reserve price with retroactive supply limitation, as described in Section 4.5.1. The simulator structure is described in Section 4.7.1.

In both systems we set the on-demand price to 1. In the constant reserve price system we set the reserve price to 0.4. In the AR(1) reserve price system we set the reserve prices according to the reserve price algorithm defined in Section 4.5.3, with a band of $[0.4, 0.45]$. To run the simulation, we need to know not only what the new reserve price should be, but also when it should be changed. To this end, we deconstructed the price change timing, as explained in Section 4.7.4.

To fully model a spot pricing system, three input data sets or models are required: for available machine supply, for instance demand, and for client bids. We modeled the machine supply as a fixed-size, because spot instances are a good practice for a quick-launch buffer: those machines which need to be kept running, in case an on-demand or reserved instance is requested. We do not expect spot-instance machine supply to represent the full variation of on-demand and reserved instance demand. We used real grid and cloud traces for instance demand (Section 4.7.2), and three client bid models (Section 4.7.3). The simulation results are presented in Section 4.7.5.

4.7.1 Simulator Event-Driven Loop

We created a trace-based event-driven simulator, where events are: (1) instance submission and termination and (2) price changes (due to a scheduled change or to a waiting instance with a bid higher than the spot price). We ran the grid trace-driven simulation on 70 CPUs, according to the number of CPUS in the trace. Since CPU was over-committed on the cloud traces but physical memory was not, we defined each cloud's capacity as the maximal amount of memory concurrently used in its trace. We ended the simulation when the last input-trace job had been submitted.

4.7.2 Workload Modeling

We fed the simulation with tasks with run-times in the range of 10 minutes to 24 hours, taken from several large system traces. According to Iosup et al. [5], a typical EC2 instance overhead is two minutes. We deem clients unlikely to wait two minutes and pay for a full hour for an activity which lasts only a few minutes, so we only used tasks longer than 10 minutes. We assume spot instances are usually used for relatively short-running instances, with longer running instances more likely to be deployed on more stable offerings such as on-demand and reserved instances. Thus we omitted tasks longer than 24 hours. We discuss the task length cut-off point in Section 4.7.5.

We used traces from one grid and three clouds. In the simulation, each task was interpreted as a single instance, submitted at the same time and requiring the same run-time as in the original trace to complete. The grid trace is 20K tasks from the LPC-EGEE workload[1]. LPC-EGEE is characterized by tasks which are small in comparison to the capacity of the cluster, allowing for elasticity.

We also used traces of three pay-as-you-go IaaS clouds[2]. These clouds were partitions of IBM's RC2 cloud [116]. The partitions used different underlying physical resources and hypervisors, and it was up to the user to choose the partition. The traces were taken from 2 April 2009 to 22 August 2011 (2.5 years). During this time, the capacity of the partitions changed with demand, reaching concurrent use of thousands of CPUs (6522, 1420, and 845 for clouds 1, 2, and 3, respectively) and thousands of gigabytes of memory (10175, 1996, and 2386 for the respective clouds). Clients of these clouds were charged 2-3 cents per hour per GB for running instances. In addition, creating an instance for the first time cost 20 cents.

The workloads of these clouds are characterized by significantly longer runtimes than grid jobs: only half the cloud instances take less than 24 hours, while 98% of the tasks last less than a day on grids (LPC-EGEE, GRID5000[3]) and parallel systems (LANL CM-5[4], SDSC-Paragon[5]) that we evaluated, as seen in Fig. 4.8. Many cloud instances last months and even years. Furthermore, the clouds exhibit longer and stronger inter-arrival time correlation than typical grids, as seen in Fig. 4.9. The autocorrelations of their inter-arrival times is even larger than those of parallel systems, even though both system types are only accessible to a limited set of clients.

4.7.3 Customer Bid Modeling

Due to the lack of information on the distribution of real client bids (since we argue that Amazon's price traces supply little information of this type), we compare several bidding models, and verify that the qualitative results

[1]Graciously provided by Emanuel Medernach [99], via the Parallel workload archive [48], file LPC-EGEE-2004-1.2-cln.swf.

[2]Graciously provided by Mariusz Sabath.

[3]Graciously provided by Franck Cappello, via the Grid Workloads Archive [65], file grid5000_clean_trace.swf.

[4]Graciously provided by Curt Canada, via the Parallel workload archive, file LANL-CM5-1994-3.1-cln.swf.

[5]Graciously provided by Reagan Moore and Allen Downey, via the Parallel workload archive, file SDSC-Par-1995-2.1-cln.swf.

are insensitive to the bid modeling. All the distributions were adjusted to uniform minimal and on-demand prices.

The first model is a Pareto distribution (a widely applicable economic distribution [83, 130]) with a minimal value of 0.4, and a Pareto index of 2, a reasonable value for income distribution [130]. The second model is the normal distribution $\mathcal{N}(0.7, 0.3^2)$, truncated at 0.4. The third is a linear mapping from runtimes to $(0.4, 1]$, which reflects client aversion to having long-running instances terminated.

4.7.4 Price Change Timing

Price changes in the simulation are triggered according to the cumulative distribution function (CDF) of intervals between them, collected during January–July 2010, and given in Fig. 4.10 (solid line). This period was characterized by quiet times—prices never changed before 60 minutes or between 90 and 120 minutes since the previous price change. It is interesting to note that such quiet times can be monetized by clients to gain free computation power with a probability of about 25%, by submitting an instance with a bid of the current spot price 31 minutes after a price change. The instance would then have a 50% possibility of undergoing another price change within 30-60 minutes. If that change is a price increase, the instance would be terminated, and the client would gain, on average, 45 minutes of free computation. Clients do not exploit this loophole in our simulation.

Fig. 4.10 also presents the evolution of the timing of price changes for the us-east region. The next algorithm (in place from July 2010 until 8 Feb 2011) allowed for a quiet hour after a price change. The following one (starting 9 Feb 2011) matches an exponential distribution with a 1.5 hour rate parameter, with five quiet minutes. This almost memory-less algorithm prevents abuse of the timing algorithm. A similar evolution of the algorithm took place in other regions on different dates. On Linux instances in regions other than us-east, an interim algorithm was used between the second and third algorithms, such that the quiet hour was abolished before the transfer to the algorithm of 2011.

4.7.5 Simulation Results

Simulation results in terms of price-availability graphs are presented in Fig. 4.7.5-4.7.5, for different input traces, bid models and price setting mechanisms. The functions of simulations with the $AR(1)$ reserve price feature a linear segment and a knee in high availability, as do the availability functions of EC2 during the third epoch, which are shown in Figs. 4.1, 4.2, and 4.3. The constant reserve price functions do not exhibit this behavior. Rather, they are jittery, like the high price regime of the us-east.windows.m1.small graph in Fig. 4.3, and the second epoch graph in Fig. 4.15. These results are not sensitive to our of choices of bidding model and workload.

Furthermore, the availability of the reserve price in the constant reserve price simulations is high (0.2-0.9), as it is in the second epoch (0.63 in Fig. 4.15). In contrast, the availability of the minimal price in the $AR(1)$ reserve price simulations and in the third epoch tends to zero as the number of discrete prices within the band grows.

These macro-economic qualitative differences can be better understood by focusing on three classes of availability graphs that resemble one another and do not present straight lines: (1) the constant minimal reserve price simulations, (2) the second epoch, and (3) the high regime of the third epoch (in particular us-east.windows.m1.small). Since the graphs of the first class reflect client bids, the qualitative resemblance suggests that the last two also reflect client bids: during the second epoch, a constant reserve price algorithm is used, and the demand for us-east.windows.m1.small exceeds the supply so much that excess demand is no longer masked by the dynamic reserve price.

To investigate the effect of truncating long running instances from the traces (mainly from the cloud traces), we ran the AR(1) simulations with different maximal run-time truncations (1 day, 2 days and 100 days). As can be seen from the price-availability graphs (Fig. 4.16), raising the upper truncation point of the trace lowers the availability at the knee. The truncation does not affect the important features discussed earlier (the straight line and the existence of the knee). From the EC2 traces we learn that the knee is usually high (above 0.9, with the exception of some m1.small instances). Thus we conclude that the workload of Amazon's EC2 spot instances is consistent with relatively short instances, and that our choice of truncating the traces at 24 hours is reasonable.

We consider these simulation results a constructive indication that most prices in the EC2 traces during the third epoch are set using an $AR(1)$ reserve price, which is not market driven. The simulation results also suggest that Amazon set prices via a market-driven auction with a constant reserve price during the second epoch (December, 2009 until January, 2010), and that prices above the band are market-driven. (In the traces we studied, prices are above the band only 2% of the time on average.)

4.8 Dynamic Reserve Price Benefits

The dynamic $AR(1)$ reserve price mechanism has several long-term, wide-range benefits that may justify its use. Like a constant minimal or reserve price, it guarantees that on-demand instances are not completely cannibalized by spot instances. Yet it also allows the provider to sell instances on machines which would otherwise run idle, to provide elasticity for the fixed price instances. Spot instances, which can be quickly evacuated, still reduce the costs associated with idle servers, with no real harm to the main offering of on-demand instances.

Using a hidden reserve price allows the provider to change the reserve price with no obligation to inform the clients, an obligation which cannot be avoided when using a minimal price. A dynamic reserve price is better than a constant minimal price, because it maintains an impression of constant change, thus preventing clients from becoming complacent. It forces them to either bid higher than the band or tolerate sudden unavailability. It also serves to occasionally clear queues of low bids within the band, a purpose that is not served by a constant reserve price that is equal to the ceiling price. Furthermore, Vincent [139] argues that in common value English and second price auctions, a random reserve price encourages participation, and thus the exchange of more information about the value of the goods.

A random reserve price might also serve other goals, if the public is unaware of its use. By creating an impression of false activity (demand and supply changes), the random reserve price can mask times of low demand and price inactivity, thus possibly driving up the provider's stock. A large enough band covering the spectrum of probable prices could also mask high demand and low supply, and thus help to maintain the illusion of an infinitely elastic cloud. However, if the artificial band is relatively small, as in the case of Amazon EC2 spot prices, the provider's use of an $AR(1)$ process for setting the price within the band is a strong indication of low demand.

4.9 Reexamination of Prior Work

We will now review the literature on pay-as-you-go IaaS cloud workload traces (and spot prices in particular), reexamining past conclusions in light of our results. We will also review literature on computation markets and on reserve prices, examining the implications of these works on our results.

Using Spot Price Traces for Client Strategy Evaluation Most studies that use price traces use them to evaluate client strategies. The relevance of such work to future deployment of instances needs to be re-evaluated when the nature of the traces changes (i.e., when a new epoch starts). Andrzejak, Kondo and Yi [15, 154] used data from the transition period between the second and third epoch for their evaluation. They focused on eu-west, which suffered most from this transition. In their work on migration [153], the same authors interchangeably used data from before and after the change in the price change algorithm on July 25, 2010, as did Voorsluys et al. [141], who analyzed the performance of their spot instance broker using traces from March 2010 to February 2011.

In their simulations, Mattess, Vecchiola, and Buyya [95] evaluated client strategies using an EC2 spot instance trace of the third epoch only, attributing the different trace behavior prior to January 18th, 2010 to Christmas and to the recent introduction of spot instances. Chohan et al. [34] analyzed price histories from the third epoch only, because of the pricing bug that was fixed in mid-January 2010 [10]. The bug allowed instances with prices higher than the regional spot price to be terminated due to congestion in their availability zone (which is a part of the region), while keeping the regional price low. The authors attributed the qualitative change of prices between the second and third epoch to the bug fix. However, this bug fix is unlikely to have caused the qualitative price changes we observe during January 2010, namely, the appearance of the pricing band.

Brebner and Liu [26] represented the cost of spot instances as a constant, which equals the average of several months of the price trace, but did not state the duration or length of the history they used. It is thus impossible to determine which epochs they used, and what the given average values represent.

Zhao et al. [159] and Mazzucco and Dumas [97] assumed spot instance prices are market-driven, and modeled some of them to be used as a client decision aid. These models are no longer relevant once a drastic policy change is made.

Using Spot Price Traces to Learn about the Market Zhang et al. [157] assumed Amazon uses a market-driven auction, which led them to conclude that spot price histories reflect actual client bids. On this basis they sought resource allocations to instance types which optimized the provider's revenue. Chen et al. [32], who tested provider scheduling algorithms, likewise assumed EC2 price traces represent market clearing prices. We consider these assumptions doubtful, in light of our findings that 98% of the time, on average, EC2 price traces are the reserve prices, and as such do not provide a lot of information about real client bids, nor are necessarily clearing prices.

Free Spot and Futures Markets Amazon's spot instances are not a free market. Price traces of free spot and future markets [105, 134] differ from EC2 spot price traces: they do not have a hard minimal price and are not anchored in the bottom of the price range. Rahman, Lu and Gupta [110] evaluated free spot market options using EC2 traces, and noted that the "data does not show enough fluctuations as expected in a free market."

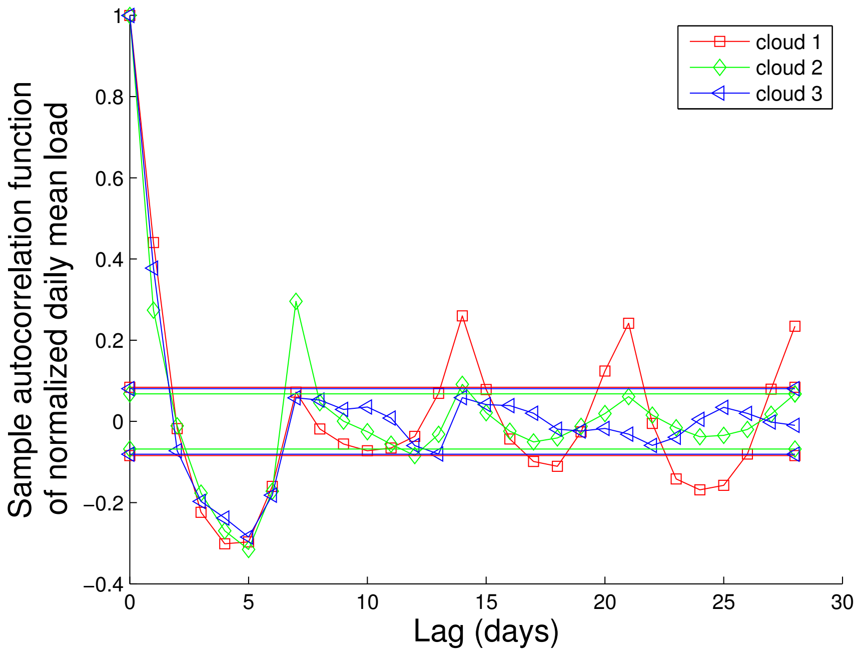
4.10 Conclusions

Amazon EC2 spot price traces provide more information about Amazon than about its clients. We have shown that during the examined period Amazon probably set spot prices using a random $AR(1)$ (hidden) reserve price. This price might have been the basis of a market-driven mechanism, in which high prices might have reflected market changes, but most low prices, within a band of prices, were usually indicative only of the dynamic reserve price.

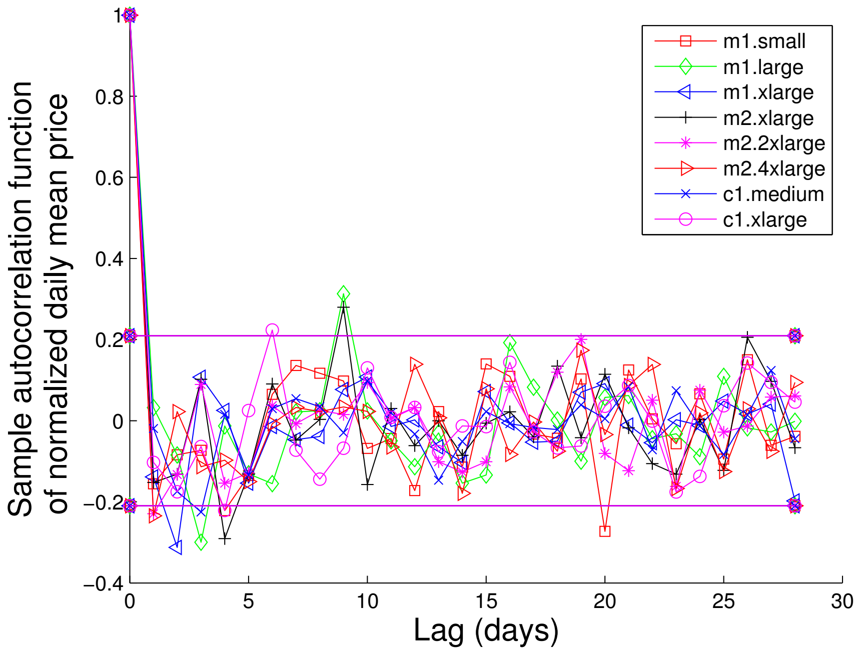
Understanding how Amazon prices its spare capacity is useful for clients, who can decide how much to bid for instances; for providers, who can learn how to build more profitable systems; and for researchers, who can differentiate between prices set by an artificial process and prices likely to have been set by real client bids. We have shown that many price trace characteristics (e.g., minimal value, band width, change timing) are artificial and might change according to Amazon's decisions. Thus, researchers should be aware of the epochs present in their traces when using those traces to model future price behavior or to evaluate client algorithm performance. We have shown that indiscriminately using Amazon's current traces to model client behavior is unfounded on average 98% of the time for the examined period.

4.11 Epilogue

Amazon's EC2 spot instance pricing mechanism underwent a radical change between the first submission of this paper and its first acceptance. Several days after its acceptance, the spot instance prices underwent another extreme change, and the pricing band disappeared from the traces altogether. For example, in the trace shown in Fig. 4.17, the spot price is constant throughout October 2011, except for a single change in the minimal price. While these radical qualitative changes are further evidence of the former prices being artificially set, the October prices are consistent with a constant minimal price auction, and are no longer consistent with an $AR(1)$ hidden reserve price.



(a) Memory utilization of three clouds



(b) Price of eight ap-southeast.linux types

Figure 4.6: Autocorrelation of mean daily values (memory utilization or prices), with respective approximate confidence bounds are displayed as horizontal lines in the same colors as the autocorrelation curves. The daily values are normalized by their week's mean value.

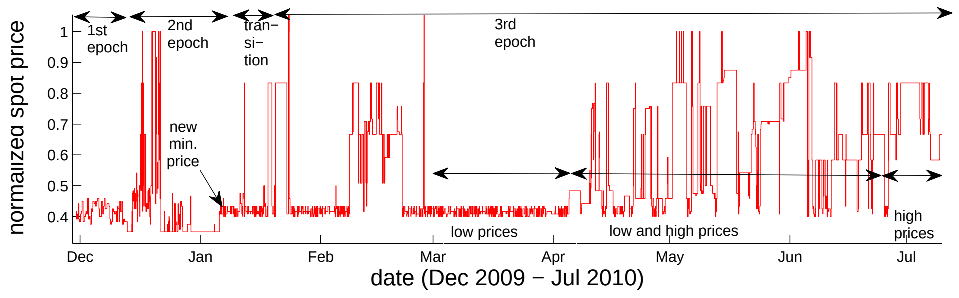


Figure 4.7: Price history for us-east.windows.m1.small. Three time epochs are shown, with a transition period between the second and third epochs. The spot price is presented as a fraction of the on-demand price for the same instance.

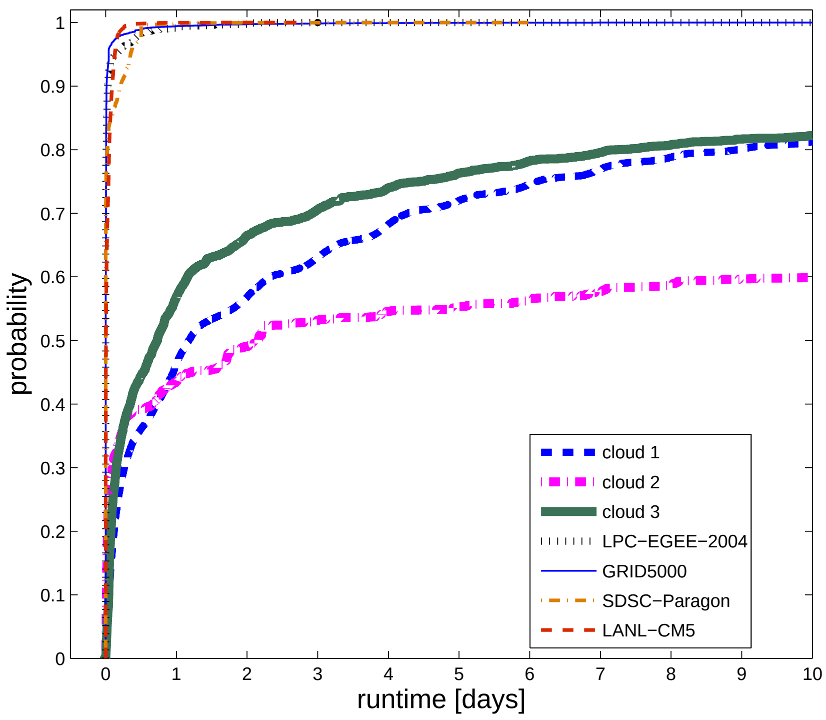


Figure 4.8: CDF of instance or task runtimes on clouds, parallel systems and grids

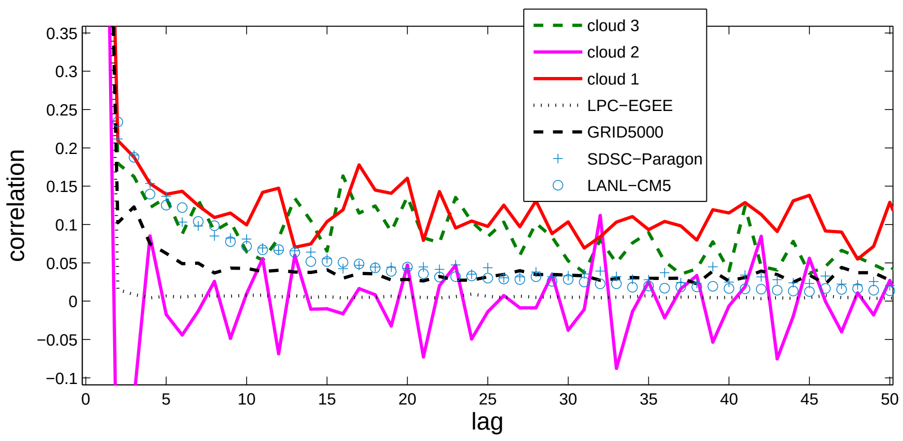


Figure 4.9: Task/instance inter-arrival time autocorrelation on clouds, parallel systems (LANL CM-5, SDSC), and grids (LPC-EGEE, GRID5000).

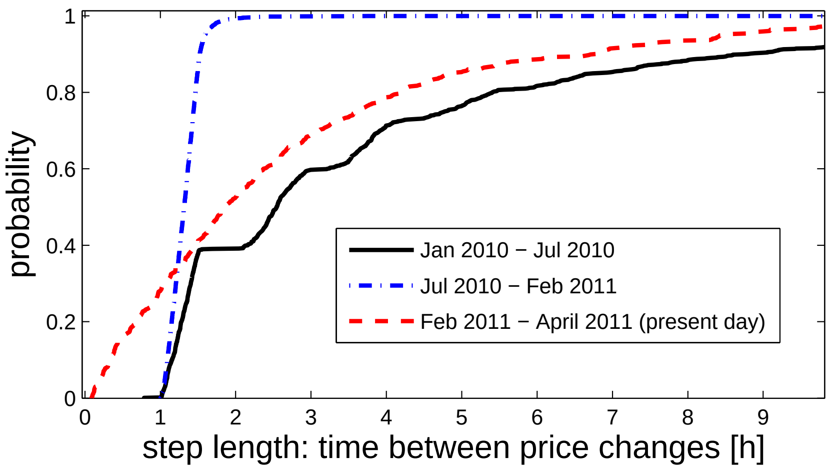


Figure 4.10: CDF of time interval between price changes for different versions of the price change scheduling algorithm. Input: us-east.linux.m1.small.

