Universiteit Antwerpen

Cost-aware Resource Management in Clusters and Clouds

Proefschrift voorgelegd tot het behalen van de graad van Doctor in de Wetenschappen – Informatica, bij de faculteit Wetenschappen, aan de Universiteit Antwerpen, te verdedigen door Ruben Van den Bossche.

PROMOTOREN:
prof. Dr. Jan Broeckhove
Dr. Kurt Vanmechelen

Ruben Van den Bossche

RESEARCH GROUP COMPUTATIONAL MODELING AND PROGRAMMING

uantwerpen.be

Dankwoord

Het is zover. Dit werk is het eindpunt van een lange tocht. Eén die niet gaat om het afleggen van een bepaalde afstand of het behalen van een doel, maar wel om het verkennen van onontgonnen gebied. Het doorploeteren van de jungle op plaatsen waar nog niemand ooit voet zette. Je omarmt je onwetendheid, en waagt voorzichtige stapjes in het onbekende op zoek naar het antwoord op je vragen, naar wetenschap. Onderweg val je, vloek je, en keer je terug op je stappen, maar niet zonder een teken achter te laten voor de volgende die zich aan deze jungle waagt. Vaak heb je het bij het verkeerde eind, maar de euforie die je voelt wanneer onwetendheid plaats maakt voor inzicht of kennis is onbeschrijfelijk. Ik ben trots dat ik bereikt heb wat ik heb bereikt, en dat ik daaraan met dit boek ook fysiek gestalte kan geven.

Een doctoraat maak je zelf, stap voor stap, maar een doctoraat maak je –gelukkig– niet alleen. Daarom zijn er een aantal mensen die ik wil bedanken voor wat ze voor mij de voorbije jaren hebben betekend.

Op een tocht als deze is ervaring onontbeerlijk. Gelukkig had ik twee promotoren, prof. Dr. Jan Broeckhove en Dr. Kurt Vanmechelen, als reisgidsen aan mijn zijde. Jan, Kurt, bedankt voor de kans die ik kreeg om te doctoreren. Bedankt om de richting aan te wijzen en de ervaring en kennis te delen. Om een duwtje in de rug te geven als het wat moeizamer ging, en om steeds klaar te staan voor mijn vragen of bekommernissen.

Onderweg zijn er soms twijfels, over de route, de snelheid, de middelen en de methoden. Soms zijn er tegenslagen, die je moet verwerken. Moed die je moet vinden om de draad weer op te pikken of een nieuwe weg in te slaan. Als dat het geval was, staat je familie altijd klaar om je onvoorwaardelijk te steunen in wat je doet. Sanne, moeke, papa, Jolijn, bedankt om in mij te geloven, om mij de kans te geven te studeren, om bemoedigende woorden te spreken als het moeilijk ging en om trots op mij te zijn als het goed ging.

Een tocht is leuker als je weet dat je er niet alleen voor staat, en als je weet dat ook collega's timmeren aan hun eigen weg. Sam, Sean, Silas, Przemek, Kurt, Delphine, Peter, Wim, Steven, Jan en Frans; bedankt voor de samenwerking in de voorbije zes jaar. Bedankt om mee te zoeken naar problemen, om als klankbord te fungeren, om me met raad en daad bij te staan, maar ook om van *den bureau*, de gang, het werk een aangename omgeving te maken. Bedankt ook om me de kunst van het lesgeven bij te brengen, en om me bij mijn lesopdrachten als assistent te vertrouwen wanneer

dat kon en bij te staan wanneer dat nodig was. Het begeleiden van studenten in hun vorming als toekomstig informaticus was niet alleen vaak een welkome afwisseling, maar ook een taak die ik met plezier deed en waaruit ik steeds veel voldoening haalde.

De voorbije elf jaar aan de unief zijn geweldige jaren geweest. Daar een punt achter zetten kan niet zonder nog één keer achterom te kijken. Aan al mijn studiegenoten, kotgenoten, reisgenoten, praesidiumgenoten, appartementgenoot, vrienden, vriendinnen en collega's en met alle TD's, etentjes, cantussen, fuiven, reizen, vergaderingen, citytrips en cafébezoeken indachtig: het was een geweldige periode, *merci*!

Er is nog veel onontgonnen terrein. Nog veel plaatsen waar niemand het bestaan vanaf weet. Het is tijd voor een nieuwe reis. Ik hoop dat jullie me allemaal opnieuw vergezellen.

Ruben
juni 2014

Contents

Contents 3

List of Figures 7

List of Tables 9

Introduction 13

1 Introduction 13
1.1 A shift towards computing as a utility . . . 13
1.2 A shift in applications, too . . . 14
1.3 Value- and cost-based scheduling . . . 15
1.4 Objective, Research Questions and Contributions . . . 16
1.5 Structure . . . 17

I Value-based Scheduling in Batch Job Cluster Systems 19

2 Fine-grained value-based scheduling 21
2.1 Introduction . . . 22
2.2 Related work . . . 23
2.3 Value-based scheduling . . . 24
2.4 Simulation . . . 27
2.5 Evaluation . . . 28
2.6 Results . . . 32
2.7 Conclusion . . . 37

II Cost-Efficient Scheduling Jobs on Hybrid Clouds 39
Introduction . . . 41
Problem Domain . . . 42
Overview . . . 45
Related Work . . . 46

3 Integer Programming Scheduling on Hybrid Clouds 49
3.1 Integer program . . . 49

3.2 Experiments . . . 50
3.3 Conclusion . . . 57

4 Heuristic for Scheduling on Hybrid Clouds 59
4.1 Experimental setup . . . 59
4.2 Scheduling Components . . . 61
4.3 Conclusion . . . 68

5 Queue-based Scheduling Heuristics 71
5.1 Algorithm Design . . . 71
5.2 Experimental Setup . . . 77
5.3 Results . . . 80
5.4 Conclusion . . . 87

III Contract Portfolio Optimization using Load Prediction 89
Introduction . . . 91
Overview . . . 92
Related Work . . . 93

6 Genetic Programming Load Prediction 95
6.1 Problem Domain . . . 95
6.2 Related Work . . . 96
6.3 Algorithm Design for Contract Portfolio Optimization . . . 97
6.4 Data and Experimental Setup . . . 98
6.5 Results . . . 101
6.6 Conclusion . . . 105

7 A Case Study 107
7.1 Problem Domain . . . 108
7.2 Purchase Management Algorithm . . . 109
7.3 Experimental Setup . . . 111
7.4 Results . . . 116
7.5 Conclusion . . . 118

8 Reserved Contract Procurement Heuristic 119
8.1 Problem Domain . . . 120
8.2 Purchase Management Algorithm . . . 121
8.3 Workload Prediction . . . 125
8.4 Experiment Setup . . . 127
8.5 Results . . . 131
8.6 Conclusion . . . 137

Conclusions and Future Research 141

9 Conclusions and Future Research 141

A Samenvatting 145

B Other Publications **147**

C Scientific Resume **149**

Bibliography **151**

List of Figures

2.1 Valuation function 29
2.2 Experiment 1 33
2.3 SDSC Blue Horizon - Experiment 2 34
2.4 SDSC Blue Horizon - Experiment 3 - Delay tolerance 5 35
2.5 SDSC Blue Horizon - Experiment 5 36
2.6 Component view of the HICCAM model 43

3.1 Binary Integer Program for Hybrid cloud scheduling 51
3.2 Public Cloud - Average cost per workload hour 53
3.3 Public Cloud - Runtimes 54
3.4 Public Cloud with network costs - Average job share 55
3.5 Hybrid Cloud 56
3.6 Hybrid Cloud - Runtimes 57

4.1 Impact of data costs. 63
4.2 Impact of data locality. 65
4.3 Impact of hybrid scheduling policies on *Cost* and *Outgoing data*. 68
4.4 Impact of overestimation of *runtime estimation errors* on *Application deadlines met*. 69
4.5 Impact of overestimation of *runtime estimation errors* on *Average cost per application*. 70

5.1 Hybrid Scheduler - Schematic View 74
5.2 Impact of data costs. 81
5.3 Impact of data locality. 82
5.4 Impact of hybrid scheduling policies on *Deadlines, Cost, Outgoing data* and *Turnaround Time*. 84
5.5 Influence of *Data set size* on hybrid scheduling 85
5.6 Impact of *Runtime Estimation Errors* on hybrid scheduling. 86
5.7 Influence of *N* on hybrid scheduling. 86

6.1 The data of the four websites plotted over time. Solid lines indicate the part of the data that was used as training set. Dashed lines indicate the part of the data that was used as test set. 100

6.2 Pareto front plots showing the complexity accuracy profiles of the LinkedIn.com models. The left plot shows all models of the SR run. The right plot shows only the models using the driving variables for a quality box of complexity 200 and accuracy 0.1. 101
6.3 The GP prediction results for the four websites (shown in solid black). The actual page view data is indicated by the dashed red lines. 103
6.4 Pareto front of models that use the driving variables – for the LinkedIn data. 104
6.5 The extra cost percentage compared to the optimal, for hosting the four websites as calculated by the cloud scheduling simulator. 104

7.1 Example of a Current contract state (CCS) with "Infinite Contracts" view. 110
7.2 The server instance traces 114
7.3 Cost reduction of PMA with Full Knowledge predictor, relative to the cost of RO. 116
7.4 Cost of PMA with different prediction techniques, relative to that of PMA with Full Knowledge predictor. 117

8.1 Illustration of renewal problem 123
8.2 Example of the "Infinite Contracts" policy. 124
8.3 PMA vs. RO (Iteration interval: weekly, Purchase Lookahead Period: 30 days) 131
8.4 tinypic.com Case 132
8.5 Prediction Model evaluation (Iteration interval: weekly, Purchase Lookahead Period: 30 days) 133
8.6 Influence of δ on PP. (Iteration interval: weekly, Purchase Lookahead Period: 30 days) 134
8.7 gardenweb.com Case 135
8.8 Influence of iteration interval and Purchase Lookahead Period 136
8.9 Average runtimes 138

List of Tables

2.1 PWA Workload traces 30
2.2 Experiment Parameters 32

3.1 Instance types 52
3.2 Public Cloud - Prices 52
3.3 Application Parameters 53
3.4 Public Cloud with network costs - Prices 54
3.5 Hybrid Cloud - Prices 56

4.1 Workload model distributions 61
4.2 Available bandwidth from UA to EC2 64

5.1 Amazon EC2 Instance types and prices (d.d. June 1, 2011) 78
5.2 GoGrid Instance types and prices (d.d. June 1, 2011) 78
5.3 Amazon EC2 and GoGrid network prices (d.d. June 1, 2011) 78
5.4 Available bandwidth from UA to Cloud Providers 79
5.5 Private Instance types 79
5.6 Workload model parameters 81
5.7 Amazon EC2 prices for `m1.small` in the US-East Region (d.d. January, 1 2014). 92

6.1 Assumed cloud offerings for instance type `m1.small`. 102
6.2 Overview of the prediction errors for the data sets. The prediction errors are show for the training data, the test data and the first, second and third 30 day period of the test data. 105

7.1 Available contract types for `m1.xlarge`. 113

8.1 The list of server load traces. 129
8.2 Available contract types for `m1.xlarge`. 130

Introduction

CHAPTER 1

Introduction

1.1 A shift towards computing as a utility

To understand the current trends in the IT industry, we first have to take a trip back in time. With the advent of the mainframe in the 60s and 70s, time-sharing became the most prominent computing model, allowing multiple users to interact with a single computer. IBM and other mainframe providers offered access to compute power on a metered basis to banks and other large organizations. John McCarthy was said to be the first person to publicly suggest in 1961 that so called time-sharing of those mainframes could result in an environment in which computing power could be sold through the utility business model. According to [1], a service is a utility if users consider it a necessity; high reliability of the service is critical; ease of use is a significant factor; the full utilization of capacity is limited and services are scalable (leading to economies of scale). Could it be that people would once rely on computing as they relied on utilities such as electricity and telephone?

The rise of the microprocessor and personal computer in the 80s made procurement of compute infrastructure accessible for all organizations, thereby causing the idea of computing as a utility to be put on hold. In the 80s and 90s, organizations with a need for compute power had few options except to invest in the procurement and maintenance of proprietary hardware. When a single PC was not sufficient to fulfill the needs of a single user, multiple computers were connected to each other through a local area network, thereby forming a compute cluster. Such a cluster consists of a set of computers which are either loosely or tightly coupled, and on which centralized management software is installed to orchestrate the deployment of applications on one or multiple nodes at the same time. When such clustered resources are shared by multiple users in an organization, organizations are forced to put systems in place for capacity planning and access regulation in order to cope with saturation. This is hard for an organization: ranking a team's tasks can be tedious and often depends on a lot of different factors and interests. Likewise, capacity planning requires insight on –and often predictions of– future load. Even with such systems in place, the mismatch between the immutability of cluster infra-

structure and the dynamic nature of load will often cause under- or over-utilization of resources and thus lost potential.

One way to mitigate these effects is an increase in scale. In the early 2000s, grid computing promised access to a much larger network of resources by loosely coupling multiple heterogeneous, geographically dispersed clusters owned by different organizations. Organizations, especially academic institutions and research organizations, used each other's compute infrastructure as an extension of their own. This allows companies with distinct use patterns and surges at different moments in time to level off excess load to other organizations. The word *grid* was used in analogy with the electric power grid; the utility model was back in the picture. When sharing resources between multiple administratively separated organizations, an arrangement has to be made on the responsibility, liability and (cost) accountability. Moreover, agreements have to be made on used middleware, security, license and data management. The entry barriers for an organization in such a grid agreement are substantial, and require a significant amount of time, effort and commitment of the different parties involved. For utility computing to really succeed, these agreements should transcend simple bilateral agreements and exceed the scale of a single organization, and should allow for a separation of supply and demand. In addition, a system that takes care of access and application control, capacity planning and load surge management is crucial for the success of such a grid model. Due to the lack of sharing agreements and high entrance barrier, and despite the large amount of government funds invested in it, grid computing never really found its way outside of its own community.

Utility computing finally found broad acceptance a few years later with the advent of *cloud computing*, in which access to external computing power is made available by large cloud providers on a metered basis, with no required upfront commitment. This revolution was driven, among other things, by the fact that the performance overhead of virtualization technology –which allows an operating system to run isolated from and in parallel with other operating systems on a single computer– had now diminished to a negligible level. Infrastructure-as-a-Service (IaaS) providers, such as Amazon EC2, allow consumers to rent virtual machines on a pay-as-you-go basis with a time granularity of a minute up to an hour. This provided a solution to grid challenges such as cost accounting, resource isolation and access management, obstacles that prevented grid computing to become a success. The barrier to get started with cloud computing is minor compared to grid computing: all a user needs to launch a virtual machine is a credit card and internet access. Resources in cloud computing are typically available *on demand*, quickly enough to absorb load spikes, which takes away the burden of long-term provisioning. This elasticity of resources combined with a fixed price regardless of the scale of deployment is unprecedented in the history of IT, and is currently transforming a large part of the IT industry.

1.2 A shift in applications, too

In the past 15 years, organizations have increasingly faced problems –both in research and business areas– that require a large amount of compute power. Fueled by the ever-increasing amount of computing power, companies and organizations have come to rely on computing for many of their business-critical tasks. Thereby,

they face a rising complexity in the management, admission control and planning of their shared computing resources and applications. This complexity increases when users have different *Quality-of-Service (QoS)* constraints regarding the achieved level of performance of their applications.

An organization's application set is typically versatile, and different kinds of applications require different approaches to scheduling and deployment. An important characteristic in which applications differ is their "lifetime". Some applications consist of a limited amount of computation after which their job is done. Such *batch-type applications* are set up to be run to completion with little or no manual intervention. Examples of such batch jobs are parallel Monte Carlo runs for particle physics research, in silico analysis of protein folding processes for drug design, solving optimization problems for financial or risk modeling, image and video rendering or transcoding and log file processing. A typical QoS constraint for a batch application is a deadline for the application to finish. Server-type applications on the other hand typically run for a long time and respond to requests across a computer network. Examples include web servers, database servers and storage servers. Such server applications commonly have QoS constraints such as a maximum response time or minimum uptime.

Managing applications deployed on an organization's shared infrastructure while taking into account its Quality-of-Service constraints is significantly different depending on the application type. Therefore, it is necessary to further categorize batch-type applications. A single execution of a batch application is often referred to as a *job* or *task*. When one application comprises multiple batch jobs grouped together, its type can be further categorized based on their interdependence. When those jobs can be executed to completion in any order and independently of each other, they are called a *bag-of-tasks* (BoT). A typical example of a BoT application is a *parameter sweep*, which consists of a single job that is executed with different parameters in a user-defined range. Other possible types of batch applications are *workflows*, in which jobs have precedence constraints, or *concurrent applications*, when all jobs have to run contemporaneous and –preferably– on a tightly coupled system.

1.3 Value- and cost-based scheduling

There are no ready-made solutions to manage the shared use of the organization's computing infrastructure –whether they belong to a private cluster, grid or cloud provider– in an optimal way. Traditionally, scheduling policies are used to tackle such problems. These policies control whom, when and for how long users or applications are given access to compute resources. In the management of compute infrastructure, a scheduling policy may take into account different –often conflicting– interests. It may aim to optimize for goals such as fairness between its users, application turnaround time, throughput, waiting time or deadlines met, or it may even attempt to find a balance between two or more of those objectives.

One approach to solve these scheduling challenges in the context of an organization's compute cluster with limited resources is the incorporation of the value a user attributes to its computation. In a value-based scheduling algorithm the user-value is expressed by the user upon the request of an amount of compute power. To encourage users to express a value truthfully, the use of a value-based scheduling

algorithm often requires an accounting system that charges users according to the expressed value and obtained service level.

With the emergence of IaaS cloud providers such as Amazon EC2 in 2006, the flexibility for organizations increased drastically. Most cloud providers promise unlimited scalability and immediate availability, a promise that has its limitations in practice but nevertheless offers opportunities that were previously unthinkable.

This shifts the problem for an organization from a scheduling problem pur sang towards a cost minimization problem. This problem is made even more complex when taking into account the increasing amount of cloud providers, instance types and pricing plans available. Amazon, for example, offers two additional pricing plans next to the default pay-as-you-go plan: their *spot* market offers dynamically priced instance hours based on supply and demand, and the procurement of reserved instances allows consumers to run servers at a discounted rate in return for an upfront payment.

Although job scheduling problems typically deal with fixed capacity infrastructure, this increase in flexibility does not make job scheduling obsolete. On the contrary, the procurement of one or more reserved contracts inherently introduces a fixed capacity infrastructure again, albeit with more flexibility in return for a higher cost. Besides, public cloud providers are often adopted by organizations and companies to be used in combination with the organization's existing private infrastructure. The setup in which public cloud resources are used to supplement private clusters is referred to as a *hybrid cloud* model. The use of hybrid clouds introduces the need for a scheduling approach that combines cluster scheduling and the aforementioned cost minimization problem to determine which workloads are to be outsourced, and to what cloud provider.

1.4 Objective, Research Questions and Contributions

The objective of this thesis is to explore, develop and evaluate scheduling algorithms that aim to maximize value or minimize cost for usage in clusters and clouds, in the context of a multi-user organization. This thesis answers the following research questions:

- **To what extent does fine-grained value-based cluster scheduling outperform coarse-grained scheduling in terms of generated value for a bag-of-tasks application with a deadline?**
  Our contribution compares the performance, in terms of generated user value, of fine-grained value-based scheduling approaches with a coarse-grained scheduler based on priority queues. We elaborate on the experimental parameters that lead to higher efficiency gains for fine-grained techniques and pinpoint in which settings such techniques can bring significant efficiency gains. Based on real-world workload traces, we evaluate to what extent these settings are fulfilled in practice. Our findings are compared to those presented in related contributions, and the dissimilarities are discussed.

- **Is a linear programming approach suitable and computationally feasible to solve the cost minimization problem of running a set of bag-of-tasks within deadline in a hybrid cloud environment?**

Our contribution analyzes whether the concerned optimization problem can be tackled in the context of resource provisioning for batch workloads. The model considers preemptible but non-provider-migratable workloads with a hard deadline that are characterized by memory, CPU and data transmission requirements. A linear programming formulation of the optimization problem is presented, after which we evaluate how the runtime of the program's solver scales with changes in the different properties of the problem.

- **How can heuristics be used to schedule a set of BoT applications with a deadline constraint and an associated data set on public and private cloud resources in a cost minimizing and scalable way?**
  Our contributions focus on the optimization problem of allocating resources from both a private cloud and multiple public cloud providers in a cost-efficient manner, with support for data locality. The application model under study consists of non-preemptible and non-migratable bag-of-task applications with a hard deadline. We analyze how this optimization problem can be tackled in the context of resource provisioning for batch workloads through the use of different scheduling algorithms, and investigate the impact of workload model parameters as well as runtime estimation errors on the cost reductions that can be achieved.

- **How can a procurement algorithm obtain cost reductions by acquiring IaaS reserved contracts in a fully automated manner using load predictions?**
  Our contributions introduce an algorithm for automated IaaS contract procurement that takes into account an organization's existing contract portfolio. To guide contract acquisition through workload prediction, different time series forecasting techniques (including genetic programming-based models and both seasonal and non-seasonal models based on ARIMA and exponential smoothing) are applied. An illustrative case study and an extensive empirical evaluation of the proposed algorithm and forecasting techniques are presented.

1.5 Structure

This thesis consists of three parts:

Part I: **Value-based Scheduling in Batch Job Cluster Systems**
A considerable amount of scientific contributions exist on the topic of value-based scheduling in the context of clusters. In Chapter 2, the added value of using such a value-based scheduling technique in comparison with a coarse-grained technique in form of a queue-based scheduler is evaluated. We examine the influence of fine-grained user-valuations on the scheduler's performance, and evaluate whether such scheduling techniques outperform coarse-grained techniques in terms of realized value.

Part II: **Cost-Efficient Scheduling Jobs on Hybrid Clouds**
When using a public cloud provider to supplement private infrastructure, scheduling approaches similar to the ones used in clusters are no longer

sufficient. The decision which workloads to outsource to what cloud provider in a hybrid cloud setting is far from trivial. It should maximize the utilization of internal infrastructure while minimizing the cost of the cloud infrastructure. Part II explores this optimization problem with a focus on batch-job applications with a hard deadline constraint.

In Chapter 3, this problem is tackled using a binary integer program, and data transfer costs are taken into account. Chapter 4 and Chapter 5 present heuristics to cope with the aforementioned problem. Data transfer times are taken into account, and *online* arrival of applications is introduced.

Part III: **Contract Portfolio Optimization using Load Prediction**

The proliferation of pricing plans and resource properties at IaaS providers turned the procurement and management of an organization's cloud resources into a complex and time-consuming task. Some cloud providers offer "reserved contracts" in addition to a simple hourly server rate. These reserved contracts allow for a cost reduction in the hourly cost in return for an upfront payment. Part III of this thesis investigates whether the automatic procurement of such reserved contracts based on predictions of future load is feasible and beneficial in terms of cost.

Chapter 6 uses a Genetic Programming approach to forecast an organization's future load and calculate an optimal contract portfolio. In Chapter 7 and 8 the problem space is broadened to incorporate the current set of contracts an organization possesses. An algorithm is presented, and multiple time series models are applied to generate load predictions. Chapter 7 includes a case study of four real-world web application traces, in Chapter 8 the evaluation is expanded to comprise a statistical evaluation of the results for over fifty traces.

Part I

Value-based Scheduling in Batch Job Cluster Systems

CHAPTER 2

Benefits of fine-grained value-based scheduling on clusters

This chapter is published as "An evaluation of the benefits of fine-grained value-based scheduling on general purpose clusters", R. Van den Bossche, K. Vanmechelen and J. Broeckhove in Future Generation Computer Systems 27 (2011) [2].

Abstract

General purpose compute clusters are used by a wide range of organizations to deliver the necessary computational power for their processes. In order to manage the shared use of such clusters, scheduling policies are installed to determine if and when the jobs submitted to the cluster are executed. Value-based scheduling policies differ from other policies in that they allow users to communicate the value of their computation to the scheduling mechanism. The design of market mechanisms whereby users are able to bid for resources in a fine-grained manner has proven to be an attractive means to implement such policies. In the clearing phase of the mechanism, supply and demand for resources are matched in pursuit of a value-maximizing job schedule and resource prices are dynamically adjusted to the level of excess demand in the system. Despite their success in simulations and research literature, such fine-grained value-based scheduling policies have been rarely used in practice as they are often considered too fragile, too onerous for end-users to work with, and difficult to implement. A coarse-grained form of value-based scheduling that mitigates aforementioned disadvantages involves the installation of a priority queuing system with fixed costs per queue. At present however, it is unclear whether such a coarse-grained policy underperforms in value realization when compared to fine-grained scheduling through auctions, and if so, to what extent. Using workload traces of general purpose clusters we make the comparison and investigate under which conditions efficiency can be gained with the fine-grained policy.

2.1 Introduction

In many organizations, clusters fulfill the ever increasing need for computational power and data storage. Their scaling potential and performance have ensured that clusters have remained an important part of an organization's IT infrastructure, despite the advances in the performance of personal computers. As a consequence of their scale and cost, clusters are typically shared by a number of users whose resource requirements vary over time. The possibilities for parallel execution on these systems, combined with their ability to multiplex and enqueue user workloads, allow them to deliver high performance under a high level of system utilization. The key software component that determines the *efficiency* under which such clusters operate is the job scheduling system. Efficiency can hereby be expressed as a function of a wide variety of metrics such as system utilization, job turnaround times or job throughput.

In many clusters, job scheduling has long allowed for a prioritization of jobs through the definition of a discrete number of job queues. A job that is submitted to a high-priority queue can thereby gain precedence over jobs in lower-priority queues. The right to submit a job to a particular queue can be constrained by for example the user's priority level or the (user-estimated) runtime of the job. In that case, the highest priority queues are typically only available to jobs with relatively short runtimes, in order to avoid small jobs to be delayed by a few very long-running jobs.

In recent years, a renewed user-oriented view on efficiency has fueled the development of job scheduling systems that attempt to directly take the *value* that a user attributes to the completion of its computation into account [3, 4, 5, 6, 7, 8, 9, 10, 11]. Job scheduling systems that adhere to such a value-oriented efficiency goal are termed *utility-based* or *value-based* scheduling systems. In value-based scheduling approaches, one allows for the direct expression of user-value, irrespective of any constraints on job runtimes or other job characteristics. In order to discourage users to consistently express the highest possible value for their jobs, an accounting system charges users according to the service level they have obtained.

We distinguish between two different ways for users to express their valuations. In a *coarse-grained* value-based scheduling policy, users can signal one of a few discrete, predefined values to the scheduling mechanism. An example implementation of such a policy is a *priority queue system* in which a particular charging rate is associated with each queue. On the other hand, a *fine-grained* value-based scheduling policy allows users to express their valuation within a continuum. This allows the scheduler to enforce a more precise prioritization on the executed jobs.

Many works have investigated economically-inspired approaches to job scheduling in which market mechanisms are used that allow a user to express his valuation in a fine-grained manner [12, 13, 14, 15, 16, 17]. An advantage of such an approach is that the service cost can be set dynamically depending on the level of congestion in the cluster and the value that other users attribute to their jobs. In this manner, supply and demand for resources can be balanced, leading to optimal economic efficiency. This level of efficiency can be attained without the need for an administrator to manually intervene and, for example, configure the queue prices in a priority queuing system.

A disadvantage of fine-grained value-based schemes is that users are now asked to formulate their valuation as a value in a continuum. These valuations are often dependent on the expected completion times of the job. This is a non-trivial task that can be considered time consuming, especially if there are possibilities for a user to obtain a more attractive service level to cost ratio, by acting strategically. Although the installment of *(pseudo–)incentive compatible* [18] market mechanisms can remove the options for such strategic behavior, value elicitation in a non-strategic setting can remain a user burden.

Many studies have shown the efficiency gains, in terms of generated user value, that value-centric scheduling approaches can bring compared to traditional system-centric scheduling approaches such as First-Come-First-Serve (FCFS) or Round-Robin (RR) scheduling [3, 9, 19, 20, 21, 22]. A round-robin scheduler assigns equal time slices to each job in the system. However, only a limited number of studies have compared the performance of a fine-grained value-based scheduling system with a coarse-grained system that is based on priority queues. We have found that when such comparisons are made, insufficient attention is often given to the specificity of the experimental parameters that lead to high efficiency gains for the fine grained approaches. The goal of this paper is to present such a comparison and pinpoint in which settings fine-grained user valuations can bring significant efficiency gains. Our additional aim is to evaluate to what extent such conditions for increased efficiency are fulfilled in practice, based on real-world cluster workload traces.

2.2 Related work

Chun et al. [5] compare a market-based algorithm called *FirstPrice* with a three-queue priority system called *PrioFIFO*. They model synthetic workloads on a small cluster consisting of 32 nodes, and combine time-varying bids with static bids. Their study shows that for sequential jobs, coarse-grained static valuations are just as effective as fine-grained, time-varying valuations. As parallelism in the workload increases, so do the benefits of the fine-grained time-varying approach. Chun et al. report increases in realized utility up to 250% for highly parallel workloads. We take a closer look at these results in Section 2.6.

AuYoung et al. [6] compare a batch scheduler with four priority queues based on the Maui-scheduling algorithm [23] and a conservative backfilling algorithm with an implementation of a periodic combinatorial auction (CA) [24]. Maui is an open source job scheduler for clusters. In a combinatorial auction, users submit bids specifying resource combinations accompanied with an amount of money they are willing to pay for that combination. Periodically, the auction clears and determines a number of winning bids. The problem of clearing such a combinatorial auction in a value-maximizing manner is known to be NP-complete [25]. AuYoung et al. use a greedy approximation algorithm in order to clear the market in a timely fashion. Their simulation was based on a one week workload trace of the SDSC Blue Horizon cluster, a publicly available trace from the Parallel Workloads Archive [26]. They obtained up to 400% improvement with the CA compared to the batch scheduler. The differences between the batch scheduler and the combinatorial auction-based approach are however big: while the combinatorial auction uses advance reservations in order to optimize the schedule, the batch scheduler schedules jobs in an

online manner. It is therefore difficult to indicate what underlying cause elicits this difference. Because the CA's implementation details and algorithm, the clearing period, the allocation window and the workload data are not specified in [6], we were unable to reproduce their results.

In [4], Lee et al. generate complex piecewise-linear utility functions for jobs in workload traces. They compare a heuristic based on a genetic algorithm (GA) that maximizes the aggregate value generated in the system according to these utility functions with the performance of the *Priority-FIFO* algorithm (similar to the *PrioFIFO* algorithm in [5]), and an EASY and conservative backfilling algorithm. A genetic algorithm is a search heuristic that emulates the natural evolution process, and is used to generate solutions to search and optimization problems. Backfilling allows short jobs to skip ahead in the queue provided they do not delay any other job in the queue (conservative) or they do not delay the job at the head of the queue (EASY). Using a workload containing 5000 jobs from the SDSC Blue Horizon cluster, Lee et al. show that the GA heuristic outperforms FIFO with backfilling and *Priority-FIFO* on aggregate utility by respectively 14% and 6%. Lee et al. focus on *load* and *decay type* of the user's utility function as the parameters that cause these differences and point out that both *Priority-FIFO* and the GA heuristic perform well in high load conditions. The higher the load, the greater the performance difference is between value-based and traditional schedulers.