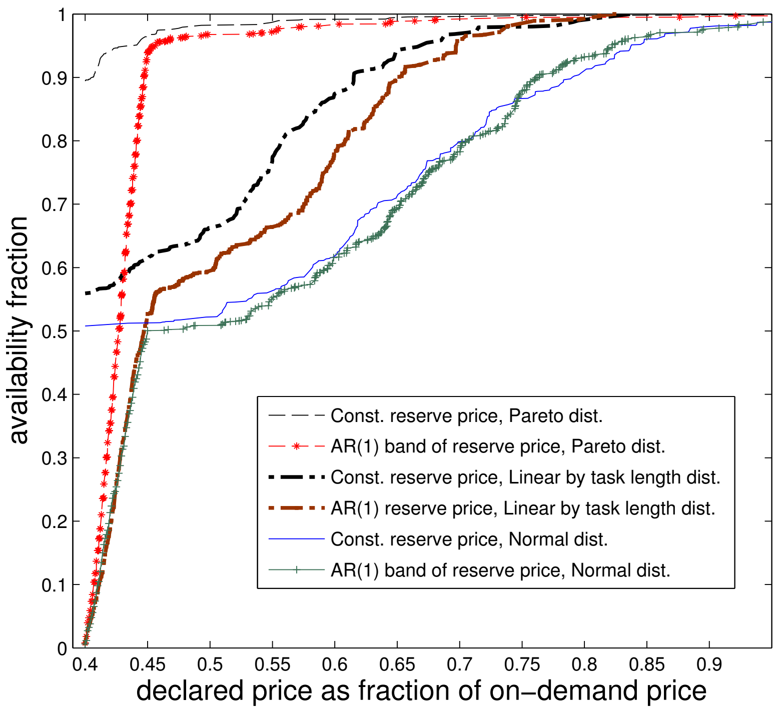


Figure 4.11: Simulation results for various bidding models, with constant and $AR(1)$ reserve price, on the basis of a grid trace (LPC-EGEE)

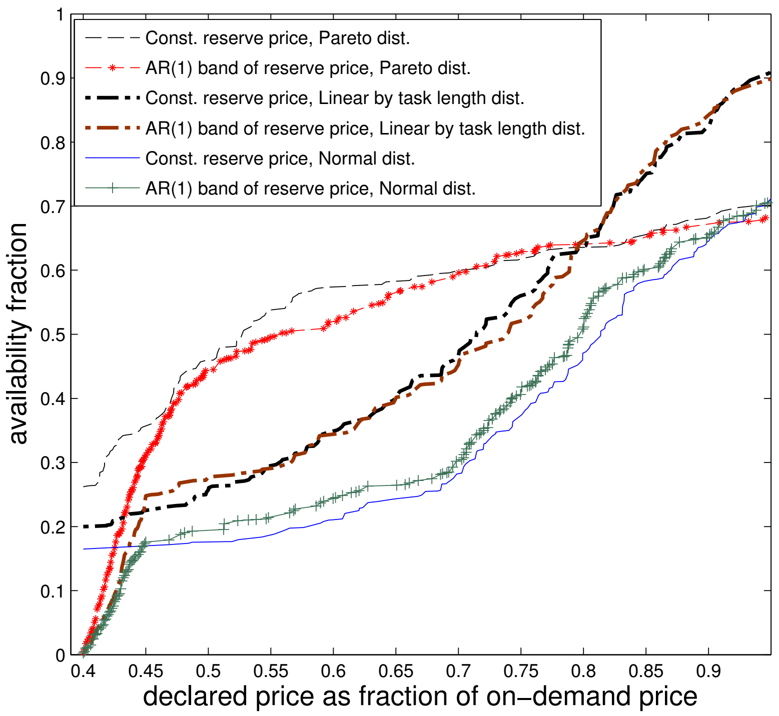


Figure 4.12: Simulation results for various bidding models, with constant and $AR(1)$ reserve price, on the basis of a trace of cloud 1

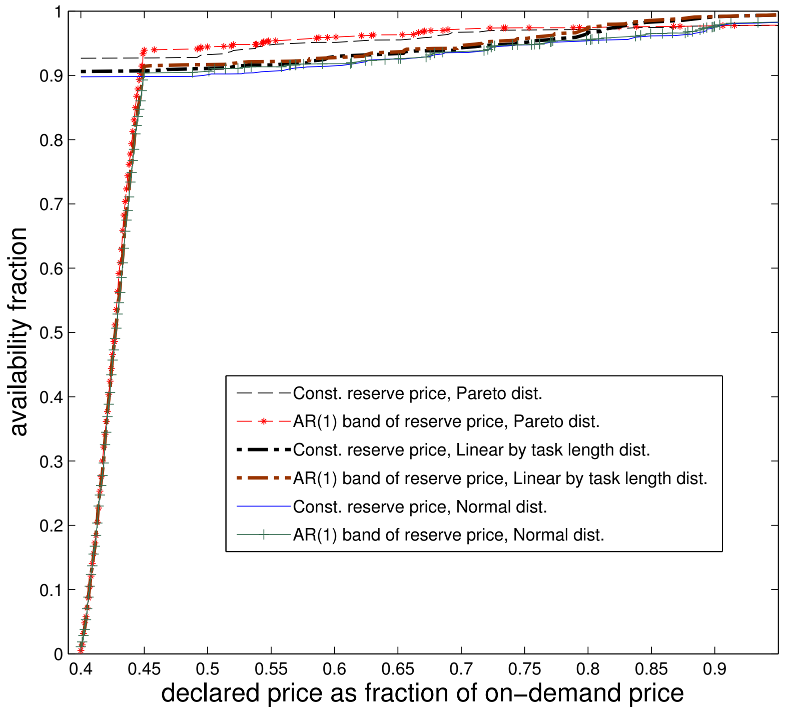


Figure 4.13: Simulation results for various bidding models, with constant and $AR(1)$ reserve price, on the basis of a trace of cloud 2

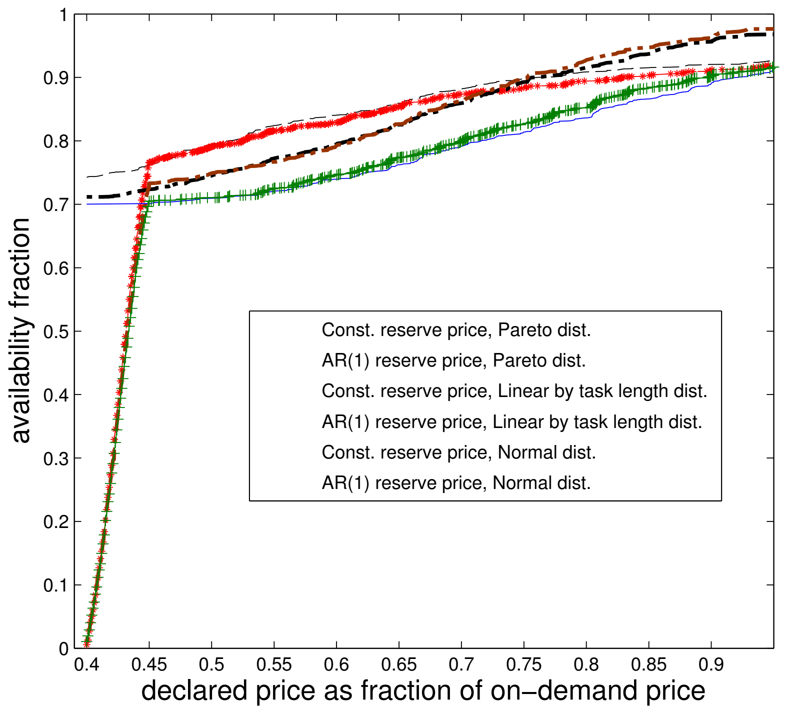


Figure 4.14: Simulation results for various bidding models, with constant and $AR(1)$ reserve price, on the basis of a trace of cloud 3

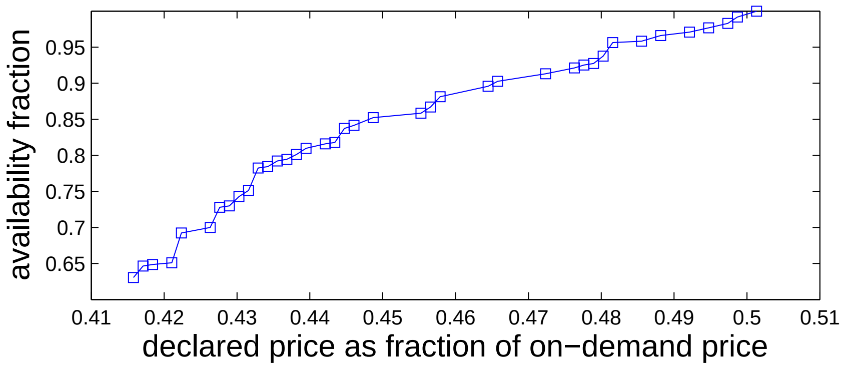


Figure 4.15: Availability as a function of the declared price during the second epoch for us-west.linux.m1.xlarge.

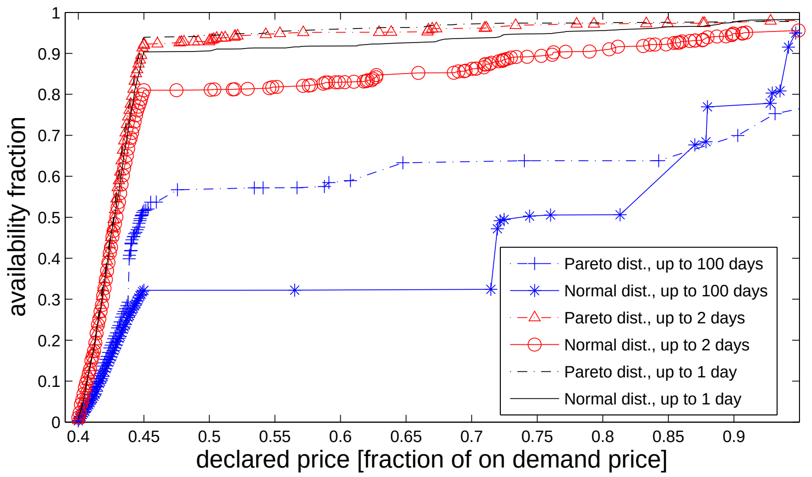


Figure 4.16: Impact of running time truncation of the cloud 2 trace on price-availability graphs for simulations with Pareto and normally distributed bids and AR(1) reserve price

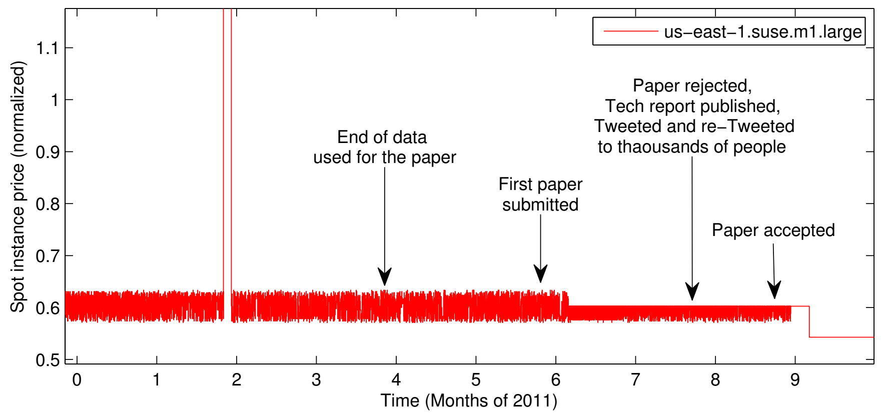


Figure 4.17: The history of this paper and the price trace of suse.m1.large on us-east during 2011

Chapter 5

The Resource-as-a-Service (RaaS) Cloud

5.1 abstract

Infrastructure-as-a-Service (IaaS) cloud providers typically sell virtual machines that bundle a fixed amount of resources, such as the core count, the memory size, and the I/O bandwidth. The resource bundles are usually unchanging throughout the lifetime of the virtual machines. We foresee that this type of rigid resource allocation will change in the near future. Instead of fixed bundles, cloud providers will increasingly sell resources individually, reprice them, and adjust their quantity every few seconds in accordance with market-driven supply-and-demand conditions; virtual machines will accordingly purchase and utilize the changing resources dynamically, while they are running. We term this nascent economic model of cloud computing the Resource-as-a-Service (RaaS) cloud, and we contend that its rise is the likely culmination of recent trends in the construction of IaaS clouds and of the economic forces operating on cloud providers and clients.

> “When the quantity of any commodity which is brought to market falls short of the effectual demand, [...] some [...] will be willing to give more.” (Adam Smith, *An Inquiry into the Nature and Causes of the Wealth of Nations*)

5.2 Introduction

Cloud computing is taking the computer world by storm. Today, Infrastructure-as-a-Service (IaaS) clouds, such as Amazon EC2, allow anyone with a credit card to tap into a seemingly unlimited fountain of computing resources by renting virtual machines for several cents or dollars per hour. According to a Forrester Research report [114], the yearly cloud computing market is expected to top $241 billion in 2020, compared to $40.7 billion in 2010, a sixfold increase. What will these 2020 clouds look like? Given the current pace of innovation in cloud computing and in other utilities such as smart grids and wireless spectra, substantial shifts are bound to occur in how providers design, operate, and sell cloud computing resources, and in how clients purchase and use those resources.

IaaS cloud providers sell fixed bundles of CPU, memory, and I/O resources packaged as server-equivalent virtual machines. We foresee that, instead, providers will continuously reprice and adjust the quantity of the individual resources with a time granularity as fine as seconds; the software stack within the virtual machines will accordingly evolve to productively operate in this dynamic, ever-changing environment. We call this new model of cloud computing the *Resource-as-a-Service (RaaS)* cloud. In a RaaS cloud, provider-governed economic mechanisms will control clients' access to resources. Hence, clients will deploy economic agents that will continuously buy and sell computing resources in accordance with the provider's current supplies and other clients' current demands.

We identify four existing trends in the operation of IaaS cloud computing platforms, that underlie the transition we foresee: the shrinking duration of rental, billing, and pricing periods (Section 5.3.1), the increasingly fine-grained resources offered for sale (Section 5.3.2), the increasingly market-driven pricing of resources (Section 5.3.3), and the provisioning of useful service level agreements (SLAs) (Section 5.3.4). We believe the economic forces operating on both providers and clients (Section 5.4) will continue pushing these trends forward. Eventually, as the trends near their culmination, these forces will unify today's IaaS cloud computing models into a single economic model of cloud computing. We call this unified model the RaaS model of cloud computing (Section 5.5). We conclude by outlining the challenges and opportunities the RaaS cloud presents (Section 5.6).

5.3 Recent IaaS Trends

5.3.1 Duration of Rent and Pricing

Before cloud computing, the average useful lifetime of a purchased server was approximately three years. With the advent of Web hosting, clients could rent a server on a monthly basis. With the introduction of on-demand EC2 instances in 2006, Amazon radically changed the time granularity of server rental, making it possible to rent a server equivalent for as little as one hour. This move was good for the provider, because, by incentivizing the clients to shut down unneeded instances, it allowed for better time-sharing of the hardware. It also benefited the clients, who no longer needed to pay for *wall clock time* they did not use, but only for *instance time* that they did use.

This trend—of renting server-equivalents for increasingly shorter time durations—is driven by economic forces that keep pushing clients to improve efficiency and minimize waste: if a partial instance-hour is billed as a full hour, you might waste up to an hour over the lifetime of every virtual machine (a *per-machine penalty*). If a partial instance-second is billed as a full second, then you will only waste up to a second over the lifetime of every virtual machine. Thus, shorter durations of rent and shorter billing units reduce client overhead and open the cloud for business for shorter workloads. Notably, low overheads encourage *horizontal elasticity*— changing the number of concurrent virtual machines—and draw clients who require this functionality to the cloud.

The trend towards shorter times is also gaining ground with regard to pricing periods. Amazon spot-instances, announced in 2009, may be repriced as often as every five minutes [2], although they bill by the price at the beginning of the hour. CloudSigma, announced in 2010, reprices its resources exactly every five minutes.[1]

New providers charge by even finer time granularity: Gridspot[2] and ProfitBricks,[3] both launched in July 2012, charge by three-minute and one-minute chunks, respectively. Google App Engine's new policy is to bill instances by the minute, with a minimum charge of 15 minutes,[4] and as of May 2013 Google Compute Engine charges by the minute with a minimum of ten minutes instead of by hours.[5]

We draw an analogy between cloud providers and phone companies, which have progressed over the years from billing landlines per several minutes to billing cell phones by the minute, and then, due to customer pressure or court orders, to billing per several seconds and even per second. Similarly, car rental (by the day) is also giving way to car sharing (by the hour), and it is recommended that wireless spectrum sharing have a shorter period base [47].

We expect this trend of shortening times to continue such that eventually, cloud providers will reprice computing resources every few seconds and charge for them by the second. Providers might compensate themselves for overheads by charging a minimal amount or using *progressive prices* (higher unit-prices for shorter rental times). Such durations are consistent with peak demands that can change over seconds when a site is "slashdotted" (linked from a high-profile Web site).[6]

5.3.2 Resource Granularity

In most IaaS clouds, clients rent a fixed bundle of compute, memory, and I/O resources. Amazon and Rackspace[7] call these bundles "instance types," GoGrid[8] calls them "server sizes," and Google Compute Engine[9] calls them "machine types." Selling resources this way provides clients with a familiar abstraction of a server-equivalent. This abstraction is starting to unravel, and in its place we see the beginnings of a new trend towards finer and finer resource granularity. In August 2012,[10] Amazon began allowing clients to dynamically change the available I/O resources for already-running instances.[11] Google App Engine charges I/O operations by the million and offers progressive network prices, which are rounded down to small base units before charging (1 byte, 1 email, 1 instance-hour).[12] CloudSigma (2010), Gridspot (2012), and ProfitBricks (2012) offer clients the ability to compose a flexible bundle from varying amounts of resources, similar to building a custom-made server out of different mixtures of resources such as CPUs, memory, and I/O devices.

Renting a fixed combination of cloud resources cannot and does not reflect the interests of clients. First, as server size is likely to continue to increase—hundreds of cores and hundreds of gigabytes of memory per server in a few years—an entire server-equivalent may be too large for some customer needs. Second, selling a fixed combination of resources is only efficient when the load customers need to handle is both known in advance and constant. As neither condition is likely, the ability to dynamically mix-and-match different amounts of compute, memory, and I/O resources benefits the clients.

We expect this trend towards finer resource granularity to continue, such that all of the major resources (compute, memory, and I/O) will be rented and charged for in dynamically changing amounts and not in fixed bundles: clients will buy seed virtual machines with some initial amount of resources, and then supplement these initial allocations with additional resources as needed.

Studying these trends, we extrapolate that, in the near future, resources will be rented separately with fine resource granularity for short durations. As rental durations grow shorter, we expect efficient clients to automate the process by deploying an economic agent (described in Section 5.5), which will make decisions in accordance with the current prices of those resources, the changing load the machine should handle, and the client's subjective valuation of those different resources at different points in time. Such agents are also considered a necessary development in smart grids [112] and wireless spectrum [161] resource allocation. Two elements are likely to ease the adoption of economic agents: client size (larger clients are more likely to invest in systematic savings, which accumulate for them to large numbers), and the availability of agents that are off-the-shelf and customizable (e.g., open source).

5.3.3 Market-Driven Resource Pricing

Virtualization and machine consolidation are beneficial when at least some resources are shared (e.g., heat sink, bus, last-level cache), and others are time shared (e.g., when a fraction of a CPU is rented, or physical memory is overcommited). However, the performance of a given virtual machine can vary wildly at different times due to interference and bottlenecks caused by other virtual machines that share resources whose use is not measured and allocated [56,102,135]. For example, Google App Engine's preliminary model, charging for CPU time only and not for memory, made the scaling of applications that use a lot of memory and little CPU time "cost-prohibitive to Google,"[13] because consolidation of such applications was hindered by memory bottlenecks. Hence, in 2011, Google App Engine was driven to charge for memory (by introducing memory-varied bundles), which became, as a result, a measured and allocated resource.

Moreover, interference and bottlenecks depend on the activity of all the virtual machines involved, and are not easily quantified in a live environment in which the guest can only monitor its own activity. Even after the guest benchmarks its performance as a function of the resource bundle it rented, neighbors sharing those same resources might still cause that performance to vary [135]. Thus, there is a discrepancy between what providers provide and what clients would actually like: in practice, what clients care about is their virtual machines' subjective performance.

To bridge this gap, researchers have proposed to sell client performance instead of consumed resources [20, 60, 102, 107]. This approach is only applicable where performance is well defined, and where the client applications are fully visible to the provider (as is the case in Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS) clouds), or the client virtual machines fully cooperate with the provider, as may be the case in private IaaS clouds. However, IaaS cloud providers and clients are separate economic entities. In general, they do not trust each other, and do not cooperate without good reason. Hence, guaranteeing client performance levels is not applicable to a public IaaS cloud, where allocated resources affect the performance of different applications differently, where the very definition of performance is subjective, where client virtual machines are opaque, and where the provider cannot rely on clients to tell the truth with regard to their desired and achieved performance. If the provider guarantees a certain performance level, it is in the client's interest to claim the performance is still too low, so that the provider will add resources.

We believe that public clouds will have to forsake the approach of charging users a predefined sum for resource bundles of unknown performance. For high-paying clients, providers can raise prices and forgo resource overcommitment. For low-paying clients, a cheap or free tier of unknown performance can be offered. However, for mid-range clients, providers will have to follow one of the following routes to handle the problem of unpredictable resource availability: (1) tackle the hard task of precisely measuring all the system's resources to quantify the real use each virtual machine made of them, and then charge the clients precisely for the resources they consumed, or (2) switch to a market-driven model.

A market-driven model is based on how clients value the few monitored resources. It does not necessitate precise measurement of resource use on the part of the provider—only the final outcome, the client's subjective valuation of its performance, matters. Clients, in turn, will have to develop their own model to determine the value of a smaller number of monitored resources. The model needs to implicitly factor in virtual machine interference over non-monitored resources. For example, clients might use a learning algorithm that produces a time-local model for the connection between monitored resources and client performance. Though highly expressive, the client's model need not be complicated: it is enough that the client can adjust the model to the required accuracy level. Hence, the minimal client model can be as simple as a specific sum for a specific amount of resources: below these requirements, the client will not pay. Above them, the client will not add money. The client willingness to pay will affect the prices and the resource allocation. Unlike previously proposed models, this economic model can accommodate real-world, selfish, rational clients.

5.3.4 Tiered Service Levels

Tiered service [103], where different clients get different levels of service, can be found in certain scientific grids. Jobs of clients with low privileges may be preempted (aborted or suspended) by jobs of clients with higher privileges. Although clouds did not, at first, offer such prioritized service but rather supplied service at only one fixed level (on-demand), Amazon has since introduced different priority levels within EC2. The higher priority levels are accorded to the reserved (introduced March 2009) and on-demand instances. Spot instances (introduced December 2009) provide a continuum of lower service levels, since Amazon prioritizes spot instances according to the bid price stated by each client. Gridspot (2012) operates in a similar manner. As in grids, these priorities are relative, so it is hard to explicitly define their meaning in terms of absolute availability. For example, the availability of on-demand instances depends on the demand for reserved instances. The PaaS provider Dotcloud (announced in 2010)[14] and Google App Engine[15] also offer different SLA levels for different fees.

Having clients with different priorities is useful to the provider, who can provide high-priority clients with elasticity and availability at the expense of lower-priority clients, while simultaneously renting out currently-spare resources to low-priority clients when high-priority clients do not need them. Likewise, different priorities allow budget-constrained cloud clients cheap access to computing resources with poorer availability. Mixing clients of different relative priorities will allow the providers to simultaneously achieve high resource utilization and maintain adequate spare capacity for handling sudden loads.

Extrapolating from the progression of SLA terms we have seen to-date, we expect that in the RaaS cloud clients will be able to define their own priority level, choosing from a relatively priced continuum. Moreover, if prices are market driven, and priority levels reflect the client's willingness to pay, then we expect that clients will be able to change their desired priority levels as often as prices change.

It is possible to extend the prevalent SLA language—"unavailability of a minimal period X, which is at least a fraction Y of a service period Z"—to express different absolute levels by controlling the parameters X,Y,Z [20]. Yet, we extrapolate that as more cloud providers adopt flexible SLAs, they will continue the existing trend of relative priorities, and not venture into extending the absolute SLA language to several tiers.

5.4 Economic Dynamics

In the previous section, we surveyed several ongoing trends and tried to surmise where they will lead us next. We now survey the economic forces operating on clients and on providers and their implications. We believe these forces caused the phenomena previously discussed and will continue pushing today's IaaS clouds forward until today's clouds turn into RaaS clouds.

5.4.1 Forces Acting on Clients

As clients purchase more cloud services their bill increases. When bills are large, clients seek systematic savings. The best way to achieve this is by paying only for the resources they need, and only when they need them. The more flexible the provider offerings, the better control clients have over their costs and the resulting performance. As providers offer increasingly fine-grained resources and service levels, clients are incentivized to develop

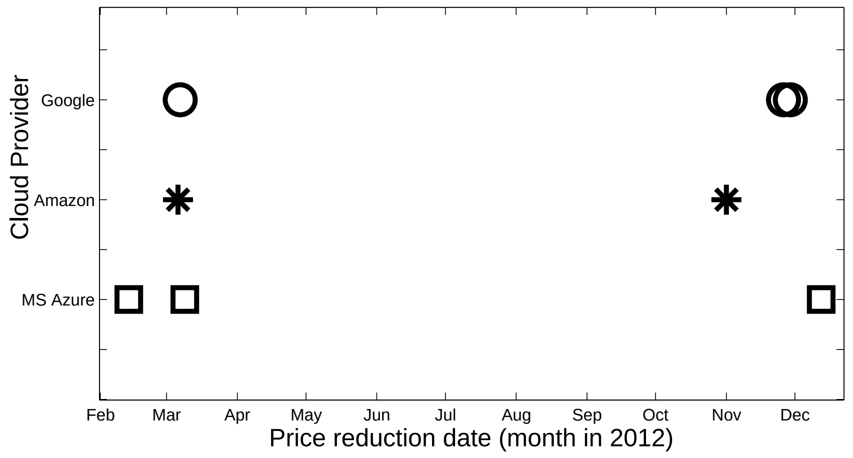


Figure 5.1: Correlated cloud price reduction dates for three major cloud providers during 2012

or adopt resource provisioning methods. As the time scales involved shorten, manual provisioning methods become tedious, increasing the clients' incentive to rely on computerized provisioning agents [154] to act on their behalf.

5.4.2 Forces Acting on Providers

Competition between IaaS cloud providers is increasing, as indicated by recent cloud price reductions. During previous years, Amazon reduced its prices in correlation with new instance type announcements, and only by 15%, while hardware costs dropped by 80% [137]. However, as shown in Figure 5.1, the timing of price cuts in 2012 by three major cloud providers is correlated, a phenomenon referred to as a "cloud price war".[16]

The competition is aided by the commoditization of cloud computing platforms. Commoditization eases application porting between providers. An example for such commoditization is the open source OpenStack,[17] which is the foundation of both Rackspace's public cloud and HP's. OpenStack also offers Amazon EC2/S3 compatible APIs. As changing providers becomes easier, and as hungry new providers join the fray, competition increases and providers are forced to lower prices.

5.4.3 Implications of Increased Competition

As competition increases and prices decrease, providers attempt to cut their costs,[18] in an effort to maintain their profit margins. At any moment, given the available revenue-creating client workload, the provider seeks to minimize its costs (in particular, power costs) by idling or halting some machines or parts thereof [49]. It does so by consolidating instances to as few physical machines as reasonably possible. When resources are overcommited due to consolidation and clients suddenly wish to use more resources than are physically available on the machine, the result is resource pressure.

The move towards tiered service and fine rental granularity is driven, in part, by the need to reduce costs and the accompanying resource pressure. When clients change their resource consumption on the fly, providers who continue to guarantee absolute Quality of Service (QoS) levels have to reserve a conservative amount of headroom for each resource on each physical server. This headroom—spare resources—is required just in case all clients simultaneously require all the resources promised them. Clients who change their resource consumption on-the-fly do not pay for this headroom unless and until they need it, so keeping it around all the time is wasteful.

Under the fixed bundles model, if the host chooses to overcommit resources, some clients will get less than the bundle they paid for. If the headroom is too small and there is resource pressure, this underprovisioning will be felt by the client in the form of reduced performance, and the illusion of a fixed bundle will be broken.

Extending the current absolute SLA language to several tiers only reduces some of the headroom. To get rid of the headroom completely, providers must resort to prioritization via tiered service levels, which only guarantees clients relative QoS. Relative QoS requires that clients change their approach. Relative QoS should thus be introduced gradually, allowing clients to control the risk to which they agree to be exposed.

Here is a concrete example of how a provider might nowadays waste its resources, and how a future provider might increase the utilization of its powered-up servers and reduce its power costs. Let us consider a 4GB physical machine, running an instance that once required 3GB of memory, and now only uses 2GB. A new client would like to rent an instance with 2GB. Under the IaaS model, the new client cannot be accommodated on this machine. 1GB goes unsold, and 2GB go unused. With tiered SLAs and dynamic resources, the first client can temporarily reduce its holdings to 2GB, and the provider then can rent 2GB to the new client. If conflicts arise later due to memory shortage, the provider can choose how much memory to rent to each client on the basis of economic considerations. No memory goes unused, and no extra physical server needs to be booted.

5.5 The RaaS Cloud

We have presented the distinct trends operating in IaaS clouds, along with the economic forces that govern them. We believe that the combined effect of all these trends and forces is leading to a qualitative transformation of the IaaS cloud into what we call the *Resource-as-a-Service (RaaS) cloud*. We present here our unified view of the RaaS cloud, and discuss possible steps on the path to its realization.

5.5.1 Trading in Fine-Grained Resources

Seed virtual machine In RaaS clouds, the client purchases upon admittance a seed virtual machine. The seed virtual machine has a minimal initial amount of dedicated resources. All other resources needed for the efficient intended operation of the virtual machine are continuously rented. This combination of resource rental schemes—prepurchasing and multiple on-demand levels—benefits the clients with the flexibility of choice. To draw from experiments on human preference in Internet service provider payment plans, clients who are presented with both flat-rate and usage-based resource rental options tend to make use of the full range of choices [6].

Fine-grained resources The resources available for rent include CPU, RAM, and I/O resources, as well as emerging resources such as computational accelerators (e.g., GPGPUs and FPGAs) and Flash devices. CPU capacity is sold on a hardware-thread basis, or even as number of cycles per unit of time; RAM is sold on the basis of memory frames; I/O is sold on the basis of subsets of I/O devices with associated I/O bandwidth and latency guarantees. Such devices include network interfaces and block interfaces. Accelerators are sold both as I/O devices and as CPUs. A subset of an I/O device may be presented to the virtual machine as a direct-assigned SR-IOV Virtual Function(VF) [53] or as an emulated [12] or para-virtual device. Every resource comes with a dynamically changing price tag. Resource rental contracts are set for a minimal fixed period, which does not have to coincide with the repricing period. The host may offer the guests renewal of their rental contract at the same price for an additional fixed period.

Host economic coordinator To facilitate continuous trading, the provider's cloud software includes an economic coordinator representing the provider's interests. This coordinator operates an economic mechanism which defines the resource allocation and billing mechanism: which client gets which resources and at what price. Several auctions were proposed to such ends, e.g., by Kelly [75], Chun and Culler [35], Lazar and Semret [81], Waldspurger et al. [146], and Lubin et al. [89]. In addition, the coordinator may act as a clearing house and support a secondary market of computing resources inside the physical machine, as SpotCloud[19] offers to do for fixed-bundle virtual machines and as Kash et al. [73] propose to do for the wireless spectrum.

Guest economic agent To take part in auctions or trade, clients' virtual machines must include an economic agent. This agent represents the client's business needs. It rents the necessary resources—given current requirements, load and costs—at the best possible prices, from either the provider or its *neighbors*—virtual machines collocated on the same physical machine, possibly belonging to different clients. When demand outstrips supply, the agent changes its *bidding strategy* (in cases where the provider runs an *auction*) or negotiates with neighbors' agents, mediating between the client's requirements and the resources available in the system, ultimately deciding how much to offer to pay for each resource at any given time.

Subletting Clients can secure resources early and sublet them later if they no longer need them. The resource securing can be done either by actively renting resources long term or by negotiating a future contract with the host. Either way, resource subletting also lays the ground for resource futures markets among clients. Clients can sublet to other clients on the same physical machine using infrastructure provided by the host's coordinator: the clients agree to redivide resources between them and inform the coordinator, who transfers the local resources from one guest to another (as Hu et al. [62] do for bandwidth resources). In addition to trading with a limited number of neighbors, clients can sublet excess resources to anyone, in the form of nested full virtual machines [21], a concept which is gaining more and more support. Examples resembling subletting exist today in the Amazon EC2 Reserved Instance Marketplace,[20] in CloudSigma's reseller option,[21] and in DotCloud, which is reselling EC2's resources with an added value.[22] The subletting option reduces the risk for clients who commit in advance to rent resources. It also partially relieves the provider of the burden of retail sales, improves its utilization, and can increase its revenue through seller fees.[23]

Legacy clients IaaS providers can introduce RaaS capabilities gradually, without forcing their clients to change their business logic. Legacy clients, without an economic agent, can still function in the RaaS cloud just as they do in an IaaS cloud. They can simply rent large RaaS seed machines, which serve as IaaS instances. IaaS virtual machines function in a RaaS cloud just as well as they do in an IaaS cloud. However, to get the RaaS benefits of vertical elasticity and reduced costs, clients will need to adapt.

Private clouds Should the provider and clients all belong to the same economic entity (e.g., as might happen in a company's private cloud), then the economic mechanism is not used for actual payments, but still reflects the relative importance of the different clients and the subjective costs of resources (electricity, for example).

5.5.2 Prioritized Service Levels

Priorities for headroom only In the RaaS cloud, the client gets an absolute guarantee (for receiving the resources and for the price paid) only for its minimal consumption, which is constant. Additional resources are provided on a priority basis in market prices. A risk-averse client can prepay for a larger amount of constant resources, trading low costs for peace of mind. From the provider's point of view, the aggregate constant consumption provides a steady income source. Only resources which may go unused (the headroom) are allocated on the basis of market competition.

Vertical elasticity: Robin Hood in reverse RaaS clients are offered on-the-fly, fine-grained, fine-timed *vertical elasticity* for each instance: the ability to expand and shrink the resource consumption of each virtual machine. The resources required for this vertical elasticity are limited by the physical resources contained in a single machine, because migrating running virtual machines from one physical machine to another will likely remain less efficient than dynamically balancing the available resources between virtual machines co-existing on the same physical machines. Hence, during peak demand times, to enable one client to vertically upscale a virtual machine, the additional resources must be taken from a *neighbor*. Instead of static priorities, in the RaaS cloud providers use the willingness of clients to pay a certain price for resources at a given moment (e.g., bids) to decide which client gets which resource. Thus market forces dictate both the constantly changing prices of resources and their allocation. In effect, the RaaS cloud provider does the opposite of Robin Hood: it takes from the poor and gives to the rich.

A few good neighbors The RaaS virtual machine's vertical elasticity is determined, via a market mechanism, by its neighbors' willingness to pay. The neighbors also determine the cost of the elastic expansion. Due to the inherent inefficiencies of live virtual machine migration, RaaS clouds must include an algorithm for placing client virtual machines on physical machines. The algorithm should achieve the right mixture of clients with different SLAs on each physical machine in the cloud, such that high-priority clients always have low-priority clients beside them, to provide them with more capacity when their demands peak. The low-paying clients can use the high-paying clients' leftover resources when they do not need them, keeping the provider's machines constantly utilized. Another objective of the allocation algorithm is to allow the low-priority clients enough aggregate resources for their needs. A low priority client can be expected to tolerate a temporary loss of service every so often, but if the physical resources are strictly smaller than the mean demand, such a client will never get enough resources to make meaningful progress. Therefore, to retain the low-priority clients, the placement algorithm must provide them with enough resources to make (some) progress.

Full house The RaaS provider also influences the quality of service that the RaaS client experiences by limiting the *maximal possible aggregate demand* for physical resources on the machine. Demand can be limited by controlling the number of virtual machines per physical machine and the maximal vertical elasticity to which each virtual machine is entitled. When the maximal possible aggregate demand is lower than the supply, resources are wasted, but all virtual machines can freely expand. As the maximal possible aggregate demand exceeds the supply, clients will be less likely to succeed in vertical expansion when they need it, or might be forced to pay more for the same expansion. Hence, RaaS clients are willing to pay more to be hosted in a physical machine with lower maximal possible aggregate demand. This encourages RaaS providers to expose information about the aggregate demand and supply on the physical machine to its clients.

5.6 Implications, Challenges, Opportunities

The RaaS cloud gives rise to a number of implications, challenges, and opportunities for both providers and clients, which did not exist in markets of entire virtual machines [7,105,110,123,134,156]. Broadly speaking, the new research areas can be divided into two categories: technical mechanisms and policies.

The RaaS cloud requires new mechanisms for allocating, metering, charging for, reclaiming, and redistributing CPU, memory and I/O resources between untrusted, not-necessarily-cooperative clients every few seconds. These mechanisms must be efficient and reliable. In particular, they must be resistant to side-channel attacks from malicious clients [115]. Hardware mechanisms are especially needed for fine-grained resource metering in the RaaS cloud.

The RaaS cloud requires new system software and new applications. Usually, current operating systems and applications are written under the assumption that their underlying resources are fixed and always available. In the RaaS cloud, virtual machines never know the precise amount of resources that will be available to them at any given second. Thus, the software running in those virtual machines must adapt to changing resource availability and exploit whatever resources the software has, when it has them. Assume a client application that just got an extra 2Gbps of networking bandwidth at a steal of a price, but only for one second. To use it effectively while it is available, all the software layers, including the operating system, run-time layer, and application must be aware of it.

The RaaS cloud requires efficient methods of balancing resources within a single physical machine, while taking into consideration the different guaranteed service levels. Bottleneck-resource allocation [44,51,57] is a step towards allocation of resource bundles, but it still requires an algorithm for setting the system share to which each client is entitled.

The resource balancers are most efficient when guests with different service levels are collocated on the same physical server. Hence, workload balancers, which balance resources across entire cloud data centers, will need to consider the virtual machines, flexibility and SLA in addition to the current considerations (static resource rquests only).

Under dynamic conditions, the intra-machine RaaS mechanisms will quickly respond to flexibility needs, holding the fort until the slower live migration can take place. However, live migration must take place to mitigate the resource pressure on the effectively most stressed machines, and allow clients to change their flexibility bounds. Large IaaS providers apparently manage without live migration [115]: the high rate of initialization and shutdown of virtual machines makes the initial balancer effective enough. However, the fine time granularity of the changes in the RaaS cloud means live migration is going to be required more often. Hence, the RaaS cloud will require efficient methods for live migration of virtual machines and for network virtualization.

On the policy side, the RaaS cloud requires new economic models for deciding what to allocate, when to allocate it, and at what prices [39]. Ideally, they should optimize the provider's revenue or a social welfare function: a function of the benefit of all the clients. The clients may measure their benefit in terms of starvation, latency, or throughput, but the mechanisms should optimize the impact of those performance metrics on the welfare of the clients, for example by maximizing the sum of client benefits or by minimizing the unhappiness of the most unsatisfied client.

These new economic models should also consider that resources may complete or substitute one another in different ways for different clients. For one client resources might be *economic complements*: if, for each thread the application requires 1GB RAM and 1 core, a client renting 2GB and 2 cores will only be interested in adding a combination of 1GB and 1 core. For another client, resources might be *economic substitutes*: every additional GB allows the application to cache enough previous results to require one core less. So when cores are expensive, a client that is renting 2GB and 2 cores will be able to release one core and rent another GB instead.

These mechanisms should be incentive compatible: truth telling regarding private information should be a good course of action for the clients, so that the provider can easily optimize the resource allocations. The mechanisms should be collusion-resistant: a virtual machine should not suffer if several of the virtual machines it is co-located with happen to belong to the same client. Like approximation algorithms for multi-unit auctions [43, 140], they should be computationally efficient at large scale, so that solving the resource allocation problem does not become prohibitive.

The mechanisms should preserve the clients' privacy [98] as well as minimize the price-of-anarchy [80]: the waste incurred by using a distributed mechanism. Moreover, in order to work in the real world, the economic mechanisms must accommodate realistic clients' willingness to pay, which is a function of their performance measurements. The mechanism must support such measured functions, which are not necessarily mathematically nice and regular (e.g., contain steps [108]). Another real-world demand is simplicity. If researchers combined some of the works mentioned above to create a cumbersome mechanism with satisfactory theoretical qualities, that still would not guarantee its acceptance by the market: the providers and the clients.

In conclusion, making the RaaS cloud a reality will require solving hard problems spanning the entire gamut from game theory and economic models to system software and architecture. The onus is now on us, the cloud computing research community, to lead the way and build the mechanisms and policies that will make the RaaS cloud a reality.

Notes

[1]`http://www.cloudsigma.com`

[2]`http://gridspot.com`

[3]`http://www.profitbricks.com`

[4] `https://developers.google.com/appengine/kb/billing\#time_granularity_instance_pricing`, accessed December 2012.

[5]`https://cloud.google.com/pricing/compute-engine`, accessed May 2013.

[6]"Fifty percent of the time the site is down in seconds—even when we've contacted site owners and they've told us everything will be fine. It's often an unprecedented amount of traffic, and they don't have the required capacity."–Stephen Fry, `http://tinyurl.com/StephenFrySeconds`.

[7]`http://www.rackspace.com/cloud/public/servers/techdetails/`

[8]`http://www.gogrid.com`

[9]`https://cloud.google.com/pricing/compute-engine`

[10]`http://aws.amazon.com/about-aws/newsletters/2012/08/14/august-2012/`

[11]`http://aws.amazon.com/ebs/`

[12] `https://developers.google.com/appengine/kb/billing`

[13]Greg D'Alesandre, `http://tinyurl.com/D-Alesandre` .

[14]`https://www.dotcloud.com/pricing.html`

[15]`https://cloud.google.com/pricing/`

[16]`http://tinyurl.com/cloud-price-war`

[17]`http://openstack.org`

[18]James Hamilton, "Amazon cycle of innovation" slide, `http://tinyurl.com/james-hamilton`

[19]`http://spotcloud.com`

[20]`http://aws.amazon.com/ec2/reserved-instances/marketplace/`

[21]`http://www.cloudsigma.com/cloud-computing/what-is-the-cloud/171`

[22]`http://docs.dotcloud.com/0.9/faq/`

[23]`http://aws.amazon.com/ec2/reserved-instances/marketplace/`

Chapter 6

Ginseng: Market Driven Memory Allocation (Memory-as-a-Service)

6.1 Abstract

Physical memory is the most expensive resource in use in today's cloud computing platforms. Cloud providers would like to maximize their clients' satisfaction by renting precious physical memory to those clients who value it the most. But real-world cloud clients are selfish: they will only tell their providers the truth about how much they value memory when it is in their own best interest to do so. Under these conditions, how can providers find an efficient memory allocation that maximizes client satisfaction?

We present Ginseng, the first market-driven framework for efficient allocation of physical memory to selfish cloud clients. Ginseng incentivizes selfish clients to bid their true value for the memory they need when they need it. Ginseng continuously collects client bids, finds an efficient memory allocation, and re-allocates physical memory to the clients that value it the most. Ginseng achieves a $\times 6.2$–$\times 15.8$ improvement in aggregate client satisfaction when compared with state-of-the-art approaches for cloud memory allocation. It achieves 83%–100% of the optimal aggregate client satisfaction.

6.2 Introduction

Infrastructure-as-a-Service (IaaS) cloud computing providers rent computing resources to their clients. As competition between providers gets tougher and prices start going down, providers will need to continuously and ruthlessly reduce expenses, primarily by improving their hardware utilization. Physical memory is the most constrained and thus precious resource in use in cloud computing platforms today [54, 60, 61, 92, 102, 145]. One way for providers to significantly reduce their expenses is by using less memory to run more client guest virtual machines on the same physical hosts.

Whereas today cloud computing clients buy a supposedly-fixed amount of physical memory for the lifetime of their guests, nothing stops their provider from overcommitting this memory. Clients today have no idea and no way to discern how much physical memory they are actually getting. Clients would much prefer to have full visibility and control over the resources they receive [3, 106]. They would like to pay only for the physical memory they need, when they need it [17, 52]. By granting clients this flexibility providers can increase client satisfaction.

Therefore, finding an efficient allocation of physical memory on each cloud host—an allocation that gives each guest virtual machine precisely the amount of memory it needs, when it needs it, at the price it is willing to pay—poses benefits for both clients, whose satisfaction is improved, and providers, whose hardware utilization is improved.

Previous physical memory allocation schemes assumed fully cooperative client guest virtual machines, where the host knows precisely what each guest is doing, how much benefit additional memory would bring to it, and the importance of that guest's workload to the client [54, 60, 61, 102]. However, when it comes to commercial cloud providers and their paying IaaS clients, none of these assumptions are realistic. Real-world clients act *rationally* and *selfishly*. They are *black boxes* with private information such as their performance statistics, how much memory they need at the moment, and what it is worth to them. Rational, selfish black-boxes will not share this information with their provider unless it is in their own best interest to do so.

When white-box models are applied to selfish guests, the guests have an incentive to manipulate the host into granting them more memory than their fair share. For example, if the host gives memory to those guests that will benefit more from it, each guest will say it benefits from memory more than any other guest. If the host gives memory to those guests that perform poorly with their current allocation, each guest will say it performs poorly. If the host allocates memory on the basis of passive black-box or grey-box measurements [72, 86, 92, 145] such as page faults, guests have an incentive to bias the measurement results, e.g., by inducing unnecessary page faults. Furthermore, black-box methods compare the guests only by technical qualities such as throughput and latency, which are valued differently by different guests under different circumstances.

In this work we address the cloud provider's fundamental memory allocation problem: How should it divide the physical memory on each cloud host among selfish black-box guests? A reasonable meta-approach would be to give more memory to guests who would benefit more from it. But how can the host compare the benefits of additional memory for each guest?

We make the following three contributions. **Our first contribution** is **Ginseng**, a market-driven memory allocation framework for allocating memory efficiently to selfish black-box virtual machines. Ginseng is the first cloud platform to optimize overall client satisfaction for black box guests.

Our second contribution is the **Memory Progressive Second Price (MPSP) auction**, a game-theoretic market-driven mechanism which induces auction participants to bid (and thus express their willingness to pay) for memory according to their true economic *valuations* (how they perceive the benefit they get from the memory, stated in monetary terms). In Ginseng, the host periodically auctions memory using the MPSP auction. Guests bid for the memory they need as they need it; the host then uses these bids to compare the benefit that different guests obtain from physical memory, and to allocate it to those guests which benefit from it the most. The host is not manipulated by guests and does not require unreliable black-box measurements.

Ginseng is the first full implementation of a single-resource Resource-as-a-Service (RaaS) cloud [3]. It is ready for a world of *dynamic-memory applications*—applications that can improve their performance when given more memory on-the-fly over a large range of memory quantities and can return memory to the system when needed. Dynamic-memory applications are still scarce. **Our third contribution** is a **dynamic-memory version of Memcached**, a widely-used key-value storage cloud application, as well as **MemoryConsumer**, a dynamic memory benchmark we developed.

Ginseng achieves a $\times 6.2$ improvement in aggregate client satisfaction for MemoryConsumer and $\times 15.8$ improvement for Memcached, when compared with state-of-the-art approaches for cloud memory allocation. Overall, it achieves 83%–100% of the optimal aggregate client satisfaction.

6.3 System Architecture

Ginseng's system architecture is depicted in Figure 6.1. Ginseng is a market-driven framework for allocating memory in the cloud using guest bids. Challenges in auctioning memory, as opposed to other kinds of resources, and how Ginseng overcomes them, are described in **Section 6.4**.

Ginseng has a host component and a guest component. The host's auctioneer receives guest bids using the protocol specified in **Section 6.5**, performs the MPSP auction described in **Section 6.6**, and uses a balloon [86] to change each guest's memory allocation according to the auction's results. Ginseng does not specifically depend on a balloon; it only requires that the host supports some underlying mechanism for memory borrowing. We implemented Ginseng for cloud hosts running the KVM hypervisor [76].

Guests utilize an economic learning agent to rent more or less physical memory. Each guest's agent acts on its behalf according to its valuation-of-memory function within the framework of the MPSP protocol. The guest is free to use any agent it wishes provided it speaks the MPSP protocol. We describe the guest agent we implemented in **Section 6.7**. Since all communication between the agents and the auctioneer is over TCP/IP, the agents and the auctioneer could run anywhere; our prototype runs them inside the guests and in the host, respectively.