Libra is an economy-based job scheduling system for clusters, based on a proportional share scheduling algorithm. In a proportional share algorithm every job has a weight, and jobs receive a share of the available resources proportional to the weight of each job. Sherwani et al. [27] present the details of this scheduler, as well as a detailed performance analysis. They show that Libra's proportional share algorithm performs better than a FIFO scheduler. No comparison is however made with a priority queue scheduling policy.

Many others [9, 19, 20, 21, 22] have compared fine-grained value-based and traditional scheduling methods. In these studies little or no attention was paid to the use of priority queues as an alternative for fine-grained value-based schedulers.

2.3 Value-based scheduling

We assume that all users have a certain understanding of the value they associate with the jobs they want to run on a cluster. The expression of value is done through a medium common to all users of the shared resource. A real or virtual *currency* is used to fulfill this role. The communication of these user valuations to the scheduling mechanism allows for the construction of an economically efficient schedule, in the sense that scarce resources are allocated to users who value them the most. In order to prevent users from consistently communicating the highest possible value to the scheduler, users are endowed with limited budgets that they use to pay for the execution of their jobs. Unless incentive compatible mechanisms are used, the value the user communicates to the scheduler does not necessarily correspond to the user's *private value* for the job's execution. In non-incentive compatible mechanisms, users have a potential gain by not revealing their private value truthfully. Techniques such as second pricing [28] or k-pricing [29] can be used to achieve incentive compatibility. In the context of this work however, we do not consider

such strategic behavior and focus on the differences between a coarse-grained and fine-grained expression of user valuations.

In a fine-grained value-based scheduling policy the expression of a user's valuation is typically done by means of a *bid*. A bid can be time-varying (as presented in [5, 30, 4]) or static (as in [31, 10, 32, 33, 34]). With time-varying bids, the user submits a bid as a monotonically decreasing function of the job's completion time. The scheduling policy can then take the evolution of the job's value with respect to the completion time into account to optimize the schedule. A static bid does not include this information.

In a coarse-grained value-based scheduling policy a user picks one of the discrete values defined by the policy to express his valuation. In the priority queue system that we consider as a model for the coarse-grained approach in this paper, this involves a choice for the job queue whose charging rate best fits the user's private valuation. Expressing a time-varying valuation in such a priority queuing system would be burdensome, as it would require resource requests to hop from one queue to another. Therefore, in order to obtain a fair comparison between both scheduling approaches, we use a model in which a user expresses his valuation as a static value, an amount of money he is willing to pay for a job he submits to the cluster.

The jobs submitted on a general purpose cluster are not always trivially parallel[1] and may therefore require co-allocation, i.e. the simultaneous allocation of multiple processors to one job. Adding co-allocation support in a scheduling algorithm gives rise to significantly higher delays and lower utilization due to schedule fragmentation, because jobs have to wait for sufficient processors to become available before being scheduled. We define the *parallelization degree* of a job to be the number of processors that need to be allocated to the job. The higher the average parallelization degree of a workload, the higher the chance utilization on a cluster will drop due to fragmentation and higher delays.

In order to minimize fragmentation and maximize the utilization without violating the value-based decisions made by the scheduler, both algorithms presented in this section implement EASY backfilling as described in [35]. EASY backfilling allows short jobs to skip ahead in the queue provided they do not delay the job at the head of the queue. If the first job in the queue cannot start, the backfilling algorithm attempts to identify a job that can backfill. Such a job must require no more than the currently available processors, and it must not delay the first job in the queue: either it terminates before the time the first job is expected to begin, or it only uses nodes that are left over after the first job has been allocated its processors. To implement this feature, we have to assume the knowledge of the exact runtime of each job. Determining estimates of task runtimes is a complex problem that has been extensively researched [36, 37, 38, 39, 40]. For workloads that are executed repeatedly (e.g. in-silico analysis of protein folding processes for a pharmaceutical company) and for which there is a clear relationship between the application's parameters and its runtime (e.g. a parameter sweep application), these authors show that it should be possible to build fairly accurate models. Depending on the type of the application, we acknowledge that making exact predictions can be difficult and that the error on the predicted runtimes can be significant. Examining the influence of errors in the

[1] Applications are called trivially parallel when the jobs that form the application are fully decoupled and have no precedence constraints.

runtime of jobs on a backfilling algorithm falls beyond the scope of this paper. The resulting scheduling error made by the backfilling algorithm would however be the same for both our fine-grained and coarse-grained algorithms. Therefore, this issue has no influence on the results of the comparison made in this contribution.

Because only few cluster systems currently allow jobs to be interrupted by the scheduler once they are started, our scheduling algorithms does not use preemption. Once the execution of a job has started, it cannot be interrupted to give place to a job with a higher value.

Priority Queues

We have chosen to adopt a priority queuing scheme as an implementation of a coarse-grained value-based mechanism as it provides a good fit for this scheduling model and it is used in practice. In this priority queue implementation, all jobs from a higher priority queue are executed before jobs in lower priority queues, while requests in each queue are handled in a FIFO-manner. Each queue has a predefined and fixed charging rate associated with it, so that low valued jobs map in a low priority queue and high valued jobs map into a high priority queue. This algorithm is also used in [4] and [5].

When building such a queue-based system, two parameters are to be determined. Both parameters have a significant impact on the total value generated by the queues.

First, the *number of queues n* determines the granularity of our system. There are extremes for this parameter where $n = 1$, an equivalent to a FIFO-queue where no valuation information is used to schedule the jobs, and $n = J$ with J the total number of jobs, thus creating an equivalent for a fine-grained system in which each separate valuation is taken into account.

The second parameter is the configuration of the *queue's charging rates*. When users associate a value v with the execution of a job, they submit their job to queue i with the highest charging rate P_i lower than v. Given a scheduling mechanism with n queues and lower bound a and upper bound b for the user valuation distribution, we define P_i as given in equation 2.1.

$$\forall i \in \{0 \ldots n-1\} : P_i = a + i \cdot \frac{b-a}{n} \tag{2.1}$$

Note that the above queue price setting mechanism is not necessarily optimal. An optimal, value maximizing price setting mechanism is however highly dependent on the distribution of the user valuations and it is therefore hard to find in the general case.

Auction

As an implementation of a fine-grained value-based scheduling mechanism, we use an approach based on a first-price auction. In such an auction, the user communicates a bid to the auctioneer and pays for the execution of his job in accordance to his bid. From a value-maximizing point of view, a schedule is optimal when a job with value v can only be delayed by jobs with a value higher than v. Our fine-grained scheduling policy is therefore implemented as an auction in which all bids are sorted

in a list. The scheduler considers each bid separately and greedily schedules the bid with the highest value, then the bid with the second highest value, and so on.

In this paper we want to compare the impact of the choice for a coarse- or a fine-grained value-based scheduling policy. Therefore, we try to minimize the differences between both approaches. We believe that this simple auction model without preemption provides a good basis for comparison with the aforementioned priority queue approach. Note that, due to the online and greedy nature of the policy and the absence of preemption, optimal efficiency in terms of generated user value is not guaranteed. Such a "spot market" model without preemption is however common in literature [41, 31, 8, 19]. More complex models would include preemption [42] or allow for advance reservations [14, 6].

2.4 Simulation

In this section we discuss the details of the simulated environment that we use to compare both scheduling policies. The use of simulation enables us to efficiently study the consequences of varying parameters on the performance of both algorithms in a controlled manner.

Workload

We use real-world workload traces from the Parallel Workloads Archive [26] to model the workload in our simulation. This archive contains raw log information regarding the workloads on parallel machines. There are more than 20 traces available. In order to obtain results that are independent of specific workload characteristics, we have selected three traces with a significantly different workload. We discuss these traces in detail in Section 2.5.

In addition to general simulation information such as number of processors, number of users and total duration, we also extracted the User ID, the job's submission time, the number of processors requested and allocated and the run time for each job in the trace.

Valuation distribution

While traces are very useful to model realistic workloads, this is not the case for modeling valuation distributions. The currently available workload traces include little or no information on the users' valuations for each job. The only information on user valuations available in some of the workload traces is the ID of a priority queue in the cluster's scheduling system. While converting from a continuously distributed valuation to a priority queue system is easy, the reverse operation is much more complicated. Lee et al. [30, 4] attempt to use these user-assigned priority levels and wait times to generate complex utility functions for each job in a workload trace. We however believe that the choice for a certain priority level is user-specific, and it does not necessarily give a precise picture of the user's real valuation for that job. Different users have different methods for mapping their tasks on a queue, and some users probably don't have any method at all. In our opinion, the user's choice for a certain priority level is in most cases not a purely value-based decision, but may instead be influenced by for example the number of queues, the available charging rates,

the load at the moment of job submission, the implemented currency system and budget distribution or additional constraints such as a restriction on the maximum runtime or maximum parallelization degree of a job.

Because of lack of robust real-world statistical data on this matter, we have chosen to model our valuation distribution as a normal distribution with a mean μ and a varying density σ.

Grid Economics Simulator

We have evaluated the proposed scheduling algorithms in a simulated market environment delivered by the Grid Economics Simulator (GES) [43]. GES is a Java based simulator that has been developed in order to support research into different market organizations for economic cluster and grid resource management. The simulator supports both non-economic and economic forms of resource management and allows for efficient comparative analysis of different resource management systems. In order to run experiments efficiently, the simulator provides a framework based on Sun's Jini [44] technology to distribute experiment workloads over clusters and desktop machines, which is used in this study. In each experiment we have conducted 5 iterations for every sample point of the independent variable to obtain statistically significant results. Further on, we present and discuss average values for each of these sample points. Because the relative standard deviations of all simulations were very low, with an upper bound of 1.25%, they are not mentioned in the discussion of the simulation results.

2.5 Evaluation

In this section we describe the metric we use to compare coarse- and fine-grained scheduling systems presented before. We also discuss the factors which potentially have an impact on the difference between both approaches, and describe our experimental setting.

User delay tolerance metric

A common metric in the evaluation of value-based scheduling mechanisms is the *aggregate value* that is generated by the execution of all the jobs in the schedule. In our simulation, all jobs in the workload will be fully executed. If we derive the aggregate value from the static values users communicate to the scheduling policy, the total generated value will be the same with all scheduling policies; only the sequence in which the jobs are processed will differ. We already discussed that a value-based schedule is optimal when a job with value v can only be delayed by jobs with a value higher than v. In order to evaluate scheduling policies in that respect, we need a metric that gives more weight to a faster turnaround time of a job by taking into account the *delay* a job suffers. Moreover, we want this delay to be evaluated in relation to the value the user associates with the job.

In this contribution, we use a decreasing valuation function to measure the delay in relation to the value the user associates with the job. The initial value is thereby assumed to be equal to the public value the user communicates to the cluster's scheduler, and remains constant for the duration of the job. When a user's

submitted job is executed immediately, the generated value is equal to this public value. If, however, the job's execution cannot start immediately, the generated value decreases. We model the slope of this decay as a decreasing function of the user's patience, measured as multiples of the job length.

Following [5, 45], we therefore introduce a linearly decreasing function modeling the *delay tolerance* of the user. According to a survey in [30], users sometimes have even more complex job valuation functions. Lee et al. observed that many of the user-provided functions show a very steep drop in value in the moments after job submission, with a leveling off later. To examine the influence of the shape of the decreasing delay tolerance function on our comparison, we also introduce a exponential decreasing function modeled after one of the proposed utility functions in [30]. Both curves are illustrated in Figure 2.1.

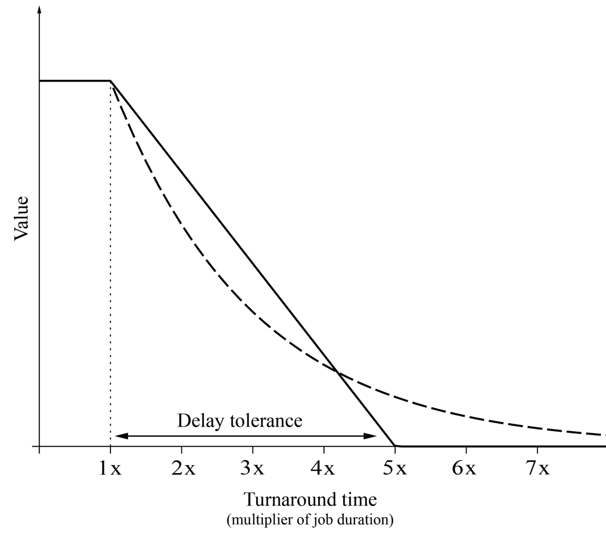


Figure 2.1: Valuation function

In the linearly decreasing function, the delay tolerance equals the time until the delivered private value is 0. The exponential function initially decreases faster than the linear function, but never becomes zero. The Equations 2.2 and 2.3 show respectively the piecewise linear function f and the piecewise exponential function g, where v is the user's public value for the job and δ is the user's delay tolerance.

$$f_\delta(x) = \begin{cases} v & : \quad 0 \le x \le 1 \\ -\frac{v}{\delta} \cdot (x-1) + v & : \quad 1 \le x \le \delta + 1 \\ 0 & : \quad x \ge \delta \end{cases} \tag{2.2}$$

$$g_\delta(x) = \begin{cases} v & : \quad 0 \le x \le 1 \\ v \cdot e^{\frac{1-x}{\tau}} & : \quad x \ge 1 \end{cases} \tag{2.3}$$

In order to obtain the value of τ in function of the delay tolerance δ in Equation 2.3, we need to define the relationship between functions f and g. We state that the

integral from 0 to infinity of f and g for the same delay tolerance δ should be equal, as stated in Equation 2.4. Solving this equation as shown in Equation 2.5–2.8 results in a value of $\tau = \frac{\delta}{2}$.

$$\int_0^{\infty} f_\delta(x)\,dx = \int_0^{\infty} g_\delta(x)\,dx \tag{2.4}$$

$$\int_1^{\delta+1} -\frac{v}{\delta}\cdot(x-1)+v\,dx = \int_1^{\infty} v\cdot e^{\frac{1-x}{\tau}}\,dx \tag{2.5}$$

$$\frac{v\cdot\delta}{2} = \lim_{x\to\infty} -v\cdot\tau e^{\frac{-x}{\tau}} \tag{2.6}$$

$$\frac{v\cdot\delta}{2} = v\cdot\tau \tag{2.7}$$

$$\frac{\delta}{2} = \tau \tag{2.8}$$

In the remainder of this paper, we use the aggregate value generated by the schedule as a metric to compare the performance of our scheduling policies. Following [4, 5], this aggregate value is defined as given in Equation 2.9, where $value_j(delay_j)$ denotes the value that the execution of job j with completion delay $delay_j$ generates for the user.

$$\text{Aggregate value} = \sum_{j\in jobs} \text{value}_j(\text{delay}_j) \tag{2.9}$$

The higher the value for this metric in a certain scheduling policy is, the smaller is the delay experienced by the higher valued jobs. It is important to understand that these decreasing valuation functions are only a measure for the delay suffered by a job in the scheduling algorithm. Therefore, these functions are private, they will never be communicated to the cluster's scheduling algorithm, and optimizations to increase the total generated private user value can only be made by correctly scheduling all jobs with a higher value before any job with a lower value.

Workload Traces

The traces from the Parallel Workloads Archive used in this contribution are listed in Table 2.1. All traces are characterized by a long duration and relatively high load. The long duration is beneficial to the robustness of our experiments, while the relatively high load is necessary to bring out the qualitative differences in terms of value realization between different scheduling policies.

Table 2.1: PWA Workload traces

Name	Duration	# Jobs	# Users	# CPU's	ϕ
SDSC Blue Horizon	32 months	243.314	468	1152	3.78%
SDSC SP2	24 months	59.725	437	128	10.75%
HPC2N	42 months	202.876	257	240	2.64%

We pointed out earlier that the delay experienced by the jobs is an important metric in our comparison and that the average parallelization degree of a workload will probably be a determining factor in this regard. It is therefore important that the parallelization degree of a job is evaluated in relation to the total number of processors in the cluster. For a job that needs 100 processors for example, it is indeed clear that scheduling the job on a cluster with 128 processors will induce a much higher delay for all subsequent jobs than scheduling the same job on a cluster with 1152 processors. We therefore introduce the measure ϕ as the average parallelization degree of the workload, expressed as a percentage of the size of the cluster. If we take a closer look at the cluster traces available in the Parallel Workloads Archive, we observe that for all cluster traces ϕ has a 95% confidence interval of [2.84%, 5.94%].

The SDSC Blue Horizon log has been previously used by many other scheduling studies, including [6, 4, 46, 47]. Its value for ϕ lies in the confidence interval, which makes it a suitable starting point for our comparison. We will also take a look at a trace from the SDSC SP2 cluster, which is smaller than the Blue Horizon cluster. The average parallelization degree for the SP2 trace is much higher ($\phi = 10.75\%$) than in most other traces. Next to these traces from the San Diego Super Computer Center, we also included a trace containing jobs from the High-Performance Computing Center North (HPC2N) in Sweden. The HPC2N trace is more recent than the other traces, and it has an average parallelization degree below the confidence interval ($\phi = 2.64\%$). On this cluster, the average job size is thus relatively small.

Experiments

In experiment 1, we will evaluate the difference in aggregate value between our auction-based scheduler and a priority based scheduler when considering a varying number of queues ranging from 1 to 5. We also investigate the impact of an increasing user delay tolerance. Prices for the queues are set as specified in Section 2.3 in an interval between 100 and 10000. We assume the user's delay tolerance function to be linearly decreasing, and the user's valuations to be normally distributed with $\mu = 5000$, $\sigma = 2000$ and a lower bound of 100.

One of the primary goals of this work is to pinpoint which experimental parameters have a significant impact on the differences between coarse- and fine-grained scheduling approaches. Therefore we will take a closer look at a few of these parameters we expect to affect the differences between them. Lee et al. [4] already emphasized the influence of the load and the decay type of user's valuation function. We concentrate on:

- *Parallelization degree* - In experiment 2 we increase the parallelization degree of our workload trace with a factor ρ, while keeping all other parameters the same as in experiment 1.
- *Load* - We increase the load in experiment 3 by scaling the inter-arrival times of the jobs with a factor θ.
- *Type of user delay tolerance function* - In experiment 4 we change the slope of the user's delay tolerance function from linear to exponential, as shown in Figure 2.1.

Table 2.2: Experiment Parameters

Ex.	Val. func.	μ	σ	ρ	θ
1	linear	5000	2000	1	1
2	linear	5000	2000	$[1,4]$	1
3	linear	5000	2000	1	$[0.7,1]$
4	exponential	5000	2000	1	1
5	linear	5000	$[1000,4000]$	1	1

- *Valuation distribution* - In experiment 5, we vary the spread σ of the normal valuation distribution.

The simulation details of the experiments are summarized in Table 2.2.

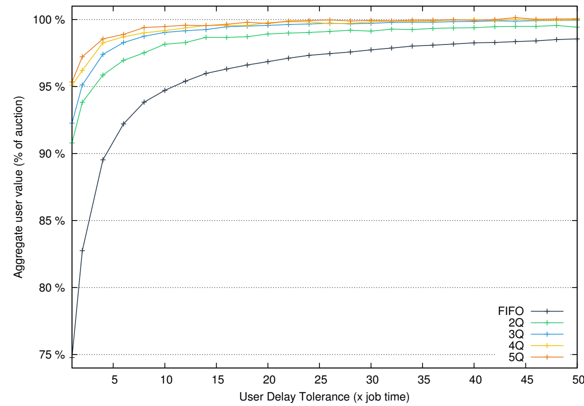
2.6 Results

Before taking a look at the influence of the specific parameters presented in Section 2.5, we will discuss the results of the first experiment shown in Figure 2.2. The figures show the aggregate user value generated by the schedule, as a percentage of the value generated by the auction-based scheduling policy.

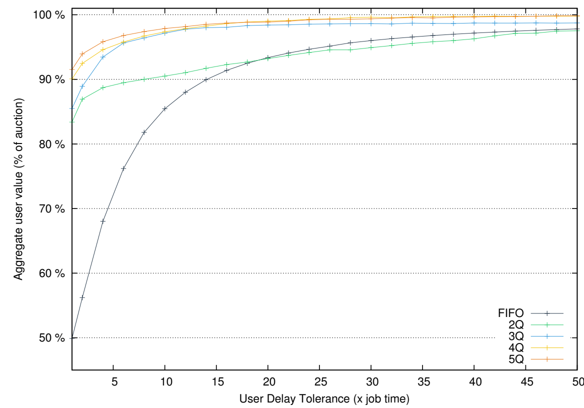
In Figure 2.2a we present the results of the SDSC Blue Horizon trace. We observe that in case of a small delay tolerance, the value-based algorithms outperform the FIFO scheduler with a 15% margin. The number of queues in a priority queue system seems to have only a small impact on the generated value. When the number of queues increases, and thus the granularity of the coarse-grained system decreases, the generated user value grows. It thereby quickly approaches the value of the fine-grained auction.

Furthermore, when the user delay tolerance increases, the difference between the performance of the priority queue policy and the auction-based policy diminishes. The priority queue systems clearly perform better than the FIFO scheduler, and even approximate the performance of the fine-grained auction. The advantage of using a fine-grained scheduler in this case is thus limited, and a small number of queues are sufficient to attain a satisfactory level of prioritization in the schedule.

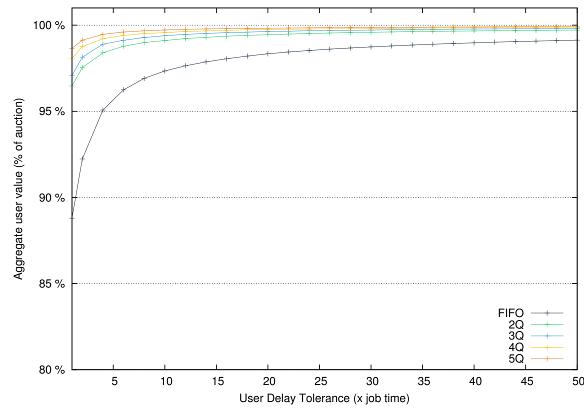
We also show the results for both the SP2 and the HPC2N trace in Figure 2.2b and Figure 2.2c. We recall that the SP2 cluster trace had a high average parallelization degree ϕ, and the HPC2N trace had a lower value for ϕ. As expected from our assumption that the average parallelization degree is a determining workload factor for the performance of the scheduling systems, we observe that for SP2 the relative aggregate value of both the FIFO and the queue schedulers are much lower than with the SDSC Blue Horizon workload. The FIFO and 2Q algorithm perform badly, but policies with a higher granularity accomplish to limit the loss in aggregate value compared to the fine-grained policy to less than 15%. On the other hand, the HPC2N results are the opposite. The FIFO algorithm's aggregate user value is always within 12% of the fine-grained algorithm, and the queues achieve values less than 4% below the auction's result.



(a) SDSC Blue Horizon



(b) SDSC SP2



(c) HPC2N

Figure 2.2: Experiment 1

It is important to note that the general trend in function of user delay tolerance in all three workload traces is the same. In order to identify the workload characteristics that influence these trends, we now isolate each of these characteristics by altering the first SDSC Blue Horizon workload model in the next experiments.

Parallelization degree

In this section, we analyze the effect of an increased parallelization degree on the realized user value. In order to perform this analysis based on the SDSC Blue Horizon trace, we increase the parallelization degree of the workload by multiplying the number of requested processors of each job with a factor ρ. We thereby make sure the number of requested processors remains smaller or equal to the total number of processors available in the cluster. In order to keep the total workload equal, we proportionally decrease the job's runtime with the same factor. The results of this experiment are presented in Figure 2.3. The solid lines represent a delay tolerance of 5, dashed lines are used for a delay tolerance of 20.

We clearly observe that the parallelization degree of a workload has a negative influence on the performance for both the FIFO and 3Q algorithms compared to the auction. In Figure 2.3, the value generated by the FIFO algorithm sinks away to 18.7% of the value generated in the auction setting with $\rho = 4$. However, the 3Q algorithm does relatively well with a value loss of only 20.1%. We observe similar but more moderate behavior with a delay tolerance of 20.

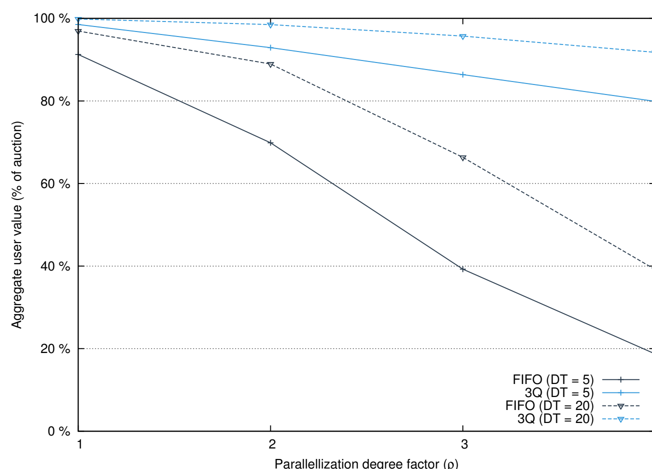


Figure 2.3: SDSC Blue Horizon - Experiment 2

The study by Chun et al. [5] confirms our findings that the workload's parallelization degree is an important factor when it comes to the difference between coarse- and fine-grained systems. They show that, for parallel workloads, a fine-grained system delivers up to 2.5 times higher performance than a priority queue scheduler, which is much more than in our experiments. They however modeled the parallelization degree as a uniform distribution over the total number of available processors. For the cases in which the fine-grained value-based *FirstPrice* policy significantly outperformed the coarse-grained *PrioFIFO* policy in their study, ϕ ranged from

12.5% to 50%. In the Blue Horizon trace used in our experiment ϕ equals only 3.78%, and we found earlier that for all PWA traces ϕ has a 95% confidence interval of [2.84%, 5.94%]. The average parallelization degrees used in [5] are thus significantly higher than those of the workload traces available in the PWA, which explains the differences between their results and ours.

Load

It is important for a scheduler to perform well in heavy load conditions. The SDSC Blue Horizon workload already has a fickle load pattern, sometimes reaching quite heavy loads. We further increase the load in experiment 3, by manually scaling the inter-arrival times of the jobs with a factor $\theta = \{0.9, 0.8, 0.7\}$. That way, the simulation's duration is compressed to respectively 90%, 80% and 70% of the original duration, while the job runtimes and number of jobs remain constant. Consequently, the load is inversely proportional to the inter-arrival time compression factor θ, with the highest load occuring when θ is small.

Results are shown in Figure 2.4. We again take a look at the situation in case of a user's delay tolerance of 5. The FIFO queue seems to be very sensitive to an increasing load. We hereby can confirm the findings from Lee et al. [4] that an auction-based or priority queue-based algorithm performs well in high load conditions. The greater the load, the greater the difference between value-based and traditional schedulers such as FIFO.

The advantages of a fine-grained approach over a coarse-grained priority queue system are however much less significant. The 3Q algorithm does much better than the FIFO-queue, with a decrease of 28.4% compared to the fine-grained scheduler. Using 5 queues, the coarse-grained method is able to limit this loss to 11.8%.

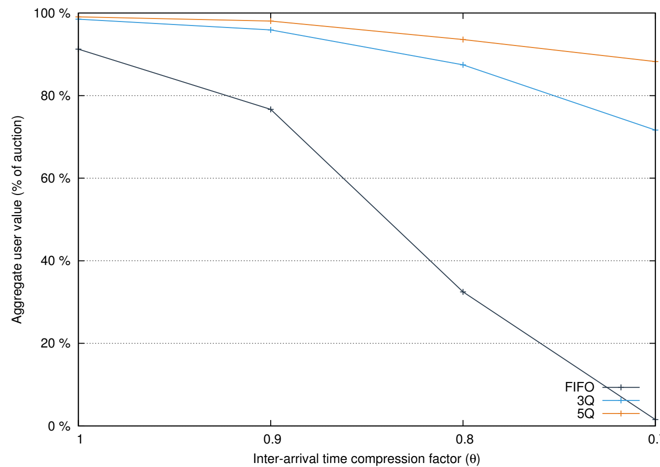


Figure 2.4: SDSC Blue Horizon - Experiment 3 - Delay tolerance 5

Type of user delay tolerance function

In Section 2.4 we presented two related delay tolerance functions: a piecewise linear and a piecewise exponential function. The exponential function decreases faster

in the beginning, denoting an impatient user, but never really becomes zero. No matter how long the user has to wait, he still associates some small amount of value with the execution of his job. When we compare the simulation results of the exponential function with the linear function's results, we observe the aggregate value realized from experiment 4 to be a little bit lower than in experiment 1. The average relative difference between both results is 0.36%, and has an upper bound of 2.41%. It is worth mentioning that, when the load in the system increases as simulated in experiment 3, the relative differences between both user delay tolerance functions for coarse- and fine-grained approaches remain small. As a consequence of these very small differences, the choice for a faster decreasing user delay tolerance function seems to have only a limited influence on the difference between coarse- and fine-grained scheduling mechanisms.

Note that the absolute differences between both delay tolerance functions are more significant. For example, using an exponentially decreasing valuation function, the fine-grained approach generates up to 4.61% less value than using a linearly decreasing function. These absolute differences are however not extensively studied in this work, as they have no impact on our comparative study between fine- and coarse-grained value-based scheduling approaches.

Valuation distribution

A much cited advantage of value-based scheduling algorithms over traditional schedulers is the ability to cope with a large spread of user valuations. The higher the difference between a high value and a low value job is, the bigger become the consequences of an inefficient schedule. We evaluate this claim in experiment 5, by varying the spread σ of our normal valuation distribution while keeping all other experimental parameters equal. The results are presented in Figure 2.5. The solid lines again represent a delay tolerance of 5, dashed lines are used for a delay tolerance of 20.

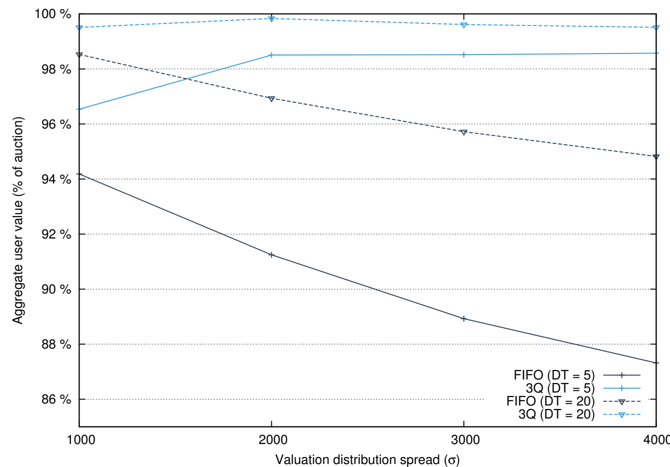


Figure 2.5: SDSC Blue Horizon - Experiment 5

As the figure shows, an increasing spread in the valuation distribution has a significant impact on the generated value of the FIFO-scheduler. Our coarse-grained priority queue system with 3 queues however, is able to cope with these larger differences in user valuations. Its relative performance compared to the fine-grained approach is almost constant.

2.7 Conclusion

Many authors have presented a comparison between a value-based scheduling implementation and traditional schedulers, such as FCFS or SJF, thereby generally showing large efficiency gains in favor of the value-based schedulers. However, only a few studies have compared fine-grained value based scheduling to scheduling with a coarse-grained value representation through priority queues.

A fine-grained approach allows users to express their valuation as a value in a continuum, which allows the scheduler to maximize the overall generated value by the system. In a coarse-grained approach, on the other hand, users attach a priority to the execution of their job by picking one of a few discrete, predefined values. The latter is much easier, more stable and less cumbersome, while the first has the advantage of being self-sustaining in the sense that supply and demand can be automatically balanced, without the need for a system administrator to configure and maintain queue prices.

We modeled users to have a private valuation that is dependent on a level of *delay tolerance*, and used *aggregate utility* as a metric to compare both scheduling approaches. Using real-world workload traces from the Parallel Workloads Archive, we evaluated under which realistic conditions a fine-grained scheduler, implemented as an online auction-based mechanism with static bids and without preemption, can outperform a priority queue system. We thereby found that a priority queue approach with a relatively low number of queues can closely approximate the performance of the auction-based approach. The higher the number of queues and thus less coarse-grained the system is, the smaller the difference between coarse- and fine-grained scheduling becomes. We further found that, for our fine-grained approach to become profitable, the *parallelization degree* and *load* of the cluster's workload must increase beyond levels that are currently found in publicly available workload traces.

It is important to understand that our implementation of a fine-grained scheduling system is a somewhat simplified approach, developed to pinpoint the differences between fine- and coarse-grained strategies. More complicated fine-grained value-based schedulers will certainly be able to perform much better than our implementation, thereby significantly outperforming a priority queue system. According to our results, these efficiency gains will not be the effect of the use of fine-grained valuations, but they will rather be a consequence of the use of more complex implementation features such as advance reservations and clearing periods, time-varying or combinatorial bids and preemption.

Part II

Cost-Efficient Scheduling Jobs on Hybrid Clouds

Cost-Efficient Scheduling Jobs on Hybrid Clouds

The IT outsourcing model in which access to scalable IT services hosted by external providers can flexibly be acquired and released in a pay-as-you-go manner, has seen an increasing adoption recently under the term *cloud computing* [48, 49]. A generally adopted definition by the National Institute of Standards and Technology (NIST) describes the cloud computing model as characterized by an *on-demand self-service*, available through *broad network access*, in which providers use *resource pooling* and provide *rapidly scalable (elastic)* resources, through a *measured service* [50]. Cloud services are further categorized as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS), depending on their focus of delivering respectively IT infrastructure, frameworks for software development and deployment, or a finished software product.

Our work focuses on the IaaS service class in which a consumer can acquire access to compute, storage, and networking capacity at the provider's site. To acquire compute resources, a consumer launches a server *instance* on the cloud provider's infrastructure, thereby specifying the instance's characteristics such as the available processing power, main memory and I/O capacity. Most commonly, the notion of an instance materializes into a virtual machine that is launched on the provider's physical IT infrastructure. Virtualization technology enables the provider to increase infrastructure utilization by multiplexing the execution of multiple instances on a single physical server, and allows one to flexibly tune the individual characteristics of an instance. Nevertheless, most providers predefine combinations of instance characteristics in a fixed number of *instance types*.

In the last few years, the IaaS market has quickly matured, with providers rapidly diversifying their product offerings in terms of pricing plans, instance types and services. Amazon's Elastic Compute Cloud (EC2) offering for example, has increased the number of instance types from one to sixteen in less than six years, and moved from a single on-demand pricing model to three different pricing models with the addition of *reserved instances* and dynamically priced *spot instances*. Different types of reserved instance contracts have been introduced depending on the expected utilization of the instance, and an open market for selling such contracts was launched[2]. Likewise, it has expanded the number of geographical *regions* in which instances can be launched from one to eight. The choice of the instance's region both influences the network characteristics in terms of available bandwidth and latency, as

[2]http://aws.amazon.com/ec2/purchasing-options/reserved-instances/

well as the cost of running the instance. Finally, the number of IaaS providers has increased significantly as well, with each provider differentiating itself in terms of services offered, prices charged for different resources (ingress and egress network bandwidth, memory, CPU capacity), and performance delivered.

Consequently, consumers face an increasing complexity in making cost-optimal decisions when matching their infrastructure requirements to the IaaS services currently available. Continuing standardization efforts in virtualization technology and IaaS offerings, such as OGF's Open Cloud Computing Interface[3], are expected to further increase the options available to a consumer when acquiring resources "in the cloud". This issue is exacerbated by the fact that consumers often also own internal IT infrastructure. Through the creation of hybrid clouds [51, 52], one can use this internal infrastructure in tandem with public cloud resources, thereby capitalizing on investments made, and maintaining a high quality-of-service level by employing the public cloud to deal with peak loads.

Although tools exist that allow a consumer to deal with some of the technical issues that arise when allocating resources at multiple public cloud providers [53, 54, 55, 56], they do not tackle the optimization problem of allocating resources in a cost-optimal manner and with support for application-level quality of service constraints such as completion deadlines for the application's execution. This lack of tool support combined with the inherent complexity of cost-optimal resource allocation within a hybrid cloud setting, renders this process error-prone and time-consuming. Despite current research efforts [57, 58, 59, 60, 61, 62, 63, 64, 65], cost-optimal resource scheduling and procurement of external resources while taking into account the availability of a local IT infrastructure remains an open problem. Moreover, a structured approach is required that can optimize resource allocations in a multi-consumer context. Indeed, the addition of volume or reservation-based price reductions in the pricing options of public cloud providers allows for the further reduction of costs if an organization collectively engages in resource allocations for its entire user base.

Within the HICCAM (Hybrid Cloud Construction and Management) project, we are investigating the design of software components and algorithms for building and deploying hybrid clouds efficiently, and for automated resource provisioning and management at the level of an entire organization. Within the project's software architecture outlined in Figure 2.6, the organization's Cloud Procurement Endpoint (CPE) is responsible for managing the procurement of resources from public cloud providers. Requests for executing applications are sent to the CPE by the different decision support systems (DSS) that assist the users in making a trade-off between quality of service levels and costs.

Problem Domain

Cloud Setting

In a hybrid cloud setting, the execution of applications occurs through the deployment of virtual machine instances on resources internal to the consumer organization, as well as resources provided by public cloud providers. The characteristics of

[3] Available at http://www.occi-wg.org

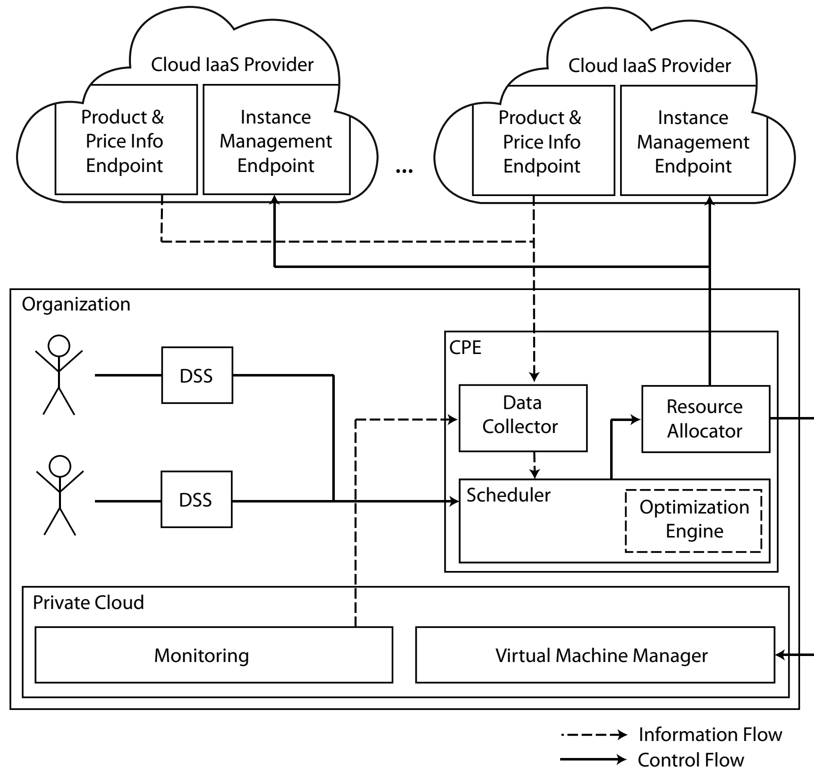


Figure 2.6: Component view of the HICCAM model

the virtual hardware on which these instances are executed are determined by the –mostly provider-specific– definition of *instance types*. An instance type fixes the number and speed of CPUs, the amount of memory, local storage and the architecture of the virtual machine instance. Note that some providers (e.g. CloudSigma) do not have fixed instance types, and allow users to customize and fine tune the cloud servers to fulfill the exact hardware needs of their applications. Selecting the best cloud server configuration for an application in a continuous spectrum of hardware properties is a task that ultimately results in one or a few server configurations with an associated cost suitable for the application. From the perspective of a scheduler, these configurations are similar to provider-specific instance types. Although simple, adding support for fine grained user-configured instance types would thus hardly affect the results in this thesis. Therefore, we assume in this study that every public cloud provider offers a number of fixed instance types and associates a price with the execution of a virtual machine on each instance type. A cloud provider charges for the time the instance has run, and commonly handles a discrete *billing interval* that has to be fully paid for. The cloud providers modeled in this contribution –Amazon EC2 and GoGrid– express and charge prices on a hourly basis. This implies that running a cloud server for 61 minutes costs the same as running the cloud server for two hours. A provider also charges for each gigabyte of network traffic used by the consumer's applications. In this regard, a distinction is made between the price for inbound and outbound traffic. For now, we do not consider the costs for additional

services such as persistent data storage, monitoring or automated workload balancing, as the cost for these services is minor compared to the computation and data transfer costs. Incorporation of storage costs becomes significant when the data set is hundreds of terabytes, or when an application needs the same data set over different, recurring, runs. The long term planning problem to decide where to store which parts of a data set is beyond this thesis' scope.

For the purposes of these chapters, the consumer's internal infrastructure is assumed to support a resource allocation model similar to that of a public cloud provider in that it also allows for the flexible deployment of virtual machines. Such a system is commonly referred to as a *private cloud* [48, 51].

We assume that a public cloud provider is able to fulfill all instance requests and thus, from a user's perspective, has an infinite amount of IaaS resources available. The private infrastructure obviously does not have an "infinite" capacity. The number of tasks running concurrently on our private cloud is therefore limited: the sum of the number of CPUs of all running instances may not exceed the total number of CPUs available. We do not explicitly take efficiency losses due to the partitioning of physical servers into multiple instances into account, as these can be dealt with by an appropriate choice of instance types and by adopting a suitable VM migration strategy. Incorporating migration overhead, virtualization overhead and virtual machine interference as well as performance differences between public and private cloud providers into our cloud model is left for future work.

Before an application can run on a cloud provider, the application's data set should be available on the site of that provider. Some providers, such as Amazon, define multiple regions in different continents to satisfy the demand for *data locality*. Chapters 4 and 5 take this data locality into account. It is assumed that all data sets reside on the organization's private cloud, and that the transfer speed from the organization to each of the available public cloud data centers is known a priori and does not fluctuate. As the data sets reside on the organization's private cloud, they are instantly available.

Workload Model

Heterogeneous workload characteristics and divergent user requirements complicate the problem of identifying the best cloud provider and instance type for a specific application. In this part, we narrow our view to only some of the characteristics and requirements, of which we believe they constitute a solid base to demonstrate the operation and implementation of a hybrid cloud scheduler. Our current application model focuses on batch type workloads. Examples of workloads that fit this model are simulation experiments that consist of a bag of independent instances of a simulation, image and video rendering codes and highly parallel data analysis codes. Common types that are not covered by our model are coupled computations that are distributed over multiple systems, workloads with a non-trivial workflow structure such as precedence constraints, and online applications such as web or database servers.

We assume that each application's workload consists of a number of trivially parallel tasks. The instance types on which tasks are executed are modeled as a set of *unrelated parallel machines* [66]. This means that every task can run on each instance type and that each instance type has a task-specific speed by which it

executes a task. When a task can be interrupted and later resumed by the scheduler without losing its state, that task is named preemptive. In Chapter 3, tasks are considered to be preemptive but non-provider-migratable. Chapter 4 and 5 assumes tasks have a non-preemptive nature.

In addition, an application is also associated with a data set. We assume that the full dataset has to be transferred to the cloud providers on which the application is running. Further on, we only focus on inbound traffic.We assume that a provider is capable of running all tasks of an application in parallel, and that there are no restrictions on the number of virtual machine instances that are simultaneously available to the organization at a public cloud provider. In this paper, applications are scheduled as a whole on only one cloud provider, as splitting applications over multiple providers increases complexity and leads to additional data transfer costs. The scheduling algorithms proposed can however be extended easily to support such a split execution, without invalidating the results of this contribution. Finally, each application is associated with a hard completion deadline by which all computational tasks in the application must have finished their execution.

Other aspects, such as security or legal constraints, lower the flexibility for making scheduling decisions and correspond to a temporary decrease in available capacity on the private cloud. As this setting is hard to configure and would not affect the relative performance of the proposed schedulers, it was not included in the model.

Determining estimates of task runtimes [36, 37, 38, 39] and modeling speedup of tasks on multiple processors [67, 68, 69, 70] are complex problems that have been extensively researched and fall beyond the scope of this paper. For workloads that are executed repeatedly because they form a structural part of an organization's business processes, we argue that it should be possible to build fairly accurate models. Such a setting allows for application-level benchmarking in order to map out the effect of using a particular instance type on application speedup. Depending on the type of the application however, we acknowledge that making exact predictions can be difficult and sometimes impossible. The extent to which a certain schedule is optimal is therefore dependent on the accuracy of the provided runtime estimates. Chapter 3 assumes that application runtimes are known a priori. Chapter 4 and 5 evaluate the sensitivity of the proposed heuristics on the accuracy of the provided runtimes.

Overview

In Chapter 3, a linear programming-based approach is used to formulate the scheduling problem faced in the CPE. Although the proposed program takes into account data transfer costs, data transfer times are assumed to be negligible. Runtimes of the applications are assumed to be known in advance. The proposed binary integer program is evaluated with regard to its computational cost for scheduling applications in both a public and a hybrid cloud setting.

Chapter 4 proposes a heuristic to tackle the hybrid optimization problem in a tractable manner. We extend our cloud model with data transmission speeds in order to take *data locality* into account during the scheduling process, and support the online arrival of applications. The influence of the different cost factors and workload characteristics on the heuristic's cost savings is evaluated, and the results' sensitivity to the accuracy of task runtime estimates is discussed.

In Chapter 5, alternative heuristics are presented to deal with the high sensitivity to runtime inaccuracies of the algorithms presented in Chapter 4.

Related work

Linear programming has been used numerous times for resource planning and scheduling [71, 72, 73, 74, 29].

The term *surge computing* was first introduced by Armbrust et al. [75]. OpenNebula [51] and Eucalyptus [76] are open source frameworks that allow to build a hybrid cloud. Buyya et al. [49] presented an early vision on the interconnection between cloud providers, an idea which was later adopted and elaborated in other contributions [48].

In Tordsson et al. [57], a binary integer program is used to tackle the problem of selecting resources among different cloud providers in a federated environment. They focus on a static approach in which online applications –non-finite applications without deadline constraints– are scheduled on cloud providers in such a way that the total infrastructure capacity is maximized, given budget and load balancing constraints.

Li et al. [58] investigate VM migration across multiple clouds while fulfilling budget, placement or load balancing constraints. They solve a linear program to calculate optimal schedules in case of changing cloud offerings or demands, taking into account migration overhead. Only static scenarios and public cloud providers are considered, and the computational complexity of solving the integer program is not evaluated.

In Lucas-Simarro et al. [59], a broker architecture is presented to deploy services across multiple cloud providers. The authors also use a binary integer program to optimize the placement of a set of virtual machines over multiple cloud providers based on cost or performance. They take into account dynamic pricing models such as the Amazon spot market, but do not consider the addition of a hybrid scheduling component.

Breitgand et al. [60] present a model for service placement in federated clouds, in which one cloud can subcontract workloads to partnering clouds to meet peak demands without costly over-provisioning. They use an Integer Program formulation of the placement program, and also provide a 2-approximation algorithm. Their workload model consists of VM requests with an associated availability QoS constraint, which is different from our bag-of-tasks workload model with deadline-based QoS constraints.

Strebel et al. [61] propose a decision model for selecting instance types and workloads to outsource to an external data center. They also use an optimization routine to create an optimal mix of internal and external resources for a given planning period. They assume that the application set is known a priori, and adopt a cost model without instance type differentiation and with a single CPU hour tariff, which differs from our work.

In Andrzejak et al. [62], a decision model is presented to determine bid prices on a resource market with varying prices, such as Amazon EC2's Spot Instances. The authors take into account the application's SLA constraints, and minimize the monetary cost for running the application.

Kailasem et al. [63] propose a hybrid scheduling technique for a data-intensive document processing workload. For such a real-time workload, the transfer time to the external cloud is comparable to its computational time. They decide which job to outsource to the public cloud provider based on bandwidth and transfer time estimates, while fulfilling various queue-specific SLA constraints. They only consider settings with a single cloud provider and a fixed number of running instances.

Lampe et al. [64] propose a cost-minimizing scheduling approach for software services based on an integer program, and consider both intra-machine resources, such as CPU or memory, as well as inter-machine resources such as network bandwidth. A hybrid setting or data locality are not studied. The approach only proves feasible for a small number of instance requests due to the high computational requirements for solving the proposed integer program.

Javadi et al. [65] present a failure-aware resource provisioning algorithm in a hybrid cloud setting, and schedule applications with deadlines in order to minimize the number of QoS violations under failing resource conditions.

Scheduling data-intensive applications has been researched in other domains as well, and related work on that topic can be found in [77, 78, 79]. Works of reference in parallel job scheduling include [80, 81]. Job scheduling with inaccurate runtimes has been previously studied in a non-cloud setting in [82, 83, 84, 85, 86, 87].

CHAPTER 3

Using Integer Programming for Scheduling on Hybrid Clouds

This chapter is published as "Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads", R. Van den Bossche, K. Vanmechelen and J. Broeckhove in Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD 2010) [88].

Abstract

With the recent emergence of public cloud offerings, *surge computing* – outsourcing tasks from an internal data center to a cloud provider in times of heavy load– has become more accessible to a wide range of consumers. Deciding which workloads to outsource to what cloud provider in such a setting, however, is far from trivial. The objective of this decision is to maximize the utilization of the internal data center and to minimize the cost of running the outsourced tasks in the cloud, while fulfilling the applications' quality of service constraints. We examine this optimization problem in a multi-provider hybrid cloud setting with deadline-constrained and preemptible but non-provider-migratable workloads that are characterized by memory, CPU and data transmission requirements. Linear programming is a general technique to tackle such an optimization problem. At present, it is however unclear whether this technique is suitable for the problem at hand and what the performance implications of its use are. We therefore analyze and propose a binary integer program formulation of the scheduling problem and evaluate the computational costs of this technique with respect to the problem's key parameters. We found out that this approach results in a tractable solution for scheduling applications in the public cloud, but that the same method becomes much less feasible in a hybrid cloud setting due to very high solve time variances.

3.1 Integer program

We introduce a binary integer program (BIP) that is based on the problem domain outlined in the previous section. The goal of our program is to deploy A applications $a_1, \ldots, a_A$ while minimizing the total cost of their execution on the consumer

organization. Each application a_k has an associated strict deadline dl_k and consists of T_k tasks $t_{k1}, \ldots, t_{kT_k}$ that are executed across the C cloud providers $c_1, \ldots, c_C$. We introduce I instance types $it_1, \ldots, it_I$. An instance type it_i has parameters cpu_i and mem_i, denoting the number of CPUs and amount of memory available.

A cloud provider c_j provides prices pi_j and po_j for each gigabyte of data traffic in and out of its data center, as well as prices p_{ij} per hour of resource consumption for each of the instance types it_i it supports. In order to cater for the resource constrained nature of private clouds, the total number of CPUs $maxcpu_j$ and amount of memory $maxmem_j$ available at a provider can be constrained. Although the private data center scheduling problem is limited to assigning resources from a resource pool to virtual machine instances, thereby ignoring the specific size and composition of individual machines in the data center, the feasibility of the proposed schedule is not affected if these issues are taken into account when drawing up the allowed instance types for the private cloud. Public cloud providers are considered to deliver an "unlimited" amount of capacity.

A task t_{kl} of application a_k has a runtime r_{kli} associated with each of the instance types it_i on which the task can run. This set of supported instance types is assumed to be a given with respect to the linear program. We assume that a matchmaking component will evaluate the hard constraints of an application (e.g. all tasks should run on servers within the EU region in order to comply with regulatory issues), and match these with instance type properties to create this set. A task t_{kl} is associated with inbound traffic volume (ni_{kl}) and outbound traffic volume (no_{kl}).

Time is explicitly represented in our programming model through the introduction of time slots with a granularity of one hour. Let S be the number of time slots in the schedule, with $S = \max_{k \in \{1,\ldots,A\}}(dl_k)$. Let $x_{klijs} = 1$ if task t_{kl} of application a_k is running in time slot s on instance type it_i of cloud provider c_j, and $x_{klijs} = 0$ otherwise. Let $y_{klij} = 1 \Leftrightarrow \exists s \in \{1, \ldots, S\} : x_{klijs} = 1$.

Our proposed BIP is presented in Figure 3.1. We introduce our objective function as given in Equation 3.1 and associated constraints given in Equations 3.2 to 3.5.

The first term in the objective function represents the data traffic costs, the second one represents the computational cost over all time slots within the schedule. Constraint 3.2 guarantees that each task is scheduled on only one instance type and cloud provider, and thereby removes the possibility of a task to resuming on a different instance type or cloud after preemption. Constraint 3.3 enforces that all the individual tasks of an application finish before the application's deadline. Constraint 3.4 and 3.5 only apply to private clouds and enforce a limit on the number of CPUs and amount of memory used at the provider for each slot in the schedule.

Note that additional constraints, such as for example requiring the application to run on a number of different providers in order to limit reliance on a single provider or data locality restrictions will probably add an additional complexity to solving the BIP, and are left for future work.

3.2 Experiments

In this section, we discuss a number of experimental settings that shed a light on the performance of our optimization approach. We thereby aim to illustrate some of the complex mappings made by the optimizer, while simultaneously providing an

minimize

$$Cost = \sum_{k=1}^{A}\sum_{l=1}^{T_k}\sum_{i=1}^{I}\sum_{j=1}^{C} y_{klij} \cdot (ni_{kl} \cdot pi_j + no_{kl} \cdot po_j) + \sum_{k=1}^{A}\sum_{l=1}^{T_k}\sum_{i=1}^{I}\sum_{j=1}^{C}\sum_{s=1}^{S} (p_{ij} \cdot x_{klijs}) \tag{3.1}$$

subject to

$\forall k \in [1, A], l \in [1, T_k]:$

$$\sum_{i=1}^{I}\sum_{j=1}^{C} y_{klij} = 1 \tag{3.2}$$

$\forall k \in [1, A], l \in [1, T_k], i \in [1, I], j \in [1, C]:$

$$\sum_{s=1}^{dl_k} x_{klijs} = y_{klij} \cdot r_{kli} \tag{3.3}$$

$\forall j \in [1, C], s \in [1, S]:$

$$\sum_{k=1}^{A}\sum_{l=1}^{T_k}\sum_{i=1}^{I} cpu_i \cdot x_{klijs} \le maxcpu_j \tag{3.4}$$

$\forall j \in [1, C], s \in [1, S]:$

$$\sum_{k=1}^{A}\sum_{l=1}^{T_k}\sum_{i=1}^{I} mem_i \cdot x_{klijs} \le maxmem_j \tag{3.5}$$

Figure 3.1: Binary Integer Program for Hybrid cloud scheduling

insight in the scheduling performance, feasibility and scalability of the proposed approach.

For the implementation and evaluation of the binary integer program, we used the modeling language AMPL[89] and IBM's linear programming solver CPLEX[1], and solved the problem with multiple data inputs. All outputs were determined by averaging the result of 20 individual samples. The experiments have been performed on six 64-bit AMD Opteron systems with eight 2Ghz cores, 512KB cache size, 32 GB memory, and Ubuntu 9.04 as operating system. Solve times are expressed in the number of seconds *CPU Time* used, as reported by the solver.

The scenarios used in the experiments below are assembled synthetically in order to provide an insight in the correct functioning and performance of our approach. Information such as the public cloud provider's prices and instance types are however loosely based on present-day real-world cloud provider data.

[1] http://www.ilog.com/products/cplex/

Table 3.1: Instance types

Name	CPUs	Memory
small	1	1.7 GB
large	4	7.5 GB
xlarge	8	15 GB

Table 3.2: Public Cloud - Prices

Prov.	small	large	xlarge	NW in	NW out
A	0.085	0.34	0.68	0	0
B	0.07	0.30	N/A	0	0
C	0.10	0.40	0.70	0	0

Public cloud

In the first experiment, 50 applications, with each between 1 and 100 tasks and a deadline between 1 hour and 1 week are scheduled on 3 public cloud providers *A*, *B* and *C* using 3 instance types *small*, *large* and *xlarge*. The properties of these instance types are summarized in Table 3.1. Note that our model allows for the creation of disparate instance types for each cloud provider, thereby even taking into account the performance differences between the providers. For clarity reasons however, we assumed in these experiments that the instance types are homogeneous for all providers.

Cloud providers associate a price with the instance types they support, as shown in Table 3.2. In our setup, provider *B* is cheaper than his competitors, but only offers *small* and *large* instance types. Provider *A* and *C* offer all instance types, with *C* being more expensive across the board. For this experiment we focus on the computational costs of running the instances on the cloud infrastructure, and thus do not take any network costs into account.

An application's tasks have a runtime for each instance type available. Runtimes on the *small* instance are normally distributed with a mean μ of 24 hours and a standard deviation σ of 12 hours. The speedup factor for *large* and *xlarge* instances is $1/f$, with f uniform between 0 and 1. The runtime for a task on a faster instance type is always smaller than or equal to the runtime on a slower one, with a minimum of 1 hour. The application's deadline is always feasible, which means that the runtime for the fastest instance type is always smaller than or equal to the deadline. Incoming and outgoing traffic for an application's task is uniformly distributed between 0 and 500 megabytes. A summary of all application parameters and distributions is given in Table 3.3.

Running this experiment shows that on average, 80.9% of the tasks are scheduled on cloud provider *B* and 18.3% on provider *A*, with absolute standard deviation less than 5.4%. The most expensive provider *C* gets no jobs. In order to minimize

Table 3.3: Application Parameters

Parameter	Value
Deadline	Uniform (1 hour, 1 week)
# Tasks per app.	Uniform (1,100)
Runtime (hours)	Normal ($\mu = 24$, $\sigma = 12$)
Runtime speedup factor	Uniform (0,1)
Network traffic (in/out) (MB)	Uniform (0,500)

costs, the solver clearly uses the cheapest provider as much as possible, and will only schedule instances on the more expensive provider A if this is necessary to meet an application's deadline constraint.

This is further illustrated in Figure 3.2. Let θ be a metric for the strictness of a deadline of a task t. θ is defined as the runtime of a task on instance type *small* divided by the task's deadline. Tasks with a tight deadline have a higher value for θ, while tasks with an easy achievable deadline have a value for θ close to zero. Partitioning the tasks based on their value for θ allows us to plot the average cost per workload hour for different deadline constraints. We thereby observe a significant increase in cost for applications with tight deadlines.

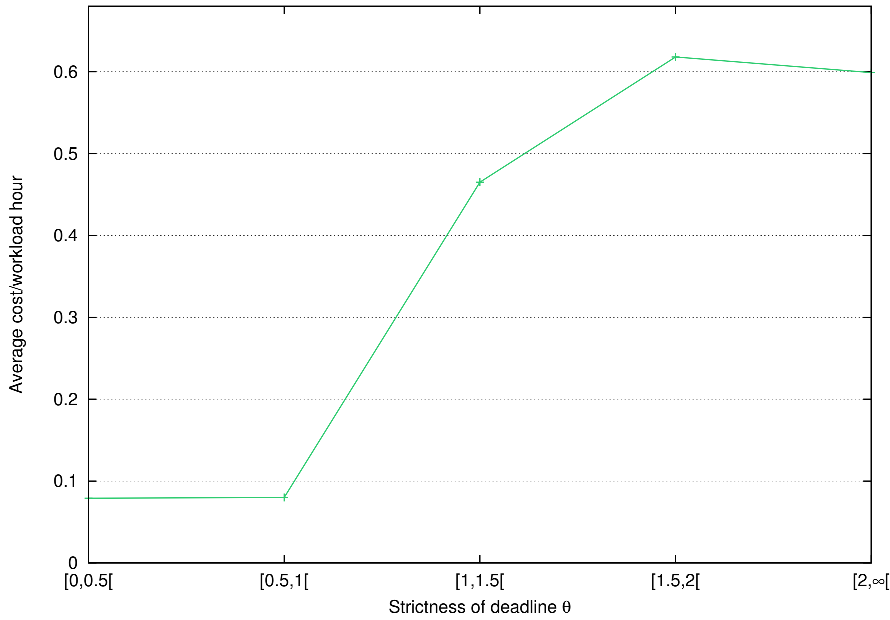


Figure 3.2: Public Cloud - Average cost per workload hour

Solving problems with an LP-solver obviously requires time. Solving one sample of this experiment using CPLEX required 30 seconds CPU time on average. Scaling

Table 3.4: Public Cloud with network costs - Prices

Prov.	small	large	xlarge	NW in	NW out
A	0.085	0.34	0.68	0.10	0.10
B	0.07	0.30	N/A	0.20	0.20
C	0.10	0.40	0.70	0	0

this experiment by increasing the number of applications while keeping all other parameters unchanged showed an almost linear increase in solving time, as illustrated in Figure 3.3.

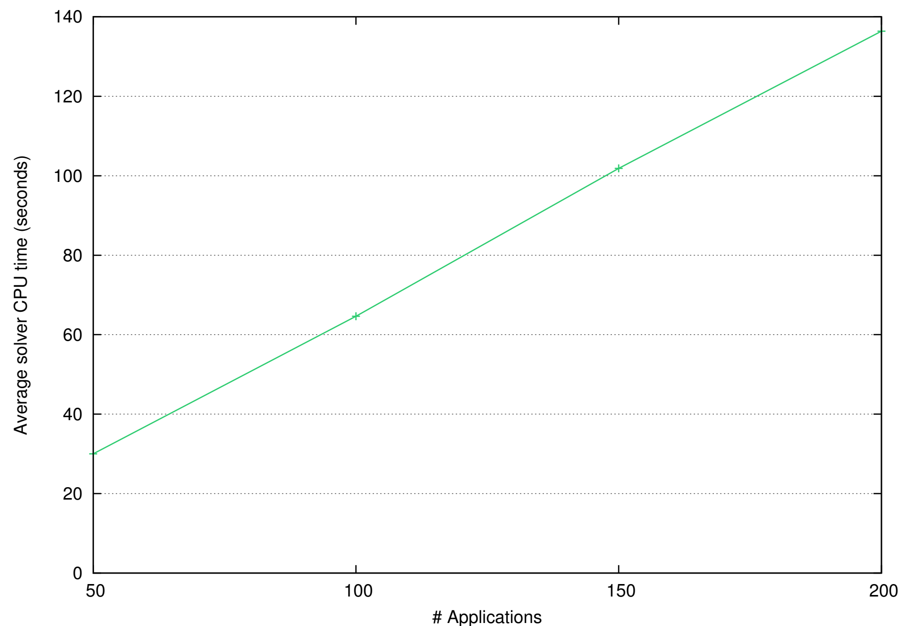


Figure 3.3: Public Cloud - Runtimes

Public cloud with network costs

In this second experiment, we add “network costs” as an extra dimension to our cloud provider model. The application parameters, cloud provider setup and computational prices are maintained from the previous experiment. Prices for network traffic are added as shown in Table 3.4. The cheapest –in terms of cost per runtime hour– provider *B* is now the most expensive one for network traffic. Provider *A*'s network prices are cheaper, and provider *C* doesn't charge at all for a task's data traffic.