6.4 Memory Auctions

Ginseng auctions memory between guests. Each guest has a different, changing, private (secret) *valuation* for memory. We define the aggregate benefit of a memory allocation to all guests—their satisfaction from 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' valuations of the memory they receive in this allocation. An efficient memory auction allocates the

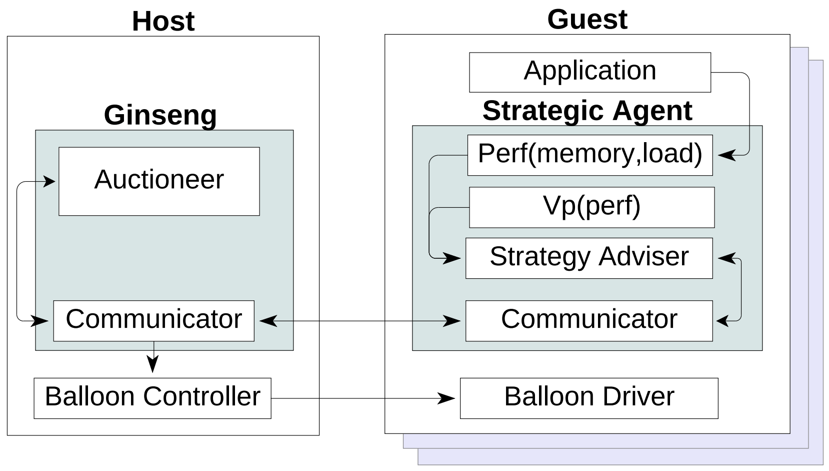


Figure 6.1: Ginseng system architecture

memory to the guests such that the social welfare is maximized. A necessary condition for a memory auction to maximize the social welfare is *Pareto efficiency*: there is no other allocation in which no guest benefits less, and at least one guest benefits more. Another requirement for a good memory auction is *fairness*: not preferring one guest over another [147]. An *ex-post fair* auction—fair even after the allocation was made—is better than an *ex-ante fair* auction, which is fair by expectation value, but may be unfair once a random choice is made in the auction.

VCG [37, 55, 138] auctions optimize social welfare by incentivizing even selfish participants with conflicting economic interests to inform the auctioneer of their true valuation of the auctioned items. They do so by the *exclusion compensation* principle, which means that each participant is charged for the damage it inflicts on other participants' social welfare, rather than directly for the items it wins. VCG auctions are used in various settings, including Facebook's repeated auctions [58, 91].

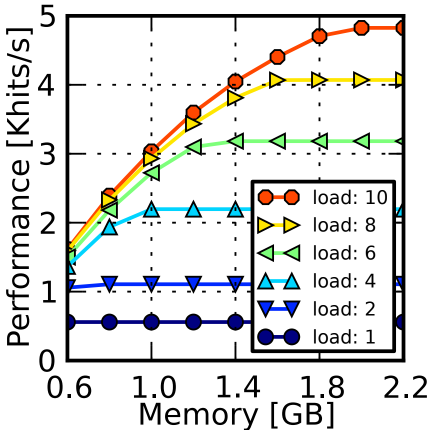
Various auction mechanisms, some of which resemble the VCG family, have been proposed for *divisible* resources, in particular for *bandwidth sharing* [75, 81, 93]. For practical reasons, bidders in these auctions do not communicate their valuation for the full range of auctioned goods. One of these VCG-like auctions is Lazar and Semret's Progressive Second Price (PSP) auction [81]. None of the auctions proposed so far for divisible goods, including the PSP auction, are suitable for auctioning memory, because memory has two characteristics that set it apart from other divisible resources: first, the participants' valuation functions may be non-concave; second, transferring memory too quickly between two participants leads to waste.

6.4.1 Non-concave Valuation Functions

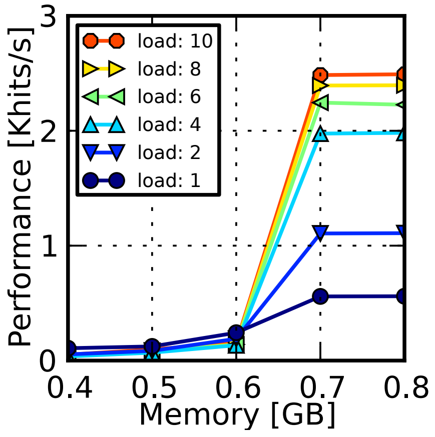
The memory valuation function, which also describes how much the guest is willing to pay for different memory quantities, is a composition of two functions: $V(\mathit{mem}, \mathit{load}) = V_p(\mathit{perf}(\mathit{mem}, \mathit{load}))$. $\mathit{perf}(\mathit{mem}, \mathit{load})$ describes the performance the guest can achieve given certain load and memory quantity. It can be measured either online [160] or offline [54, 61]. Performance is a guest-specific metric that differs between guests. It might be measured in hits per second for a webserver, transactions per second for a database, trades per second for a high-frequency-trading system, or any other guest-specific metric. $V_p(\mathit{perf})$, the guest's owner's (i.e., the client's) valuation of performance function, describes the value the client derives from a given level of performance from a given guest. This function is different for each client and is private information of that client.

If either of these functions is non-concave or not monotonically rising, the composed function may be non-concave or not monotonically rising as well. The PSP auction optimally allocates a divisible resource if and only if all the valuation functions are monotonically rising and concave. Other bandwidth auctions also rely on the monotonically rising concave property of the valuation functions.

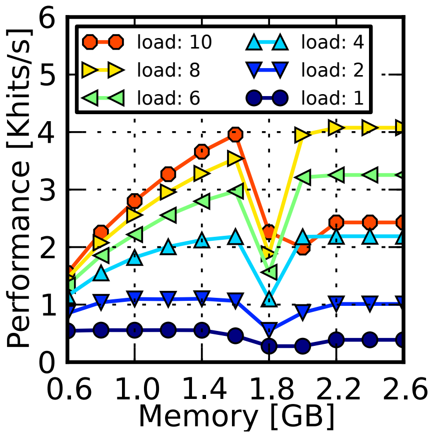
Guest performance $\mathit{perf}(\mathit{mem}, \mathit{load})$ is not necessarily a concave, monotonically rising function of physical memory. For example, in the experimental environment, our *memcached* version with dynamic cache size has a concave, monotonically rising performance graph (Figure 6.2(a)). However, the performance graph of off-the-shelf memcached in the same environment is monotonically rising, but not concave (Figure 6.2(b)). The performance graph of our dynamic memcached, in a default system configuration, is not always concave or monotonically rising (Figure 6.2(c)). Because on-line measurements of real production systems cannot be expected to always produce concave, monotonically rising performance graphs, the valuation-of-memory



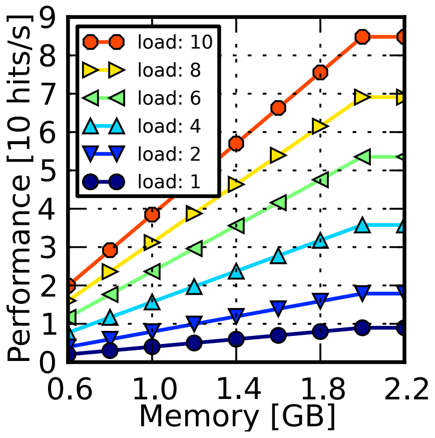
(a) Dynamic Memcached, experimental system



(b) Memcached, 500MB internal cache, default system



(c) Dynamic Memcached, default system



(d) MemoryConsumer, experimental system

Figure 6.2: Application performance ("get" hit rate for Memcached, hit rate for MemoryConsumer) as a function of guest physical memory, for different load values. Load is number of concurrent requests.

graph $V(mem, load)$ may also be non-concave or even not monotonically rising.

Auction protocols which assume monotonically rising concave valuation functions either interpret a bid of unit price and quantity (p, q) as willingness to purchase exactly q units for unit price p or as willingness to buy up to q units at price p. In the first case, the bidding language is limited to exact quantities. In the second case, if the valuation function is non-concave, the guest may get a quantity that is smaller than the one it bid for, and pay for it a unit price it is not willing to pay. If the function is not, at the very least, monotonically rising, it may even get a quantity it would be better off without.

MPSP supports non-concave and non-monotonic valuation functions by specifying *forbidden ranges*. These are forbidden memory-quantity ranges for a single price bid. The guest can use forbidden ranges to cover domains in which its average valuation per memory unit is lower than its bid price. By definition, MPSP will not allocate the guest a memory quantity within its forbidden ranges. Rather, it will optimize the allocation given the constraints. The guest can thus avoid getting certain memory quantities in advance while still maintaining its expressiveness.

6.4.2 Memory Waste

Since guest valuations change over time, auctions must expire and allow resources to be put up for auction again. Repeated bandwidth auctions (*rounds*) can be analyzed as stand-alone auctions because the benefit from increased bandwidth is immediate. In contrast, the benefit from winning more memory is not immediate.

Memory is often used for caching. To utilize increased cache sizes, guests need to retain the memory used for caches for relatively long periods of time, to increase the likelihood of cache hits. *Cycling allocations* are repeating allocation patterns involving guest and host behavior [23], where resources are transferred back and forth between guests. If subsequent memory auctions result in cycling allocations, then increasing auction frequency will yield less benefit for guests; memory they rented but did not yet have time to use is *wasted*. Hence, unlike in bandwidth auctions, memory auctions should not be analyzed separately. Instead the auctioneer should control the amount of memory exchanging hands in each auction round to balance memory waste with the time required to respond to changing guest valuations.

Reclaim Factor. In MPSP, each guest i is set up permanently with the *bare minimal physical memory* it requires to operate, denoted as $bare_i$. This memory is charged for separately by a constant hourly fee. Only *extra memory* is rented using auctions. In each round, the auctioneer reclaims a *reclaim factor* $0 < \alpha \leq 1$ of each guest's extra memory for a new auction. The guest continues to rent the rest of the extra memory it won in previous auctions at the prices for which it won it. The host can change the reclaim factor between auctions. It can increase it to improve the system's responsiveness when the memory pressure rises or is expected to rise (e.g., a new guest is launched), or when guests change bids fast, indicating fast valuation changes. Otherwise, it can decrease it to decrease the potential memory waste.

Tie Breaking. Guests are sorted by the unit-prices they bid when they queue for memory. When two or more bids are identical, the tie must be broken, preferably fairly and Pareto-efficiently.

Tied PSP guests are excluded from the allocation [81], so that if some bidders expect to be tied, they are incentivized to change their bids. A *steady state* is when the auction's personal results (a guest's won goods and payment) turn out the same in subsequent auctions in response to the same strategy. A *Nash equilibrium* is a steady state in which guests stick to their bids if they know what other guests plan to bid. Breaking ties by excluding guests prevents ties in Nash equilibria. However, in dynamic, real-life scenarios, guest bids are not always in Nash equilibrium, especially if guests do not continuously inter-communicate. Hence, we sought alternatives to this tie-breaking method, which we find unsuitable for memory auctions.

We considered three Pareto-efficient options. Dividing the memory among all the tied guests is *ex-post fair*, but it is also NP-hard, because the forbidden ranges may turn solving it into solving a knapsack problem. Preferring guests according to a random shuffle is *ex-ante fair* before each round. Preferring the current memory holder [160] is only ex-ante fair before the tie is formed, but is the most efficient tie breaker. We opted for combining the latter two approaches in Ginseng. Guests are sorted lexically by three qualities: first by bid prices, then by their current holdings, and then by a random shuffle.

6.5 Repeated Auction Protocol

In the MPSP auction, memory allocations change every round. The guest rents the memory for the full duration of one round, or more if the reclaim factor is small. Here we describe one MPSP auction round, indexed t.

Initialization. For each guest i, a reference point called the *base memory* is initialized as $base_i(0) = bare_i$ when it enters the system. The guest's initial memory allocation is its base memory.

Auction Announcement. The host computes a decay in the base memory of each guest i according to the reclaim factor $0 < \alpha \leq 1$ to

$$base_i(t) = \alpha \cdot bare_i + (1 - \alpha) \cdot final_i\,(t-1)\,, \tag{6.1}$$

where $final_i(t-1)$ is the total memory allocated to the guest in the previous round (including the *bare* memory). It computes the *free memory*—maximal amount of memory each guest can bid for—as the *excess* physical memory beyond the host's memory and the aggregate base memories. It then informs each guest of its new base, the free memory, and the auction's closing time, after which bids are ignored.

Bidding. Interested guests bid for memory. Agent i's *bid* is composed of a *unit price* p_i—memory price per MB per hour (billing is still done per second according to exact rental duration.) and a list of *desired ranges*: mutually exclusive, closed ranges of desired memory quantities $[r_i^j, q_i^j]$ for $j = 1 \ldots m_i$, sorted in ascending order. The bid means that the guest is willing to rent any memory quantity within the desired range list, in addition to its current basic holdings $base_i(t)$, for a unit price p_i. The *forbidden ranges* are those that lie between the desired ranges. To simplify the notation in the context of the same round, we drop hereafter the round indexing (t), e.g., $base_i(t)$ can be denoted as $base_i$.

Bid Collection. The host asynchronously collects guest bids. It considers the most recent bid from each guest, dismissing bids received before the auction was announced. Guests that did not bid lose the auction automatically. A guest that persists in not bidding gradually loses its extra memory, until it is left with its *bare* minimal memory.

Allocation and Payments. The host computes the allocation and payments according to the MPSP auction protocol described in Section 6.6. For each guest i, it computes how much memory it won (denoted by ${q'}_i$) and at what unit price (denoted by ${p'}_i$). The payment rule guarantees that $0 \leq {p'}_i \leq p_i$.

Accounting. In each round, a guest may win a memory *chunk*: a memory quantity with an attached rental unit-price. Over time, guests come to hold memory chunks of different sizes with different unit prices. The host holds this information as a list, sorted by unit price. The list is updated at the end of the auction round in two stages: first, α of the guest's extra memory is released (the cheapest chunks or parts thereof). Then, if the guest won memory quantity q'_i in the auction, a memory chunk of size q'_i, with a unit price of p'_i is added to the list. Note that for $\alpha = 1$ one chunk at most exists, and the accounting is trivial.

Informing Guests. The host informs each guest i of its personal results p'_i, q'_i. To improve the performance of guest learning algorithms, to be described in Section 6.7.3, the host also announces information that guests can work out anyhow, about *borderline bids*: the lowest accepted bid's unit-price and the highest rejected bid's unit-price.

Adjusting and Moving Memory. After an *adjustment period* following the announcement, the host actually takes memory from those who lost it and gives it to those who won, by shrinking and expanding their balloons as necessary. The purpose of this period is to allow each guest's agent to notify its applications of the upcoming memory changes, and then allow the applications time to gracefully reduce their use of memory, if necessary. The applications are free to choose when to start reducing their memory consumption, according to their memory-release agility. This early notification approach makes it possible for the guest operating systems to gracefully tolerate sudden large memory changes and spares applications the need to monitor second-hand information on memory pressure.

6.6 The Auction

The MPSP auction relies on finding an optimal allocation: an allocation that maximizes the *social welfare function (SW)*, defined as the sum of guest valuations for the memory they won,

$$SW = \sum_{i=1}^{N} \left(V_i(final_i(t)) - V_i(base_i(t)) \right), \tag{6.2}$$

where N is the number of guests and $V_i(\cdot)$ is guest i's memory valuation function for their current loads. To determine the optimal allocation, the

MPSP auction solves a constrained divisible good allocation problem. After we define the allocation rule algorithm and the payment rule, we proceed to discuss complexity and give an example. The correctness proof is omitted for brevity.

6.6.1 Allocation Rule

The optimal allocation is found using a constrained divisible good allocation algorithm. In each stage, a divisible good allocation is attempted: the guests are allocated their maximal desired quantities according to the tie breaking order (discussed in Section 6.4.2). If there are a guest g and a forbidden range R such that g is allocated a memory quantity inside R, then the allocation is *invalid*. If the SW value of the invalid allocation can improve the known highest valid value, two constrained allocations are recursively considered instead: one in which guest g gets a memory quantity beyond the forbidden range R, and another in which it gets less than R's starting point. The social welfare of the *valid* allocations is compared to find the optimal allocation.

6.6.2 Payment Rule

The payments follow the *exclusion compensation* principle. According to the PSP rule [81], if guest i gets some memory ${q'}_i > 0$, it pays:

$$p'_i = \frac{1}{q'_i} \sum_{k \neq i} p_k \left[{q'}_k(0, s_{-i}) - {q'}_k(s_i, s_{-i}) \right], \tag{6.3}$$

where s_i is agent i's bid and s_{-i} are the other guests' bids. Note that to compute the payment for a guest which gets allocated some memory, the constrained divisible good allocation algorithm needs to be computed again without this guest. In total, the allocation procedure needs to be called one time more than the number of winning guests.

6.6.3 Complexity

The MPSP algorithm solves an NP-hard problem, because its bidding language includes forbidden ranges. Its time complexity is $O(N^2 \cdot 2^M)$, where N is the number of guests and M is the number of all the forbidden ranges in all the bids. To find an optimal allocation, at most 2^M divisible allocations are attempted, each taking $O(N)$ to compute. For the payment rule, $O(N)$ allocations need to be computed.

However, for real life performance functions, a few forbidden ranges are enough to cover the non-concave regions (up to one for the functions we measured and for Websphere [61]). Given the small number of guests on a physical machine, the algorithm's run-time is reasonable (less than one second using a single hardware thread in our experiments). For concave functions, the complexity is reduced to $O(N^2)$, as in the PSP auction [81].

Had Ginseng been implemented on the basis of *bundles* in a *multi-unit* auction, the memory would have been divided to units. The clients would have bid for bundles of such units. The host would have had to trade off the accuracy of the final allocation with the complexity of the auction by controlling the bundle size. As the number of units grows, the final allocation is more accurate, but the auction's complexity grows. In contrast, the MPSP auction is of a continuous resource, and thus its fine-grained allocation accuracy does not increase its algorithmic complexity.

6.6.4 Example of a Single Round

Consider a system with 6 MB of physical memory and two guests, bidding $bid_1 = (p_1 = 2, r_1 = 3, q_1 = 4)$ and $bid_2 = (p_2 = 1, r_2 = 3, q_2 = 5)$. In the first stage, we sort the guests by price, and try to allocate 4MB to guest 1 and 2MB to guest 2. This is an invalid allocation, because guest 2 gets a quantity in its forbidden range, which is $[0, 3)$. We examine two constrained systems instead. (1) Guest 2 gets no more than the start of the forbidden range, which is 0. In this case, the allocation is 4MB to guest 1, with a social welfare of 8. (2) Guest 2 is guaranteed the full forbidden range, which is 3. Then the rest of the free memory is allocated by the order of prices, so guest 1 gets the other 3 MB. The social welfare in this case is 9, and this is the chosen allocation.

According to Equation 6.3, the guests pay $p'_1 = \frac{1}{3}(1[5-3]) = \frac{2}{3}$ and $p'_2 = \frac{1}{3}(2[4-3]) = \frac{2}{3}$, because in each other's absence they each would have gotten their maximal desired range.

6.7 Guest Strategy

In this section we present the bidding strategy used by the guests in the performance evaluation in Section 6.9. To simplify the notation, we drop the guest's index in the remainder of this section.

The guest is *myopic*: it wishes to maximize its *estimated utility* in this round U_{est}. The guest considers bidding for different maximal desired memory quantities. For each maximal quantity q, it is clear (as will be explained in Section 6.7.1) that the best strategy would be to bid its true valuation for the quantity. The guest then compares its estimated utility from bidding for the different maximal quantities, as described in Section 6.7.2, with the help of on-line learning algorithms (in Section 6.7.3). Our guest does not collude with its neighbors.

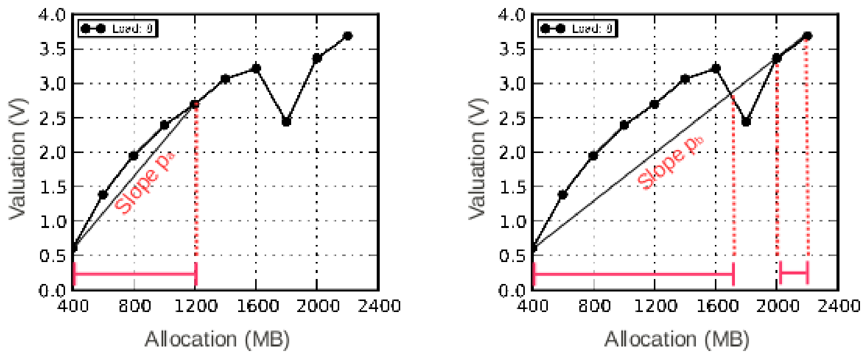
6.7.1 Choosing the bid price p

In this section we assume the maximal desired memory quantity q is given. For the simple case of an exact desired memory quantity ($m = 1, r^m = q^m = q$), for any value q, bidding the mean unit valuation of the desired quantity $p(q) = \frac{V(base+q)-V(base)}{q}$ is the best strategy, no matter what the other guests do. By bidding lower than $p(q)$, the guest risks losing the auction, but by bidding higher it risks operating at a loss (paying more than what it thinks the memory is worth).

If the valuation function is (at least locally, in the range up to q) concave monotonically rising, bidding $p(q)$ is still the best strategy for q regardless of other guests' bids: $p(q)$ is the guest's minimal valuation for the range because the unit valuation drops with the quantity. See for example Figure 6.3(a), where the valuation function is above the line connecting the valuation of 1200 MB with the base (400 MB) valuation.

For other valuation functions, where the unit valuation may rise locally with quantity, the guest avoids getting quantities for which the unit valuation is lower than the bid price by covering them with the aforementioned forbidden ranges. This coverage ensures that the client never operates at a loss. Such a case is demonstrated in Figure 6.3(b), where the range $[1700, 2000]$ MB is forbidden.

When the guest uses at least one forbidden range, bidding $p(q)$ still protects the guest from operating at a loss, but it is not necessarily the best



(a) Single range (b) Multiple range

Figure 6.3: Strategies for choice of unit price for two maximal quantities, using the same valuation function. Figure 6.3(a) demonstrates a single desired range strategy for a concave monotonically rising part of the valuation function. Figure 6.3(b) demonstrates a multiple desired range strategy for a non-concave, not even monotonically rising part of the valuation function.

strategy. For learning purposes, the guest can choose, according to its level of risk aversion, to retain q, but lower the bid price if it allows it to decrease its forbidden ranges. This will make the guest more flexible regarding the memory quantities it can take, and enable the host to give it a partial allocation in more cases, when the alternative might be not getting any memory quantity at all.

However, in a *steady state*, the guest already knows how much memory it can get for any bid it makes. The guest is incentivized to raise its bid price to a maximum, and by this increase the exclusion compensation that other guests pay, and make them more considerate. Hence, our guests always bid $p(q^m)$.

6.7.2 Choosing q^m

The guest chooses q^m that maximizes U_{est} in a steady state. U_{est} is assessed by estimating the quantity of memory it will get, which is defined for simplicity as

$$q_{est}(p, q^m) = \begin{cases} q^m & p > p_{min} \\ 0 & p \leq p_{min} \end{cases} \tag{6.4}$$

where p_{min} is the lowest price the guest can offer and still have a chance of getting any memory at all. The utility estimation also requires an estimation of the unit price to be paid for the allotted memory amount, p_{est}. The estimated utility is defined as:

$$\begin{aligned} U_{est}(q^m) &= V(base + q_{est}(p, q^m)) - Y \\ &\quad - p_{est}(q_{est}(p, q^m)) \cdot q_{est}(p, q^m), \end{aligned} \tag{6.5}$$

where Y denotes the known part of the cost (which is paid for the *base* memory), and if p_{est} is needed, it is assessed according to Section 6.7.3.

For concave valuation functions $V(\cdot)$ $U_{est}(q^m)$ is maximized when $p(q^m) = p_{min}$. In such cases, the guest must only estimate and predict p_{min} to bid optimally. For other functions, the guest needs to evaluate the full expression in Equation 6.5, including p_{est} and q_{est}, to find $\arg\max_{q^m}(U_{est})$. If several values of q^m maximize U_{est}, the guest prefers to bid with higher p values, which improve its chances of winning the auction. For non-concave valuation functions, the full expression in Equation 6.5 must be evaluated.

6.7.3 Evaluating Guest Utility

The guest evaluates p_{min} for the current round on the basis of ten recent borderline bids. q_{est} is predicted on the basis of p_{min}, according to Equation 6.4. The price to be paid, p', depends mainly on non-granted bids. To predict it, the guest maintains a historical table of (p', q') pairs, and uses it as a basic estimate for p_{est}. The p_{est} estimate is further bounded from above by the highest losing bid price in the last auction round.

6.8 Experimental Setup

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

Alternative Memory Allocation Methods. *Static* is a fixed allocation of the same amount of memory to each guest without any overcommitment. *Host-swapping* is the same as static except the host is allowed to swap guest memory to balance memory between guests as it sees fit. The *Memory Overcommitment Manager (MOM)* [86] collects data from its guests to learn about their memory pressure and continuously adjusts their balloon sizes to make the guests feel the same memory pressure as the host.