Solving the network-cost-aware version of our model with 50 applications resulted in an increase of the average solve time with less than 0.2% compared to the previous variant, in which all network traffic was free.

In the previous experiment, provider *C* received no tasks as expected, because it was the most expensive for all available instance types. Because of the low network prices of provider *C*, we now expect that –for network-intensive applications– it will sometimes be cheaper to run on this provider. On average 74.9% of the tasks are scheduled on cloud provider *B*, 14.4% on provider *A* and 11.0% on provider *C*. If we define the ratio for CPU vs. network intensiveness of a task t as the runtime of t on instance type *small* divided by the total network traffic, we can categorize our tasks as *network intensive*, *neutral* or *CPU intensive*. The proportional shares of the providers in each task segment are shown in Figure 3.4. It thereby becomes even more clear that it is only for network intensive tasks advantageous to be scheduled on provider *C*.

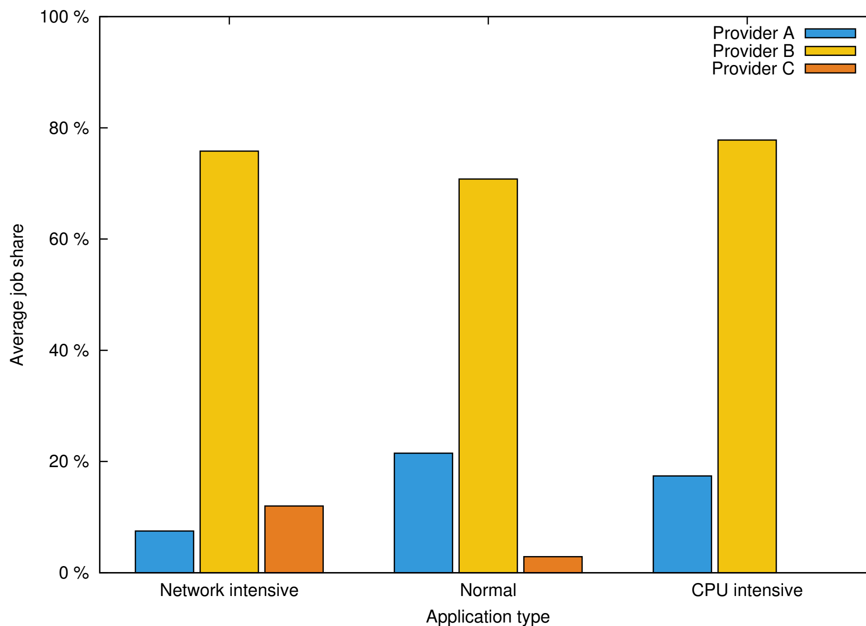


Figure 3.4: Public Cloud with network costs - Average job share

Hybrid cloud setting

The CPE not only supports scheduling on public clouds, but also enables us to model a private cloud. This can be done by adding a provider with a limited number of CPUs or amount of memory available. Because we want our solver to use the private infrastructure as much as possible, and only outsource tasks to the cloud if it is insurmountable for the deadline of the task to be attained, we associate no costs with the execution of a task on the private cloud infrastructure.

In this third experiment, we assemble a hybrid setup in which one private cloud with 512 CPUs and one public cloud provider is available. Instance types and appli-

Table 3.5: Hybrid Cloud - Prices

Prov.	small	large	xlarge	NW in	NW out
Public	0.085	0.34	0.68	0	0
Private	0	0	N/A	0	0

cation parameters are equal to the previous experiments, and are shown in Tables 3.1 and 3.3. The private cloud can only schedule *small* and *large* instances, the public cloud provider is identical to provider *A* in the previous experiments. Network costs are not considered. The prices are described in Table 3.5.

In Figure 3.5, we illustrate the cost-optimal schedule with a sample run that contains 50 applications that are submitted at the start of a week to the CPE. We plot the costs and utilization of the private cluster in a time span of a week. We observe that the public cloud is only used in the beginning, in order to finish all tasks within deadline.

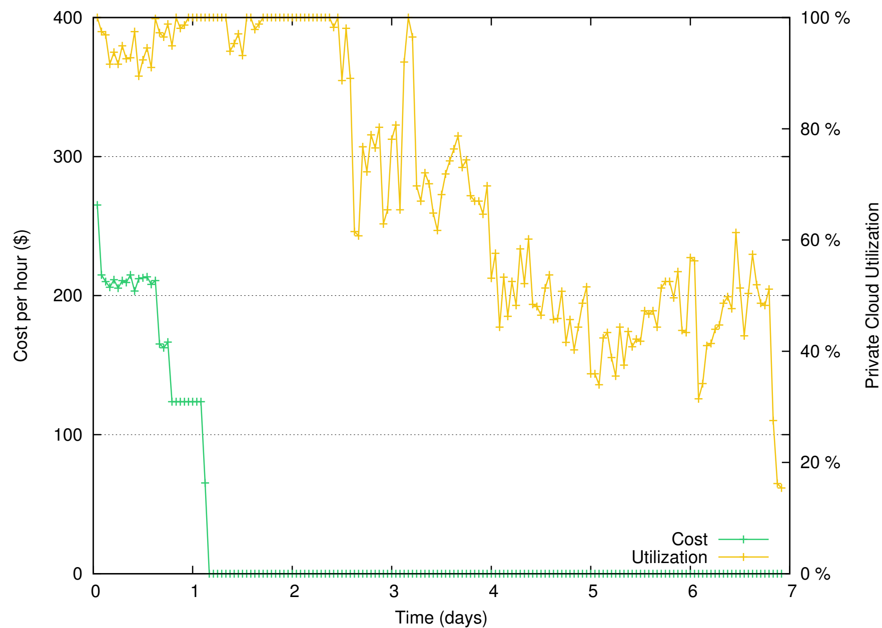


Figure 3.5: Hybrid Cloud

By creating cost-optimal schedules for a number of applications ranging from 10 to 50 with 20 samples each, we experienced solve times ranging from a few seconds to multiple hours and even days. The runtimes of these samples are grouped and shown in Figure 3.6. It shows that, for an increasing number of applications, the number of samples with a high solve time increases significantly. We have also observed a very large variation in the solver runtime for this experiment. The addition

of infrastructure with a fixed capacity has major implications on the complexity of the linear program. Indeed, the solver now needs to tackle an NP complete problem that involves scheduling a set of deadline constrained jobs with non-identical runtimes on a fixed capacity infrastructure with parallel machines [66].

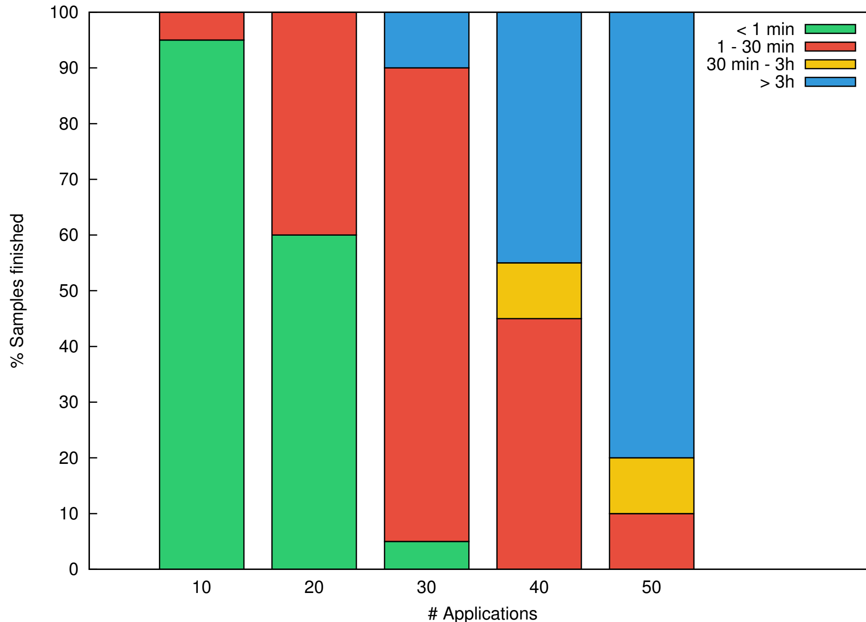


Figure 3.6: Hybrid Cloud - Runtimes

The high variances on these solve times undermine the feasibility of using a linear programming approach in our CPE model. We observed cases in which the solve time exceeded 5 days, without obtaining an optimal solution. It is however possible for a LP solver to report intermediate solutions, thereby creating opportunities to weaken the high variances observed above. The solver can be configured with an absolute or relative tolerance on the gap between the current best integer objective and the objective of the best node remaining in the search space. When the difference between these values drops below the value of this tolerance parameter, the mixed integer optimization is stopped. For one sample with a solve time of more than 27 hours, for example, the intermediate solution found after 30 minutes was less than $0.5 more expensive than the optimal solution found 27 hours later. Next to using the solver-specific facilities to cope with this problem, it should also be possible to develop more feasible heuristics that approximate the optimal solution found by the solver. The development of these heuristics will be the main focus in the next chapters.

3.3 Conclusion

In the context of hybrid cloud scheduling, we have outlined a software architecture model for the HICCAM project in order to highlight and emphasize the purpose of

the *Optimization Engine* component. This component was then described in detail, after which we presented an experimental evaluation of the proposed scheduling solution through a discussion of three cases of increasing complexity. When scheduling applications in the public cloud, our scheduling approach seems to perform very well both in terms of cost minimization, feasibility and scalability. As such, the use of cost-optimization techniques in the cloud scheduling decision process through the proposed binary integer program can support users in their decision making process and allow for (partial) automatization of this process. In addition, the large-scale employment of the proposed technique could in the longer term be an enabler for increasing price competition in the IaaS market.

The addition of network costs to our model barely influences the solver's performance but shows that, with the current relations between runtime and network traffic costs, the determining cost factor in all but very network-intensive applications is clearly the runtime. In the hybrid setting, the solver's performance decreases drastically. These issues are tackled by developing custom heuristics in the subsequent chapters.

CHAPTER 4

Heuristic for Scheduling Deadline Constrained Workloads on Hybrid Clouds

This chapter is published as "Cost-Efficient Scheduling Heuristics for Deadline Constrained Workloads on Hybrid Clouds", R. Van den Bossche, K. Vanmechelen and J. Broeckhove in Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science [90].

Abstract

Cloud computing offerings are maturing steadily, and their use has found acceptance in both industry and research. Cloud servers are used more and more instead of, or in addition to, local compute and storage infrastructure. Deciding which workloads to outsource to what cloud provider in such a setting, however, is far from trivial. This decision should maximize the utilization of the internal infrastructure and minimize the cost of running the outsourced tasks in the cloud, while taking into account the applications' quality of service constraints. Such decisions are generally hard to take by hand, because there are many cost factors, pricing models and cloud provider offerings to consider. In this work, we tackle this problem by proposing a set of heuristics to cost-efficiently schedule deadline-constrained computational applications on both public cloud providers and private infrastructure. Our heuristics take into account both computational and data transfer costs as well as estimated data transfer times. We evaluate to which extent the different cost factors and workload characteristics influence the cost savings realized by the heuristics and analyze the sensitivity of our results to the accuracy of task runtime estimates.

4.1 Experimental setup

Due to the lack of sufficient real-world data, especially with regard to application data set sizes and deadline constraints, we have to resort to a partially synthetic model for generating application instances for the experiments in Section 4.2.

Every scheduling experiment has a duration of one week, in which new applications arrive every second following a Poisson distribution with $\lambda = 0.002$. This results in an average inter-arrival time of 2000 seconds, or about 1200 applications in a week. With these parameters, we aim to generate enough load to tax the private cloud beyond its saturation point so that it becomes advantageous to use a hybrid cloud setup.

The number of tasks per application is uniformly distributed between 1 and 100. Task runtimes within one application are typically not independent of each other. Therefore, we assign an application a *base runtime*. Following [91], the application's base runtime is modeled as a Weibull distribution. The distribution's parameters are derived from the Parallel Workloads Archive's *SDSC IBM SP2* workload trace[1]. The runtime of each individual task is then drawn from a normal distribution, with the application's base runtime as the mean μ and a relative *task runtime standard deviation* σ. A task runtime standard deviation of 0% will lead to identical task runtimes, equal to the application's base runtime, while for example $\sigma = 100\%$ results in high runtime variations. Unless mentioned otherwise, the relative task runtime standard deviation was fixed at 50%.

Tasks can typically run on more than one instance type. Running a task on an instance type with for example 1 CPU and 1.7 GB memory will probably –but not always– be slower than running the same task on an instance type with 4 CPUs and 7.5 GB memory. Modeling speedup of parallel tasks on multiple processors without thorough knowledge on the operation of the application is a delicate and complex task. We model the speedup after Amdahl's law, which is used to find the maximum expected improvement of a system when only a part of it can be improved. We assume that a task has a sequential fraction with length c_1 and a parallelizable fraction with length c_2. The execution time of the task then equates to $c_1 + c_2/n$, where n is the number of processors available on the instance type. We describe the size of the parallel component as a percentage of the total runtime. We use a uniform distribution between 0% and 100%, covering both highly sequential and highly parallelizable applications. Note that in this workload model, we do not factor in the speed of the individual cores of the instance type to determine the runtime, and assume it to be homogeneous across the different instance types[2]. Next to the number of CPU cores, the amount of memory available on an instance type IT_{mem} can also influence the runtime of a task. Applications get assigned a *minimum memory requirement* App_{MemReq}. The runtime for tasks run on instance types that meet the application's memory requirement remains unaltered. If $App_{MemReq} > IT_{mem}$, a task suffers a slowdown equal to IT_{mem}/App_{MemReq}. Although this linear slowdown is optimistic and might be worse in practice, it satisfies our need to differentiate between instance types. The minimum memory requirement is normally distributed with an average of 1.5 GB and a standard deviation of 0.5 GB.

Taking into account the application's base runtime, task runtime variance, parallel component size and memory requirement, the runtimes for each of the application's tasks on each of the available instance types can be determined by the application's owner before submission to the scheduler.

[1] http://www.cs.huji.ac.il/labs/parallel/workload/

[2] Generalizing the model to non-homogeneous CPU speeds however does not impact our scheduling algorithms and could be achieved by adding an additional scaling factor for the task's runtime on a given instance type.

In our workload model, applications have a deadline before which all tasks in the application have to be completed. This deadline will determine to a large extent on which instance type the tasks will run. Without deadlines, all tasks would be executed on the slowest and cheapest instance type. We model the application deadline as a factor of its runtime on the *fastest instance type* available. A small deadline factor results in a tight deadline, a deadline factor < 1 will always result in unfeasible deadlines. To cover both tight and loose deadlines in our workload, the deadline factor is uniformly distributed between 3 and 20.

Finally, an application is also linked to a *data set* that is used by each of the application's tasks. The data set size of an application is normally distributed with averages ranging from 0 GB for purely computational tasks to 2500 GB for data-intensive applications with a very large data set. The relative standard deviation is fixed to 50%.

A overview of the distributions used in this paper is given in Table 4.1.

Characteristic	Distribution
App. arrival rate (seconds)	Poisson(0.002)
Tasks per application	Uniform(1,100)
Base runtime (hours)	Weibull($\lambda = 1879$, $\beta = 0.426$)
Task runtime standard deviation (%)	50%
App. parallel fraction (%)	Uniform(0, 100)
App. memory requirement (GB)	Normal($\mu = 1.5$, $\sigma = 0.5$)
App. deadline factor	Uniform(3,20)
App. data set size (GB)	Normal(varying μ, $\sigma = 50\%$)

Table 4.1: Workload model distributions

4.2 Scheduling Components

We introduce a hybrid scheduling solution suitable for our problem domain. The proposed solution consists of three loosely coupled components:

- The *public cloud scheduler* decides for an incoming application –with given runtimes for each of the available instance types and with a given data set size– on which of the public cloud providers to execute that application. It takes into account the cost for execution and transferring data, as well as differences in data transfer speeds between the cloud providers.
- The *private cloud scheduler* schedules an incoming application on a private cloud infrastructure, taking into account the limited amount of resources available.
- The *hybrid cloud scheduler* decides whether an incoming application can be scheduled on the organization's private infrastructure or has to be offloaded –at a cost– to the public cloud. It may consider a variety of factors, ranging

from private cloud utilization or queue wait time to the public cost of the application or budget constraints.

Each of the following subsections explain how our scheduling components work, which policies they use and demonstrate their operation using one or more scenarios. All simulations were performed with a simple Java based discrete time simulator, and executed on a system with two Intel Xeon E5620 processors with 16 GB memory. The simulation of running one hybrid scenario in which 1267 applications arrive over the course of one week took on average less than 12 minutes, with outliers for bigger private cloud sizes to 35 minutes, which is still less than two seconds per application scheduled.

Public Cloud Scheduling

The public cloud scheduling component schedules each incoming application on a public cloud provider. We therefore introduce a cost function $Cost$ (Equation 4.1) for an application a, which will consist of the computational costs and the data costs of running the application on a public cloud provider p. We define the set of tasks of application a as T_a. A provider p supports the set of instance types IT_p. The cost of running a task t on provider p with instance type it is defined as $C_{task}(t,p,it)$, and is elaborated in Equation 4.2. The time to deadline of application a is DL_a. The runtime of task t on instance type it is given by $RT_t^{it,p}$, which is rounded up to the nearest discrete time unit (i.e. 1 hour) of provider p's billing interval. The cost of running an instance of type it on provider p for one time unit is defined as C_{it}^{p}. The scheduler calculates the cost for running the application on each available cloud provider p, and schedules all tasks of an application on the cheapest cloud provider.

The first cost factor in Equation 4.1 is the computational cost C_{comp} (Equation 4.3) to run each task of application a within deadline constraints on the cheapest instance type.

$$Cost = C_{comp} + C_{data} \tag{4.1}$$

$$C_{task}(t,p,it) = \begin{cases} RT_t^{it,p} \cdot C_{it}^{p} & : \quad RT_t^{it,p} \leq DL_a \\ \infty & : \quad RT_t^{it,p} > DL_a \\ \infty & : \quad it \notin IT_p \end{cases} \tag{4.2}$$

$$C_{comp} = \sum_{t}^{T_a} \min_{it}^{IT_p} \left(C_{task}(t,p,it) \right) \tag{4.3}$$

Next to the computational costs, the cost function in Equation 4.1 also takes the data transfer costs C_{data} (cfr. Equation 4.4) for the application's data set into account. The data set size of application a is defined as D_a (in GB), and the inbound and outbound network traffic prices per GB of provider p are denoted by NW_p^{in} and NW_p^{out} respectively. Further on, we only focus on inbound traffic and assume that D_a has to be fully transferred to the cloud provider on which a is running.

$$C_{data} = D_a \cdot NW_p^{in} \tag{4.4}$$

We introduce a public cloud landscape in which users run applications on three providers, modeled exactly after the on-demand offerings of Amazon's EC2[3], Rackspace and GoGrid, with all their prices and instance type configurations d.d. March 1, 2011. The application's average data set size varies from 0 GB to 2.5 TB, with a relative standard deviation of 50%. We compare the difference between our cost-efficient scheduler with (*CE-Data*) and without (*CE-NoData*) taking into account the data transfer cost. The results are shown relative to a *Random* policy, in which for each arriving application a cloud provider is picked randomly. Figure 4.1 illustrates the impact of taking data costs into account when scheduling applications on the public cloud, and shows the influence of the data set size of an application on the differences between both policies.

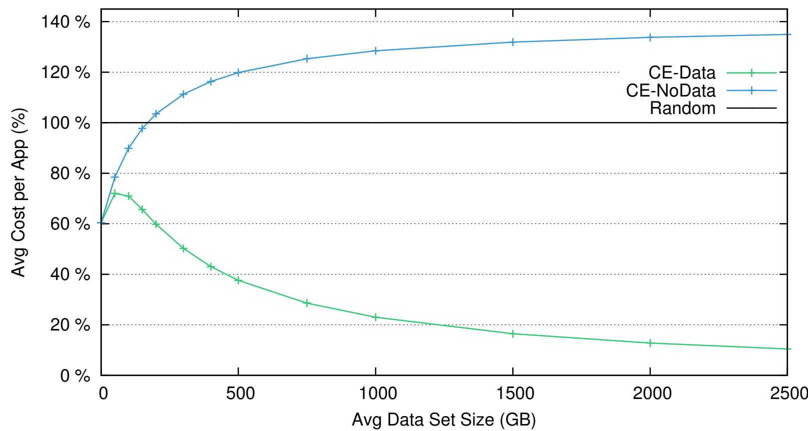


Figure 4.1: Impact of data costs.

Next to the expected cost reduction of our cost-efficient scheduler compared to a random policy, we also observe a significant cost reduction of the scheduling policy that takes data costs into account compared to the same policy that only takes note of the computational costs. The second term in cost equation 4.1 appears to be at least as important as the first.

The application's data set size does not only affect the data transfer costs to a cloud provider, but may also influence the runtimes of individual tasks. Before a task can run on a cloud provider, the application's data set should be transferred completely. If an application has a large data set, the additional time necessary to transfer the data to the cloud provider may require tasks to be scheduled on a faster instance type in order to meet the application's deadline. It is therefore important to also take the data transfer times from the consumer to the cloud provider into consideration. Some cloud providers, such as Amazon, provide multiple regions in different continents to satisfy the demand for *data locality*.

Differences in data transfer speed between the different regions can be significant. As an example, we measured the average bandwidth available from the University of Antwerp in Belgium to three of the Amazon EC2 regions using the

[3] For Amazon EC2, prices are taken from the US-east region, and their "cluster" and "micro"-instances are disregarded.

iperf[4] bandwidth measurement tool. Every half hour over the course of one day, we measured the available bandwidth to the `us-east`, `us-west` and `eu-west` regions. The averages of these measurements are shown in Table 4.2. In our case, transferring data to `eu-west` is more than twice as fast as transferring data to the cheaper `us-east` region.

Table 4.2: Available bandwidth from UA to EC2

Zone	Avg (MB/s)	Time/GB (s)	Stdev
`us-east`	15.04	68	11.91%
`us-west`	10.8	95	8.03%
`eu-west`	39.15	26	15.85%

In order for the public cloud scheduler to take data locality into account when scheduling an application, we update C_{task} in Equation 4.2 so that only these instance types are available for which the sum of the data transfer time and the task runtime is smaller than the time to deadline DL_a. Figure 4.2 demonstrates this feature. In this experiment, the Rackspace and GoGrid providers are omitted, and all applications are scheduled on one of the previously benchmarked regions of Amazon EC2. The application's average data set size ranges from 0 GB to 2.5 TB, and we use the *CE-Data* policy to calculate computational and data transfer costs. In total, 1297 applications are scheduled. The figure shows the amount of applications scheduled in each of the regions. We observe that applications without a data set are all scheduled in the `us-east` region, as it provides the cheapest compute capacity. The larger the data set, the more applications are executed in the nearby `eu-west` region in order to succesfully meet the applications' deadline constraints. The percentage of applications not meeting their deadline increases quickly as average data set sizes exceed 400 GB. This can be explained by the fact that in our workload model, the application's deadline is determined as a factor of the fastest possible task runtime, without considering data transfer times. Increasing the average data set size in an experiment consequently results in tighter deadlines and can lead to the sum of the data transfer time and the task runtime on the fastest instance type to exceed the deadline.

Private Cloud Scheduling

Our private infrastructure is modeled as a simple queuing system in which incoming requests for an application's tasks are handled in a first-come first-served manner without backfilling. The private infrastructure obviously does not have an "infinite" capacity. The number of tasks running concurrently on our private cloud is thus limited: the sum of the number of CPUs of all running instances may not exceed the total number of CPUs available. Overhead for virtualization as well as unusable leftovers due to the partitioning of physical servers into multiple instances are not considered. The private cloud supports four instance types, with respectively 4, 8, 16

[4] http://iperf.sourceforge.net/

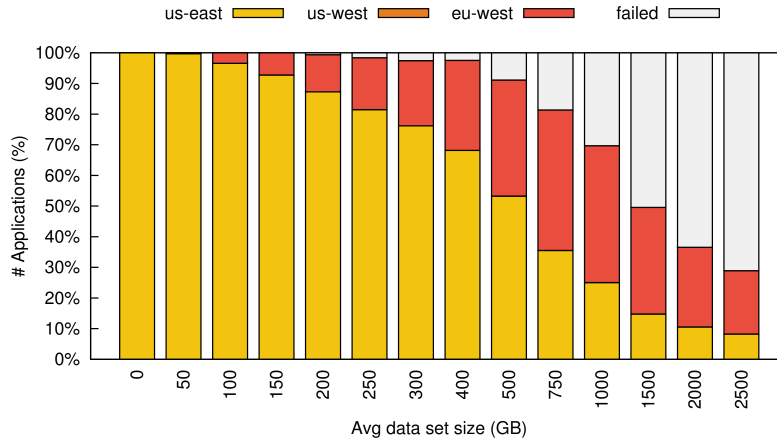


Figure 4.2: Impact of data locality.

and 26 CPU's and 4 GB, 8 GB, 16 GB and 26 GB memory. A task is scheduled on the instance type on which it induces the smallest *load*, which is defined as the runtime multiplied by the amount of CPUs of the instance type. The instance type for a task is selected at the moment the task arrives at the private scheduling component, and won't change afterwards.

This scheduling strategy does not provide an optimal schedule with respect to maximal adherence to application deadlines, cluster utilization or generated user value [2]. It does however provide a simple scheduling heuristic that allows for the exact calculation of the *queue wait time* of a task. This permits a hybrid scheduler only to schedule applications on the private cloud infrastructure that with certainty finish before deadline. This enables us to evaluate the consequences of the hybrid cloud scheduling decisions and make abstraction of the private scheduler's performance. The approach of using other metrics such as the private cluster's utilization, the number of applications in the queue or using an integrated hybrid advance reservation system are left for future work.

From the perspective of the hybrid scheduler, running an application on the private cloud is free of cost, and the applications' data sets are available free of charge and without a transfer delay. This will cause a cost-efficient hybrid scheduler to maximize the local infrastructure's utilization, and offload applications only when absolutely necessary. In an organization with an internal billing mechanism, it is however possible that users will be charged for their use of the private compute infrastructure. The development of a fair internal billing system for organizations with a hybrid cloud setup that takes into account public and private operational and capital expenses is beyond the scope of this contribution.

Hybrid Cloud Scheduling

In a hybrid setup, additional logic is required to decide which applications to run on the organization's own computing infrastructure and which ones to offload to a public cloud provider. After making this decision, the hybrid scheduler hands over

control to the public or private scheduling component. In all of the experiments in this subsection, we use a public scheduling component with the *CE-Data* policy.

The hybrid scheduling component has to decide on the allocation of application workloads to the public and private cloud components. It should provide an efficient solution with respect to the total cost for running all applications, while minimizing the number of applications missing their deadlines due to queue congestion or high data transfer times. In the remainder of this section, we propose a hybrid scheduling mechanism that takes into account the *potential cost* of an application when scheduled on a public cloud provider. We compare this *Cost Oriented* approach with a first-come first-served *Private First* scheduler.

Such a simple *Private First* scheduler adds all incoming applications to the queue of the private cloud, unless the *queue wait time* for that application indicates that meeting the application's deadline is impossible by the sole use of the private cloud. In that case, the application is scheduled on the cheapest public cloud provider. Because the runtimes of all tasks in the queue are known, calculating the expected wait time in the queue for a set of tasks is trivial.

This FCFS scheduling methodology can result in a cost-inefficient schedule when, for example, application B with a cost of $100 on the public cloud arrives shortly after application A with a potential cost of $50. It is then possible that application B has to be executed externally to meet its deadline constraints, resulting in an additional cost for the system of $50. Our *Cost Oriented* approach tries to avoid these errors by calculating on arrival the cost of running the application on the public cloud within deadline, and giving priority to the most expensive applications on the private cloud. This *potential cost* can be calculated using the public cloud scheduling algorithms described in Subsection 4.2. In this approach, applications can't be scheduled on arrival because that would again result in handling them in a first-come first-serve manner. On the other hand, delaying the scheduling decision on the hybrid scheduling level is delicate because it may result in a higher cost due to the need for more expensive instance types to complete the application within deadline. The technique of waiting until after the point where the application cost increases (and there is need for a more expensive instance type) could theoretically still result in cost reductions. In practice this turned out to happen only by exception and never outweighed the additional costs brought about by badly made decisions. The hybrid scheduler therefore uses the following strategy :

1. Determine max_{cost}: the time up to which the application's cost of execution remains constant and no instance type switch is required.

2. Determine max_{DL}: the time up to which adhering to the application's deadline remains feasible.

3. Delay the scheduling decision up to $min(max_{cost}, max_{DL})$.

Before scheduling, applications are sorted on their *potential cost*, normalized to the application's *load*[5] on the private cloud. This way, the most expensive applications are treated first. The full scheduling logic for which the pseudocode is shown in Algorithm 1, is executed every *N* seconds. A small value of *N* allows the scheduler

[5] The load of a is defined as the sum over all tasks of the runtime multiplied by the amount of CPUs of the instance type.

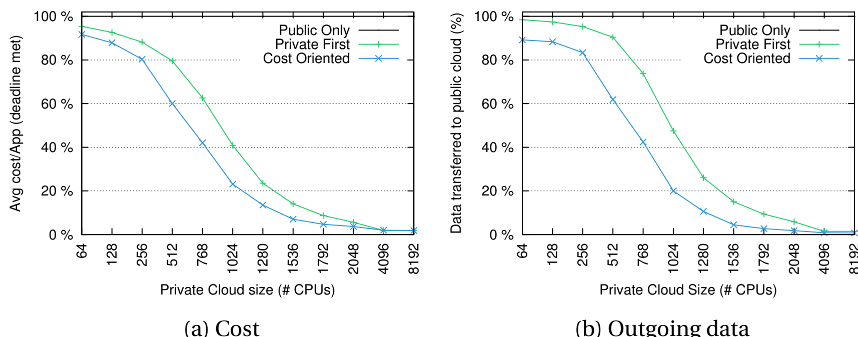
to fine-tune every scheduling decision with a high granularity, while an increased value of N may involve a smaller computational overhead. We found a value for N corresponding to 10 minutes to be suitable in our simulations.

```
Input: A = {Unscheduled applications}
SortMostExpensiveFirst(A)
foreach Application a ∈ A do
    if EstimatedQueueWaitTime(a)
      + EstimatedPrivateRuntime(a) > DL_a then
        // Deadline cannot be met on private
        ScheduleOnPublic(a);
    else if EstimatedQueueWaitTime(a)
      + EstimatedPrivateRuntime(a) + N > DL_a then
        // Deadline will become unfeasible
        ScheduleOnPrivate(a);
    else if TimeToPriceIncrease(a) - N < 0 then
        // Application will become more expensive
        ScheduleOnPrivate(a);
    else
        // Let a wait until next round
    end
end
```

Algorithm 1: *Cost Oriented Scheduling* Pseudocode

We demonstrate the operation of the hybrid scheduling components by adding a private cloud to the experimental setting with the different Amazon EC2 regions used in Section 4.2. Computational costs, data transfer costs as well as data transfer times are taken into account. The average data set size in this experiment is 50 GB with a relative standard deviation of 50%. We compare the *Cost Oriented* scheduling policy to the *Private First* policy as well as a *Public Only* scheduler, which schedules all applications on the public cloud, and a *Private Only* scheduler, which adds all applications to the private queue. In Figure 4.3a, the average cost per application executed within deadline is shown for the *Cost Oriented* and *Private First* policies, relative to *Public Only* policy. It is to be expected that adding a "free" private cluster results in significant cost reductions, but we also observe that our *Cost Oriented* logic is up to 50% less expensive than the *Private First* policy. This cost reduction can be achieved by sending significantly less data-intensive applications to the public cloud. This is illustrated in Figure 4.3b, which displays the total amount of data transferred to the public cloud, relative to the total amount of data of all applications.

In all algorithms used in this paper, we assume that the exact runtimes are known and provided by the consumer a priori. Our *Cost Oriented* algorithm, and to a lesser extent also the *Private First* algorithm, depend on the accuracy of the provided runtimes as they both use the exact queue wait time, and the *Cost Oriented* policy attempts to delay the scheduling decision as much as possible up until the application's deadline. Because user provided runtimes will never be exact, we evaluate the degree of sensitivity of the algorithms to runtime estimation errors. Therefore, scheduling algorithms now base all their decisions on *estimated runtimes* instead of exact runtimes. The average error on these runtimes ranges from 0% to 50%. In this experiment, the three Amazon EC2 regions are used as public cloud together with a private cluster of 512 CPUs. Applications have an average data set size of 50 GB. The impact of these runtime estimation errors on the number of



(a) Cost

(b) Outgoing data

Figure 4.3: Impact of hybrid scheduling policies on *Cost* and *Outgoing data*.

applications meeting their deadlines and on the average cost per application is illustrated in Figure 4.4a and Figure 4.5a respectively. We can observe a dramatic decrease in the number of deadlines met for the *Cost Oriented* approach. This is accompanied by an increase in cost per application, which is the result of the high cost that has to be paid for partially executed applications with unexpected failed deadlines.

These results constitute an impediment to the use of the proposed algorithms in a setting wherein accurate runtime estimates are difficult to attain. A countermeasure to reduce the number of failed deadlines can be to increase the conservativeness of the scheduler by deliberately overestimating the given runtimes with a certain percentage. In the following three experiments, we add an *Overestimation factor* of 10%, 50% and 100% to the estimated runtime of every task. All other experiment settings remain unchanged. The results with regard to the number of deadlines met and the average cost are shown in Figures 4.4 and 4.5. Overestimating the user provided runtimes also slightly influences the cost for scheduling an application in a public cloud environment, as the public scheduling component may have to pick a more expensive instance type. Compared to Figure 4.5a, the absolute cost for *Public Only* in Figures 4.5b–4.5d increases less than 2% for an *overestimation factor* of 10%, less than 9.1% for an *overestimation factor* of 50% and less than 12.2% for an *overestimation factor* of 100%.

It is clear that the simple technique of overestimating the user-provided runtimes can reduce the sensitivity of our algorithms for runtime estimation errors to an acceptable level, with only a limited increase in cost. The development of appropriate algorithms to further address this issue will be the focus of Chapter 5.

4.3 Conclusion

We address the challenge of cost-efficiently scheduling deadline constrained batch type applications on hybrid clouds. In Chapter 3, we presented a linear programming formulation of the optimization problem for a static set of applications. In this contribution, the problem is tackled by developing heuristics for cost-efficient scheduling that are able to operate on larger-scale problems. The proposed public and hybrid scheduling algorithms aim at providing an efficient solution with respect

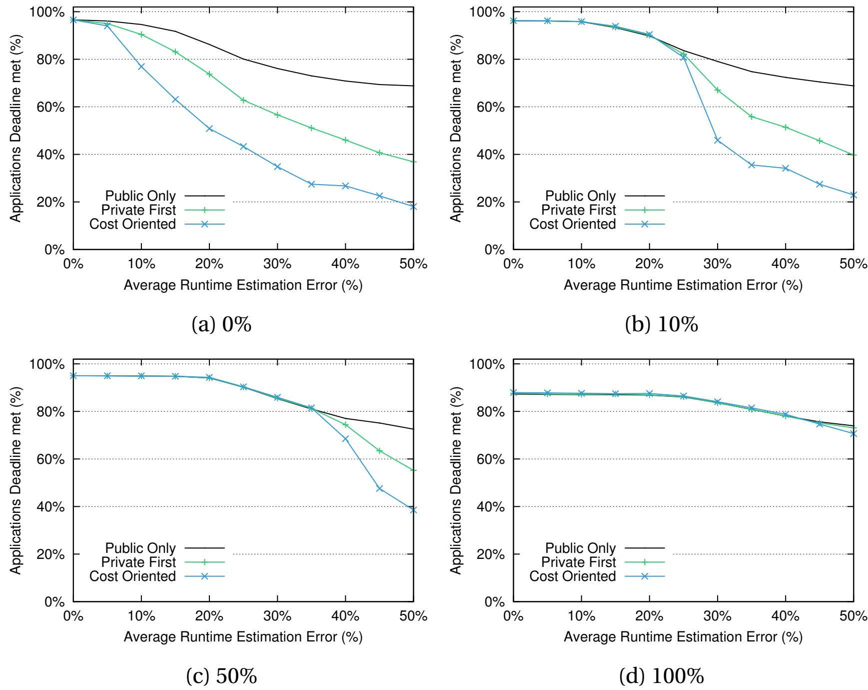
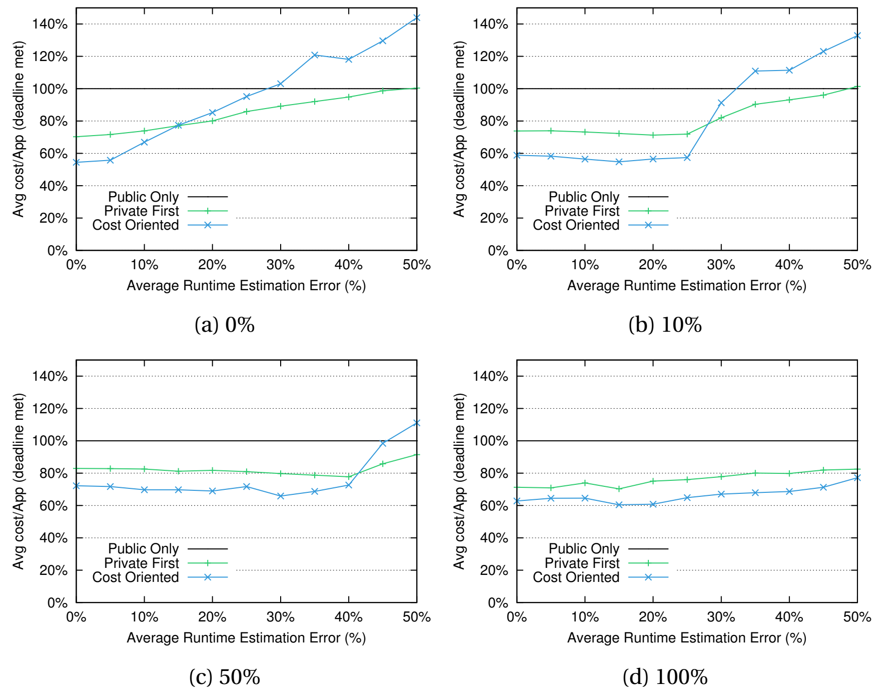


Figure 4.4: Impact of overestimation of *runtime estimation errors* on *Application deadlines met.*

to the cost for running as much applications as possible within deadline constraints. The public scheduling component thereby takes into account data transfer costs and data transfer times. A hybrid scheduling mechanism is presented to take into account the *potential cost* of an application when scheduled on a public cloud provider. We demonstrate the results of all scheduling heuristics on workloads with varying characteristics. Results show that a cost-oriented approach pays off with regard to the number of deadlines met and that there is potential for a significant cost reduction, but that the approach is sensitive to errors in the user provided runtime estimates for tasks. We demonstrated the possibility to increase the conservativeness of the scheduler with respect to these estimates in order to deal with this issue without undermining the foundations of our cost-efficient scheduling algorithm. Future work in line with this chapter consists of partially redesigning the different components of our proposed solution to be aware of and more resistant to runtime estimation errors, which is discussed in Chapter 5, as well as incorporating support for other public pricing models such as Amazon's *reserved* instances and *spot* markets.



(a) 0%

(b) 10%

(c) 50%

(d) 100%

Figure 4.5: Impact of overestimation of *runtime estimation errors* on *Average cost per application.*

CHAPTER 5

Queue-based Scheduling Heuristics for Hybrid Clouds

This chapter is published as "Online Cost-Efficient Scheduling of Deadline-Constrained Workloads on Hybrid Clouds", R. Van den Bossche, K. Vanmechelen and J. Broeckhove in Future Generation Computer Systems 29 (2013) [92].

Abstract

Cloud computing has found broad acceptance in both industry and research, with public cloud offerings now often used in conjunction with privately owned infrastructure. Technical aspects such as the impact of network latency, bandwidth constraints, data confidentiality and security, as well as economic aspects such as sunk costs and price uncertainty are key drivers towards the adoption of such a *hybrid cloud* model. The use of hybrid clouds introduces the need to determine which workloads are to be outsourced, and to what cloud provider. These decisions should minimize the cost of running a partition of the total workload on one or multiple public cloud providers while taking into account application requirements such as deadline constraints and data requirements. The variety of cost factors, pricing models and cloud provider offerings to consider, further calls for an automated scheduling approach in hybrid clouds. In this chapter, we tackle this problem by proposing a set of algorithms to cost-efficiently schedule deadline-constrained bag-of-tasks applications on both public cloud providers and private infrastructure. Our algorithms take into account both computational and data transfer costs as well as network bandwidth constraints. We evaluate their performance in a realistic setting with respect to cost savings, deadlines met and computational efficiency, and investigate the impact of errors in runtime estimates on these performance metrics.

5.1 Algorithm Design

In Chapter 4, initial steps were taken to tackle this issue by developing custom algorithms for cost-efficient scheduling. Our results showed that a cost-oriented

approach pays off with regard to the number of deadlines met and that there is potential for significant cost reductions. There was however still room for improvement in terms of the scheduling algorithm's sensitivity to estimation errors in the user provided runtimes and the efficiency of the private cloud scheduling component. In order to deal with these issues, we introduce a new hybrid scheduling approach for our problem domain. The proposed solution consists of two loosely coupled components:

- The *public cloud scheduler* decides for an incoming application –with given task runtimes for each of the available instance types and with a given data set size– on which of the public cloud providers to execute that application. It takes into account the cost for execution and transferring the data, as well as differences in data transfer speeds between the cloud providers.
- The *hybrid cloud scheduler* decides whether an incoming application can be scheduled on the organization's private infrastructure or has to be offloaded –at a cost– to the public cloud, depending on the application's deadline and potential cost.

Public Cloud Scheduling

We introduce the $Cost$ function (Equation 5.1) for an application a, that consists of the computational costs and the data-related costs of running a on a public cloud provider p. We define the set of tasks of a as T_a. A provider p supports the set of instance types IT_p. The cost of running a task t on provider p with instance type it is defined as $C_{task}(t,p,it)$, and is elaborated in Equation 5.2. The time to deadline of a is DL_a. The runtime of a task t on instance type it is given by $RT_t^{it,p}$, which is rounded up for cost calculations to the nearest discrete time unit (i.e. 1 hour) of p's billing interval. The cost of running an instance of type it on p for one time unit is defined as C_{it}^p. The scheduler calculates the cost for running the application on each available cloud provider, and schedules all tasks of an application on the cheapest provider.

The first cost factor in Equation 5.1 is the computational cost C_{comp} (Equation 5.3) to run each task of a within deadline constraints on the cheapest instance type.

$$Cost = C_{comp} + C_{data} \tag{5.1}$$

$$C_{task}(t,p,it) = \begin{cases} RT_t^{it,p} \cdot C_{it}^p & : \quad RT_t^{it,p} \leq DL_a \\ \infty & : \quad RT_t^{it,p} > DL_a \\ \infty & : \quad it \notin IT_p \end{cases} \tag{5.2}$$

$$C_{comp} = \sum_{t}^{T_a} \min_{it}^{IT_p} \left(C_{task}(t,p,it) \right) \tag{5.3}$$

The data transfer costs in Equation 5.1 are represented by C_{data} (cfr. Equation 5.4). The data set size of application a is defined as D_a (in GB), and the inbound and outbound network traffic prices per GB of provider p are denoted by NW_p^{in} and NW_p^{out} respectively.

$$C_{data} = D_a \cdot NW_p^{in} \tag{5.4}$$

Note that the application's data set size does not only affect the data transfer costs to a cloud provider, but may also influence the runtimes of individual tasks. For applications with a large data set, the additional time required to transfer the data to the provider may demand tasks to be scheduled on a faster instance type to meet the application's deadline.

In order for the public cloud scheduler to take data locality into account, we update C_{task} in Equation 5.2 so that only instance types are considered for which the sum of the application's data transfer time and the task runtime is smaller than the time to deadline DL_a. We define the transfer speed from the organization to a cloud provider p as TS_p. This results in Equation 5.5.

$$C_{task}(t,p,it) = \begin{cases} RT_t^{it,p} \cdot C_{it}^{p} & : \quad RT_t^{it,p} + \dfrac{D_a}{TS_p} \leq DL_a \\ \infty & : \quad RT_t^{it,p} + \dfrac{D_a}{TS_p} > DL_a \\ \infty & : \quad it \notin IT_p \end{cases} \tag{5.5}$$

Finding the cheapest cloud provider for an application a involves the calculation of the $Cost$ function from Equation 5.1 for each of the public cloud providers. For the sake of completeness, the algorithm is shown in Algorithm 2.

Input: Application a
Result: Cheapest cloud provider P

```
P ← ∅;
foreach Provider p do
    if Cost(p) < Cost(P) then
        P ← p;
    end
end
```

Algorithm 2: *Public Scheduling* Pseudocode

Hybrid Cloud Scheduling

In a hybrid setup, additional logic is required to decide whether to execute an application on the organization's own computing infrastructure or to offload it to a public cloud provider. This decision is made by the hybrid scheduling component that allocates applications to the public and private cloud scheduler components. It should thereby optimize the total cost for running all applications, while minimizing the number of applications missing their deadlines due to queue congestion or high data transfer times. In Chapter 4, a hybrid scheduling algorithm was presented that calculates the *potential cost* of executing an application in the public cloud within deadline, and gives priority to executing the most expensive applications on the private cloud. This scheduling decision was taken independently of the private scheduling logic, after which the hybrid scheduler handed over control to the public or private scheduling component. As optimization of the private infrastructure

was not the goal of the contribution, the private scheduling logic consisted of a straightforward queuing system in which incoming requests were handled in a first-come first-served manner.

Although the results in Chapter 4 show that a cost-oriented approach pays off with regard to the number of deadlines met and that there is potential for significant cost reductions, the approach was found to be sensitive to errors in the user-provided runtime estimates for tasks. These induce a negative impact on the performance of the private scheduler with respect to the utilization or the number of deadlines met in case of respectively over- or underestimation. A simple approach of deliberately overestimating runtimes was shown to succesfully reduce this impact, but also led to increased costs.

The hybrid scheduling approach presented in this contribution deals with the aforementioned issues by merging the private and hybrid component described in Chapter 4 and by constantly re-evaluating the schedule in order to neutralize the consequences of wrong estimates. It relies on a single queue, in which applications wait to be executed on the private cloud. The private cloud scheduler prioritizes applications according to a *queue sort policy*, while a *queue scanning algorithm* detects unfeasible applications, after which one or more applications are sent to the public cloud. An outline of the integrated scheduling policy is shown in Figure 5.1, and the operation is further explained in the remainder of this section.

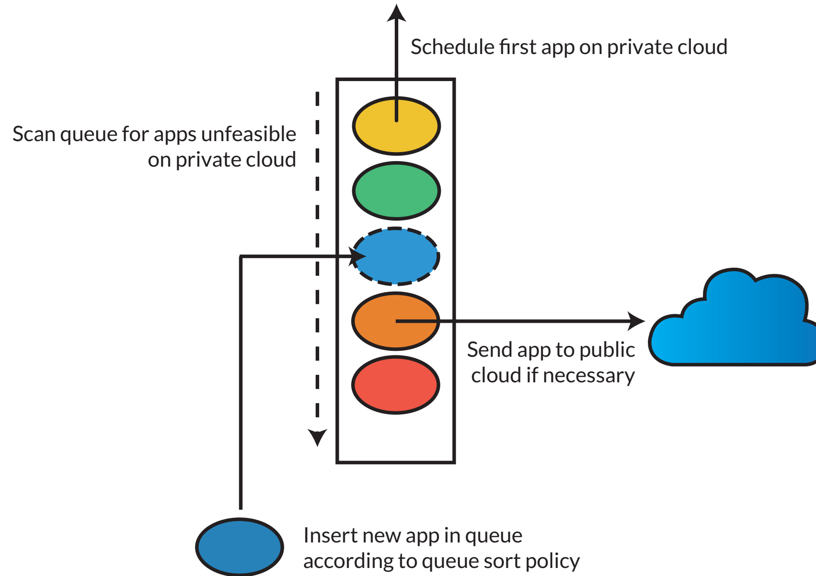


Figure 5.1: Hybrid Scheduler - Schematic View

The private cloud scheduler adopts the strategy of scheduling a task on the instance type on which it induces the smallest *load*. Let CPU_{it} be the number of CPUs of the instance type, the $load_{task}$ of a task t on an instance type it of the private cloud provider p is defined in Equation 5.6.

$$load_{task}(t, it, p) = RT_t^{it,p} \cdot CPU_{it} \tag{5.6}$$

From the perspective of the hybrid scheduler, running an application on the private cloud is considered to be free of cost, and the applications' data sets are available free of charge and without delay. The scheduler therefore attempts to maximize the local infrastructure's utilization, and offloads applications only when absolutely necessary. Cost-efficiency results for the hybrid scheduling algorithms as presented in Section 5.3 therefore indicate an upper bound on actually realized cost reductions, as fixed and variable costs for the acquisition, operation and maintenance of the private infrastructure are not accounted for. An internal costing model can be developed to distribute these costs over the user population, while also incorporating the opportunity cost of executing an individual application on the private cloud. The development of a fair internal billing system based on such a cost model is an important and still underdeveloped field, but lies beyond the scope of this thesis.

When a new application is submitted for execution, it is inserted in the queue at the right position according to a *queue sort policy*. In this contribution, both *First-Come First-Served* (FCFS) and *Earliest Deadline First* (EDF) policies are considered. With FCFS, new applications are added at the end of the queue and are handled by the scheduler in order of arrival. EDF sorts applications on their deadline, prioritizing applications that are closer to their deadline. EDF is known to be optimal in terms of the number of deadlines met when dealing with preemptive and independent jobs on uniprocessors [66]. Although we are dealing with non-preemptive "bag-of-tasks"-type applications in a heterogeneous multi-processor environment with additional data requirements, we expect the EDF policy to have a positive impact on the cost and number of deadlines met. An application that is further away from its deadline will likely be cheaper to run on the public cloud as cheaper instance types are available for the application to complete before its deadline.

The application at the front of the queue is to be executed on the private cloud as soon as sufficient resources become available. An often used optimization technique to increase utilization is *backfilling* [85]. Backfilling allows short jobs to skip ahead in the queue provided they do not delay any other job in the queue or –less conservatively– the job at the head of the queue. Backfilling can be done on the job- or CPU level. Job-level backfilling can be implemented to increase utilization in a system with multi-node parallel jobs. This is typically found in MPI-like applications, and is not applicable to our bag-of-tasks workload model. Because the private infrastructure is considered as a uniform set of CPUs and the actual selection of a physical server is not considered, the impact of CPU-level backfilling for individual tasks would be minimal, and is therefore not implemented. Moreover, backfilling typically relies on user-provided runtime estimates [84] –just like the algorithms in this contribution– and errors in these estimates would obscure whether observed scheduling behavior is caused by their impact on backfilling or on our own algorithms.

An application should be removed from the queue and sent to the public cloud when –given the current state of the private infrastructure and the other applications in the queue– the application won't be able to finish before its deadline. Such an application is termed *unfeasible*. The queue scanning algorithm shown in Algorithm 3, detects unfeasible applications by constructing a tentative schedule for the private

cloud. This schedule is based on the progress of the different applications in the queue and their estimated runtimes.

The algorithm is executed periodically every N seconds. A small value of N allows the scheduler to quickly react to changes in the expected runtime of applications, while an increased value of N leads to a smaller computational overhead.

```
Input: Q ⟵ Application Queue, sorted on queue sort policy
Input: Policy ⟵ function ∈ { UnfeasibleToPublic, CheapestToPublic }
Schedule ⟵ {Running tasks};
foreach Application a ∈ Q do
    foreach Task t ∈ a do
        time ← GetStartTime(schedule, t);
        // get instance type with least total workload within deadline
        it ← GetInstanceType(t, time);
        if it exists then
            // Deadline can be met on private
            Schedule ⟵ t;
        else
            // Application a is unfeasible
            Policy(Q, a);
            break; // and restart queue scan with removed app(s)
        end
    end
end
```

Algorithm 3: *Queue scanning* Pseudo-code

The queue scanning algorithm is linked to a policy that determines the action to take when an unfeasible application is detected. In this work, we consider two possible actions:

- *Unfeasible-to-public*: The application that proved unfeasible is removed from the queue and sent to the public cloud for execution. Data has to be transferred to the public cloud provider, and all tasks can run in parallel on different instances in order to try and meet its deadline. The pseudocode can be found in Algorithm 4.

```
Input: Q ⟵ Application Queue, sorted on queue sort policy
Input: app ⟵ Unfeasible application
Q ← Q \ {app}
∀t ∈ app : p_t, it_t ← findCheapestPublicCloudProvider(t, app)
if ∀t : p_t exists then
    ∀t ∈ app : schedule t on p_t, it_t
else
    // Deadline cannot be met
    app.status ⟵ unfeasible;
    break;
end
```

Algorithm 4: *UnfeasibleToPublic* Pseudo-code

- *Cheapest-to-public*: The scheduler calculates the cost that each application which preceeds the unfeasible application would induce, when it is run in the public cloud. It thereby ignores applications that are already running on the private cloud as well as applications with a smaller workload than

the unfeasible application, and selects the application with the lowest cost (normalized to the workload of the application) for execution in the public cloud. The pseudocode for this policy can be found in Algorithm 5.

```
Input: Q ⟵ Application Queue, sorted on queue sort policy
Input: P ⟵ Private cloud provider
Input: app ⟵ Unfeasible application
cheapestApp ⟵ ∅
foreach Application a ∈ Q do
    ∀t ∈ a : p_t, it_t ← findCheapestPublicCloudProvider(t, a)
    ∀t ∈ a : private_it_t ← min_{it}^{IT_P} (load_task(t, it, P))
    if a > app // according to sort policy
    then
        continue;
    else if Σ_{t∈a} load_task(t, private_it_t, P) < Σ_{t∈app} load_task(t, private_it_t, P) then
        continue;
    else if Cost(a) ≥ Cost(cheapestApp) then
        continue;
    else
        cheapestApp ⟵ a;
    end
end
if cheapestApp exists then
    Schedule cheapestApp on public cloud;
    Q ← Q \ {cheapestApp};
else
    // Deadline cannot be met
    app.status ⟵ unfeasible;
    break;
end
```

Algorithm 5: *CheapestToPublic* Pseudo-code

5.2 Experimental Setup

We evaluate the proposed scheduling algorithms using a Java-based discrete time simulator[1]. Simulations are executed on an *AMD Opteron 6274*-based system with 64 CPU cores and 196 GB of memory.

Simulation runtimes depend on the scheduling techniques and the private cloud size used in the simulation setup, as well as the queue scanning interval N. In Section 5.3, we evaluate the influence of N on the simulation runtime.

Standard deviation on the outputs of the simulations were low overall, and are therefore not mentioned in the discussion of the simulation results.

Cloud setting

We introduce a public cloud landscape in which users run applications on two providers, modeled after the on-demand offerings of Amazon's EC2 and GoGrid, with all their prices and instance type configurations. For Amazon EC2, prices are taken from the US-east, US-west and EU-west regions, and their "cluster" and "micro"-instances are disregarded. For reference, a listing of these instance types and prices can be found in Tables 5.1, 5.2 and 5.3.

			Price/hour		
Name	CPUs	Memory	us-east	us-west	eu-west
m1.small	1	1.7 GB	$0.085	$0.095	$0.095
m1.large	4	7.5 GB	$0.34	$0.38	$0.38
m1.xlarge	8	15 GB	$0.68	$0.76	$0.76
c1.medium	5	1.7 GB	$0.17	$0.19	$0.19
c1.xlarge	20	7 GB	$0.68	$0.76	$0.76
m2.xlarge	6.5	7 GB	$0.50	$0.57	$0.57
m2.2xlarge	13	34.2 GB	$1.00	$1.14	$1.14
m2.4xlarge	26	68.4 GB	$2.00	$2.28	$2.28

Table 5.1: Amazon EC2 Instance types and prices (d.d. June 1, 2011)

			Price/hour
Name	CPUs	Memory	EU West
X-Small	0.5	0.5 GB	$0.095
Small	1	1 GB	$0.19
Medium	2	2 GB	$0.38
Large	4	4 GB	$0.76
X-Large	8	8 GB	$1.52
XX-Large	16	16 GB	$3.04

Table 5.2: GoGrid Instance types and prices (d.d. June 1, 2011)

	Price/GB	
Cloud provider	Data in	Data out
EC2 (all regions)	$0.10	$0.15
GoGrid	$0.00	$0.29

Table 5.3: Amazon EC2 and GoGrid network prices (d.d. June 1, 2011)

As stated in this Part's problem domain, we assume that the transfer speed from the organization to each of the available public data centers is known a priori and does not fluctuate. To obtain realistic data, we measured the average bandwidth available from the University of Antwerp in Belgium to the cloud providers used in this contribution. In line with other contributions [93, 94, 95, 96], we used the iperf[2] bandwidth measurement tool. Every hour over the course of one day, we measured the available bandwidth to Amazon's `us-east-1`, `us-west-1` and `eu-west-1` regions, as well as the GoGrid `EU West` datacenter. The averages of these measurements are shown in Table 5.4. In our case, transferring data to `eu-west-1` is more than 15 times faster than transferring data to the cheaper `us-east-1` region, and more than four times faster than transferring the same data set to the European GoGrid datacenter.

[1] Code and experimental data are available upon request with the corresponding author.
[2] http://iperf.sourceforge.net/

Cloud provider	Avg (MB/s)	Time/GB (s)	Stdev
EC2 us-east-1	1.54	665	40.87%
EC2 us-west-1	1.19	864	42.43%
EC2 eu-west-1	24.16	42	23.58%
GoGrid EU West	5.91	173	32.06%

Table 5.4: Available bandwidth from UA to Cloud Providers

The private cloud is assumed to reside on-premise in the users' organization, and supports the three instance types described in Table 5.5. Data sets are immediately available for applications running on the private infrastructure.

Name	CPUs	Memory
small	4	4 GB
large	8	8 GB
xlarge	16	16 GB

Table 5.5: Private Instance types

Application Workload

Due to the lack of sufficient real-world data, especially with regard to application data set sizes and deadline constraints, we need to resort to a partially synthetic model for generating application instances for the experiments in Section 5.3.

Every scheduling experiment has a duration of one week, in which new applications arrive every second following a Poisson distribution with $\lambda = 0.002$. This results in an average inter-arrival time of 2000 seconds, or about 1200 applications in a week. With these parameters, we aim to generate enough load to tax the private cloud beyond its saturation point so that it becomes advantageous to use a hybrid cloud setup.

The number of tasks per application is uniformly distributed between 1 and 100. Task runtimes within one application are typically not independent of each other. Therefore, we assign each application a *base runtime*. Following [91], the application's base runtime is modeled as a Weibull distribution. The distribution's parameters are derived from the Parallel Workloads Archive's *SDSC IBM SP2* workload trace[3]. The runtime of each individual task is then drawn from a normal distribution, with the application's base runtime as the mean μ and a relative *task runtime standard deviation* σ. With $\sigma = 0\%$ task runtimes equal the application's base runtime, while for example $\sigma = 100\%$ results in high runtime variations. Unless mentioned otherwise, this deviation was fixed at 50%.

Tasks can typically run on more than one instance type. Running a task on an instance type with for example 1 CPU and 1.7 GB memory will probably –but not always– be slower than running the same task on an instance type with 4 CPUs

[3] http://www.cs.huji.ac.il/labs/parallel/workload/

and 7.5 GB memory. Modeling speedup of parallel tasks on multiple processors without thorough knowledge on the operation of the application is a delicate and complex task. We model the speedup after Amdahl's law and assume that a task has a sequential fraction with length c_1 and a parallelizable fraction with length c_2. The execution time of the task then equates to $c_1 + c_2/n$, where n is the number of processors available on the instance type. We describe the size of the parallel component as a percentage of the total runtime and use a uniform distribution between 0% and 100%. Note that in this workload model, we do not factor in the speed of the individual cores of the instance type to determine the runtime, and assume it to be homogeneous across the different instance types[4]. Next to the number of CPU cores, the amount of memory available on an instance type IT_{mem} can also influence the runtime of a task. Applications get assigned a *minimum memory requirement* App_{MemReq}. The runtime for tasks executed on instance types that meet the application's memory requirement remains unaltered. If $App_{MemReq} > IT_{mem}$, a task suffers a slowdown equal to IT_{mem}/App_{MemReq}. Although this linear slowdown is optimistic and might be worse in practice, it satisfies our need to differentiate between instance types. The minimum memory requirement for a task is normally distributed with an average of 1.5 GB and a standard deviation of 0.5 GB.

Taking into account the application's base runtime, task runtime variance, parallel component size and memory requirement, the runtimes for each of the application's tasks on each of the available instance types can be determined before submission to the scheduler.

Each application is associated with a deadline that determines to a large extent the instance type used for executing the tasks. Without deadlines, all tasks would be executed on the cheapest –and often slowest– instance type. At application arrival, we determine its time-to-deadline as a factor of the application's runtime on the *fastest instance type* available. To cover both tight and loose deadlines in our workload, this *application deadline factor* is uniformly distributed between 3 and 20.

Finally, an application is linked to a *data set* that is used by each of the application's tasks. Its size is normally distributed with averages ranging from 0 GB for purely computational tasks to 2500 GB for data-intensive applications, with a relative standard deviation of 50%.

We model the inaccuracies in estimating the runtime of tasks by assigning each application a *Runtime Accuracy* factor $acc \in [0,1]$.The estimate distribution is then modeled as a Gaussian distribution with μ the exact task runtime and $\sigma = \mu \cdot (1 - acc)$. For the experiments in which runtime estimation errors are not considered, acc is set to 1.

An overview of the workload model parameters used in this paper is given in Table 5.6.

5.3 Results

In this Section, the scheduling algorithms proposed in Section 5.1 are evaluated using experiments according to the workload model and cloud setting described in

[4]Generalizing the model to non-homogeneous CPU speeds however does not impact our scheduling algorithms and could be achieved by adding an additional scaling factor for the task's runtime on a given instance type.