Workloads. To experiment with overcommitment trade-offs, we needed benchmarks of *dynamic memory applications*: applications that can improve their performance when given more memory on-the-fly over a large range of memory quantities, and can return memory to the system when needed. We experimented with a modified *dynamic memcached* and with *MemoryConsumer*, a dedicated dynamic memory benchmark. Both applications interacted with the Ginseng guest agent to dynamically adjust their heap sizes when they won or lost memory.

Dynamic memcached is a version of memcached that changes its heap size on the fly to respond to guest memory changes. Memcached was driven by a *memslap* client. The application's performance is defined as the "get" hit rate. [1]

MemoryConsumer is a dynamic memory benchmark. It tries to write to a random 1MB-sized cell out of a range of 1950 cells. If the address is within the range of memory currently available to the program, 1MB of data is actually written to the memory address, and it is considered a hit. After each attempt, whether a hit or a miss, it sleeps for 0.1 seconds, so that misses cost time. The application's performance is defined as the hit rate.

We profiled the performance of each workload with varying amounts of memory to create its $perf(mem, load)$ function. We measured performance under different loads for four concurrent guests without memory overcommitment, as also done by Hines et al. [61]. We gradually increased and decreased the physical memory in small steps, waiting in each step for the performance to stabilize. For memcached we waited and measured the performance for 200 seconds, and for MemoryConsumer for 60 seconds. The $perf(mem, load)$ graphs can be seen in Figure 6.2(a) for the dynamic Memcached and Figure 6.2(d) for MemoryConsumer.

Load. We defined "load" for memcached and MemoryConsumer as the number of concurrent requests being made. We used two load schemes: *static loads*, where each guest's load is constant over time, and coordinated *dynamic*

[1]Dynamic-memcached is available from `https://github.com/ladypine/memcached`.

loads. In coordinated dynamic loads, each pair of guests exchange their loads every T_{load}. The load-exchange timing is not coordinated among the different guest pairs in the experiments. Loads are in the range $[2, 10]$. The total load is always the number of guests $\times 6$, so that the aggregate hit rate of different experiments will be comparable.

Machine Setup. We used a cloud host with 12GB of RAM and two Intel(R) Xeon(R) E5620 CPUs @ 2.40GHz with 12MB LLC. Each CPU has 4 cores with hyper-threading enabled, for a total of 16 hardware threads. The host ran Linux with kernel `2.6.35-31-server #62-Ubuntu`, and the guests ran `3.2.0-29-generic #46-Ubuntu`. To reduce measurement noise, we disabled EIST, NUMA, and C-STATE in the BIOS and Kernel Samepage Merging (KSM) [16] in the host kernel. To prevent networking bottlenecks, we increased the network buffers. We dedicated hardware thread 0 to the host and pinned the guests to hardware threads $1 \ldots N$. When the host also drove the load for memcached, memslap processes were randomly (uniformly) pinned to threads $(N + 1) \ldots 15$.

Memory Division. 0.75GB were dedicated to the host. Thus, in static allocation experiments, each guest got $11.25GB/N$, where N denotes the number of guests. To allow guests to both grow and shrink their memory allocations, we configured all guests with a high *maximal memory* of 10GB, most of which was occupied by balloons, leaving each guest with a smaller *initial memory*. In Ginseng experiments, we started the guests with initial memory equal to their *bare* memory (0.6GB) and limited the sum of current memories to $11.25GB$.

When using host-swapping based methods (static with host-swapping and MOM), extensive host-swapping caused the host to freeze when the maximal guest memory was set to 10GB. Hence we also compared against hinted (white-box) methods, in which the maximal memory of each guest was configured as 2GB instead of 10GB. In our experiments, 2GB is the most memory any rational guest would ask for, since performance remains flat with any additional memory beyond 2GB. This white-box configuration, which is based on our knowledge of the experiment design, is intended to get the best performance out of the alternative memory allocation methods. The initial and maximal memory values are summarized in Table 6.1.

Reducing Guest Swapping. Bare metal operating systems shield applications from memory pressure by paging memory out and by clearing

Method/Memory (GB)	Initial	Maximal
Ginseng	*bare*	10
Static	$11.25/N$	—
MOM	*bare*	10
Host-swapping	10	—
Hinted MOM	*bare*	2
Hinted host-swapping	2	—

Table 6.1: Guest configuration: initial and maximal memory values for each overcommitment method.

buffers and caches, but dynamic-memory applications should be exposed to memory pressure to respond to it. To this end we minimized guest swapping by setting `vm.min_free_kbytes` to 0.

Reducing Indirect Overcommitment. Bare metal operating systems keep some memory free, in case of sudden memory pressure. In a virtualized system, the hypervisor can indirectly overcommit this memory by giving it to other operating systems while it is not in use; the hypervisor relies on its ability to page out guests if and when sudden memory pressure occurs. Since we focus on direct overcommitment (e.g., using balloons) we set the tunable knob `vm.overcommit_memory` to 1 in our guests, thus reducing the amount of memory they maintain which the host can indirectly overcommit.

Time Scales. Three time scales define the usability of memory borrowing and therefore the limits to the experiments we conducted: a typical time that passes before the change in physical memory begins to affect performance, T_{memory}; the time between auction rounds, $T_{auction}$; a typical time scale in which conditions (e.g., load) change, T_{load}. Useful memory borrowing requires $T_{load} >> T_{memory}$. This condition is also necessary for on-line learning of memory valuation. To evaluate T_{memory}, we performed large step tests, making abrupt sizable changes in the physical memory and measuring the time it took the performance to stabilize. We empirically determined good values for T_{load} on the basis of step tests results: 1000 seconds for memcached experiments, whereas for MemoryConsumer 200 seconds are enough. We also used those step tests to verify that major faults (swapping) were insignificant, and to verify that the performance measurement method was getting enough time to evaluate the performance. For example, memslap required 200 seconds to start experiencing cache misses.

In realistic setups providers should set $T_{auction} < T_{load}$. Therefore, we set $T_{auction}$ to 12 seconds. In each 12-second auction round the host waited 3 seconds for guest bids and then spent 1 second computing the auction's result and notifying the guests. The guests were then allowed 8 seconds to prepare in case they lost memory.

6.9 Performance Evaluation

This section attempts to answer the following four questions: (1) which allocation method provides the best social welfare? (2) how does the reclaim factor affect social welfare? (3) what are the host revenue, wasted memory, ties, and inefficiency in a Ginseng system? (4) how important (and accurate) is off-line profiling of guest performance?

6.9.1 Comparing Social Welfare

We begin by evaluating the social welfare achieved by Ginseng vs. each of the five other methods listed in Table 6.1 for a varying number of guests on the same physical host. We evaluate Memcached guests and MemoryConsumer guests in separate sets of experiments. In each experiment set, guests were subject to dynamic loads. Each Memcached experiment lasted 60 minutes, with $T_{load} = 1000$ seconds. Each MemoryConsumer experiment lasted 30 minutes with $T_{load} = 200$ seconds. For each experiment we present average results of 5 experiments. The reclaim factor was set to 1. Ginseng guests use the strategy described in Section 6.7.

In both benchmarks, *perf*(*mem*) is a concave function. To evaluate Ginseng's abilities over non-concave functions, we used performance valuation functions $V_p(perf)$ that make the resulting composed valuation function $V(mem)$ non-concave.

In the first experiment set (MemoryConsumer), each guest i's valuation function is defined as

$$V_i(mem) = f_i \cdot (perf(mem))^2 , \tag{6.6}$$

where the f_i values were drawn from the *Pareto distribution*, a widely used model for income and asset distributions [129]. We used a Pareto index of

1.1, which is reasonable for income distributions [130], and a lower bound of $10^{-4}\frac{\$}{Khit}$.

The "square of performance" valuation function is characteristic of on-line games and social networks, where the memory requirements are proportional to the number of the users, and the income is proportional to user interactions, which are proportional to the square of the number of users. The composed valuation function is drawn in Figure 6.4(a).

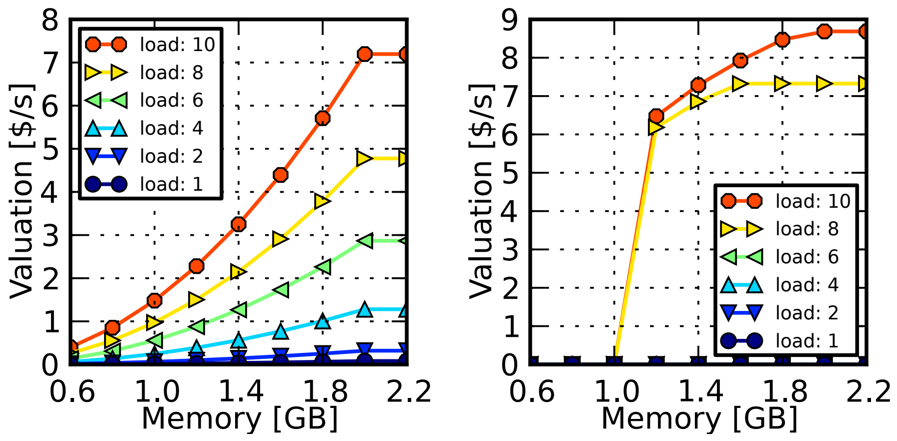
In the second experiment set (dynamic memcached), each guest i's valuation function is defined as $V(mem) = f_i \cdot perf(mem)$, where the f_i values were distributed according to a *Pareto distribution* with a Pareto index of 1.36, another reasonable value for income distributions, bounded in the range $[10^{-4}, 100]\frac{\$}{Khit}$. The bounding represents the fact that on-line trading does not span the whole range of human transactions: some are too cheap or too expensive to be made on-line. The highest coefficient was set as:

$$f_1 = \begin{cases} 0.1\frac{\$}{Khit} & perf(mem) < 3.4\frac{Khit}{s} \\ 1.8\frac{\$}{Khit} & otherwise. \end{cases} \tag{6.7}$$

This sort of piecewise-linear valuation functions characterizes service level agreements that distinguish usage levels by unit price. The valuation function for the first guest is shown in Figure 6.4(b).

The social welfare of the different experiments is compared in Figure 6.5. The figures contain two upper bounds for the social welfare, achieved through a simulator, which is presented in Section 6.9.3. The tighter bound results from a simulation of Ginseng itself, and the looser bound results from a white-box on-line simulation. The MOM and host-swapping methods yield negligible social welfare values for these experiments, and are not presented.

As can be seen in Figure 6.5, Ginseng achieves much better social welfare than any other allocation method for both workloads. It improves social welfare by up to $\times 15.8$ for memcached and up to $\times 6.2$ for MemoryConsumer, compared with both black-box approaches (static) and white-box approaches (hinted-mom). Since each guest is allocated a fixed amount of memory (*bare*) on startup, as the number of guests increases, the potential for social welfare increases, but our host has less free memory to auction; hence the relative peak in social welfare for 7 guests (MemoryConsumer). In the Memcached experiment the relative peak is flat because the first guest's valuation is



(a) MemoryConsumer (b) Dynamic Memcached

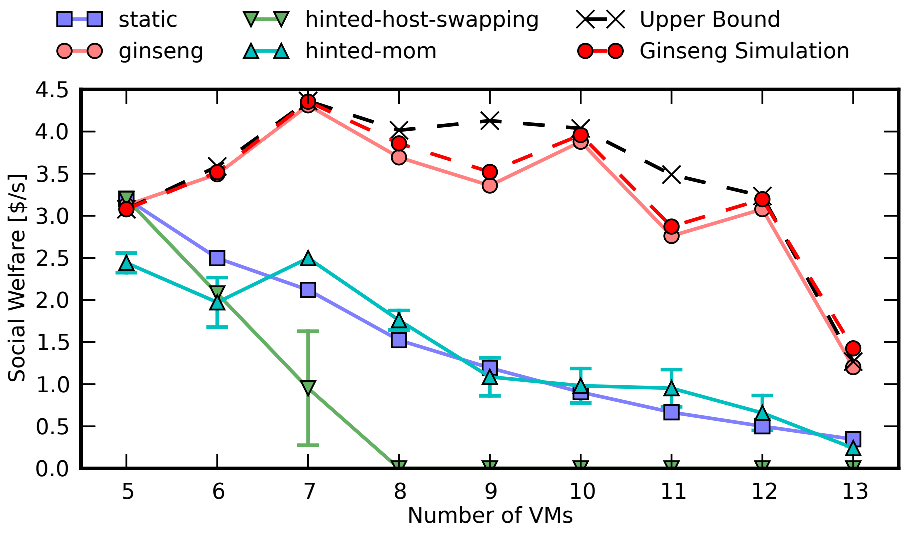
Figure 6.4: Valuation functions for different loads

significantly larger than the rest. In both experiment sets, Ginseng achieves 83%–100% of the optimal social welfare. The sharp decline in Ginseng's social welfare for 13 guests comes when Ginseng no longer has enough free memory to answer even the needs of the most valuable guest.

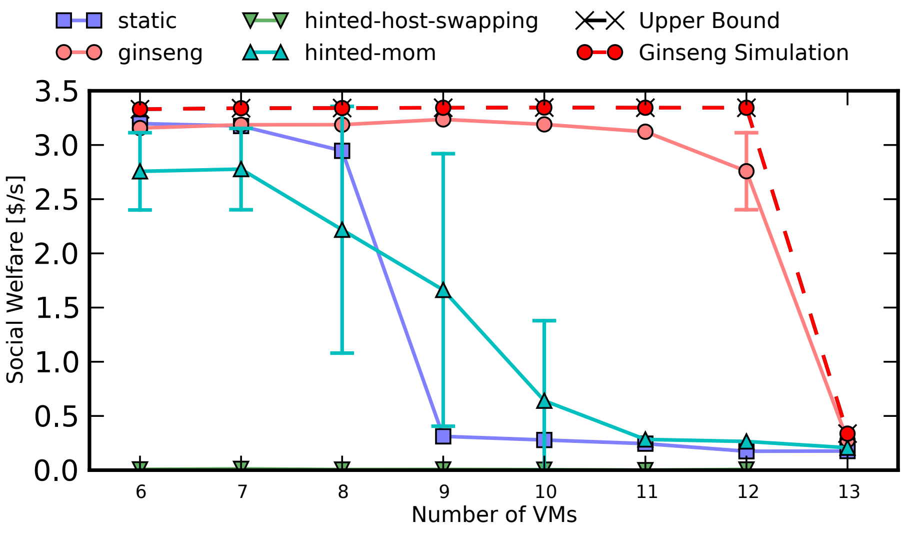
As can be seen in Figure 6.6, in which the performance of the different methods is compared, the improvement that Ginseng delivers does not come at a cost when the aggregate performance is considered: Ginseng's aggregate performance is roughly equivalent to the performance of the better methods, namely hinted-MOM and static division.

6.9.2 Reclaim Factor Analysis

To examine the impact of the reclaim factor on social welfare in a real system, we combined a statically loaded memcached guest, which is vulnerable to allocation cycles, with a dynamically loaded MemoryConsumer guest, whose load changed every 60 seconds. Each guest got $bare = 0.8GB$. The results of this experiment for various reclaim factors are given in Figure 6.7. As can be seen in Figure 6.8, lowering the reclaim factor reduces the penalty that MemoryConsumer suffers when conditions change and it needs to change its strategy. When the reclaim factor is lower, the system gets sluggish and does not stabilize before the load changes again.



(a) MemoryConsumer, valuation is square of performance



(b) Memcached, first guest valuation is piecewise linear

Figure 6.5: Social welfare (mean and standard deviation) under different allocation schemes as a function of the number of guests, for dynamic load experiments. The dashed lines indicate simulation-based upper bounds on Ginseng's social welfare.

In real systems there is a tradeoff between system responsiveness and limitation of allocation cycles that does not exist in simulations. This experiment proves the importance of the reclaim factor as a knob for the host to control the system's stability.

6.9.3 Simulated Experiments

To evaluate various aspects of Ginseng's performance, we augmented the experimental results with simulated experiments. The simulator was created by re-using Ginseng's algorithmic core with simulated guests that use the same strategy as real guests; our simulations can be seen, therefore, as emulations of the Ginseng process.

In our simulations we measured social welfare, sum of guest utilities, and host revenue. In addition we measured an upper bound on waste, ties, and the inefficiency, as explained below.

Waste and Ties. The simulations did not account for the impact of fast ownership changes on the actual value obtained from memory, because the performance of simulated guests stabilized immediately once the memory size changes. We defined the *upper bound on memory waste due to ownership changes* as the maximal total allocated memory minus the static allocations over the last 40 auction rounds:

$$Waste(t) = \max_{\tau=t-40}^{t} \sum_{i=1}^{N} final_i(\tau) - \sum_{i=1}^{N} \min_{\tau=t-40}^{t} final_i(\tau).$$

Waste can be caused by fast load changes ($T_{load} \lesssim T_{memory}$) as well as by *cycling allocations.* Ties do not cause cycles because when they are broken, preference is given to the previous owner, leading to a stable solution. Since the simulations lasted 1000 rounds, a moving window of 40 rounds filtered out transient effects while still catching large cycles.

Inefficiency. The simulation environment also enabled an on-line white-box computation of the optimal allocation which we compare with Ginseng's experimental results. The optimal allocation results from a centralized constraint-satisfaction algorithm, with which the guests share their full valuation functions. The social welfare that originates from the optimal allocation is denoted by $SW_{\max}$. We computed the *inefficiency*, defined as $1 - \frac{SW}{SW_{\max}}$, using the simulation results. Inefficiency quantifies the aggregate valuation degradation experienced due to the mechanism design and the bidding language.

Simulation Setup. We ran the simulations with 10 MemoryConsumer guests, with identical linear valuation functions, with *bare* memories of $0.8GB$ and with *static loads* that sum up to a total load of 60, the same total load as in the 10-guest experiments. We performed a parametric sweep over the reclaim factor and the total physical memory of the system, in the range of 11–20GB, corresponding with decreasing memory overcommitment ratios of 4–1. We defined the *memory overcommitment ratio* as the sum of each guest's maximal demand for rented memory (at any point in time during the experiment) divided by the memory that was available for rent at $t = 0$. The higher the overcommitment ratio, the fiercer is the competition for memory.

Simulation Results can be seen in Figure 6.9. In the static MemoryConsumer simulations, the reclaim factor has a low impact on the host revenue and sum of guest utilities, and no impact on the social welfare and the inefficiency. The inefficiency ranges from 0 in a well provisioned system to 35% for an overcommitment ratio of 3.5. The inefficiency can be reduced by using a richer bidding language [93]. There are no ties in the MemoryConsumer simulations, and usually no waste either. We attribute the lack of ties to the different slopes of the MemoryConsumer performance graphs for the different loads (in Figure 6.2(d)). In contrast, memcached performance graphs share the same slope in their lower parts (Figure 6.2(a)), and indeed in memcached simulations (not shown due to lack of space) up to 80% of the simulation rounds resulted in ties. This is consistent with our design assumption in Section 6.4.2, that ties do happen in real life, and supports our claim that they must be efficiently dealt with.

Discussion: Host Revenue. Ginseng does not attempt to maximize host revenue directly. Instead, it assumes that the host charges an admittance fee for cloud services and maximizes the aggregate client satisfaction (the social welfare). Maximizing social welfare improves host revenues indirectly because better-satisfied guests are willing to pay more. Likewise, improving each cloud host's hardware (memory) utilization should allow the provider to run more guests on each host. Nevertheless, it is interesting to examine the host's direct revenues.

For small overcommitment ratios (< 1.3) the host revenue is negligible ($< 5\%$ of the maximal social welfare): the guests' profits (Figure 6.9(b)) equal their valuations (Figure 6.9(a)). As the overcommitment ratio increases, host revenue decreases because there is less memory to rent. When the host revenue is zero and the social welfare is high, as is the case for the low overcommitment range, the system is functioning well and is in a state of equilibrium, where guests are more considerate of their neighbors thanks to the exclusion compensation principle. Our guests reach such equilibria using indirect negotiations that result from their learning strategy (in Section 6.7.3). More sophisticated guests may directly negotiate to ease their way into an equilibrium [18].

We also ran simulations with dynamic loads, for an overcommitment ratio of 1.5, changing the dynamicity of the system by controlling the ratio of T_{load} and $T_{auction}$. According to Figure 6.9(f), the social welfare in the simulation improve as the system is less dynamic and as the reclaim factor is increased.

6.9.4 Impact of Off-Line Profiling

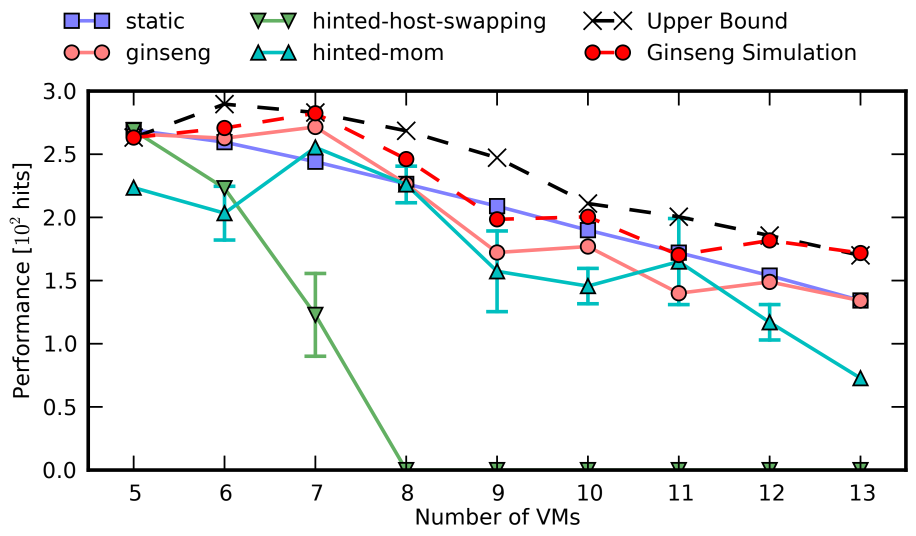
In our experiments we used performance graphs that were measured in advance in a controlled environment. In real life, artificial intelligence methods should be used to collect such data on-the-fly. Since the accuracy of the best on-the-fly methods is bounded by the accuracy of hindsight, we can bound the impact of refraining from on-the-fly evaluation on the performance graphs. In Figure 6.10 we compare our benchmarks' predicted performance (according to measured load and memory quantities, and using Figure 6.2) with performance values measured during Ginseng experiments for the same loads and memory quantities. The experimental values were collected after the memory usage stabilized (more than T_{memory} after a memory change). The comparison shows that the profiled data is accurate enough, as can be seen when comparing Ginseng's experiment results to its simulations in Figure 6.5.

6.10 Conclusions

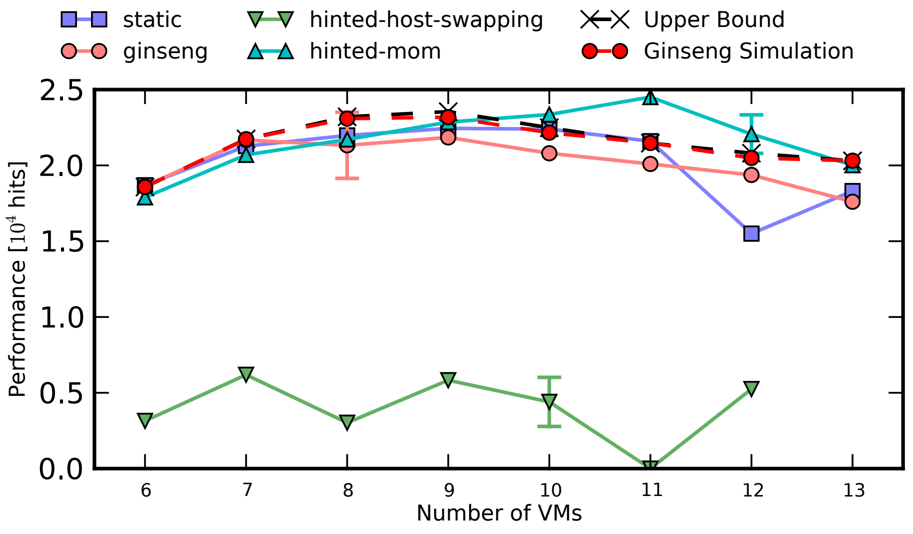
Ginseng is the first cloud platform that allocates physical memory to selfish black-box guests while maximizing their aggregate benefit. It does so using the MPSP auction, in which even guests with non-concave valuation of memory are incentivized to bid their true valuations for the memory they request. Using the MPSP auction, Ginseng achieves an order of magnitude of improvement in the social welfare.