Parameter	Value
App. arrival rate (seconds)	Poisson(0.002)
Tasks per application	Uniform(1,100)
Base runtime (hours)	Weibull($\lambda = 1879$, $\beta = 0.426$)
Task runtime standard dev. (%)	50%
App. parallel fraction (%)	Uniform(0, 100)
App. memory requirement (GB)	Normal($\mu = 1.5$, $\sigma = 0.5$)
App. deadline factor	Uniform(3,20)
App. data set size (GB)	Normal(varying μ, $\sigma = 50\%$)
Runtime accuracy	1 *or* Uniform(0,1)

Table 5.6: Workload model parameters

Section 5.2.

Public Cloud Scheduling

We first illustrate the impact of data transfer costs in our experimental setup by comparing the difference between our cost-efficient scheduler with (*CE-Data*) and without incorporating these costs in the provider selection decision (*CE-NoData*). Data transfer speeds are not yet considered. Figure 5.2 illustrates this impact when scheduling applications on the public cloud, and shows the influence of the data set size of an application on the differences between both policies. The results are shown relative to a *Random* policy, in which for each arriving application a cloud provider is picked randomly. The application's average data set size μ varies from 0 GB to 2.5 TB with a relative standard deviation σ of 50%.

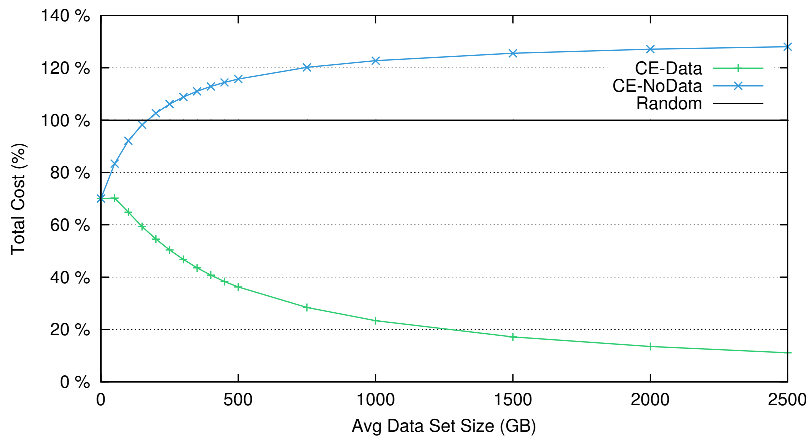


Figure 5.2: Impact of data costs.

Without data costs, the cost-efficient scheduler obtains a cost reduction of 30% compared to the random policy. We also observe a significant cost reduction of the scheduling policy that takes data costs into account compared to the same policy that only reckons with computational costs. This experiment also gives an indication

of the share that compute and transfer costs have in the total cost: with an average data set size of 50 GB, the computational costs for *CE-Data* cover 85% of the total costs, while the share in *CE-NoData* decreases to 65%. For μ =500 GB this share shrinks to 81% and 16%, respectively.

Next to the computational cost and the data transfer cost, a third important factor to consider in a public cloud setting is *data locality*. The experimental setting is the same as the previous experiment, but data transfer speeds are now considered and assumed to be equal to the measured averages of the bandwidth benchmarks in Section 5.2. We use the *CE-Data* policy to calculate computational and data transfer costs. In total, 1198 applications are scheduled. Figure 5.3 shows the amount of applications scheduled in each of the regions.

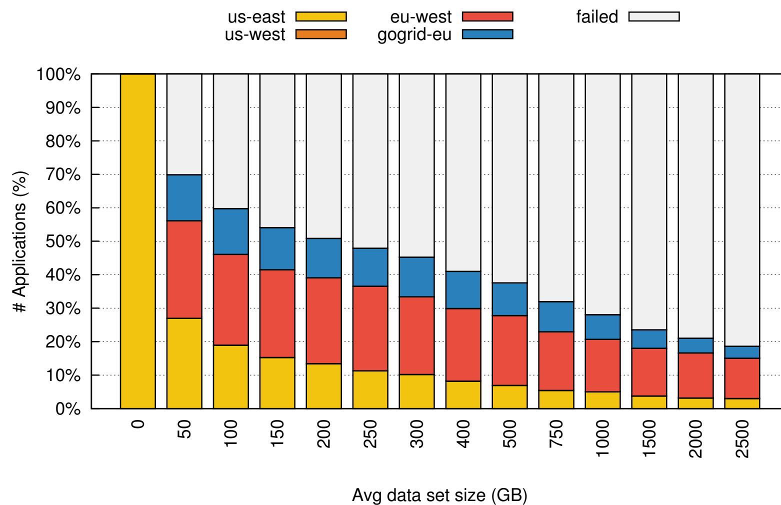


Figure 5.3: Impact of data locality.

We observe that applications without a data set are all scheduled in the `us-east-1` region, as it provides the cheapest compute capacity. Applications with a large data set are scheduled in the EC2 `eu-west-1` region in case of an application with a tight deadline, because the data set can be transferred to this datacenter more quickly. Applications with large data sets and loose deadlines are scheduled in the GoGrid datacenter, because of the zero cost for inbound traffic. The percentage of applications not meeting their deadline increases quickly as average data set sizes exceed 400 GB. This can be explained by the fact that in our workload model, the application's deadline is determined as a factor of the fastest possible task runtime, without considering data transfer times. Increasing the average data set size in an experiment consequently results in tighter deadlines and can lead to the sum of the data transfer time and the task runtime on the fastest instance type to exceed the deadline.

Hybrid Cloud Scheduling

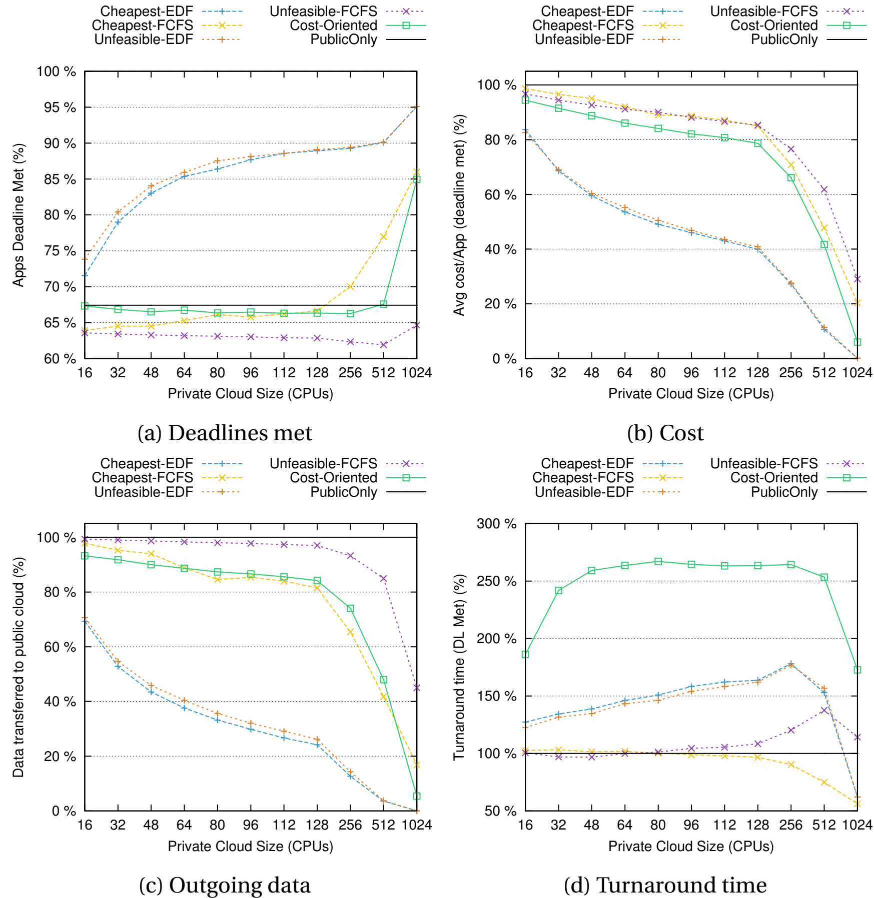
The hybrid scheduling components proposed in this contribution appear in four guises, composed by the combination of the action when an unfeasible application

is found (*Unfeasible-to-public* or *Cheapest-to-public*) and the queue sort policy (*EDF* or *FCFS*). These are compared to the *Cost Oriented* scheduling policy from Chapter 4 as well as a *Public Only* scheduler that schedules all applications on the public cloud using the algorithm and cost calculation described in Section 5.1. In all of the experiments in this subsection, we use a public scheduling component with the *CE-Data* policy.

We demonstrate the operation of the hybrid scheduling components by adding a private cloud to the experimental setting used in Section 5.3. In the first experiment, computational costs, data transfer costs as well as data transfer times are taken into account, but runtimes are still assumed to be known a priori. Therefore, the value of N –the interval between two consecutive runs of the queue scanning algorithm– is of low importance and is set to 5 minutes. This scenario corresponds to the experimental setting in Figure 5.3 with an average data set size of 50 GB and with the addition of a private cluster of varying sizes.

The results for all scheduling policies in this experiment are shown relative to *Public Only* policy, and can be found in Figure 5.4. Figure 5.4a shows the number of applications that were completed within their deadline with varying sizes of the private cloud. While the Public-Only scheduler misses 32% of the deadlines due to the extra time needed to transfer the data, the incorporation of a private cloud –on which such transfer times do not apply– allows the hybrid scheduler to meet more deadlines by decreasing the amount of outgoing data. This is also clearly depicted in Figure 5.4c. In addition, it is noteworthy from Figure 5.4a that the EDF-based policies succeed to better take advantage of a private infrastructure in terms of number of deadlines met. This is the result of both the intrinsic properties of EDF to process more applications within deadline constraints, and of the operation of the queue scanning algorithm that sends significantly more applications with loose deadlines –which mostly reside at the tail of the queue– to a public cloud provider. In Figure 5.4b, the average cost per application executed within deadline is displayed. It is to be expected that adding a private cluster results in significant cost reductions, but we also observe that the cost savings when using an EDF-based scheduler over the other schedulers are very large. The cause of this difference lies in the fact that EDF manages to efficiently schedule up to twice as much applications on the private cluster as any other scheduling policy. The difference between the *Cheapest-to-public* and the *Unfeasible-to-public* approaches is significant in case of the *FCFS* approaches, but is limited to only a few percent for EDF-based approaches. In our synthetic setting, the substitutability of the applications in terms of their computational cost is high. Therefore, a "wrong" scheduling decision taken by the *Unfeasible-to-public* policy has only a minor impact on the total cost reduction. One side effect of the *Cheapest-to-public* approach is that significantly less data-intensive applications are sent to the public cloud. This is illustrated in Figure 5.4c that shows the total amount of data transferred to the public cloud, relative to the amount of data transferred to the cloud in a public-only scenario. Finally, the average turnaround time –the time between the submission and the end of the last task of an application– is shown for applications that met their deadline in Figure 5.4d. The *Cost-Oriented* algorithm leads to high turnaround times as it postpones the execution of an application's tasks as long as possible. The turnaround time for the FCFS-schedulers is lower, but this comes at a cost of a higher number of deadline misses. The EDF-schedulers' turnaround times are lower than those of the *Cost-*

Oriented policy but higher than those of the *FCFS* policies. The diverging turnaround times of Cheapest-FCFS and Unfeasible-FCFS for larger private cloud sizes are caused by the difference in handling data-intensive applications: when Cheapest-FCFS outsources the cheapest applications, it consequently selects applications with smaller data sets. This can be seen in Figure 5.4c. Applications with less data transfer cause less delay due to data transfer time, ultimately resulting in a lower turnaround time.



(a) Deadlines met

(b) Cost

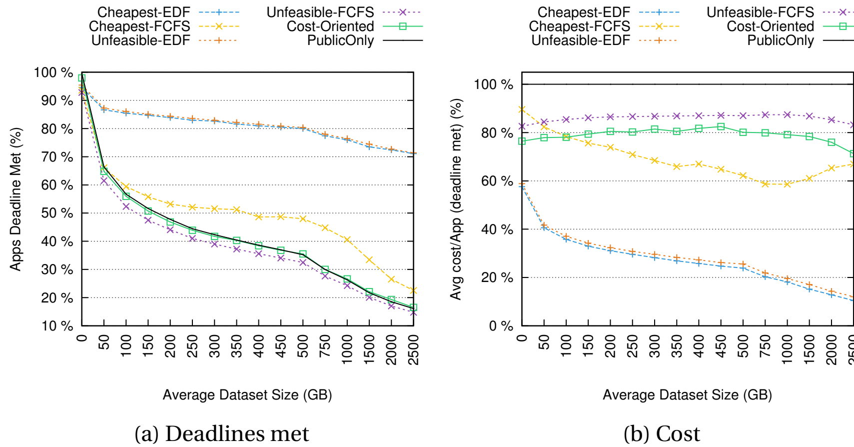
(c) Outgoing data

(d) Turnaround time

Figure 5.4: Impact of hybrid scheduling policies on *Deadlines, Cost, Outgoing data* and *Turnaround Time.*

We further explore the characteristics of the proposed scheduling algorithms by examining the influence of the data set size on the potential cost reduction. As the data set size increases, the number of failed deadlines will increase due to the longer data transmission times when outsourcing an application. This is shown in Figure 5.5a. The more efficient placement capabilities of the EDF-based policies are robust with relation to the data set size. Figure 5.5b shows the cost savings the algorithms achieve with a private infrastructure of 128 CPUs and with varying average data set sizes, relative to the cost of scheduling the whole application set

on a public cloud provider. With bigger data sets, the difference in cost between the *EDF* algorithms and the *FCFS* policies quadruples. The cost reduction of the *Cheapest-EDF* scheduling policy relative to the *Unfeasible-EDF* policy increases from 2% (without data) up to 13% (with an average data set of 2.5TB).



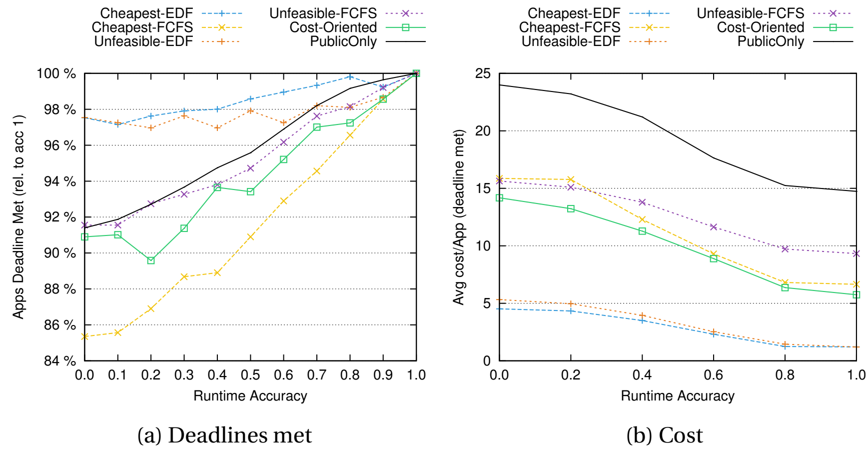
(a) Deadlines met (b) Cost

Figure 5.5: Influence of *Data set size* on hybrid scheduling

All of the algorithms discussed in this chapter depend to a greater or lesser extent on the accuracy of the provided runtimes as they are used to calculate the potential cost of running the application on the public cloud, as well as to select the instance type necessary to finish tasks within deadline. Moreover, the private scheduling algorithm relies on runtime estimates to determine the *start time* of a task when searching for unfeasible tasks. The *Cost-Oriented* algorithm (Chapter 4) already proved to be sensitive to estimation errors, because it attempts to delay the scheduling decision as much as possible up until the application's deadline.

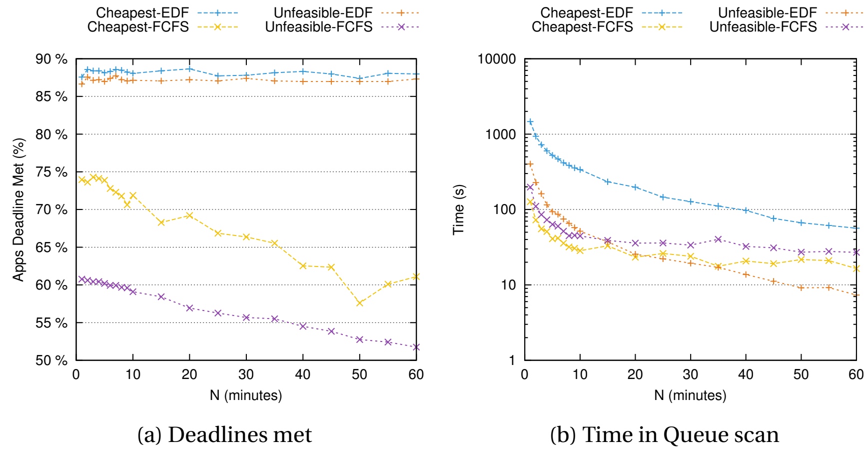
We evaluate the degree of sensitivity of the proposed algorithms to runtime estimation errors in a setting that supplements the previously used cloud providers with a 512 CPU private cluster. Applications have an average data set size of 50 GB. For different values of acc, we evaluate the number of deadlines met and the average cost per application that finishes before its deadline. Figure 5.6a shows the value of these two metrics relative to the scenario in which $acc = 1$. A decrease in the accuracy causes the number of applications that meet their deadline to decrease. The decrease is limited for EDF to less than 3%, while the same metric declines about 5 times faster for the cost-oriented approach. The Cheapest-FCFS policy performs in relative terms even worse, and loses the performance gains it had over the Unfeasible-EDF and Cost-Oriented policies when $acc = 1$.

In Figure 5.6b, the average cost per application is shown for different values of acc. All hybrid schedulers seem to follow the increasing trend set by the public scheduling component, which selects larger and thus more expensive instance types for overestimated tasks. In relative terms, the increase for the EDF-based queue sort policies is stronger than the other scheduling policies because it manages to execute more expensive –overestimated– applications, which are marked as unfeasible by the other scheduling algorithms.



(a) Deadlines met (b) Cost

Figure 5.6: Impact of *Runtime Estimation Errors* on hybrid scheduling.



(a) Deadlines met (b) Time in Queue scan

Figure 5.7: Influence of N on hybrid scheduling.

Up until now, the time interval N between two consecutive executions of the queue scanning algorithm was chosen to be 5 minutes. However, in the presence of runtime estimation errors, the value of N has an influence on the schedulers' ability to quickly anticipate on a change in runtimes, but also on the computational overhead of the scheduling logic. In the following, we evaluate for different values of N ranging from 1 minute to 1 hour the influence on the number of deadlines met as well as the time spent in the queue scanning mechanism. We again vary *acc* according to a uniform distribution between 0 and 1. The results in Figure 5.7 show that increasing the value for N leads to a significant decrease in the number of applications that meet their deadline, which is due to the decreased number of times the algorithm is able to detect and correct deviations of the estimated runtimes. This decrease is not present when ordering the queue based on deadline, because the outsourced applications are generally further away from the front of the queue and are therefore exposed to less tight deadlines. For other scenarios with

tighter deadlines, the EDF-based policies exhibit a decrease, but more moderate than the FCFS-policies. For example, in a scenario with a *deadline factor* between 1.2 and 6, the Cheapest-EDF and Unfeasible-EDF achieve 3% less deadlines in $N = 60$ than in $N = 1$. Furthermore, it is noticeable that the computational overhead for *Cheapest-EDF* is up to an order of magnitude larger than the FCFS-based policies. This overhead is caused by iteratively calculating the cost for a large number of applications. In spite of the difference it should be noted that a total scheduling time of 1400 seconds ($N = 1$) for almost 2000 applications over the course of a two weeks scheduling period is negligibly small. Depending on the available computational power to schedule applications and the arrival rate in the system, it should be possible for a system administrator to set the value of N so that the amount of deadlines met can be maximized.

5.4 Conclusion

Supplementing the private infrastructure of an organization with resources from public cloud providers introduces the problem of cost-efficiently and automatically managing application workloads within such a hybrid cloud environment. This paper presents scheduling algorithms to deal with this optimization problem for deadline-constrained bag-of-task type applications while taking into account data constraints, data locality and inaccuracies in task runtime estimates. In Chapter 3, we presented a linear programming formulation of the optimization problem for a static set of applications. In this contribution, online hybrid scheduling algorithms that operate on larger-scale problems with additional data constraints are presented and evaluated in terms of deadlines met, cost-efficiency, computational efficiency, application turnaround time, and robustness with regard to errors in runtime estimates. Our results quantify the additional gains in cost-efficiency that can be achieved by adopting an EDF approach on the private cloud. We demonstrate that further cost reductions are realized if cost is used as a discriminating factor for selecting outsourced applications. In addition, an EDF scheduling policy for the private cloud is shown to significantly increase robustness with respect to runtime estimation errors, at an additional cost in turnaround time. As such, our results can provide guidance for system administrators to assess the trade-offs of adopting an EDF policy for hybrid cloud deployments that focus on bag-of-task type applications. Finally, our analysis of computational cost indicates that the proposed algorithms are able to schedule a large number of applications within a practical timeframe.

Future work in line with this contribution consists of further generalizing the workload model, increasing the detail of the private cloud model and incorporating other public pricing models such as Amazon's *reserved* and *spot* instances.

Part III

Contract Portfolio Optimization using Load Prediction

Contract Portfolio Optimization using Load Prediction

Businesses and non-profit organizations increasingly rely on cloud computing to handle (part of) their applications' workloads. This has led to a shift in IT expense management policies from purchasing hardware and support contracts to establishing and maintaining company policies for cloud management and cost control. At present however, few tools are available to support decision makers in this task. Meanwhile, the need for automation is gradually increasing as cloud providers are rapidly complexifying their products and associated contracts.

An IaaS cloud provider offers virtual machine *instances* for rent to its customers. Instances are commonly offered according to a set of predetermined resource specifications, called *instance types*. An instance type defines the architecture, the number of CPU cores and processing speed, the amount of memory, the network and disk bandwidth and the hard disk size of that type. Although some cloud providers offer more flexibility by allowing consumers to configure the different resource dimensions independently, we specifically deal with fixed instance types in these chapters.

Instance usage is generally charged for in a pay-as-you-go manner by associating every instance type with a *price per time unit*. Amazon EC2, for example, adopts a price-per-hour policy, rounding up partial hours of usage, while other providers – such as Rackspace or Google Compute Engine– charge usage in 1 minute increments, rounding up to the next minute. Besides this *on-demand* charging policy, some providers –Amazon and Rackspace, among others– also offer *reserved* contracts of a given length (1 or 3 years at Amazon EC2, 6 to 36 months at Rackspace). These imply a significantly lower price per time unit in exchange for an up-front payment.

To illustrate the potential cost savings and complexity that comes with the selection of a reserved contract type, Table 5.7 illustrates all available contracts and prices for an Amazon EC2 `m1.small` instance type in the US-East region. Column *Utilization optimum* contains the instance utilization level (as a percentage of the total time in the contract period) for which the reserved contract is the most cost-effective. The last column shows the potential cost savings for each reserved instance plan, compared to the use of on-demand instances. For example, the one-year medium utilization reserved contract is the cheapest for an instance that runs at least 69%

Usage level	Up-front cost	Hourly price	Period	Utilization optimum	Cost reduction vs. on demand
Heavy	$169	$0.014	1 year	83%–100%	33%–45%
Medium	$139	$0.021	1 year	69%–82%	27%–33%
Light	$61	$0.034	1 year	27%–68%	0%–26%
On demand	n/a	$0.06	1 year	0%–26%	0%
Heavy	$257	$0.012	3 years	80%–100%	55%–64%
Medium	$215	$0.017	3 years	46%–79%	42%–54%
Light	$96	$0.027	3 years	12%–45%	0%–41%
On demand	n/a	$0.06	3 years	0%–11%	0%

Table 5.7: Amazon EC2 prices for `m1.small` in the US-East Region (d.d. January, 1 2014).

and at most 82% of the contract period, which entails a cost reduction of 27% up to 33%.

Next to potential cost savings, reserved contracts can also increase quality of service. In EC2 for example, the availability of instances purchased under a reserved contract is guaranteed during the contract's length, while requesting (a large number of) on-demand instances might fail under congestion.

The availability of multiple reservation plans introduces complexity for the customer in balancing the flexibility of on-demand instances with the potential cost savings and increased quality of service of long-term commitments. We will only focus on the first two aspects of this problem, relying on prediction techniques to forecast server loads in order to find the right balance from a cost perspective. The main reason for this is that very limited data is available on availability differences and that the impact of a decreased availability depends heavily on the enterprise and its applications.

Overview

Chapter 6 uses Genetic Programming to forecast future load patterns, which are used by a resource manager to calculate an optimal contract portfolio. The proposed approach assumes a fixed contract length of 90 days, and does not take into account existing contracts. Its performance is evaluated using real-world traces with the daily page view data of four well-known web applications.

Chapter 7 introduces a heuristic that allows for online optimization of an organization's contract portfolio, given a future load predicted with time series forecasting techniques. The algorithm takes into account an organization's existing contracts, and uses contracts with a length of 1 year. Its operation is illustrated as a case study using four web application traces, similar to the approach in Chapter 6. Our analysis explores the cost impact of a Double-Seasonal Holt-Winters prediction technique compared to a clairvoyant predictor and compares the algorithm's performance with a stationary contract renewal approach.

Chapter 8 extends the algorithm presented in Chapter 7 with an additional parameter and two extra renewal policies. The evaluation comprises an extensive study of over 50 workload traces, simulated with four time series forecasting models and a reactive prediction technique in combination with four different renewal policies.

The techniques are evaluated on their performance in terms of cost reduction, as well as on their robustness and computational complexity.

Related Work

Chaisiri et al. [97] formulate a stochastic programming model to create an optimal reserved contract procurement plan. The aim of their algorithm is to cover as much demand as possible with reserved contracts. They take into account multiple cloud providers, including private ones with limited resources, and show that the algorithm makes a good trade off between reserved and on-demand instances. However, they assume that the demand distribution is known and constant, and do not take into account multiple contracts with the same length and different utilization levels. They also do not consider scenarios of demand fluctuation, in which it is more beneficial to use on demand instances to absorb some of the load peaks.

Shen et al. [98] introduce a cloud-based online hybrid scheduling policy for both on-demand and reserved instances. Their algorithm relies on Integer Programming to find an optimal solution, given the future workload distribution. The derivation of this predicted workload distribution is not discussed. Their work also differs from ours in the workload model: short jobs are scheduled on virtual machines using common job scheduling heuristics.

Hong et al. [99] propose a *commitment straddling* approach, in which they determine the optimal distribution of on-demand and reserved instances. They assume the Cumulative Distribution Function of the future load is known a priori and calculate the function values at the utilization optima for a reserved instance. Multiple reserved utilization levels are not considered. Additionally, they do not discuss methods for forecasting the CDF, and only use short web traces (< 6 months) in their evaluation.

In Tian et al. [100], a linear program is used to minimize the provisioning cost for deploying a web application on both reserved and on-demand instances. The web application's load can be classified as high, normal and low, and the authors assume that the long-term load is constant: last year's utilization (expressed as the number of hours in each load class) is used as the prediction for the upcoming year.

Other efforts in optimizing cost in a cloud environment include [92, 57, 101, 102]. These do not take into account reserved contracts, but focus on optimizing virtual machine placement over multiple public and/or private clouds.

CHAPTER 6

Genetic Programming-based Load Models

This chapter is published as "Optimizing a Cloud contract portfolio using Genetic Programming-based Load Models", S. Stijven, R. Van den Bossche, K. Vladislavleva, K. Vanmechelen, J. Broeckhove and M. Kotanchek in Genetic Programming Theory and Practice XI, 2014 [103].

Abstract

Infrastructure-as-a-Service (IaaS) cloud providers offer a number of different tariff structures. The user has to balance the flexibility of the often quoted pay-by-the-hour, fixed price ("on demand") model against the lower-cost-per-hour rate of a "reserved contract". These tariff structures offer a significantly reduced cost per server hour (up to 50%), in exchange for an up-front payment by the consumer. In order to reduce costs using these reserved contracts, a user has to make an estimation of its future compute demands, and purchase reserved contracts accordingly. The key to optimizing these cost benefits is to have an accurate model of the customer's future compute load – where that load can have a variety of trends and cyclic behavior on multiple time scales. In this chapter, we use genetic programming to develop load models for a number of large-scale web sites based on real-world data. The predicted future load is subsequently used by a resource manager to optimize the amount of IaaS servers a consumer should allocate at a cloud provider, and the optimal tariff plans (from a cost perspective) for that allocation. Our results illustrate the benefits of load forecasting for cost-efficient IaaS portfolio selection. They also might be of interest for the Genetic Programming (GP) community as a demonstration that GP symbolic regression can be successfully used for modeling discrete time series and has a tremendous potential for time lag identification and model structure discovery.

6.1 Problem Domain

In this contribution, the focus is on the long term planning problem of purchasing server instances with different service levels and prices from a public cloud provider

in a cost-efficient manner, based on predictions of future load by using Genetic Programming.

The tariff structure of a cloud provider is represented as a generalization of Amazon's on-demand and reserved pricing plans. We assume that a cloud provider proposes a number of offerings, with an associated *up-front cost* (≥ 0) in exchange for a cheaper *hourly rate* during a given *period* ρ (in days). An offering includes one of the two *charging methods*:

As-you-go After paying the upfront cost, the customer is charged only for the actual usage (hours) of the concerned instance. If an instance runs for h hours, the customer is billed the upfront cost and h times the hourly rate.

Every hour The customer makes a commitment to use the instance continuously. Even if the instance is not running, the hourly rate is charged. If an instance type offering with this charging method is purchased, the cost at the end of the period is always the same, regardless of the number of hours the instance has actually been running.

An "on-demand" plan, in which no up-front cost is charged and server hours are charged as-you-go can be modeled by setting the up-front cost = 0 and the period $\rho = \infty$.

The contract portfolio management approach in this chapter supposes that the purchase decision is taken only once each period, and that all purchases are made at the first day of that period. The expansion of the scheduling algorithms to a "sliding" approach in which previous ongoing purchases are taken into account is left to future work.

Divergent user requirements and heterogeneous workload characteristics complicate the problem of identifying the best cloud provider for a specific application. The approach presented here is confined to only some of the characteristics and requirements, which we believe constitute a solid base to demonstrate the impact and performance of the prediction and scheduling mechanics.

Our current application model focuses on web application workloads, because they have a long lifetime which makes the use of prediction methods relevant.

6.2 Related Work

Symbolic regression via genetic programming (GP) has been used for time series prediction since the seminal book by Koza [104]. In a majority of cases GP is given a task to learn the time series trend and reconstruct a dynamic system that generated time series using lagged (delayed) vectors. Given m lagged vectors $v_{t-1}, v_{t-2}, \ldots v_{t-m}$ induce a function f, such that

$$v_t = f(\mathbf{v}) = f(v_{t-1}, v_{t-2}, \ldots v_{t-m}). \tag{6.1}$$

Parameter m is often called an embedding dimension, lag parameter or a sliding time window. Sometimes another *delay* parameter τ is taken into account, and delayed lagged vectors $v_{t-\tau}, v_{t-2\tau}, \ldots v_{t-m\tau}$ are considered for model induction.

With few exceptions, like [105, 106], GP employs a standard symbolic regression representation. Each individual is a superposition of primitives (being mathematical operators) with terminals sampled from a set of lagged vectors as well as constants.

Many use GP to complement autoregressive and moving average models to capture the nonlinearity of time series. [107] created a separate GP individual for each sample from the prediction horizon, but the vast majority of publications focus on generating single step predictions (predicting the next value) and iterating them over the test set. The problem with iterative predictions (when predicted values are used as inputs for subsequent evaluations) is that they tend to magnify very quickly (see [108]).

In this work we attempt to generate accurate load predictions over a horizon of at least 90 unseen samples (days). Furthermore, all 90 samples are required to construct an optimal contract portfolio, i.e. the target prediction has to be in a form

$$(v_T, \ldots, v_{T+\rho})^T \in \mathbb{R}^\rho, \quad \rho = 90, \tag{6.2}$$

given the training data $(v_1, \ldots, v_{T-1})^T$.

As pointed out by [107] and [106], the main advantage of GP for discrete time series prediction lies in its ability to discover dependencies among samples, and automatically select only relevant variables for the models.

We have not found any GP references which used more than 20 consecutive samples (lags) as candidate inputs. One, four and ten lags are the most popular ([109, 106, 110]). [111] use a time window of the previous 20 data points, but no variable selection results are reported and only predictive accuracy of models rather their interpretability was measured.

We believe that recent algorithmical improvements in the effectiveness of the variable selection capability of GP should allow exploration of much larger spaces of lagged variables. In this work we push GP towards $m = 365$ inputs, and let driving variables be automatically discovered.

6.3 Algorithm Design for Contract Portfolio Optimization

In this section, an algorithm is outlined to find the optimal allocation of server instances over the available pricing plans, given predicted (or real) load data for the upcoming period. The algorithm should try to minimize cost, and output a purchase suggestion in the form of a number of server instances for each pricing plan.

The inputs to the algorithm consist of the required instances with an associated instance type for each hour in the scheduling period, and the set of available pricing plans. The algorithm first calculates the utilization (in hours) for each of the instances running in the period, after which the best pricing plan for this utilization level is calculated based on the cost function described in Section 6.1. The cheapest plan is added to the suggested purchases.

This approach, illustrated in Algorithm 6, gives preference to the instances with the highest utilization throughout the relevant period. As a consequence, the result set is first filled with pricing plans that offer the highest cost reduction in return for the biggest upfront cost, and falls back on the plans with a lower or zero upfront cost to cope with temporary load surges. This results in a very cost-efficient result set.

result ← ∅
for all $h \in \rho$ **do**
  for all $\iota \in h$ **do**
    utilization ← 0
    for $\alpha \leftarrow h$ **to** ρ **do**
      if $\iota \in \alpha$ **then**
        utilization ← utilization + 1
        $\alpha \leftarrow \alpha \setminus \iota$
      end if
    end for
    cheapest ← ∅
    for all $\phi \in \Phi$ **do**
      if $Cost^{\iota}_{\phi}$(utilization) < $Cost^{\iota}_{\text{cheapest}}$(utilization) **then**
        cheapest ← ϕ
      end if
    end for
    result ← result ∪ {cheapest}
  end for
end for
return result

Algorithm 6: Purchase Suggestion Algorithm

6.4 Data and Experimental Setup

Data

Server utilization data in cloud infrastructures is unfortunately not made available by cloud providers. In our experiments, we therefore use the publicly available traces[1] of the daily number of page views for a number of well-known websites. We assume that the number of page views relates directly to the load that gets generated and hence to the number of server instances required to handle that load. As the page view data consists of daily averages, we assume the load is also constant through the day, that the web application is deployed on a number of homogeneous instances of the same instance type, and that number of instances is scaled linearly according to the *page views per instance* ratio.

In order to evaluate the performance and robustness of the predictions and of the scheduling algorithm, we use the daily page view data from four well-known web applications with different characteristics (see also Figure 6.1):

1. **stackoverflow.com:** The page view data shows robust growth with a lower number of views in the weekends and during the Christmas period.

2. **imgur.com:** For this website we used the visitor data as the page view data contained odd properties, likely due to implementation details. The data shows robust growth until the middle of 2012, after which the growth seems to stagnate.

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

3. **linkedin.com:** The page view data shows a strong seasonal pattern with drops in the weekends and holidays and a robust growth until the beginning of 2012, which diminishes in the last 14 months.

4. **tinypic.com:** The page view data shows rapid decline with lower number of views in the weekends, but no strong seasonal pattern.

The data is in many cases non-stationary (means, variances and covariances change over time) and in all cases heteroscedastic (of different variability, i.e. not allowing constant error variance when modeled using ordinary regression[2]), which poses a good challenge to genetic programming, see Figure 6.1.

Experiment setup

We use the algorithm outlined in Section 6.3 with different inputs of expected future load to assess the added value of using predicted future loads to purchase a portfolio of reserved contracts and minimize infrastructure costs. The results of the following predicted input scenarios are compared in Section 6.5.

1. **Optimal:** This scenario is the “reference” scenario, in which the customer has full knowledge of the future, and selects an optimal portfolio based on the real future load.

2. **Prediction:** The customer purchases reserved contracts by taking into account the GP predicted load for the scheduling period of ρ days.

3. **One period back:** The customer purchases the same instances as were needed in last ρ days.

4. **One year back:** The customer purchases the same instances as were needed in the same period of ρ days exactly one year ago (assuming that that utilization levels would be the same as in the period one year ago).

In all experiments of this contribution, we use a prediction and scheduling period of $\rho = 90$ days. Although not veracious at the time of writing, it can be argued that a cloud provider could be interested in providing shorter-time reservation periods in addition to the 1-year and 3-year periods currently available at Amazon EC2. This would enable customers with highly varying load patterns, which are currently exclusively using on-demand instances, to also make up-front commitments for shorter periods. The support for multiple offerings with different periods and the incorporation of longer-term predictions is left for future work.

The inputs for the prediction scenario are calculated using symbolic regression via GP implemented in *DataModeler* ([112]). We use Pareto-aware symbolic regression to optimize both accuracy and simplicity of the models. The fitness function for prediction accuracy was defined as $1 - R^2$ where R^2 is the square of the scaled correlation between prediction and response. As complexity measure we use the sum of subtree sizes of the model expression, which is equivalent to the total number

[2] as in a model $Y_t = \mathcal{M}(t) + \sigma(t)\epsilon_t$

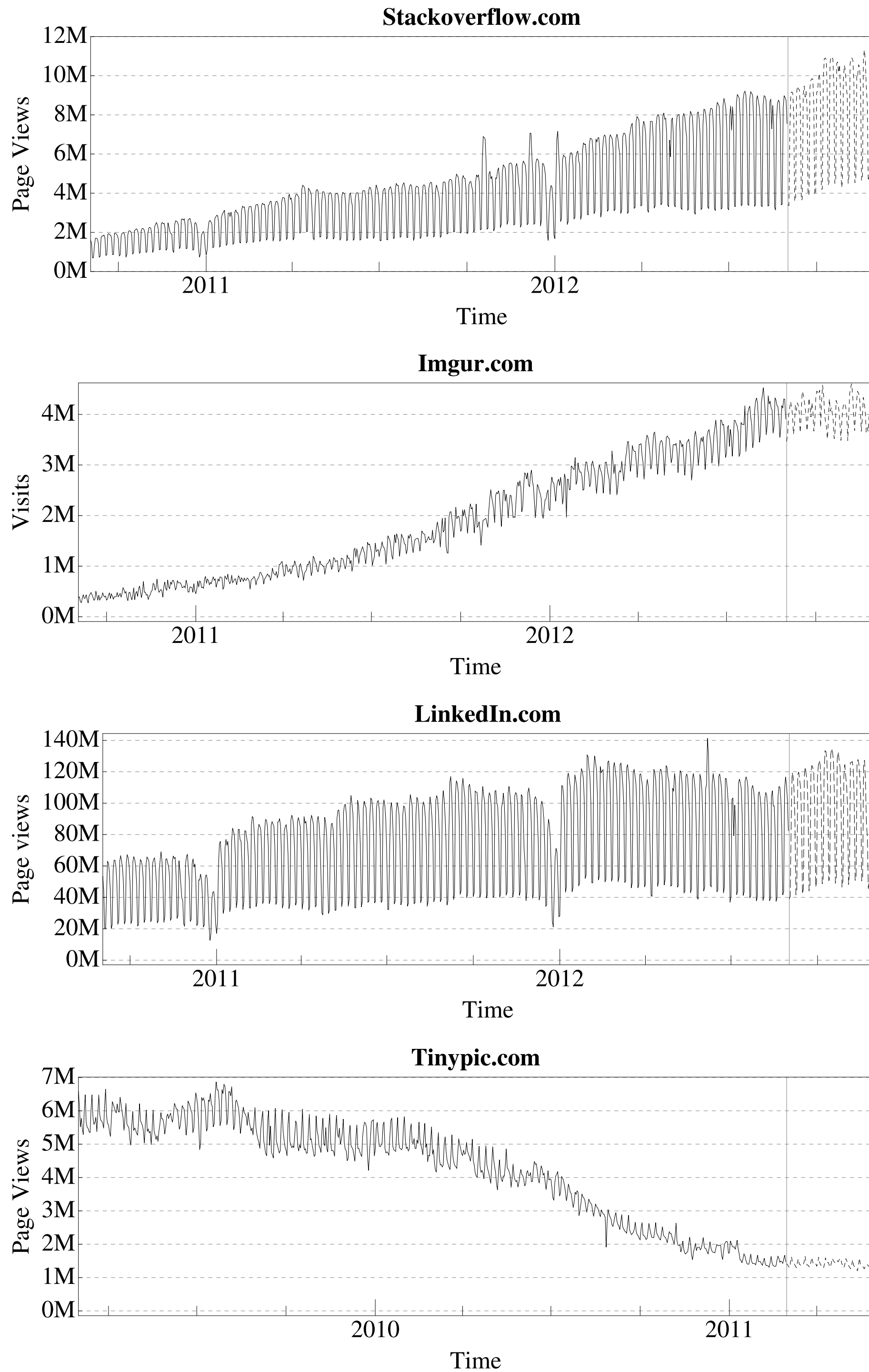


Figure 6.1: The data of the four websites plotted over time. Solid lines indicate the part of the data that was used as training set. Dashed lines indicate the part of the data that was used as test set.

of links traversed starting from the root to each of the terminal nodes of the parse tree.

In practice, data transformation and normalization methods precede analysis of time series ([106], [109]). We, however, applied symbolic regression to raw data to prepare for online modeling of streamed series.

To let GP automatically discover significant lags, we convert the original load vector to a data matrix in the following way: for a fixed length of a sliding time window m, each load value v_t was augmented to a row vector $(v_t, vt-1,\ldots,v_{t-m}) \in \mathbb{R}^{1\times m}$. Omitting the first m samples of the response vector $\mathbf{v} = (v_1,\ldots,v_T)^T$ (for which not all lags are available) leads to a data matrix with $T-m$ rows and m columns.

The lag parameter m is fixed to $m = 365$ days in all experiments.

To generate predictions 90 days ahead without iteratively accumulating prediction errors, we exclude lags $(v_{t-1},\ldots,v_{t-90})$ from the modeling and use the first column v_t of the augmented data table as the response. The modeling task for the lag window m and the scheduling horizon ρ is now formulated as a search for a model or a model ensemble f, such that:

$$v_t = f(v_{t-\rho-1}, v_{t-\rho-2}, \ldots v_{t-m}) + \epsilon, \quad m > \rho. \tag{6.3}$$

Each experiment presented here consists of two modeling stages of 40 independent symbolic regression runs, with each run executing for 900 seconds. The results of stage one experiments define the driving variables and variable combinations (optimal time lags), stage two experiments use only driving variables as inputs. Final solution ensembles were constructed from stage two models. Prediction is then evaluated on the test time period (see Figure 6.1), used for constructing an optimal contract portfolio, and compared with a baseline on-demand strategy applied to real loads in the test period.

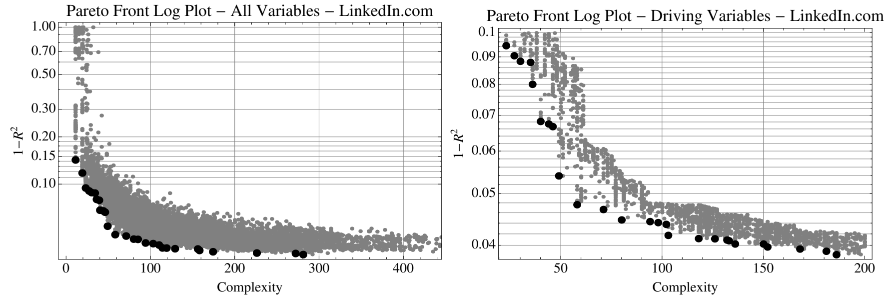


Figure 6.2: Pareto front plots showing the complexity accuracy profiles of the LinkedIn.com models. The left plot shows all models of the SR run. The right plot shows only the models using the driving variables for a quality box of complexity 200 and accuracy 0.1.

6.5 Results

SR results

Exclusion of the 90 most recent lags from the modeling allowed us to forecast the loads 90 days into the future using the available training data without the need

Usage level	Up-front cost	Hourly price	Period ρ	Charging method
Heavy	$42.25	$0.014	90 days	every hour
Medium	$34.75	$0.021	90 days	as-you-go
Light	$15.25	$0.034	90 days	as-you-go
On demand	N.A.	$0.06	N.A.	as-you-go

Table 6.1: Assumed cloud offerings for instance type `m1.small`.

to iterate predicted loads and accumulate prediction error. The quality of predictions was evaluated with respect to $1 - R^2$, the Normalized Root Mean Square Error (NRMSE) and the Hit percentage (HIT) ([109]).

The forecasts for the testing horizon for all four websites are depicted in Figure 6.3. The *stackoverflow.com* predictions are of high quality as the previously observed growth in page views (from Figure 6.1) continues in the test period. The *imgur.com* predictions do well for the first month of the predicted period, but afterwards the growth stagnates and the prediction overshoots because this behavior is not observed in the training data. The *linkedin.com* predictions follow the growth curve as observed in the training data and are therefore very close to the actual data. The *tinypic.com* data stagnates at the end of the training period and continues to do so in the test period. SR predictions follow the declining trend of the training set and therefore predict a page view count which is lower than the observed count. On the other hand, the predictions are still closer to the test data than the data of the previous period. Table 6.2 shows the prediction errors for both the training and test data.

For the sake of completeness, Figure 6.4 depicts a Pareto front of all models generated for *linkedin.com* data as well as a subset of models that use the driving variables.

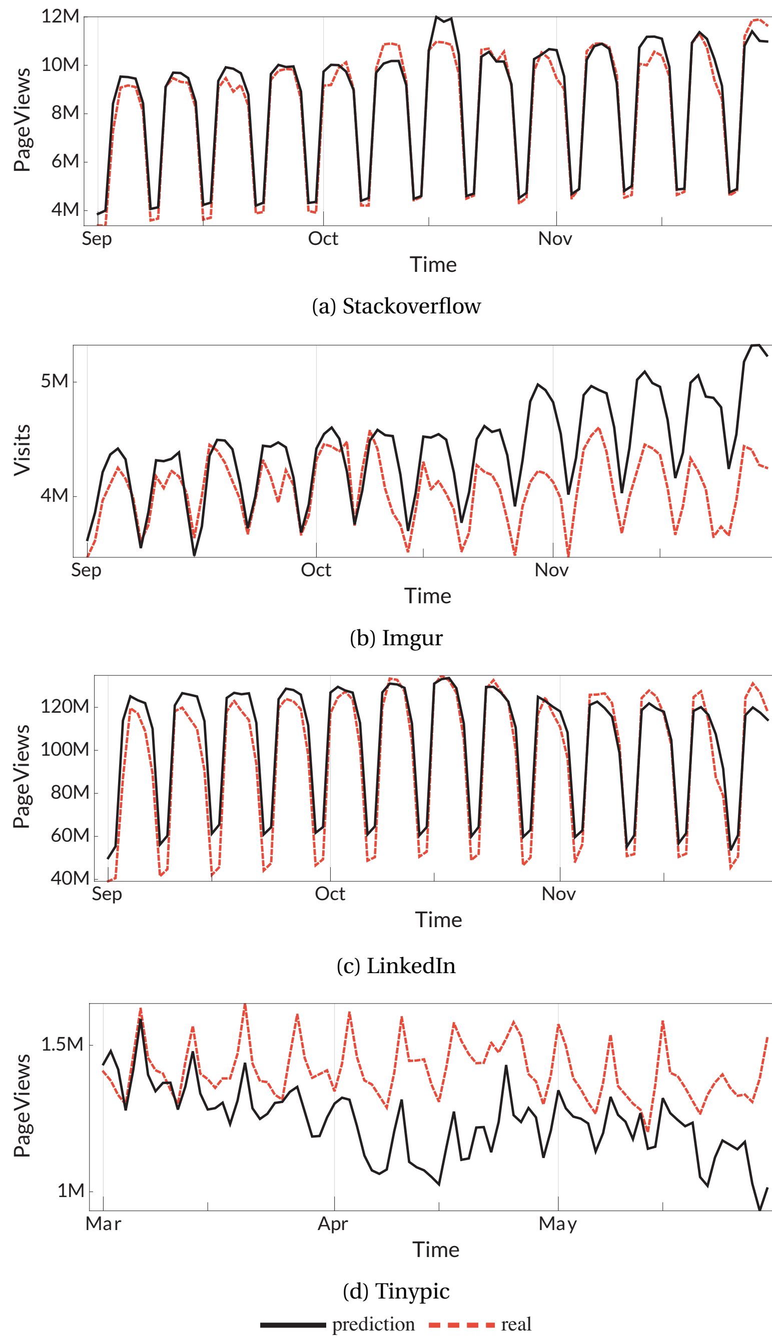
Validation through simulation

The main goal of this chapter is to explore the benefits of forecasting the computational load to make data-driven decisions about purchasing IaaS contracts. To achieve this goal, we compare three data-driven load forecasting scenarios to a straightforward paid on-demand service. The reference cost of an on-demand service for the scheduling period is estimated using real load from the test data. The three prediction scenarios employ loads predicted by GP, actual loads of the previous scheduling period, and actual loads of the same period one year back.

To map page views to the computational cost, we assume that the *page views per instance* ratio is known in advance for a certain instance type and that the ratio is constant for different page requests.

The prices for the different offerings are derived from the Amazon EC2 reserved prices for the `m1.small` instance type, in which the period is reduced to 90 days and the upfront cost is proportionally reduced with three quarters. These prices and conditions are listed in Table 6.1.

For each of the forecasting scenarios we construct an optimal contract portfolio and compare the cost of this portfolio to the baseline cost of the optimal portfolio



(a) Stackoverflow

(b) Imgur

(c) LinkedIn

(d) Tinypic

Figure 6.3: The GP prediction results for the four websites (shown in solid black). The actual page view data is indicated by the dashed red lines.

LinkedIn.com – Model Selection Report

	Complexity	$1-R^2$	Function
1	23	0.094	$4.666 + \frac{-28.093}{7+\text{delay}_{238}+\text{delay}_{364}}$
2	30	0.088	$1.657 + -0.098\,(-3.150 + \text{delay}_{238} + \text{delay}_{364})^2$
3	34	0.088	$0.733 + -0.123\,(\text{delay}_{238} + \text{delay}_{364})\,(-5 + \text{delay}_{287} + \text{delay}_{364})$
4	36	0.080	$6.325 + \frac{-58.134}{10+\text{delay}_{217}+\text{delay}_{238}-\text{delay}_{294}+\text{delay}_{364}}$
5	39	0.079	$0.617 + -0.147\,(\text{delay}_{238} + \text{delay}_{364})\,(-5 + \text{delay}_{287} + \text{delay}_{364}{}^2)$
6	40	0.071	$2.040 + -0.244\,(\text{delay}_{116} + \text{delay}_{275} + (2.021 - \text{delay}_{364})^2)$
7	46	0.066	$1.646 + -0.103\,(\text{delay}_{116} + \text{delay}_{275} + (-2.848 + \text{delay}_{238} + \text{delay}_{364})^2)$
8	49	0.063	$2.702 + -0.183\,(\text{delay}_{116} + \text{delay}_{296} + \text{delay}_{301}{}^2 + (3 - \text{delay}_{364})^2)$
9	53	0.062	$2.934 + -0.205\,(\text{delay}_{116} + \text{delay}_{275} + (3 - \text{delay}_{364})^2 + \text{delay}_{364} + \text{delay}_{364}{}^2)$
10	62	0.059	$1.012 + 0.260\,(-\text{delay}_{109} + \text{delay}_{238} - \text{delay}_{294}{}^2 - \text{delay}_{296} + 3\,\text{delay}_{364})$
11	63	0.052	$3.580 + -0.261\,(\text{delay}_{116} + \text{delay}_{296} + \text{delay}_{301}{}^2 + (3 - \text{delay}_{364})^2 + 2\,\text{delay}_{364})$
12	72	0.048	$3.000 + -0.215\,(\text{delay}_{116} - \text{delay}_{238} + \text{delay}_{275} + \text{delay}_{301}{}^2 + (2.999 - \text{delay}_{364})^2 + 2\,\text{delay}_{364})$
13	77	0.048	$1.058 + 0.215\,(-\text{delay}_{116} + \text{delay}_{238} - \text{delay}_{296} - \text{delay}_{301}{}^2 + 4\,\text{delay}_{364} - \text{delay}_{364}{}^2)$
14	115	0.048	$1.009 + 0.101\,(-2\,\text{delay}_{109} + \text{delay}_{217} + \text{delay}_{231} + \text{delay}_{238} - \text{delay}_{294}{}^2 - 2\,\text{delay}_{296} - 2\,\text{delay}_{301}{}^2 + 8\,\text{delay}_{364} - \text{delay}_{364}{}^2)$
15	129	0.046	$1.018 + 0.101\,(-\text{delay}_{109} - \text{delay}_{116} + \text{delay}_{231} + 2\,\text{delay}_{238} - \text{delay}_{275} - \text{delay}_{294}{}^2 - \text{delay}_{296} - 2\,\text{delay}_{301}{}^2 + 8\,\text{delay}_{364} - \text{delay}_{364}{}^2)$

Figure 6.4: Pareto front of models that use the driving variables – for the LinkedIn data.

constructed for real data. The results for all four websites are presented in Figure 6.5.

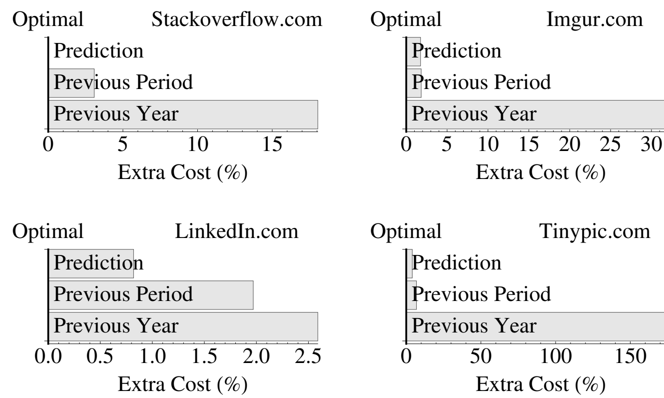


Figure 6.5: The extra cost percentage compared to the optimal, for hosting the four websites as calculated by the cloud scheduling simulator.

Figure 6.5 illustrates that optimal contracts generated using GP-based load forecasts (bars labeled as "Prediction") provide superior results with respect to cost benefits compared to the actual optimal purchasing decisions for all four websites. In three out of four cases GP-based contracts are significantly better than the contracts which anticipate the same loads as in the previous period of ρ days.

For clarity reasons, the cost when using only on demand instances is not shown in Figure 6.5, but they exceed the cost in the "optimal" reserved scenario by 52%, 72%, 51%, and 73% for Stackoverflow.com, Imgur.com, LinkedIn.com, and tinypic.com respectively. For the LinkedIn website the on-demand costs under our assumptions and prices from Table 6.1 would be \$117,094, the contract portfolio in the optimal scenario would cost \$77,338, and the prediction based contract for any GP-enthusiast would cost \$77,975.

These figures show that it is beneficial for IaaS customers to take into account reserved contracts and consider using prediction models of future load to make the right purchases. Genetic programming proves to be able to build such load prediction models fairly well.

Data set		$1-R^2$	**NRMSE**	**HIT**
Stackoverflow.com				
Training Data		0.0325	0.0509	0.838
Test Data	day 01-30	0.0073	0.0714	0.793
	day 31-60	0.0334	0.0726	0.826
	day 61-90	0.0331	0.0700	0.931
	day 01-90	0.0254	0.0576	0.843
Imgur.com				
Training Data		0.0499	0.0496	0.764
Test Data	day 01-30	0.2350	0.1941	0.862
	day 31-60	0.5354	0.3506	0.828
	day 61-90	0.3680	0.6109	0.828
	day 01-90	0.5258	0.4123	0.843
LinkedIn.com				
Training Data		0.0573	0.0612	0.942
Test Data	day 01-30	0.0276	0.1643	0.966
	day 31-60	0.0102	0.0886	0.897
	day 61-90	0.0401	0.0944	0.966
	day 01-90	0.0453	0.1080	0.944
Tinypic.com				
Training Data		0.0233	0.0431	0.805
Test Data	day 01-30	0.7009	0.3407	0.724
	day 31-60	0.7132	0.8235	0.621
	day 61-90	0.8601	0.5604	0.862
	day 01-90	0.8291	0.4756	0.741

Table 6.2: Overview of the prediction errors for the data sets. The prediction errors are show for the training data, the test data and the first, second and third 30 day period of the test data.

6.6 Conclusion

In this chapter we illustrated the value of data analysis for taking smarter data-driven decisions for the application of purchasing cloud computing services. We showed that the most successful strategy for selecting contract portfolios is based on upfront reservation of compute instances using forecasted utilization levels obtained by genetic programming.

We showed that symbolic regression via genetic programming (implemented in DataModeler) routinely filters out handfuls of driving variables out of several hundreds of candidate inputs. We also demonstrated the competitive advantage of running multi-objective symbolic regression to evolve not only accurate but also ‘short’ highly interpretable relationships.

For future work we will avoid the limiting decisions taken to design experiments in this chapter – fixing the sliding time window and maximal number of lags m, and fixing the length of the analysis window T. [113] proposed a good heuristic to estimate the optimal values for these parameters (albeit no lagged variables were used as potential inputs). We believe that both the analysis window as well as the sliding time window can be evolved in the same symbolic regression process.

CHAPTER 7

Optimizing Reserved Contract Procurement: A Case Study

This chapter is accepted for publication as "Optimizing IaaS Reserved Contract Procurement using Load Prediction", R. Van den Bossche, K. Vanmechelen, J. Broeckhove, Proceedings of the 7th IEEE International Conference on Cloud Computing, July 2014.

Abstract

With the increased adoption of cloud computing, new challenges have emerged related to the cost-effective management of cloud resources. The proliferation of resource properties and pricing plans has made the selection, procurement and management of cloud resources a time-consuming and complex task, which stands to benefit from automation. This contribution focuses on the procurement decision of reserved contracts in the context of Infrastructure-as-a-Service (IaaS) providers such as Amazon EC2. Such reserved contracts complement pay-by-the-hour pricing models, and offer a significant reduction in price (up to 70%) for a particular period in return for an upfront payment. Thus, customers can reduce costs by predicting and analyzing their future needs in terms of the number and type of server instances. We present an algorithm that uses load prediction with automated time series forecasting based on a Double-seasonal Holt-Winters model, in order to make cost-efficient purchasing decisions among a wide range of contract types while taking into account an organization's current contract portfolio. We analyze its cost effectiveness through simulation of real-world web traffic traces. Our analysis investigates the impact of different prediction techniques on cost compared to a clairvoyant predictor and compares the algorithm's performance with a stationary contract renewal approach. Our results show that the algorithm is able to significantly reduce IaaS resource costs through automated reserved contract procurement. Moreover, the algorithm's computational cost makes it applicable to large-scale real-world settings.

7.1 Problem Domain

Our problem domain concerns an organization that needs to execute workloads originating from a number of applications under its control. The organization relies on cloud provider(s) to handle (a part of) this workload and pursues the goal of optimizing its contract portfolio based on the current and future load generated by these applications.

We define an organization's *load* l_t^{it} as the aggregated number of instances of type it that it has running at time instant t. The internal resource usage of an instance and the type(s) of application(s) it executes are assumed to be unknown, as they depend on specific application-level domain knowledge. In addition, we assume that the workloads directed to instances of different types are independent of each other, as an application's migration of one instance type to another is a decision that should not be taken without domain knowledge and is out of scope of this thesis. This assumption is aligned with the Amazon EC2 configuration wherein a load balancer and auto-scaler collaborate to automatically scale up and down an instance pool of a given instance type in order to process the application workload in accordance with configured QoS constraints (e.g. response time). Consequently, we will omit the instance type qualification in our notation in what follows.

We model service levels and tariff structure as a generalization of the *on-demand* and *reserved* offerings of Amazon EC2. A contract type is modeled as $c = \langle n, u, h, l, p \rangle$, with c_n a unique name identifying the type, $c_u \geq 0$ the contract's *up-front cost*, c_h the *hourly rate*, c_l the contract's *length*, and c_p one of the following *charging policies*:

- AYG (**As-You-Go**) : After paying c_u, the customer is charged only for the actual usage of the instance. If an instance runs for h hours during the contract's lifetime, the customer is billed $c_u + h \times c_h$.
- EH (**Every Hour**) : After paying c_u, the customer is charged c_h per hour during the contract's lifetime, even if the instance is not running.

An "on-demand" policy as offered by EC2 can be modeled as $\langle OD, 0, c_h, \infty, AYG \rangle$.

A purchased reserved contract of type c is denoted by $\bar{c} = c \cup \langle ptime \rangle$ with $\bar{c}_{ptime}$ the contract's purchase time. On-demand contract types with $c_u = 0$ are available. Customers in possession of a set of reserved contracts R_t and a running server count l_t at time t, have the reserved instance hourly rates applied first as these are the cheapest. If $l_t \leq |R_t|$, all running instances are charged at hourly rates of the contracts in $|R_t|$. We thereby match instances to contracts in the order of the cheapest hourly rate contract first. If $l_t > |R_t|$, the additional instance(s) are billed at the on-demand contract's hourly rate for the given instance type.

In order to take advantage of reserved instances with an optimal purchasing plan, long-term predictions –as long as the longest available contract length γ– have to be made. Predictions of future load are made based on historical load data. Selecting the required historical data length to make reliable forecasts for seasonal data is hard, and depends on the amount of randomness in the data [114]. In this chapter, we assume that a load history with a duration of twice the longest seasonal period –i.e. 2 years– is available.

As we do not focus on the differences in availability, we assume that the capacity of the cloud provider's data centers is infinite, that boot times for instances are

negligible, and there is no difference in instance availability between on demand and reserved instances. We adopt the EC2 charging policy of rounding up usage of partial hours.

7.2 Purchase Management Algorithm

The scope of our Purchase Management Algorithm (PMA) is to decide on the number of new contracts purchased, on the type of the contract(s), and on the renewal of existing contracts. The PMA is *iterative*, in that it executes on a regular basis (e.g. daily, weekly, monthly, etc.).