Although Ginseng focuses on selfish guests, it can also benefit altruistic guests (e.g., when all guests are owned by the same economic entity). In this case, economic valuations can distinguish between guests that perform the same function for different purposes, such as a test server vs. a production server.

Ginseng is the first concrete step towards the Resource-as-a-Service (RaaS) cloud [3]. In the RaaS cloud, all resources, not just memory, will be bought and sold on the fly. Extending Ginseng to resources other than physical memory remains as future work.



(a) MemoryConsumer, valuation is square of performance



(b) Memcached, first guest valuation is piecewise linear

Figure 6.6: Performance (mean and standard deviation) under different allocation schemes as a function of the number of guests, for dynamic load experiments. The dashed lines indicate the performance according to the simulations that yield an upper bound on the social welfare, as indicated in Figure 6.5.

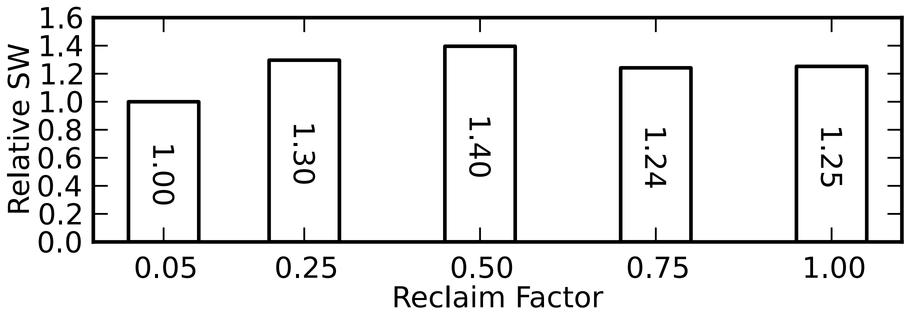


Figure 6.7: Impact of reclaim factor on social welfare for a mixed workload of memcached and MemoryConsumer

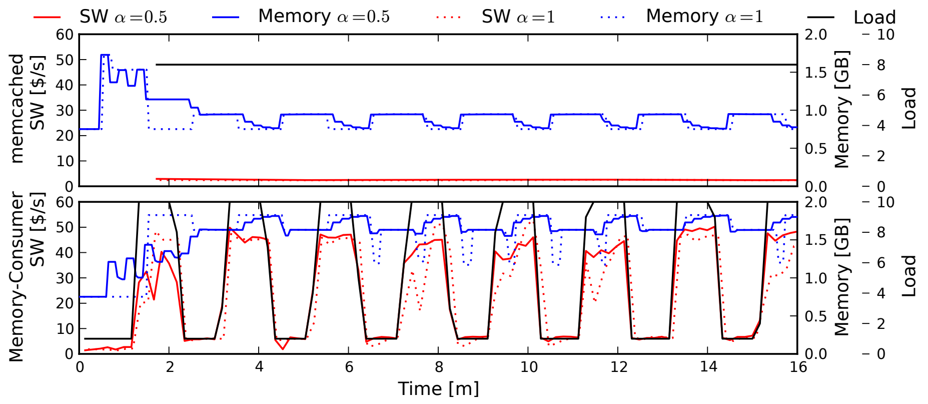
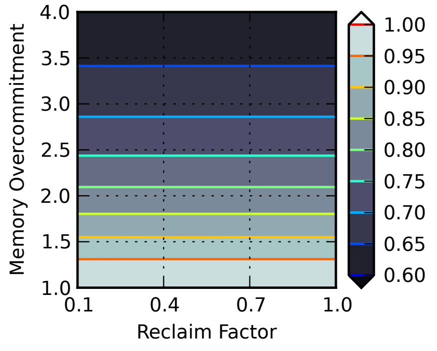
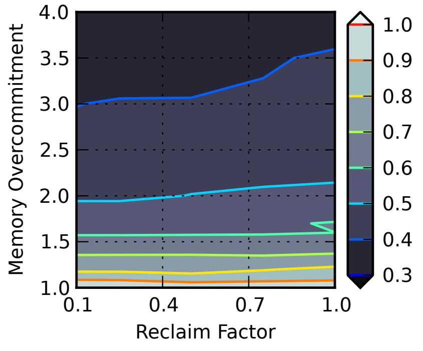


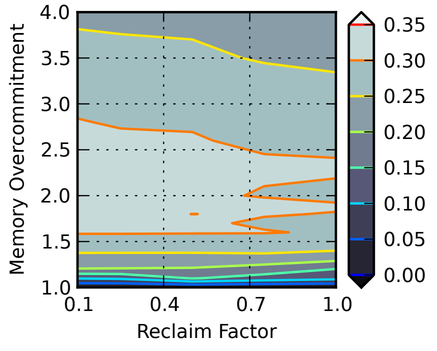
Figure 6.8: Two mixed-workload experiment traces of utility and memory allocation



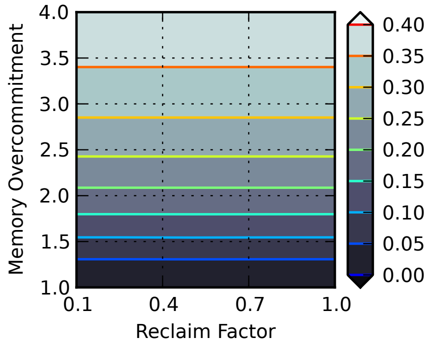
(a) Social welfare



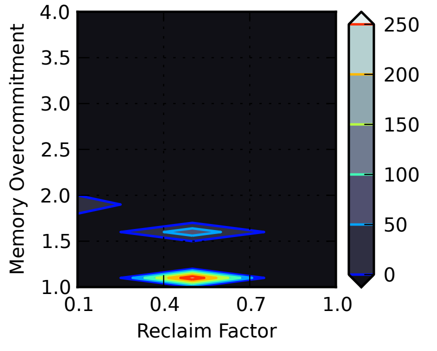
(b) Sum of guest utilities



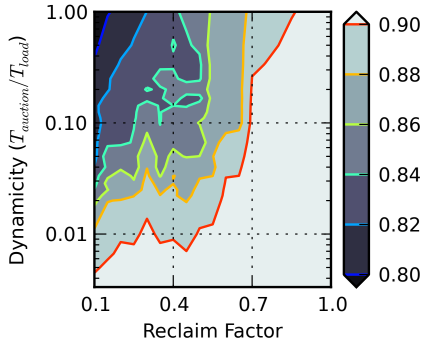
(c) Host revenue



(d) Inefficiency

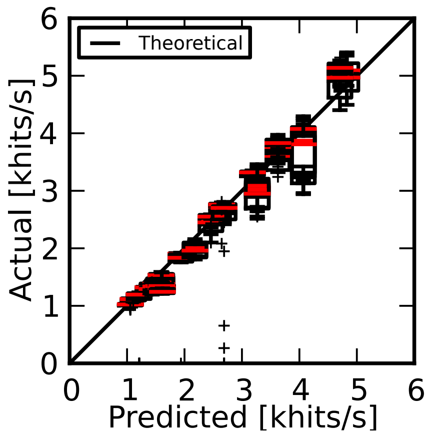


(e) Waste (MB)

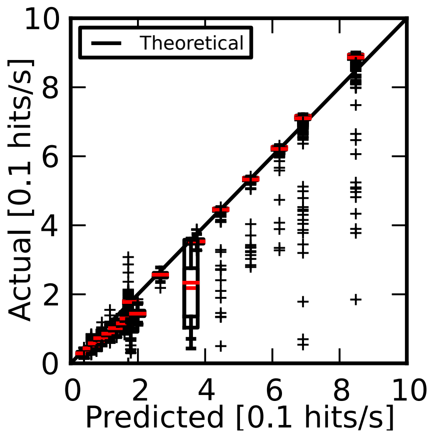


(f) Social welfare (dynamic)

Figure 6.9: Impact of reclaim factor and overcommitment ratio on Ginseng time-averaged performance for MemoryConsumer guests. Figure 6.9(f) shows the impact of the reclaim factor and dynamicity on social welfare in dynamic simulations for an overcommitment ratio of 1.5. Social welfare, revenue and profit values are normalized by the maximal social welfare achieved in the parametric sweep.



(a) memcached



(b) MemoryConsumer

Figure 6.10: Comparison of predicted performance values (according to the profile graphs, given load and memory allocation) with measured performance.

Chapter 7

RaaS: Additional Research Directions

In this section we outline several research directions that might build on the concept of RaaS, with a focus on our implementation of Ginseng. We begin with challenges within the Ginseng scope, continue to outline an extension of Ginseng to a multi-resource system, turn to analyze side-channel attacks that can be made on Ginseng and their ramifications, and conclude with a proposal for a mechanism with improved stability.

7.1 Single Resource

7.1.1 Game-Theoretic Challenges

The MPSP auction is uncharted territory with regard to game theory: the reclaim factor, which reduces waste, introduces private guest-states that change over time and affect the guests' valuation of additional memory chunks. When the base memory changes, the forbidden ranges and the bid price change as well (because the base memory and its valuation are their point of reference). In addition, valuations may change at random due to dynamic loads. In this work we only analyzed guest strategies with a horizon of one auction round. In simpler problems of repeated games without private states, there may be rational strategies which are irrational to play as a stage game (single round). This may also be the case here: there are strategies that are irrational in the stateless game (with $\alpha = 1$), but are rational in the private-state game. For example, if a guest expects a fast increase of demand for memory, it can plan ahead and bid for more memory than it currently needs. It will benefit from keeping its payments lower for several rounds, until the system reaches a new equilibrium. Even in a stateless game, prediction of other guests' bids may incentivize a guest to lie about its valuation in a repeated VCG auction. Analysis of such strategies calls for new theoretic approaches.

7.1.2 Guest Logic

The current guest agent is rather simplistic. It does not communicate with its neighbors, nor does it make future plans (even when a low reclaim factor is used). There are many directions in which the the guest agent can evolve. The guest agent's accuracy can improve through learning. For example, if the agent is enhanced with a load anticipation capability, it will be able to bid according to predictions rather than according to the current load. If it communicates with the application, it can learn of an upcoming increased need for resources. If the guest communicates with its neighbors, it can trade in resources that it previously rented for a low price, and sell them for a higher price. If the guest collects data on its performance on the fly, it will be able to update the *perf*(*mem*) function on-line, and thus adapt it to changing conditions. For example, if the network has become a bottleneck and a memory increase does not improve the performance, the guest can learn this on-line and stop bidding for large memory increases.

7.1.3 Host Logic

The reclaim factor is a knob that requires a policy, a heuristic to operate it. It can be changed dynamically, but the host needs to decide how to do so. The dynamic changing of the reclaim factor must be made in light of the total amount of memory that was bid for, the available memory and the provider's plans for adding or removing guests. Another type of information that might affect the reclaim factor is black-box measurements, which might be used to assess the rate at which conditions change.

7.1.4 Provider Logic: Global Cloud View

The host logic must be combined with a global cloud view mechanism that matches guests to hosts. The matching should include a per-host pricing of entrance to that host. Hosts with lower resource pressure should have a higher entrance fee. The global view algorithm must also be combined with a live migration recommendation system that optimizes the social welfare while considering the migration overhead.

7.1.5 Minimal Price

When the host does not lose from renting a resource, maximizing the social welfare of all the guests (the aggregate valuation) also maximizes the aggregate utility of the guests and the host:

$$SW = \sum_{i=1}^{N} U_i + U_{host} = \sum_{i=1}^{N} (V_i(final_i) - p'_i) + \sum_{i=1}^{N} p'_i = \sum_{i=1}^{N} V_i(final_i). \quad (7.1)$$

In this case, renting spare resources for extremely low prices (and even for free) still improves the social welfare, as long as the renting guest benefits from the resource.

However, the host may prefer not to rent the resource. Hosts that can power down unused resources (e.g., memory segments or cores) value them at least as the difference of the active and suspended resource operational costs. In addition, the host may consider the wear and tear (in particular in Flash devices) and missed opportunity costs (for being less responsive to future resource pressure). In such cases, the social welfare is higher when the memory is rented only to clients who value it more than the host does. To this end, the host's valuation of resources can be represented in the auction as a special guest, whose bid is the minimal price for the full system's memory.

7.1.6 Memory Shedding

When the reclaim factor is smaller than 1, the guest's base memory changes with each auction round. To make for a Pareto-efficient system, the guest must be allowed to promptly shed any non-required memory, without waiting for its base memory to dwindle over time. Otherwise, the gradual decay might lead the base memory into one of the guest's forbidden ranges, thus forcing the guest to pay for memory that degrades its performance. When the guest's requirements change quickly but the reclaim factor is small, shedding unrestricted amounts of extra memory within a single round will protect the guest from system sluggishness without hurting the other guests.

Memory shedding also enables the guest to adjust the price it pays for memory rented long ago. In the chunks method, memory is tagged according to the price for which it was rented. In time, the most expensive memory chunks remain. For concave monotonically rising functions, the payment converges from below to the exact valuation of the memory. However, if prices drop and memory pressure is lower, the guest should be able to announce memory shedding. This means that the guest is willing to lose all the memory it is currently renting, but if it wins it, it wins it for a low price (the current market price).

7.2 Multi-Resource Allocation

Ginseng can be expanded to a full RaaS [3] implementation, allocating multiple resources simultaneously (e.g., memory, I/O, and CPU) [3]. In this section we outline the algorithm for a multi-resource RaaS, without implementing it. The implementation is left for future work.

The expansion to multi-resource introduces the notions of *economic complements*: resources are called economic complements if guests would like to rent more of one resource when they rent more of the other. For example, consider an application that utilizes a core and 500MB in each thread. Guest A has 500MB and one core, and guest B has 1000MB and one core. They both perform at a rate of a single thread. However, guest B values an additional core more than guest A, because given an additional core, guest B will double its performance, while guest A's performance will remain the same. Gutman and Nissan [57] assume such utility functions, denoted *Leontief utilities*. However, resources may also be *economic substitutes*: resources are called *economic substitutes* if guests would like to rent less of one resource when they rent more of the other. For example, when using a caching application such as memcached, a guest which rents a large memory quantity will require less bandwidth to get the stored items, or it might require less CPU cycles to compute them again. Furthermore, the same resources might be substitutes for one guest, and complements to another. This dependency might be of high order. However, the guest does not have to state its full valuation function, but rather state its local manifestation, since Ginseng's bidding language includes the designation of desired ranges. For a small enough range, an approximation to the first order of the guest's valuation of resources is accurate enough. Formally, a guest's valuation of D resources $\vec{d} \in R_+^D$ can be multi-linearly approximated locally as

$$V(\vec{d}) = \sum_{v \in \{0,1\}^D} a_v \prod_{k=1}^{D} d_k^{v_k}. \tag{7.2}$$

We denote the coefficients a for short as a $\vec{a} \in R^{2^D}$. For the simplest multi-resource case, Equation 7.2 is reduced to a bilinear function:

$$V(d_1, d_2) = a_{00} + a_{01} d_1 + a_{10} d_2 + a_{11} d_1 d_2. \tag{7.3}$$

We set the valuation of the reference point $V(\vec{0}) = 0$, and thus $a_{\vec{0}} = 0$ and $2^D - 1$ free coefficients are left. This setting is consistent with using only one free coefficient in the single-resource case (the bid price p).

7.2.1 Bidding Language

The *extended bidding language* for D resources includes the unit-price coefficients $\vec{a}$, and a list of m desired ranges (D-dimensional boxes) in which the unit-price coefficients are valid. Formally, the bid is of the form

$$\begin{aligned} & \vec{a} && , (\vec{r}^1, \vec{q}^1) \ldots (\vec{r}^m, \vec{q}^m) && (7.4) \\ \textit{where} \quad & \vec{a} && \in R^D && (7.5) \\ \textit{and} \quad & \forall && j = 1 \ldots m, \quad \vec{r}^j, \vec{q}^j \in R_+^D. && (7.6) \end{aligned}$$

The bid is interpreted as a willingness to pay a global price according to Equation 7.2 for resources within the desired ranges. The resource unit-prices are computed as the partial derivatives derived from Equation 7.2:

$$p_k = \frac{\partial V(\vec{d})}{\partial d_k} \quad \forall 1 \le k \le D. \tag{7.7}$$

For the two resource case, we derive the resource unit-prices from Equation 7.3:

$$p_1 = \frac{\partial V(d_1, d_2)}{\partial d_1} = a_{01} + a_{11} d_2 \tag{7.8}$$

$$p_2 = a_{10} + a_{11} d_1 \tag{7.9}$$

Note that although the valuation might decrease with any resource (e.g., the valuation of cores might decrease with additional bandwidth), the prices must still be positive. If the prices are non-positive, the guest should not bid at all in this range.

7.2.2 Allocation Rule

The multi-resource allocation rule is an extension of the single resource rule. The single resource allocation rule is computed by attempting a divisible-good allocation, which is efficiently done by sorting the bids, and then splitting the case along a 2-tree structure if a forbidden range was split by the divisible good algorithm. However, multi-resource bids are not necessarily sortable by unit price.

Consider an auction for memory and bandwidth. There are two guests, A and B. Guest A bids a higher unit-price for bandwidth than guest B, while for memory guest B's unit price exceeds A's. Now suppose there is enough bandwidth for both guests, but memory is insufficient, so it is a *bottleneck resource* of the system. In this case, the multi-resource auction should be reduced to the single resource MPSP, the bids should be reduced to unit-prices for memory when bandwidth is at the full desired ranges, and guest B should be sorted before guest A. However, if bandwidth is the bottleneck resource, then unit-prices for memory matter, and the multi-resource auction should be reduced to a single-resource bandwidth auction, preferring guest A over guest B. These examples demonstrate the problem of sorting points in a multi-dimensional space. In the case of the bilinear bids the problem is even harder. In each bid, one resource's unit-price depends on the amount of other resources allocated to the guest. This means one bid's unit-price might exceed another bid's unit-price for one allocation, but be lower for another. Hence, even the divisible multi-resource allocation problem cannot be solved using a simple sorting of bids by their unit-prices, as done in the single resource auction.

We define the divisible multi-resource allocation problem as a linear programming problem: Find vectors $\vec{d_i} \in R_+^D \ \forall i = 1 \dots N$ that maximize the social welfare $SW = \sum_{i=1}^{N} V_i(\vec{d_i})$ under the constraints

$$0 \leq \sum_{i=1}^{N} d_{ik} \leq A_k \qquad \forall k = 1 \dots D \tag{7.10}$$

$$\left(\min_{j=1}^{m_i} r_{ik}^j \right) \leq d_{ik} \leq \left(\max_{j=1}^{m_i} q_{ik}^j \right) \qquad \forall i = 1 \dots N, \quad k = 1 \dots D, \tag{7.11}$$

where A_k denotes the amount of resource k available for auction. That is, the solution does not allocate more resources than the host can allocate, and the resources allocated to each guest are within a D-dimensional box that covers all of the guest's desired ranges.

The *indivisible multi-resource allocation* problem can be solved using a branching algorithm, which is an extension of the 2-tree in the single resource auction. In each step of the tree, a divisible multi-resource allocation is attempted. If the divisible allocation is a valid indivisible allocation, it is evaluated as a candidate for the optimal allocation. However, the resulting allocation may be *invalid*: it may allocate a guest with resource amounts outside any of its desired ranges, as demonstrated in the example in Figure 7.1. In this case, the guest's box, as defined for the divisible multi-resource algorithm, is divided to sub-boxes that do not contain the guest's allocated resource amounts.

The simplest division covers each desired range in a separate box, since each desired range is convex. However, there is an opportunity for complexity reduction for large numbers of desired ranges. An alternative method for sub-box division is demonstrated in Figure 7.4. First, a forbidden box is defined around the allocated amounts, such that it is a maximal box that has no intersection with any desired range. The maximal box is not unique: its shape is determined by the order of the dimensions in which the forbidden box's sizes are maximized, as demonstrated by the different maximal forbidden boxes in Figures 7.2 and 7.3.

Once the forbidden bounding box is chosen, the guest's covering box minus the forbidden range is expressed as a union of disjoint boxes. When possible, the division lines are chosen such that they do not divide any desired range. This is how the solid blue lines in Figure 7.4 were chosen. Then the bounding boxes are shrunk to minimal boxes covering the desired ranges, as demonstrated by the solid black rectangles in Figure 7.4. A covering box might shrink to zero volume (like the non-existent box which could have covered the high memory regimes in Figure 7.4) because it does not contain any desired ranges. Such shrunken boxes do not lead to a branch.

The choice of the forbidden box and branching option can affect the computations required to find the optimal allocation, but not the correctness of the algorithm or the maximal social welfare. The chosen box and branching option might also affect the order in which optimal allocations are discovered, so that if an order criterion is used in the process (e.g., in case of a tie, prefer the previously found allocation), the final allocation might also be affected.

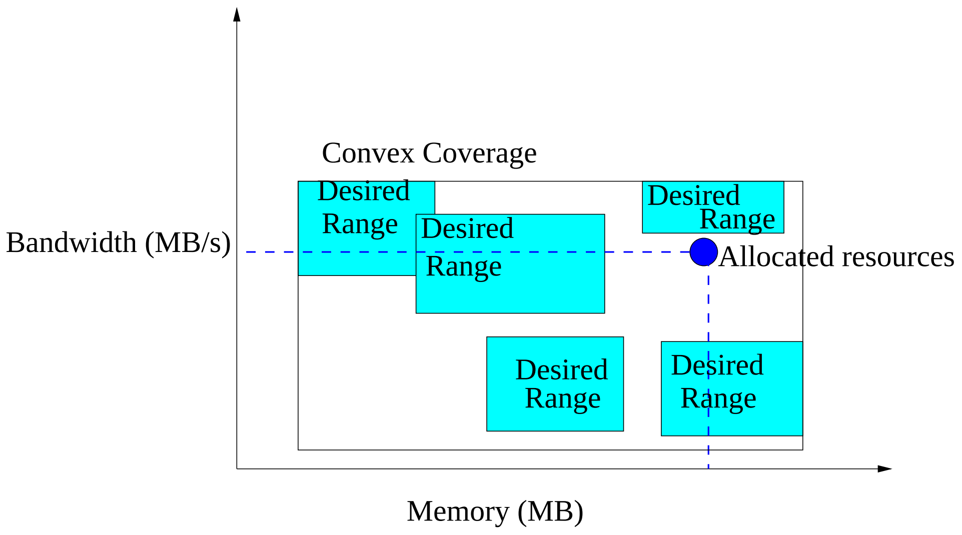


Figure 7.1: Indivisible multi-resource allocation. The blue point indicates the result of the divisible multi-resource auction for one guest, located outside this guest's divisible ranges, but inside those ranges' covering box.

7.3 Side-Channel Attacks

Memory overcommitment and dynamic memory allocation methods are a side channel through which information can leak from one guest to its neigh-

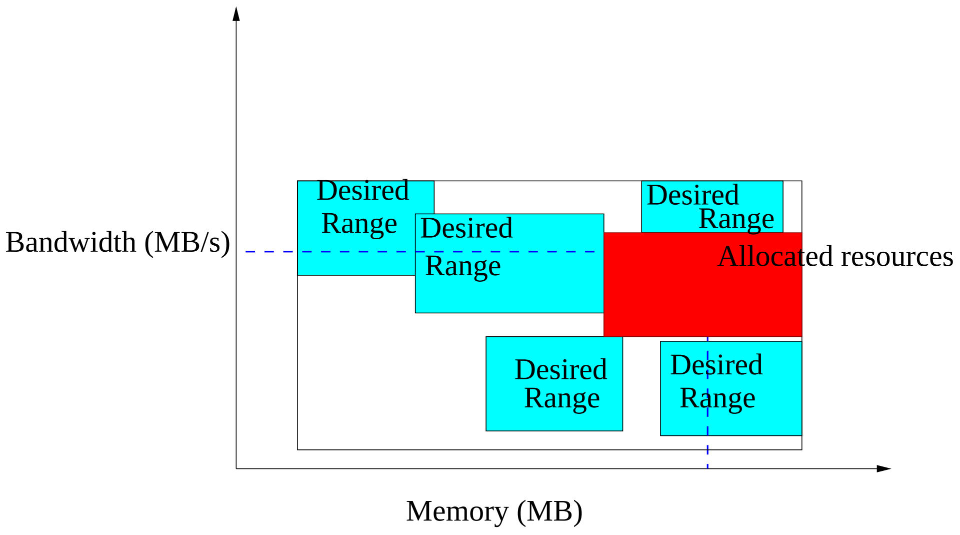


Figure 7.2: Forbidden bounding box choice (red box)—maximizing the bandwidth dimension first.

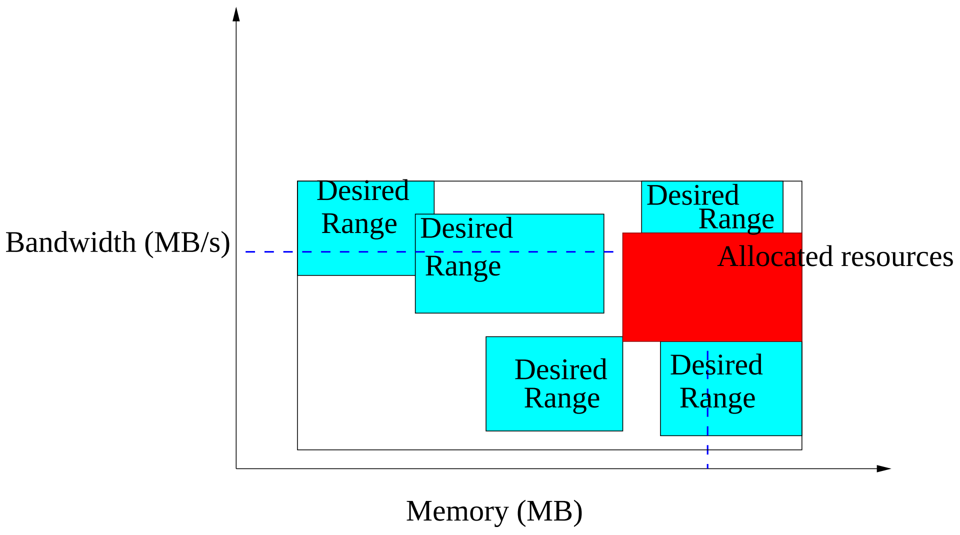


Figure 7.3: Forbidden bounding box choice (red box)—maximizing the memory dimension first.

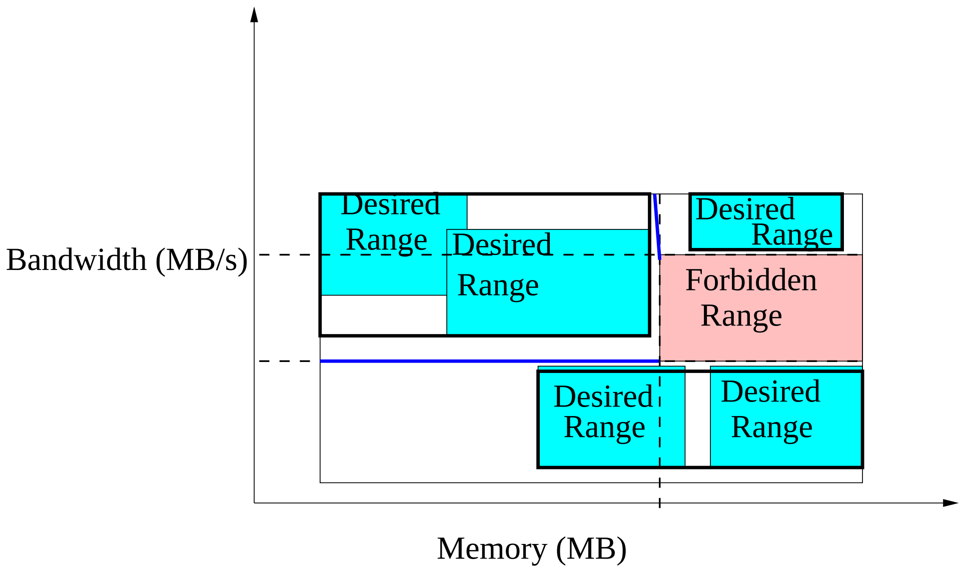


Figure 7.4: Indivisible multi-resource allocation branching. The pink box is a forbidden range that includes the allocated resource amounts.

bor. A hostile guest might then use this information, or even just the fact that it might leak, to harm other guests or the host. In this section, the term *neighbor* denotes a non-malicious guest, co-located with a malicious guest on the same physical machine.

7.3.1 Information Leakage

In Ginseng, the *borderline bids* (the bid unit-prices of the accepted bid with the lowest unit-price and of the rejected bid with the highest unit-price) are announced by the host, because they are important to the convergence of the bidding process. Were they not announced, the guest could still approximate them by recording how much it was charged. One way to approximate them is for the malicious guest to bid for a small memory quantity while gradually raising its bid price until it is allocated some memory, thus discovering the lowest accepted bid's unit-price, which would then equal its own bid. The unit-price that this malicious guest is charged would be the highest rejected bid's unit-price.

In a system with only two guests, a guest can easily deduce full information about its neighbor's memory valuation function. It can bid repeatedly for all of the system's memory with a gradually increasing low bid price. In a steady state (that is, if the neighbor does not change his bid), the guest can delimit the unit-price in its neighbor's bid between two of its own bids' unit-prices. Its own highest bid that did not win the full memory is a lower bound for the unit-price of the neighbor's bid, and its own lowest fully-winning bid's unit-price serves as an upper bound. These bounds can be as close as the guest wishes, and thus reveal the unit-price of the neighbor's bid to any desired accuracy. This method can also inform the guest of its neighbor's required memory for this price by comparing the memory quantities it won in those two bids that close on the neighbor's bid. This method will inform the guest of one p, q^m point on the neighbor's valuation function. By collecting several such points, the guest can learn the neighbor's valuation function for the relevant ranges (ranges which the neighbor used for bidding).

The malicious guest can even monitor changes in its neighbor's memory valuation over time. As we demonstrated earlier, changes in the neighbor's load can result in changes in its memory valuation function. In addition, valuation functions may change due to the subjective importance of the neighbor's workload. By comparing the evolution of the function $p(q^m)$ over time, the guest can learn when the neighbor needs memory more (that is, when the memory is more important to the neighbor). Such information about times in which resources are more crucial to the neighbor's operation might be used by the malicious guest to initiate hostile, disruptive activity. However, the information gathering activities themselves damage the neighbor, causing it to lose an auction or only partially win it.

In a system with more than two guests, deducing the neighbors' valuation functions is harder but not impossible, because the information gathering process can be repeated, and thus noise (coming form other neighbors) can be cleaned. Furthermore, to disrupt the system, a malicious guest does not need to know its neighbors' full valuation function. It is enough for the malicious guest to discover the memory quantity requested by all the other guests together and the highest unit-price of the bids.

7.3.2 Disruptive Activity

A hostile client may design a hostile guest that we denote a *soldier*. The soldier causes damage to its neighbors while maintaining its own costs reasonably low. The lower its costs are, the more soldiers the hostile client can afford to operate.

Soldiers can invalidate their neighbors' caches. If the soldier bids a high unit-price for a large memory quantity, it takes hold of memory that was previously rented to other guests. The ownership change requires that the host itself clear the memory of its contents, to prevent information leak between guests. When the soldier allows the neighbor to win (in the next auction), the memory is already wiped out, and needs to be slowly filled again with cached items and files. The soldier pays for the rental of a large memory quantity, but only for a single auction round's duration, and only as much as it is worth to the neighbor. If memory shedding is allowed, even a low reclaim factor will not increase the soldier's costs.

Such soldiering activity disrupts the system's stability. It invalidates the neighbors' assumption of a steady state, and slows down the system's convergence to an efficient allocation. As a result of the instability, memory is, in effect, wasted, because the neighbors benefit less than they could have from the memory that they rent. The hostile activity also raises the borderline bids, and thus causes simple advisors, such as the one we developed, to respond by bidding higher (often for a smaller amount of memory). Thus the effect of the temporary artificial memory pressure is imprinted on the learning algorithms for several more auction rounds, until it decays. Overall, a soldier's activity on a host degrades the quality of service on this host for all the other neighbors. To cause more damage, the soldier might trigger its hostile activity when its neighbors' memory valuation is high.

7.3.3 Prevention of Disruptive Activity

The host would like to identify such disruptive behavior, contain and prevent it. However, positive identification is hard, because such disruptive bidding may be the result of benign fast load changes, as demonstrated in the experiment in Figure 6.8. Soldiers might be better identified by examining the correlation of their bidding strategy with that of their neighbors'. However, correlation between guests' bids does not necessarily indicate malicious intent. It can also be a benign response to the introduction of instability, e.g., the addition of a new guest on the same host or a notable bid change made by a third guest (that increases the resource pressure). Such benign guest bidding strategies need to be filtered out to identify malicious guests.

Under uncertainty, without proof of the maliciousness of a guest, the provider will be reluctant to prevent such bidding. A safer measure on the part of the provider would be to contain the suspicious guest: prevent it from harming other clients, while avoiding harm to the suspicious client itself. Such containment can be achieved by live migration of the suspicious guest to a host that only holds guests belonging to the same client.

Causing the provider to co-locate one client's guests on the same physical machine, without any guest belonging to any other client, is the equivalent of Amazon's dedicated instances, for which Amazon currently charges \$10 per hour per region [8]. Dedicated instances have many benefits. They have fast inter-guest communication. They can be used with shared memory mechanisms (e.g., shared memory MPI), which are faster than MPI over Ethernet. They might also be used more efficiently for scientific computing. In addition, dedicated instances are protected against various side channel attacks [115].

Chapter 8

Conclusion

In this work we set out to pursue the goal of efficient sharing of computing resources. Starting with a combination of grids and a cloud, we tried to improve the overall efficiency by finding the most efficient client strategies. We were able to control the inefficiency by assigning a cost to it. However, changing only one side of the client-provider equation was not enough—the most efficient strategies still suffered the inherent waste of replication.

Our journey in pursuit of this goal lead us to examine cloud computing models, in which payments are explicit and cost saving is an intuitive client goal. We discovered that Amazon's spot instances, the cloud model which we first perceived as the most efficient sharing method in a cloud environment, were actually not market driven, and hence not very efficient: artificially raised prices maximize neither the provider's nor the clients' revenue. However, spot instances themselves are still a good candidate for efficient sharing of computing resources.

To reach the holy grail of efficient resource sharing, we charted a road map that follows current cloud trends to their culmination in the RaaS cloud. We outlined how memory would be allocated and shared in the RaaS cloud, and set out to implement its prototype. Our early results were promising: a $\times 6.2$–$\times 15.8$ improvement in aggregate client satisfaction when compared with state-of-the-art approaches for cloud memory allocation. We are convinced that the cloud industry is indeed marching towards the Resource-as-a-Service cloud model, and that this change will make the sharing of computing resources more efficient.

Appendix: Software

Spot Price Simulation

There are many functions that the Matlab scripts can perform, most of which were used for the analysis of the spot instance traces. However, what might be useful to other researchers is the simulation of spot instance prices given workload traces (such as those available from the Parallel Workloads Archive or the Grid Workload Archive).

The script `paper_sims.m` loads a trace in the SWF format, using `get_trace.m`. The body of the simulation is done in the script `cloud_2nd_price_loads.m`. The simulation is event driven, with two kinds of events: the arrival of a new instance bid and the change of the reserve price (if the reserve price is set to be random). On each such event, the running and waiting instance bids are sorted, and the number of sold instances that maximizes the host revenue is computed. Then instances are killed and/or admitted accordingly. The input key `user_values` controls the distribution from which the user bids are taken.

This is an example of the script's printout:

```
next random step chosen at time 9825050.000000 as 2309.000000

spot_price =

0.9376

done with 10892/20001 free hosts 32/70 waiting bids 8777
done with 10892/20001 free hosts 32/70 waiting bids 8777
done with 10892/20001 free hosts 32/70 waiting bids 8778
done with 10892/20001 free hosts 32/70 waiting bids 8779
done with 10892/20001 free hosts 32/70 waiting bids 8780
done with 10892/20001 free hosts 32/70 waiting bids 8781
done with 10892/20001 free hosts 31/70 waiting bids 8781
```

As demonstrated in the printout, many of the bids might never get a chance to run. Realistic users would quit the queue with such bids, but this behavior is not implemented in the simulation. We end the simulation when the last instance request arrives. We do so because we do not wish to describe the gradual decline of occupancy that results from the ending of the trace.

The spot instance analysis was performed on the basis of data from now inactive Web sites whose data is nonetheless still available from the following locations:

SpotWatch `https://s3-eu-west-1.amazonaws.com/ruben.ruben/SpotWatch.tar`

CloudExchange `http://files.evercu.be/cloudexchange.tgz`

The simulator code is available as free software from `http://www.cs.technion.ac.il/~ladypine/spotprice.tar.gz`.

Dynamic memcached

Memcached provides key-value hashing for values of a wide range of sizes. It stores the items in slabs according to their size. Each slab group can hold items whose size is within a fixed range. This structure works well for a fixed distribution of item sizes, but might pose a problem if the item size distribution changes: memcached might be unable to store items of one size due to lack of space, while storing obsolete items of other sizes.

Dormando, one of memcached's lead developers, created a branch of memcached 1.4 in which it is possible to move a slab between groups, so that it can hold items of a different size. This mechanism cleans all the items from a slab of one group and adds it as a new slab to another group. The slab group from which a slab will be taken is chosen on the basis of recent utilization statistics.

We extended this version of memcached, allowing it to shrink the heap size by cleaning slabs and not reassigning them. The heap shrink is triggered by a change of the desired heap size, which can now be communicated to memcached on the fly, using the command "-m" (the same syntax which is used for setting the heap size when initializing the program.) The aggressiveness of the shrinking operating is controlled by the `automove_level`. In the most aggressive mode, a slab group from which slabs will be taken is chosen even without gathering utilization statistics, to enable a quick response to memory pressure.

Another method which hastens memory shrinkage is to order the release of several slabs together. First we define a slab group's *equal share* of released slabs as the ratio of the number of slabs that must be released and the number of active slab groups. This is the number of slabs that should be released from each active slab group if there is no preference for any slab group, so that the release is done evenly. Then we determine the first candidate for slab release on the basis of available utilization data. Since this data is deleted when read, the first candidate is usually also the best candidate. The best candidate might not be strictly preferable to other candidates, but at least it is not worse than them. From this candidate we release its a number of slab which equals the *equal share*, as defined earlier. If all the items are of the same size, there is only one active slab group. In this case, all the slab release commands are given at once, without gathering more utilization statistics. When there is more than one slab group, the rest of the releases are determined on the basis of the poorer utilization data gathered from this point on.

Heap shrinkage can also be manually activated from the command line, by issuing the command:"slab reassign N -S", where N is the slab index to shrink, and S is the number of slabs to take from this item size.

This is the printout of the `slabs_shrink.t` test, showing an allocation of slabs for two item sizes, followed by a shrinkage, and expansion and then another shrink operation.

```
ok 1
ok 2
ok 3 - verbose is not 0
ok 4 - slab 31 evicted is nonzero
ok 5 - slab 25 evicted is nonzero
gap 4718848 for gap 5 to reach from 6291456 to 2097152 when currently
using 6816000
```

```
ok 6 - slab shrink was ordered
emergency source changed from 0 to 25
emergency source changed from 0 to 31
ok 7 - slabs shrunk is nonzero
ok 8 - slab 25+31 pagecount changed
Slabs class 31 25
Changed from 4 2
to 1 1
limit_maxbytes 2 total malloced 2
ok 9 - stored key
ok 10 - stored key
ok 11 - slab expand was ordered
ok 12 - slab 31 pagecount increased - using the increased memory limit
Slabs class 31 25
Changed from 1 1
to 8 2
gap 7864576 for gap 8 to reach from 20971520 to 3145728 when currently
using 11010304
ok 13 - slab shrink was ordered
emergency source changed from 0 to 25
emergency source changed from 0 to 31
ok 14 - slabs shrunk is nonzero
ok 15 - slab 25+31 pagecount changed
Slabs class 31 25
Changed from 8 2
to 1 1
```

Memcached is free software, released under the BSD license. Our version is available from `https://github.com/ladypine/memcached`.

Bibliography

[1] Hussein A. Abbass, Ruhul Sarker, and Charles Newton. PDE: A Pareto-frontier differential evolution approach for multi-objective optimization problems. In *CEC*, 2001.

[2] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing Amazon EC2 spot instance pricing. In *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011.

[3] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Raas: Resource as a service. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[4] K. Agrawal, A. Benoit, L. Magnan, and Y. Robert. Scheduling algorithms for linear workflow optimization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, 2010.

[5] Nezih Yigitbasi Alexandru Iosup and Dick Epema. On the performance variability of production cloud services. In *Cluster, Cloud and Grid Computing (CCGrid)*, 2011.

[6] Jörn Altmann and Karyen Chu. How to charge for network services—flat-rate or usage-based? *Computer Networks*, 36(5):519 – 531, 2001.

[7] Jörn Altmann, Costas Courcoubetis, George Stamoulis, Manos Dramitinos, Thierry Rayna, Marcel Risch, and Chris Bannink. GridEcon: A market place for computing resources. In *Grid*

Economics and Business Models, volume 5206 of *Lecture Notes in Computer Science*, pages 185–196. Springer Berlin / Heidelberg, 2008.

[8] Amazon EC2 dedicated instances. `http://aws.amazon.com/dedicated-instances`. [Accessed Apr, 2013].

[9] Amazon EC2 spot instances. `http://aws.amazon.com/ec2/spot-instances`. [Accessed Aug, 2011].

[10] Spot instance termination conditions? `http://tinyurl.com/2dzp734`, Mar 2010. Online AWS Developer Forums discussion. [Accessed Apr, 2011].

[11] AMD. ACP — the truth about power consumption starts here. white paper, 2007. `http://www.amd.com/us/Documents/43761C_ACP_WP_EE.pdf`.

[12] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

[13] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *e-Science*, pages 196–203, 2005.

[14] Artur Andrzejak, Derrick Kondo, and David P. Anderson. Exploiting non-dedicated resources for cloud computing. In *NOMS'10*.

[15] Artur Andrzejak, Derrick Kondo, and Sangho Yi. Decision model for cloud computing under SLA constraints. In *IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010.

[16] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Ottawa Linux Symposium (OLS)*, pages 19–28, 2009.

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[18] Lawrence M. Ausubel and Paul Milgrom. *Combinatorial auctions*, chapter The lovely but lonely Vickrey auction, pages 17–40. 2006.

[19] Junjik Bae, Eyal Beigman, Randall Berry, Michael L. Honig, and Rakesh Vohra. An efficient auction for non concave valuations. In *9th International Meeting of the Society for Social Choice and Welfare*, 2008.

[20] Salman A. Baset. Cloud SLAs: Present and future. *ACM SIGOPS Operating Systems Review (OSR)*, 46(2), Jul 2012.

[21] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 423–436, 2010.

[22] A. Benoit, Y. Robert, A. L. Rosenberg, and F. Vivien. Static strategies for worksharing with unrecoverable interruptions. In *IPDPS*, 2009.

[23] Christian Borgs, Jennifer T. Chayes, Nicole Immorlica, Kamal Jain, Omid Etesami, and Mohammad Mahdian. Dynamics of bid optimization in online advertisement auctions. In *International Conference on World Wide Web (WWW)*, pages 531–540, 2007.

[24] Sem Borst, Onno Boxma, Jan Friso Groote, and Sjouke Mauw. Task allocation in a multi-server system. *J. of Scheduling*, 6(5):423–436, 2003.

[25] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[26] Paul Brebner and Anna Liu. Performance and cost assessment of cloud services. In *Service-Oriented Computing*, volume 6568 of *Lecture Notes in Computer Science*, pages 39–50. 2011.

[27] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2002.

[28] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13).

[29] Henri Casanova. On the harmfulness of redundant batch requests. In *HPDC*, pages 255–266, 2006.

[30] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a generic framework for large-scale distributed experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.

[31] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[32] Junliang Chen, Chen Wang, Bing Bing Zhou, Lei Sun, Young Choon Lee, and Albert Y. Zomaya. Tradeoffs between profit and customer satisfaction for service provisioning in the cloud. In *HPDC*, 2011.

[33] Ran Chen and Hao Li. The research of grid resource scheduling mechanism based on pareto optimality. In *Software Engineering (WCSE), 2010 Second World Congress on*, 2010.

[34] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Chandra Krintz. See spot run: using spot instances for mapreduce workflows. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[35] Brent N. Chun and David E. Culler. Market-based proportional resource sharing for clusters. Technical report, Berkeley, CA, USA, 2000.

[36] Walfredo Cirne, Francisco Brasileiro, Daniel Paranhos, Luís Fabrício W. Góes, and William Voorsluys. On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Comput.*, 33(3), 2007.

[37] Edward H. Clarke. Multipart pricing of public goods. *Public Choice*, 11(1):17–33, Sep 1971.

[38] Daniel Cordeiro, Pierre-François Dutot, Grégory Mounié, and Denis Trystra. Tight analysis of relaxed multi-organization scheduling algorithms. In *IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

[39] Amir Danak and Shie Mannor. Resource allocation with supply adjustment in distributed computing systems. In *Int'l Conference on Dsitributed Computing Systems (ICDCS)*, 2010.

[40] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley and Sons, first edition, 2001.

[41] Yang Ding, Mahmut Kandemir, Padma Raghavan, and Mary Jane Irwin. A helper thread based EDP reduction scheme for adapting application execution in cmps. In *IPDPS*, 2008.

[42] Menno Dobber, Robert D. van der Mei, and Ger Koole. Dynamic load balancing and job replication in a global-scale grid environment: A comparison. *IEEE Trans. Parallel Distrib. Syst.*, 20(2), 2009.

[43] Shahar Dobzinski and Noam Nisan. Mechanisms for multi-unit auctions. *Journal of Artificial Intelligence Research*, 37:85–98, 2010.

[44] Danny Dolev, Dror G. Feitelson, Joseph Y. Halpern, Raz Kupferman, and Nathan Linial. No justified complaints: on fair sharing of multiple resources. In *Innovations in Theoretical Computer Science Conference (ITCS)*, pages 68–75. ACM, 2012.

[45] Jack J. Dongarra, Emmanuel Jeannot, Erik Saule, and Zhiao Shi. Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In *Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2007.

[46] Benjamin Edelman, Michael Ostrovsky, and Michael Schwarz. Internet advertising and the generalized second-price auction: Selling billions of dollars worth of keywords. *American Economic Review*, 97(1):242–259, March 2007.

[47] John P. Holdren et al. Realizing the full potential of government-held spectrum to spur economic growth. Technical report, The President's Council of Advisors on Science and Technology, 2012.

[48] Dror Feitelson. Parallel workloads archive. Website. `http://www.cs.huji.ac.il/labs/parallel/workload/index.html`.

[49] Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Are sleep states effective in data centers? In *International Green Computing Conference (IGCC)*, 2012.

[50] Gaurav D. Ghare and Scott T. Leutenegger. Improving speedup and response times by replicating parallel programs on a snow. In *JSSPP '04*.

[51] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2011.

[52] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management (CNSM)*, pages 9–16. IEEE, 2010.

[53] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Dan Tsafrir, and Assaf Schuster. ELI: Bare-metal

performance for I/O virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2012.

[54] Abel Gordon, Michael Hines, Dilma Da Silva, Muli Ben-Yehuda, Marcio Silva, and Gabriel Lizarraga. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *ASPLOS RESoLVE '11: Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE) workshop*, 2011.

[55] Theodore Groves. Incentives in teams. *Econometrica*, 41(4):617–631, Jul 1973.

[56] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2008.

[57] Avital Gutman and Noam Nisan. Fair allocation without trade. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 2, pages 719–728, 2012.

[58] John Hegeman. Facebook's ad auction. Talk at Ad Auctions Workshop, May 2010.

[59] Joseph L. Hellerstein, Walfredo Cirne, and John Wilkes. Google cluster data. Website, 2011. `http://code.google.com/p/googleclusterdata/`.

[60] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *IFIP/IEEE Symposium on Integrated Management (IM)*, 2009.

[61] Michael Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In

CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science, 2011.

[62] Liting Hu, Kyung Dong Ryu, Dilma Da Silva, and Karsten Schwan. v-bundle: Flexible group resource offerings in clouds. In *Int'l Conference on Dsitributed Computing Systems (ICDCS)*, 2012.

[63] Alexandru Iosup, Catalin Dumitrescu, Dick H. J. Epema, Hui Li, and Lex Wolters. How are real grids used? The analysis of four grid traces and its implications. In *GRID*, 2006.

[64] Alexandru Iosup, Mathieu Jan, Omer Ozan Sonmez, and Dick H. J. Epema. On the dynamic resource availability in grids. In *GRID*, 2007.

[65] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, and Dick H. J. Epema. The Grid Workloads Archive. *Future Generation Comp. Syst.*, 24(7):672–686, 2008. `http://gwa.ewi.tudelft.nl/pmwiki/`.

[66] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and D. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. on Parallel and Distrib. Sys.*, 22, 2011.

[67] Alexandru Iosup, Ozan Sonmez, and Dick Epema. Dgsim: Comparing grid resource management architectures through trace-based simulation. In *Euro-Par '08*.

[68] Keith R. Jackson, Lavanya Ramakrishnan, Karl J. Runge, and Rollin C. Thomas. Seeking supernovae in the clouds: a performance study. In *HPDC*, pages 421–429, 2010.

[69] Bahman Javadi and Rajkumar Buyya. Comprehensive statistical analysis and modeling of spot instances in public cloud environments. Technical Report CLOUDS-TR-2011-1, Cloud Computing and Distributed Systems Laboratory, The University of Melbourne, 2011.

[70] Emmanuel Jeannot, Erik Saule, and Denis Trystram. Bi-objective approximation scheme for makespan and reliability optimization on uniform parallel machines. In *Euro-Par 2008*, volume 5168 of *Lecture Notes in Computer Science*, chapter 94, pages 877–886. 2008.

[71] Ramesh Johari and John N. Tsitsiklis. Efficiency loss in a network resource allocation game. *Mathematics of Operations Research*, 29(3):407–435, 2004.

[72] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 14–24, 2006.

[73] Ian A. Kash, Rohan Murty, and David C. Parkes. Enabling spectrum sharing in secondary market auctions. In *Workshop on the Economics of Networks, Systems, and Computation*, 2011.

[74] Rama Katkar and David H. Reiley. Public versus secret reserve prices in ebay auctions: Results from a pokémon field experiment. *Advances in Economic Analysis and Policy*, 2006.

[75] Frank Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997.

[76] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)*, pages 225–230, 2007. `http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf`.[Accessed Apr, 2011].

[77] Derrick Kondo, Andrew A. Chien, and Henri Casanova. Resource management for rapid application turnaround on enterprise desktop grids. In *SC'04*, 2004.

[78] Derrick Kondo, Gilles Fedak, Franck Cappello, Andrew A. Chien, and Henri Casanova. Characterizing resource availability in enterprise desktop grids. *Future Generation Com. Sys.*, 23(7).

[79] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick H. J. Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *CCGRID*, 2010.

[80] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. pages 404–413, 1999.

[81] Aurel Lazar and Nemo Semret. Design and analysis of the progressive second price auction for network bandwidth sharing. *Telecommunication Systems - Special issue on Network Economics*, page http://comet.columbi, 1999.

[82] Young Choon Lee, Riky Subrata, and Albert Y. Zomaya. On the performance of a dual-objective optimization model for workflow applications on grid platforms. *IEEE Trans. Parallel Distrib. Syst.*, 20(9):1273–1284, 2009.

[83] Moshe Levy and Sorin Solomon. New evidence for the power-law distribution of wealth. *Physica A*, 242:90–94, 1997.

[84] Huagang Li and Guofu Tan. Hidden reserve prices with risk-averse bidders. Technical report, University of British Columbia, 2000.

[85] Tong Li and Isabelle Perrigne. Timber sale auctions with random reserve prices. *Review of Economics and Statistics*, 85(1):189–200, 2003.

[86] Adam G. Litke. Memory overcommitment manager. website, 2011. `https://github.com/aglitke/mom`.

[87] Huan Liu. A measurement study of server utilization in public clouds. In *Int'l Conference on Cloud and Green Computing (CGC)*, 2011.

[88] Tim Lossen. Cloud exchange. `http://cloudexchange.org/`. [Accessed Apr, 2011].

[89] Benjamin Lubin, David C. Parkes, Jeff Kephart, and Rajarshi Das. Expressive power-based resource allocation for data cen-

ters. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

[90] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63:1105–1122, November 2003.

[91] Brendan Lucier, Renato Paes Leme, and Eva Tardos. On revenue in the generalized second price auction. In *International Conference on World Wide Web (WWW)*, 2012.

[92] Dan Magenheimer. Memory overcommit... without the commitment. In *Xen Summit*. USENIX association, June 2008.

[93] Patrick Maillé and Bruno Tuffin. Multi-bid auctions for bandwidth allocation in communication networks. In *IEEE INFOCOM*, 2004.

[94] Patrick Maillé and Bruno Tuffin. Multi-bid versus progressive second price auctions in a stochastic environment. *Quality of Service in the Emerging Networking Panorama*, pages 318–327, 2004.

[95] Michael Mattess, Christian Vecchiola, and Rajkumar Buyya. Managing peak loads by leasing cloud infrastructure services from a spot market. In *IEEE Int'l Conference on High Performance Computing and Communications (HPCC)*, 2010.

[96] Christopher A. Mattson and Achille Messac. Pareto frontier based concept selection under uncertainty, with visualization. *Optimization and Engineering*, 6(1), 2005.

[97] Michele Mazzucco and Marlon Dumas. Achieving performance and availability guarantees with spot instances. In *IEEE Int'l Conference on High Performance Computing and Communications (HPCC)*, 2011.

[98] Frank McSherry and Kunal Talwar. Mechanism design via differential privacy. In *The 48th Annual Symposium on Foundations of Computer Science (FOCS)*, 2007.

[99] Emmanuel Medernach. Workload analysis of a cluster in a grid environment. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 2005.

[100] Achille Messac, Amir Ismail-Yahaya, and Christopher A. Mattson. The normalized normal constraint method for generating the Pareto frontier. *Structural and Multidisciplinary Optimization*, 25(2), 2003.

[101] Grzegorz Miłoś, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference (ATC)*, 2009.

[102] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2010.

[103] Andrew Odlyzko. Paris metro pricing for the internet. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 140–147, New York, NY, USA, 1999. ACM.

[104] Ana-Maria Oprescu and Thilo Kielmann. Bag-of-tasks scheduling under budget constraints. In *CloudCom*, 2010.

[105] Fernando Martinez Ortuno and Uli Harder. Stochastic calculus model for the spot price of computing power. In *Annual UK Performance Engineering Workshop (UKPEW)*, 2010.

[106] Zhonghong Ou, Hao Zhuang, Jukka K Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of amazon EC2. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[107] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2009.

[108] David C. Parkes, Ariel D. Procaccia, and Nisarg Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *The ACM Conference on Electronic Commerce (EC)*, 2012.

[109] Lucian Popa, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: Sharing the network in cloud computing. In *ACM HotNets*, 2011.

[110] Muntasir Raihan Rahman, Yi Lu, and Indranil Gupta. Risk aware resource allocation for clouds. Technical report, University of Illinois at Urbana-Champaign, 2011.

[111] Christina Ramberg. *Internet Marketplaces: The Law of Auctions and Exchanges Online.* Oxford, 2002.

[112] Sarvapali D. Ramchurn, Perukrishnen Vytelingum, Alex Rogers, and Nicholas R. Jennings. Putting the 'smarts' into the smart grid: a grand challenge for artificial intelligence. *Commun. ACM*, 55(4):86–97, April 2012.

[113] Juan Manuel Ramírez-Alcaraz, Andrei Tchernykh, Ramin Yahyapour, Uwe Schwiegelshohn, Ariel Quezada-Pina, José Luis González-García, and Adán Hirales-Carbajal. Job allocation strategies with user run time estimates for online scheduling in hierarchical grids. *J. Grid Comput.*, 9:95–116, March 2011.

[114] Stefan Ried, Holger Kisker, Pascal Matzke, Andrew Bartels, and Miroslaw Lisserman. Sizing the cloud—understanding and quantifying the future of cloud computing. Technical report, Forrester, 2011.

[115] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. 2009.

[116] Kyung Dong Ryu, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, Stefan Berger, Dilma M Da Silva, Jim Doran, Frank Franco, Alexei Karve, Herb Lee, James A Lindeman, Ajay Mohindra,

Bob Oesterlin, Giovanni Pacifici, Dimitrios Pendarakis, Darrell Reimer, and Mariusz Sabath. RC2–a living lab for cloud computing. In *Proceedings of the 24th international conference on Large installation system administration (LISA)*, pages 1–14. USENIX Association, 2010.

[117] Dmitriy Samovskiy. Amazon ec2 spot instances - a flop? `http://tinyurl.com/somic11`, Aug 2011. [Accessed Sep, 2011].

[118] Sujay Sanghavi and Bruce Hajek. Optimal allocation of a divisible good to strategic buyers. In *IEEE Conference on Decision and Control (CDC)*, 2004.

[119] Erik Saule and Denis Trystram. Analyzing scheduling with transient failures. *Inf. Process. Lett.*, 109:539–542, May 2009.

[120] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted linux environments. In *OLS '06: 2006 Ottawa Linux Symposium*, 2006.

[121] Vyas Sekar and Petros Maniatis. Verifiable resource accounting for cloud computing services. In *ACM Cloud Computing Security Workshop (CCSW)*, 2011.

[122] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *ACM Symposium on Cloud Computing (SOCC)*, page 5. ACM, 2011.

[123] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin Auyoung, Alex C. Snoeren, Amin Vahdat, and Brent Chun. Why markets could (but don't currently) solve resource allocation problems in systems. In *USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*, page 7, 2005.

[124] Mark Silberstein. Building online domain-specific computing service over non-dedicated grid and cloud resources: Superlink-online experience. In *CCGRID '11*, 2011.

[125] Mark Silberstein, Artyom Sharov, Dan Geiger, and Assaf Schuster. Gridbot: Execution of bags of tasks in multiple grids. In *SC'09*.

[126] Mark Silberstein, Anna Tzemach, Nickolay Dovgolevsky, Maáyan Fishelson, Assaf Schuster, and Dan Geiger. Online system for faster multipoint linkage analysis via parallel execution on thousands of personal computers. *The American Journal of Human Genetics*, 78(6):922–935, 2006.

[127] Daniel Paranhos Da Silva, Walfredo Cirne, Francisco Vilar Brasileiro, and Campina Grande. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Euro-Par*, 2003.

[128] Gurmeet Singh, Mei-Hui Su, Karan Vahi, Ewa Deelman, Bruce Berriman, John Good, Daniel S. Katz, and Gaurang Mehta. Workflow task clustering for best effort systems with pegasus. In *MG '08*.

[129] Wataru Souma. Universal structure of the personal income distribution. *Fractals*, 9(04):463–470, 2001.

[130] Wataru Souma. Physics of personal income. `http://arxiv.org/pdf/cond-mat/0202388`, 2002.

[131] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[132] Adel Nadjaran Toosi, Rodrigo N. Calheiros, Ruppa K. Thulasiram, and Rajkumar Buyya. Resource provisioning policies to increase iaas provider s profit in a federated cloud environment. In *IEEE Int'l Conference on High Performance Computing and Communications (HPCC)*, 2011.

[133] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Trans. Internet Technol.*, 9(1), 2009.

[134] Kurt Vanmechelen, Wim Depoorter, and Jan Broeckhove. Combining futures and spot markets: A hybrid market approach to economic grid resource management. *Journal of Grid Computing*, 9:81–94, 2011.

[135] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of hpc applications. In *ACM Int'l Conference on Supercomputing (ICS)*, 2008.

[136] Kurt Vermeersch. Spot watch. `http://spotwatch.eu/input/`. Accessed Apr, 2011.

[137] Kurt Vermeersch. A broker for cost-efficient qos aware resource allocation in EC2. Master's thesis, Universiteit Antwerpen, 2011.

[138] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16(1), 1961.

[139] Daniel R. Vincent. Bidding off the wall: Why reserve prices may be kept secret. *Journal of Economic Theory*, 65(2):575–584, 1995.

[140] Berthold Vöcking. A universally-truthful approximation scheme for multi-unit auctions. In *Annual ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[141] William Voorsluys, Saurabh Kumar Garg, and Rajkumar Buyya. Provisioning spot market cloud resources to create cost-effective virtual clusters. In *ICA3PP*, 2011.

[142] Anton Vorontsov. Add mempressure cgroup. `http://lwn.net/Articles/528687/`, Accessed April 2013, Dec 2012.

[143] N. Vydyanathan, U. Catalyurek, T. Kurc, J. Saltz, and P. Sadayappan. Toward optimizing latency under throughput constraints for application workflows on clusters. In *Euro-Par*, 2007.

[144] Nagavijayalakshmi Vydyanathan, Umit Catalyurek, Tahsin Kurc, Ponnuswamy Sadayappan, and Joel Saltz. A duplication

based algorithm for optimizing latency under throughput constraints for streaming workflows. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, 2008.

[145] Carl A. Waldspurger. Memory resource management in Vmware ESX server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[146] Carl A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, Feb 1992.

[147] Hongyi Wang, Qingfeng Jing, Rishan Chen, Bingsheng He, Zhengping Qian, and Lidong Zhou. Distributed systems meet economics: pricing in the cloud. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[148] Sewook Wee. Debunking real-time pricing in cloud computing. In *Cluster, Cloud and Grid Computing (CCGrid)*, 2011.

[149] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. Brief announcement: modelling mapreduce for optimal execution in the cloud. In *ACM SIGACT-SIGOPS symposium on Principles Of Distributed Computing (PODC)*, 2010.

[150] Joshua Wingstrom and Henri Casanova. Probabilistic allocation of tasks on desktop grids. In *IPDPS*, 2008.

[151] UW-Madison CS Dept. Condor pool. Website. `http://www.cs.wisc.edu/condor/uwcs/`.

[152] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *ACM/USENIX Int'l Conference on Virtual Execution Environments (VEE)*, pages 31–40, 2009.

[153] Sangho Yi, Artur Andrzejak, and Derrick Kondo. Monetary cost-aware checkpointing and migration on Amazon cloud spot instances. *IEEE Transactions on Services Computing*, 2011.

[154] Sangho Yi, Derrick Kondo, and Artur Andrzejak. Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2010.

[155] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI'08*, 2008.

[156] Sharrukh Zaman and Daniel Grosu. Combinatorial auction-based dynamic vm provisioning and allocation in clouds. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011.

[157] Qi Zhang, Eren Gurses, Raouf Boutaba, and Jin Xiao. Dynamic resource allocation for spot markets in clouds. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2011.

[158] Han Zhao, Xinxin Liu, and Xiaolin Li. Hypergraph-based task-bundle scheduling towards efficiency and fairness in heterogeneous distributed systems. In *IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[159] Han Zhao, Miao Pan, Xinxin Liu, Xiaolin Li, and Yuguang Fang. Optimal resource rental planning for elastic applications in cloud market. In *IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS)*, 2012.

[160] Weiming Zhao and Zhenlin Wang. Dynamic memory balancing for virtual machines. In *ACM/USENIX Int'l Conference on Virtual Execution Environments (VEE)*, pages 21–30, 2009.

[161] Xia Zhou, Sorabh G, Subhash Suri, and Haitao Zheng. eBay in the sky: Strategy-proof wireless spectrum auctions. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2008.

חלוקת משאבי מחשוב יעילה לא שיתופית

אורנה אגמון בן-יהודה

חלוקת משאבי מחשוב יעילה לא שיתופית

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

אורנה אגמון בן-יהודה

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
תמוז ה'תשע"ג   חיפה   יוני 2013

המחקר נעשה בהנחיית פרופ' אסף שוסטר בפקולטה למדעי המחשב.

אני מודה לטכניון ולמכון האסו פלאטנר על התמיכה הכספית הנדיבה בהשתל-
מותי.

תקציר

המאפיין החשוב ביותר של מחשוב הענן הוא כסף. בענן, לקוחות לא-שיתופיים משלמים לספקים תמורת משאבי המחשוב המשותפים בהם הם משתמשים, בזמן שהם משתמשים בהם. הכנסת הפיצוי הכספי למערכת פותחת דלת בפני מגוון אפשרויות של חלוקה יעילה של משאבי מחשוב משותפים. אנו חוקרים את היסו-דות הכלכליים של מערכות מחשוב ענן ומציעים מנגנונים חדשים בהם יכולים ספקים ולקוחות מציאותיים לא-שיתופיים להשתמש, כדי לחלוק משאבי מחשוב ביעילות.

אנו מדגימים כיצד לקוחות יכולים למזער בו זמנית את זמן הריצה ואת עלויות החישוב של העבודות שברצונם לבצע, על ידי הרצתם על גבי השילוב הנכון של משאבי גריד וענן. לשם כך אנו מפתחים שפה לשליחת תיקי מטלות. אנו בונים חזית פרטו של אסטרטגיות החישוב היעילות ביותר במסגרת שפה זו. אנו מראים כי שפה זו עשירה מספיק כדי לכלול פתרונות שהם יעילים יותר משיטות אינטואיטיביות המשמשות משתמשים טיפוסיים. כמו-כן אנו מראים כי אופן השימוש היעיל ביותר בשילוב של גריד וענן הוא להחזיק את התור למחשוב הענן מלא. זאת משום שתור מלא מאפשר למטלות להסתיים על הגריד בזמן שהן ממתינות לענן ולהתבטל מן התור. הדבר נכון עבור כל שקלול תמורות שבין עלות לאורך הזמן הכולל של חישוב תיק המטלות כולו.

אנו מנתחים את הדרך בה ספק ענן מוביל, אמזון, מתמחר את עודף ההיצע שלו (הנמכר תחת השם "ספוט"). אנו מציגים דרך חדשה לבחינת היסטוריות המחירים של אמזון, אשר זמינות לכל אך נבחנו עד כה כגרף תלוי זמן. לשיטתנו יש למדוד את הזמינות של כל מחיר, כלומר את שבר הזמן לאורך ההיסטוריה בו היה ניתן להשיג כוח מחשוב במחיר זה. אנו מוצאים כי קיימת שיטתיות רבה במחירים של אמזון, כאשר בוחנים את זמינות הספוטים: לגרפים יש צורה אופיינית מלאכותית, הם זהים כאשר הם מנורמלים במחיר המחשוב בשיטת מחיר "לפי דרישה", וקיימת שונות בין שיטתיות המחירים במערכות הפעלה שונות.

טרם הצגת עבודתנו רווחה האמונה כי מחירי הספוט מבוססים על הצע וביקוש. אנו מראים כי בניגוד לאמונה זו, בשנים הראשונות לקיום שיטת מכירה זו, המחירים יוצרו בעצם באופן רנדומלי על ידי אמזון. כמו כן עולה ממחקרנו כי המחירים אשר פורסמו על ידי אמזון היו מלאכותיים כאשר הדרישה הייתה נמוכה, ומצב זה התקיים בתשעים ושמונה אחוז מן הזמן בהיסטוריות המחירים שניתחנו. אנו מאששים את טענתנו באמצעות בנייה של אלגוריתם מחירים מלאכותיים, בו המחיר הנמוך ביותר שמוכן הספק לקבל משתנה לפי תהליך אוטו-רגרסיבי מן הסדר הראשון אשר מעוגן לרצועת מחירים קבועה מראש. אנו משתמשים בתהליך זה לקביעת מחירים וכן בניתוחנו את מועדי שינוי המחירים של אמזון כדי ליצור סימולציה המונעת על ידי עקבות של מטלות המוגשות לענן. באמצעות הזנה של עקבות של גריד ושל שלושה עננים לתכנת הסימולציה אנו משחזרים היסטוריות מחירים בעלות אופיינים איכותיים דומים לאופיינים של היסטוריות המחירים של אמזון. היתרון הבולט ביותר של שיטת תמחור רנדומלית אוטו-רגרסיבית הוא אשליית הלקוחות כי קיים שימוש ער במוצר. בד בבד עם פרסום המחקר שלנו שינתה אמזון את שיטת המכירה, והיא אינה מבוססת עוד על גורמים רנדומליים.

אנו מנתחים מגמות בשוק מחשוב הענן, ומוצאים כי משאבי מחשוב מושכרים לפרקי זמן הולכים ומתקצרים, בגרעיניות שהולכת ונהיית עדינה. כמו כן, אנו צופים כי מחירי משאבי המחשוב יהיו מושפעים יותר משיקולי שוק, והתמורה שהלקוחות מקבלים עבור התשלום תהיה נתונה במונחים של חוזה שרות יחסי משוכב ולא במונחים אבסולוטיים. אנו מבססים את תחזיתנו על המודלים העסקיים של חברות מחשוב ענן והשינויים בהם, כמו גם על מחקרים והמלצות של חוקרים אחרים ועל שיקולים כלכליים. יתרה מזאת, מאז פורסם מחקר זה לראשונה הואץ קצב השינוי, ותחזיות שונות שפרסמנו במאמר מוצגות כאן כבר כעובדות: למשל, במהלך השנה האחרונה פרצה מלחמת מחירים בין ענקי מחשוב הענן (מלחמה שחזינו, ואשר אותה אנו יכולים לתעד כיום), וספקים חדשים הכריזו על השכרת משאבים וחיובם לפי פרקי זמן של דקות ספורות. כתוצאה ממגמות אלו, אנו צופים שמודל מחשוב הענן הבסיסי ביותר ("תשתית כשירות") יוחלף בהדרגה עד שיהפוך למודל של "משאב כשירות". במודל זה הלקוחות משלמים את המחיר המתאים עבור המשאבים שהם צורכים אך ורק כאשר הם אכן מעוניינים לצרוך אותם. במקום לרכוש וודאות, הלקוחות משקללים את התמורות שהם יכולים לקבל תמורת כספם, ובוחרים באפשרות המתאימה להם. כך הלקוח אינו מחויב לרכוש חוזה הבטחת איכות מחייב מדי, ואילו ספק מחשוב הענן יכול להטיל את עול הדאגה לאלסטיות על לקוחותיו המשלמים פחות. מודל כזה מחייב קיומו מנגנון כלכלי בתוך כל מכונה פיזית. כדי ליהנות מגמישות המשאבים על הלקוח להפעיל סוכן חכם אשר יטפל עבורו ברכישות ואפילו בסחר משני במשאבים.

אנו מציגים אב טיפוס של מחשוב ענן במודל של "משאב כשירות", אשר משכיר ביעילות, בגרעיניות עדינה ובמהירות זיכרון ללקוחות לא שיתופיים. זיכרון הוא המשאב הקשה ביותר לחלוקה, מכיוון שהתועלת מזיכרון מופקת רק לאחר פרק זמן מסוים, לאו דווקא קבוע, בו אחסן הזיכרון את המידע. קושי נוסף בהקצאת זיכרון הוא שתועלת הלקוחות מן הזיכרון אינה בהכרח פונקציה מונוטונית עולה, ולפיכך שיטות חלוקת משאבים רציפים הקיימות בספרות אינן מתאימות למשאב רציף זה. אב הטיפוס מבוסס על מודל דמוי מכרז מחיר שני, אשר מעודד את הלקוחות להצהיר על התועלת האמתית שלהם מכמות מסוימת של משאב. באופן זה אנו מתגברים על בעיית המדידה של ביצועי הלקוח כתלות במשאב המסופק לו, מדידה אשר הלקוח עשוי לזייף את תוצאותיה והספק לא יודע כיצד לבצע. את ביצועי המערכת אנו מודדים במונחים של רווחה חברתית: סכום ההערכות של כל הלקוחות על המשאבים שהם מקבלים. אב הטיפוס משיג תוצאות של רווחה חברתית אשר משופרות בסדר גודל לעומת שיטות עדכניות אחרות.

כחלק מהערכתו של אב הטיפוס פיתחנו גרסה של תוכנה חופשית אשר מאחסנת מטמון של מפתח וערך. הגרסה שפיתחנו מסוגלת לעשות שימוש דינמי בזיכרון הערימה, להגדילה ולהקטינה בהתאם לצורכי המערכת. על ידי כך הצלחנו להשיג גמישות ביצועים כתלות בכמות הזיכרון הפיזי המצוי, כלומר התכנה מהווה דוגמה לתכנות המיועדות לענן מטיפוס "משאב כשירות".

אנו מסיימים בהצגת תכנית למימוש של ענן "משאב כשירות" עבור מערכת מרובת משאבים, בה הלקוחות עשויים לראות משאבים שונים כחליפיים (למשל, סוגים שונים של זיכרון במהירויות שונות) או כמשלימים (למשל, כאשר הלקוח יכול לצרוך בדיוק חמש מאות מגה-בית לכל מעבד שהוא מפעיל) . כמו כן אנו בוחנים התקפות אפשריות מטיפוס ערוץ משני על ענן במודל "משאב כשירות". התקפות אלה יא-פשרו איסוף מידע על שכנים החולקים מכונה פיזית כתוצאה מהשפעת השכנים על תוצאת ההקצאה של השחקן העוין: מידע הכולל את מידת התועלת של הלקוחות מן הזיכרון כמו גם שינויים בהעדפות הלקוח, אשר עשויות להעיד על זמנים בהם הלקוח פגיע יותר. אנו מתארים שימוש עוין במידע שהושג באופן זה וכן מנתחים דרכי התמודדות עם איומים אלה.