PMA Inputs

The inputs of the algorithm are:

- C : A set of contract types available to the organization. The length c_l of the longest reserved contract type (contracts with $c_u > 0$) in C is defined as γ.
- l_{hist} : A set $\{l_{t-2\gamma}, l_{t-2\gamma+1}, ..., l_t\}$ per instance type with t the current time and l_i the number of running instances at time i.
- R : The set $\{R_t, R_{t+1}, ..., R_{t+\gamma}\}$ at current time t, with R_i the set of all acquired reserved contracts for which $\bar{c}_{ptime} + \bar{c}_l > i$.
- t_{NP} : The next time the PMA runs.

The algorithm's aim is to find out if the procurement of an additional reserved contract $\bar{c}$ will have a positive effect on the total cost. This is the case if $\bar{c}_u$ is less than the total reduction in hourly costs during $\bar{c}_l$, achieved by purchasing $\bar{c}$. In this respect, dealing with a contract with the *Every Hour* charging policy is different from dealing with an *As-You-Go* policy. We therefore extend each contract type c with two additional properties: the *PMA hourly cost* (h_{pma}) and the *PMA up-front fee* (u_{pma}). $c_{h_{pma}}$ (Equation 7.1) denotes the cost of running an instance for one time unit *given that the purchase of the contract has already been made.* $c_{u_{pma}}$ (Equation 7.2) represents the minimal cost due when purchasing the reserved contract, regardless of the number of hours the associated instance will be running.

$$c_{h_{pma}} = \begin{cases} c_h & : c_p = AYG \\ 0 & : c_p = EH \end{cases} \tag{7.1}$$

$$c_{u_{pma}} = \begin{cases} c_u & : c_p = AYG \\ c_u + c_l c_h & : c_p = EH \end{cases} \tag{7.2}$$

Contract Expiry

In order to decide whether or not to buy or renew reserved contracts at the current time t_{cur}, we need to determine when and for how long the already purchased contracts are in force. Therefore, the algorithm constructs a *current contract state* (CCS) data structure based on R, that stores for every time step $t \in [t_{cur}, t_{cur} + \gamma]$ a sorted list of $\bar{c}_{h_{pma}}$ for every $\bar{c} \in R_t$. This allows for an efficient calculation of the cost

of running n instances at a time t by summation of the first n elements in the list for time step t. If the length of the list is smaller than n, it is padded with additional on-demand hourly contract fees prior to the summation.

If only the currently purchased contracts are taken into account when building the CCS at time t_{cur}, the algorithm neglects the potential to renew contracts in the future. If the predicted load remains constant or increases, this causes the calculated utilization over a potential contract's length to increase, as the non-overlapping period seems uncovered by a reserved contract. This in turn means that a cost-based algorithm –such as the one presented in this chapter– will tend towards buying contracts with a higher optimal utilization range and corresponding higher price.

Therefore, we introduce two *CCS contract views*:

- **Finite contracts** (FC): In this view, contracts are added to the CCS only in the time steps between $t \in [t_{cur}, \bar{c}_{ptime} + \bar{c}_l]$
- **Infinite contracts** (IC): Here, contracts in the distant future are assumed to be renewed automatically, and the consideration of whether the renewal of a contract is appropriate will only be made in the iteration of the algorithm just before the expiration of the contract. In the construction of the CCS, contracts with $\bar{c}_{ptime} + \bar{c}_l > t_{NP}$ are in this view treated as if $\bar{c}_l = \infty$, and are added to every list in the CCS. The $\bar{c}_{h_{pma}}$ value of reserved contracts expiring before t_{NP} are added only to the list of each time step $t \in [t_{cur}, \bar{c}_{ptime} + \bar{c}_l]$.

  Figure 7.1 illustrates this approach with $|R_{t_{cur}}| = 2$. Contract 1 has $\bar{c}_p = EH$, and therefore $\bar{c}_{h_{pma}} = 0$. It ends before t_{NP}. Contract 2 has $\bar{c}_p = AYG$ with $\bar{c}_{h_{pma}} = 0.034$, and ends after t_{NP}.

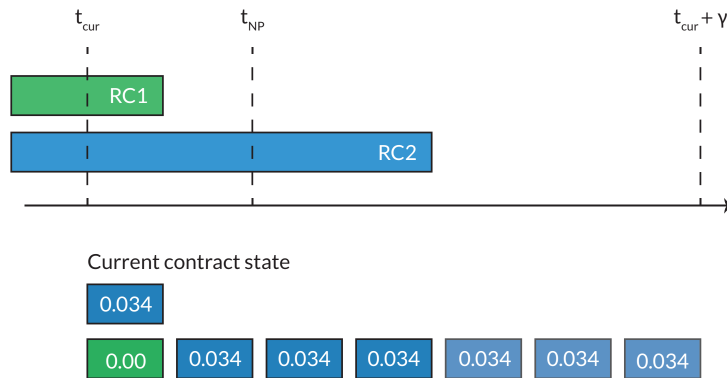


Figure 7.1: Example of a Current contract state (CCS) with “Infinite Contracts” view.

Operation

In each iteration, at time step t_{cur}, the algorithm uses a prediction technique to predict the future load $\hat{l} = \{\hat{l}_{t_{cur}+1}, \hat{l}_{t_{cur}+2}, ..., \hat{l}_{t_{cur}+\gamma}\}$, based on l_{hist}. Then, $\hat{l}$ is used

to evaluate whether augmenting R_h with an additional reserved contract is cost-effective. This is done by first calculating the expected cost, based on $\hat{l}$, given that R_h remains unaltered. Subsequently, for every $c \in C$, the total expected cost is calculated when adding a new contract of type c to R_h for each $h \in [t_{cur}, t_{cur} + c_l]$. The expected costs for adding a contract c are calculated in two ways:

- **Total cost** For each time step $t \in [t_{cur}, t_{cur} + \gamma]$, the aggregated cost per time step is calculated using the CCS and the predicted load $\hat{l}_t$ by taking the sum of the first $\hat{l}_t$ elements of the ordered list at time t in the CCS.
- **Next Purchase Time** For each time step $t \in [t_{cur}, t_{NP}]$, the aggregated cost is calculated similar to the Total cost, but instead of adding $c_{u_{pma}}$ to the subtotal, it is being prorated by adding $\frac{c_{u_{pma}} \times (t_{NP} - t_{cur})}{c_l}$.

This results in a set of $2 \cdot |C|$ aggregated cost values, two for each reserved contract and two for the scenario in which excess demand is covered with the on-demand contract. At this point in the algorithm, two decisions are taken:

1. Is it cost-beneficial to add the most cost-effective reserved contract, taking into account the resulting costs for the period $[t_{cur}, t_{cur} + \gamma]$?
2. Is it cost-beneficial to add that reserved contract when considering only period $[t_{cur}, t_{NP}]$?

For making Decision 1, the total cost when adding the most cost-effective contract $c \in C$ is compared to the status quo scenario. If a (long-term) cost reduction is achieved when purchasing c, the impact of the purchase is evaluated for the shorter-term in $[t_{cur}, t_{NP}]$. This two-stage decision process avoid premature purchases; if c does not yield an expected cost reduction for the short-term, it is better to postpone the acquisition, thereby increasing the accuracy of the load predictions.

If acquiring c yields a cost gain both short- and long-term, it is added to the list of purchase suggestions and added to R_h for every $h \in [t_{cur}, t_{cur} + c_l]$. Next, the CCS is recalculated and the algorithm iterates to evaluate whether additional contract purchases lead to further cost reductions. If the answer to Decision 1 or Decision 2 is negative, the PMA finishes and returns the list of purchase suggestions.

7.3 Experimental Setup

In order to evaluate the potential for cost savings with the PMA, a Python-based simulator is used to simulate the performance of the algorithm for a specified workload trace. The simulator executes a scenario in which contracts are bought according to the suggestions made by the PMA, instances are spun up and down based on the actual load at a given time (l_t), and the total cost for the scenario is calculated based on the contract portfolio available at every moment in time between the start and end of the simulation.

In a simulated environment, it is important to select a setting that models the real-world as closely as possible. This experimental setup focuses on the deployment of real-world web application workload traces on a cloud provider which resembles Amazon EC2.

Workload

To evaluate the PMA, we require a number of real-world load traces with diverse characteristics; with and without trends and seasonal behavior, and of different magnitude with respect to server load. The traces have to be of significant length as a load history of two years is required.

Unfortunately, long-term cloud usage traces are unavailable, as they are not published by cloud providers or organizations. Some cluster and grid job traces can be found at the Parallel Workloads Archive (PWA) [115], the Grid Workloads Archive (GWA) [82] and the Google Cluster Trace (GTC) [116]. These job traces however do not fit our needs, as they are relatively short –only three traces from the PWA are longer than 3 years– and their load is upper bounded by the available resources in the cluster or grid.

We therefore rely on data provided by Quantcast[1], an advertising company that measures the audience of a large number of websites. Audience reports, containing a historical view of the number of visits and page views per day, are freely available on their website.

Our experiments utilize these page view reports to build a server instance history that serves as input for the simulator. From the available reports, we selected four well-known web applications with different load characteristics:

1. **linkedin.com:** The page view data shows a strong seasonal pattern with drops in the weekends and holidays and a robust growth until the beginning of 2012, which stagnates in the last 2 years.

2. **stackoverflow.com:** The page view data shows robust growth with a lower number of views in the weekends and during the Christmas period.

3. **time.com:** The data for this website shows a relatively flat trend with multiple positive outliers.

4. **tinypic.com:** The page view data shows rapid decline with no strong seasonal pattern.

Each of these page view reports is converted into a server load pattern assuming a linear relation between the number of views, HTTP requests and server instances required to handle those requests. The page view reports contain daily aggregated data. Since the addition of artificial day-night patterns would reduce the significance of the real-world data without adding value to the evaluation of the algorithm, we assume the load is constant through the day. This does not impact our results because of the different timescales involved. We further assume that the web application is deployed on instances of a single type and adopt the `m1.xlarge` instance type, which is commonly used for web servers.

Given an average number of HTTP requests per page view (RPP) and a number of requests a single `m1.xlarge` server can handle (RPS), the number of page views p_t at time t can be converted to a number of server instances l_t following Equation 7.3.

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

$$l_t = \frac{p_t \cdot RPP}{RPS} \qquad (7.3)$$

According to [117], an average page view results in an RPP value of 44.56 GET requests. The number of requests per second (RPS) a server instance is able to handle is subject to a number of application-specific parameters, which are hard to determine without application knowledge or benchmarking abilities. For example, the mix of static versus dynamic content, the number of database requests and the caching techniques used by an application are unknown. As this number has no influence on our comparative results and only affects the absolute cost figures, we assume that for one `m1.xlarge` server, $RPS = 50$. The resulting load patterns used for our evaluation are presented in Figure 7.2.

In all experiments, the algorithm is executed weekly in a period between two years after the beginning of each trace and one year before the end of the trace.

Contract types

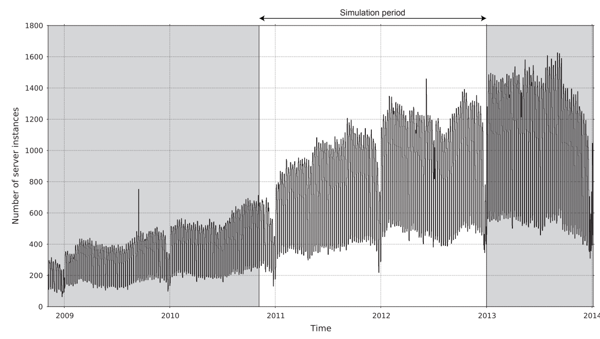
The on-demand prices and reserved contracts for an EC2 `m1.xlarge` instance are configured according to their actual prices. In the evaluation of our algorithm, we assume that only reserved contracts with a duration of one year are available, because of the unavailability of real-world trace data available for a longer period. We also believe that making predictions for a period of three years would likely call for additional domain and business-specific knowledge, as well as manual time series modeling instead of relying solely on automatic forecasting methods; an avenue for future work. The set of reserved contract types C used in the evaluation can be found in Table 7.1.

$\bar{c}_n$	$\bar{c}_u$	$\bar{c}_h$	$\bar{c}_l$	$\bar{c}_p$
Heavy	\$1352	\$0.112	1 year	EH
Medium	\$1108	\$0.168	1 year	AYG
Light	\$486	\$0.271	1 year	AYG
On Demand	\$0.0	\$0.480	1 year	AYG

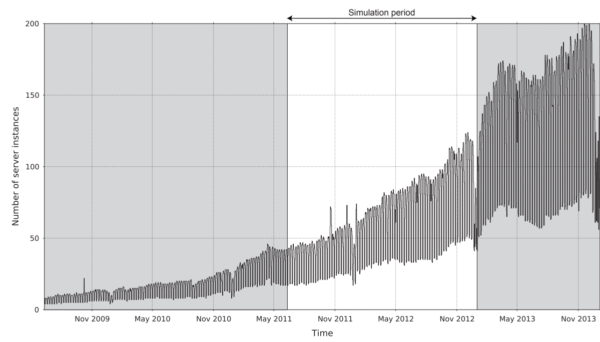
Table 7.1: Available contract types for `m1.xlarge`.

Scenario setup

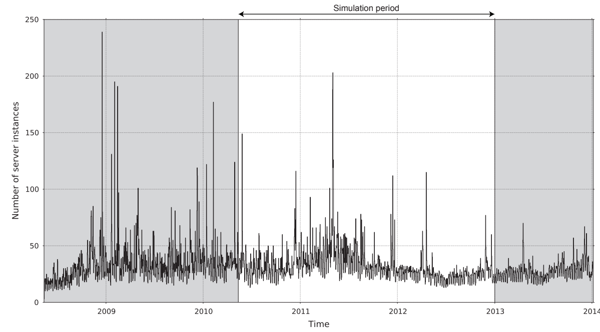
The load traces' non-zero starting point require an *initial contract portfolio* (ICP). This portfolio is a cost-optimized set of contracts: 7 days before the start of the simulation, the optimal set of reserved contracts for the last 90 days is calculated. We distribute the start times of the contracts in the ICP uniformly between 7 days and 371 days prior to the simulation's start time, in order to avoid that all contracts end at the same time. This would overemphasize the importance of the load prediction and procurement choices at that particular time, skewing the results.



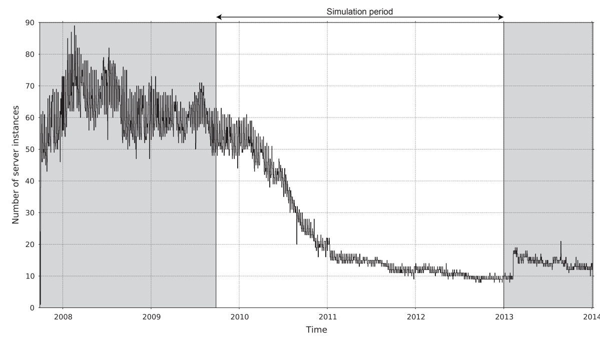
(a) Linkedin.com



(b) Stackoverflow.com



(c) Time.com



(d) Tinypic.com

Figure 7.2: The server instance traces

Prediction techniques

The added value of the prediction techniques in the PMA depends on their ability to capture the general trend of the organization's load as well as their ability to incorporate seasonal behavior, such as weekly or yearly recurring patterns. This allows for a correct estimation of the number of required instances, and its increase or decrease over time. Predicting the size and frequency of the load fluctuations on the other hand, is required to optimize contract choices with respect to the utilization level of the contract (cfr. Table 7.1).

The load patterns are a typical example of time series [118, 119]. *Time series analysis and forecasting* deals with the use of statistical models to forecast values of a future time based on the values observed in the past. In using time series models for prediction purposes, the model selection, parameter estimation and forecasting are complex and time-consuming. However, progress has been made in the accuracy of automated forecasting methods [120, 121, 122].

In order to evaluate the performance of PMA with both perfect load predictions and with different prediction techniques, the following predictors are compared in Section 7.4:

1. **Full Knowledge (FK):** This predictor has access to the real load pattern. The outcome of PMA with this predictor serves as a benchmark.

2. **Previous Period (PP):** Extrapolates the most recent period of length δ to the predicted values by simply 'repeating' the period. If δ is chosen wisely, the resulting load pattern includes simple seasonal patterns, such as a weekly or monthly patterns. This technique corresponds to the manual approach in which someone calculates the past utilization in period δ, and purchases contracts accordingly. In our evaluation, we use the PP predictor with δ equal to 7 days and 30 days.

3. **Double-Seasonal Holt-Winters (DSHW):** Taylor's DSHW method [123] decomposes the time series into a trend and two seasons, and uses *exponential smoothing* [124, 119] to construct the forecast. We use an automatic forecasting implementation of DSHW available in the *Forecast* package for R [125, 120]. We configure the method to use both weekly and yearly seasonality.

Cost calculation

In order to reliably compare the cost-effectiveness of different scenarios, the aggregated cost has to be carefully calculated, given the limited timespan of the simulation. If the PMA acquires a new reserved contract on the last day of the simulation for example, attributing the full up-front cost to the scenario's total cost would result in a negative bias, as the purchase only becomes profitable in the time period after the end of the simulation. Consequently, we introduce a cost calculation method that spreads the up-front cost for each purchase over the whole period in which it is valid. Therefore, for all contracts in which the period between $\bar{c}_{ptime}$ and $\bar{c}_{ptime} + \bar{c}_l$ does not overlap entirely with the simulation period, the up-front cost $\bar{c}_u$ and, in case of $\bar{c}_p = EH$, the hourly rates $\bar{c}_h$ are only accounted for in proportion to the time in which the contract period and simulation period overlap.

7.4 Results

We discuss the outcome of two experiments. First, for each load trace, the cost level achieved by PMA is compared to the cost incurred when simply renewing the contracts in the ICP as they expire. We use this *Renewal Only* (RO) as a reference point for our evaluation. We first configure the PMA with the Full Knowledge predictor, so it operates under "best case" conditions, and compare the results for both a CCS with a "finite contract" (PMA+FC) and CCS with a "infinite contract" (PMA+IC) view. Figure 7.3 shows the cost reduction of PMA relative to the cost of the RO policy.

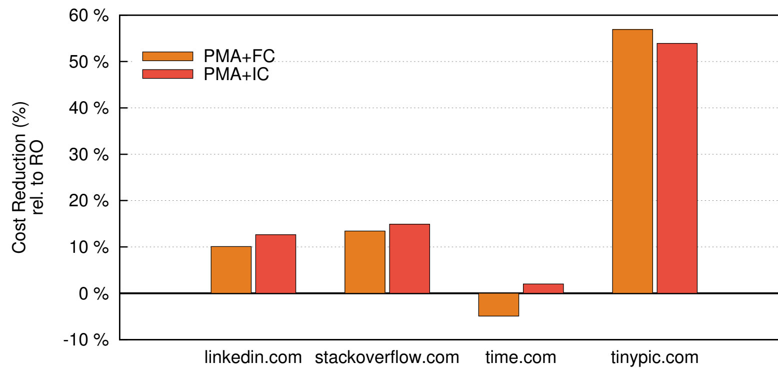


Figure 7.3: Cost reduction of PMA with Full Knowledge predictor, relative to the cost of RO.

In each of the load traces, PMA+IC outperforms RO in terms of cost. In comparison with the PMA+FC algorithm, PMA+IC performs better in all cases except for the *tinypic.com* trace. The cost reduction with respect to RO is the smallest (2.02%) in the *time.com* case, which is to be expected given its stationary character. Here, PMA+IC outperforms PMA+FC with 6.9% by assuming the renewal of existing contracts. At both *linkedin.com* and *stackoverflow.com*, PMA+IC achieves a significant cost reduction (12.6% and 14.9%) here compared to RO, and also does slightly better (+2.6% and +1.5%) than PMA+FC. In these patterns, which exhibit a sustained growth with a strong seasonal pattern, the potential cost reduction is substantial, and PMA+IC manages to compose a better contract portfolio in terms of purchase time and utilization level because of the delayed renewal decision for existing contracts. For *tinypic.com*, PMA+FC scores better than PMA+IC in terms of cost. The sharp decline in the number of server instances causes this; a contract state view in which contracts run infinitely causes the algorithm to underbuy: too few contracts and contracts with a smaller optimal utilization are bought. Compared to RO, PMA+FC obtains a significant cost reduction of 56.9%, which is due to the utilization decrease. The difference between both views for the different workload traces shows the need for a hybrid technique that automatically selects the best view depending on the current load trend. The development thereof is left for future work.

The difference in optimization potential and the impact of prediction errors between workloads with a rising and falling trend, appears to be substantial. This is

inherent to the pricing structure of reserved contracts and our iterative approach: an increasing load implies the purchase of additional contracts, a decision that can be made at every iteration of the algorithm. In an increasing trend, or when the predicted load is below the real server load, the excess demand can be covered with on-demand instances for a limited surplus in cost. However, when the trend decreases or the server load is overestimated, the up-front commitments of reserved contracts offset the gains from their lower hourly rates.

The second experiment compares the three prediction techniques discussed in Subsection 7.3. In Figure 7.4, the cost when using PMA with the Previous Period (7 and 30 days) and Double-Seasonal Holt-Winters prediction techniques is shown relative to the cost with the Full Knowledge predictor, for the various workload traces. The CCS in the algorithm is configured with the best performing view for each of the traces, which is FC for *tinypic.com* and IC for all others. Larger values represent a higher cost, relative to the cost of the full knowledge scenario at 0%.

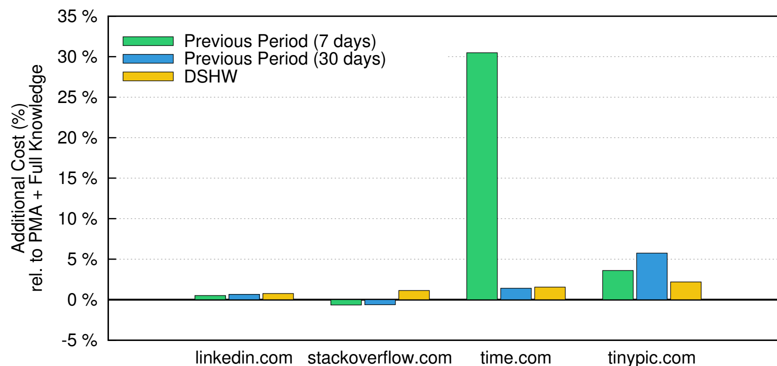


Figure 7.4: Cost of PMA with different prediction techniques, relative to that of PMA with Full Knowledge predictor.

In both the *linkedin.com* and *stackoverflow.com* cases, the three prediction techniques exhibit a cost effectiveness close (< 1%) to that of the Full Knowledge scenario. The small cost difference in favor of the PP techniques in *stackoverflow.com* is caused by one overprediction made by the DSHW technique at the end of the simulation period, which resulted in the purchase of a number of unused light instances. In the *time.com* case, it is remarkable that –compared to the DSHW predictor– the 7-day PP predictor performs significantly worse. There is a large number of spikes in that load trace and the PP technique erroneously assumes that these will reoccur. The statistical Holt-Winters approach on the other hand distinguishes between one-time and seasonal spikes, and considers the former as noise. Finally, in the *tinypic.com* trace, DSHW also outperforms the PP prediction techniques but still remains 2.19% more expensive than the Full Knowledge case.

We executed the experiments on a machine with four 12-core AMD Opteron 6234 processors and 196 GB of memory. In the *linkedin.com* case, the duration for one run of the purchase management algorithm has a 95% confidence interval between 4.97 and 106.26 seconds of CPU time, with an average runtime of 28.96 seconds. Calculating the predictions with the DSHW technique takes on average

23.57 seconds, with a 95% confidence interval of [19.48, 28.32]. These numbers show that the PMA could be executed regularly and on a large scale, and its overhead does not outweigh the potential cost savings.

The reported results depend on the start times of the contracts in the *initial contract portfolio*, which are generated according to a uniform distribution. In order to evaluate the significance of the presented results, we ran 100 iterations with a different random seed for every scenario with the *linkedin.com* workload trace. Relative standard deviations for the calculated cost of all prediction techniques were below 0.11%, the relative standard deviations of the cost relative to the stationary scenario remained below 0.76%.

7.5 Conclusion

This chapter tackles the problem of the selection, procurement and management of reserved IaaS contracts by introducing a *purchase management algorithm*. The algorithm iteratively produces contract procurement suggestions based on server load predictions while taking into account a current contract portfolio. Using simulation, we evaluated the cost reduction the algorithm achieves compared to a stationary approach. We further quantified the cost differences between (a) full knowledge predictions, (b) an automated forecasting approach using a Double-seasonal Holt-Winters model and (c) a reactive approach that extrapolates past observations. Our results show that the algorithm has a significant potential for cost reduction when provided with good predictions and the right renewal approach, which is different for load with a rising or falling trend. The results also show that the DSHW method is a promising prediction technique when used in combination with the procurement algorithm. The next chapter covers the automated selection of a renewal approach and a more comprehensive statistical evaluation using a large number of workload traces.

CHAPTER 8

Reserved Contract Procurement Heuristic with Load Prediction

This chapter is under review as "IaaS Reserved Contract Procurement Optimisation with Load Prediction" for publication in Future Generation Computer Systems, R. Van den Bossche, K. Vanmechelen, J. Broeckhove.

Abstract

With the increased adoption of cloud computing, new challenges have emerged related to the cost-effective management of cloud resources. The proliferation of resource properties and pricing plans has made the selection, procurement and management of cloud resources a time-consuming and complex task, which stands to benefit from automation. This chapter focuses on the procurement decision of reserved contracts in the context of Infrastructure-as-a-Service (IaaS) providers such as Amazon EC2. Such reserved contracts complement pay-per-hour pricing models, offering a significant price reduction in exchange for an upfront payment. This chapter evaluates whether customers can reduce costs by predicting and analyzing their future needs in terms of the number of server instances. We present an algorithm that uses load prediction to make cost-efficient purchasing decisions, and evaluate whether the use of automated time series forecasting proves useful in this context. The algorithm takes into account a wide range of contract types as well as an organization's contract portfolio, and load predictors based on Holt-Winters and ARIMA models are used. We analyze its cost effectiveness through simulation of an extensive amount of real-world web traffic traces. Results show that the algorithm is able to significantly reduce IaaS resource costs through automated reserved contract procurement, but that the use of advanced prediction techniques only proves beneficial in specific cases.

8.1 Problem Domain

Our problem domain concerns an organization that needs to execute workloads originating from a number of applications under its control. The organization relies on cloud provider(s) to handle (a part of) this workload and pursues the goal of optimizing its contract portfolio based on the current and future load generated by these applications.

We define an organization's *load* l_t^{it} as the aggregated number of instances of type it running at time instant t. The internal resource usage of an instance and the type(s) of application(s) it executes are assumed to be unknown, as they depend on specific application-level domain knowledge. In addition, we assume that the workloads directed to instances of different types are independent, as an application's migration of one instance type to another is a decision that should not be taken without domain knowledge and is out of scope of this thesis. This assumption is aligned with the Amazon EC2 configuration wherein a load balancer and auto-scaler collaborate to automatically scale up and down an instance pool of a given instance type in order to process the application workload in accordance with configured QoS constraints (e.g. response time). Consequently, we will omit the instance type qualification in our notation in what follows.

We model service levels and rates as a generalization of the *on-demand* and *reserved* offerings of Amazon EC2. A contract type is modeled as $c = \langle n, u, h, l, p \rangle$, with c_n a unique name identifying the type, $c_u \geq 0$ the contract's *up-front cost*, c_h the *hourly rate*, c_l the contract's *length*, and c_p one of the following *charging policies*:

- AYG (**As-You-Go**) : After paying c_u, the customer is charged only for the actual usage of the instance. If an instance runs for h hours during the contract's lifetime, the customer is billed $c_u + h \times c_h$.
- EH (**Every Hour**) : After paying c_u, the customer is charged c_h per hour during the contract's lifetime, even if the instance is not running.

An "on-demand" plan can be expressed as $od = \langle OD, 0, h, \infty, AYG \rangle$. We assume that an on-demand plan is available for each instance type. A purchased reserved contract of type c is denoted by $\bar{c} = c \cup \langle ptime \rangle$ with $\bar{c}_{ptime}$ the contract's purchase time. Customers in possession of a set of reserved contracts R and a running server count l_t at time t, have the reserved instance hourly rates applied first as these are the cheapest. If $l_t \leq |R|$, all running instances are charged at hourly rates of the contracts in $|R|$. We thereby match instances to contracts in the order of the cheapest hourly rate contract first. If $l_t > |R|$, the additional instance(s) are billed at the on-demand contract's hourly rate od_h. This approach is similar to the EC2 billing process.

In order to optimize the acquisition of reserved instances, long-term predictions –as long as the longest available contract length γ– have to be made. We predict future load using historical load data. Determining the history size required to make reliable forecasts for seasonal data is hard, and depends on the amount of randomness in the data [114]. We assume that a load history with a duration of twice the longest seasonal period –i.e. 2 years– is available.

As we do not focus on the differences in availability, we assume that the capacity of the cloud provider's data centers is infinite, that boot times for instances are

negligible, and there is no difference in instance availability between on-demand and reserved instances. We adopt the EC2 charging policy of rounding up usage of partial hours.

8.2 Purchase Management Algorithm

The scope of our Purchase Management Algorithm (PMA) is to decide on the number of new contracts purchased, on the type of the contract(s), and on the renewal of existing contracts. The PMA is *iterative*, in that it executes on a regular basis (e.g. daily, weekly, monthly, etc.).

PMA Inputs

The inputs of the algorithm for a given instance type are:

- C : A set of available reserved contract types. The length c_l of the longest contract type in C is defined as γ.
- od : The on-demand contract.
- l_{hist} : A set $\{l_{t-2\gamma}, l_{t-2\gamma+1}, ..., l_t\}$ with t the current time and l_i the number of running instances at time i.
- R : The set of all acquired reserved contracts.
- t_{NP} : The next time the PMA runs.
- PLP : The period in time (*purchase lookahead period*) for which possible purchases are considered relevant.

Operation

In each iteration, at time step t_{cur}, the algorithm uses a prediction technique to predict the future load $\hat{l} = \{\hat{l}_{t_{cur}+1}, \hat{l}_{t_{cur}+2}, ..., \hat{l}_{t_{cur}+\gamma}\}$, based on l_{hist}. Then, $\hat{l}$ is used to evaluate whether augmenting R with an additional reserved contract is cost-effective. For every $c \in C \cup \{od\}$, the total expected cost is calculated when adding c to the current set of purchased contracts R. If c is od, the calculated cost corresponds to the scenario in which no additional contracts are purchased. The expected costs for adding a contract c are calculated in two ways:

- **Total cost** The total cost is calculated by adding the upfront cost of c with the aggregated cost per time step (for each time step $t \in [t_{cur}, t_{cur} + \gamma]$) taking $R \cup c$ as the set of available contracts.

$$Cost_{total}(t_{cur}, R, l_{hist}, c, od) = c_u + \sum_{t=t_{cur}}^{t_{cur}+\gamma} CostPerTimestep(t, R \cup \{c\}, \hat{l}, od) \quad (8.1)$$

- **Purchase Lookahead Period** The cost for the period up to PLP is calculated similar to the total cost, except for the upfront cost. Instead of adding c_u to the subtotal, it is being prorated with regard to the length of PLP.

$$\begin{aligned} Cost_{PLP}(t_{cur}, R, l_{hist}, c, od) = & \frac{PLP}{c_l} \cdot c_u \\ & + \sum_{t=t_{cur}}^{t_{cur}+PLP} CostPerTimestep(t, R \cup \{c\}, \hat{l}, od) \end{aligned} \quad (8.2)$$

The details of the *CostPerTimestep*-function are explained in Section 8.2.

These cost calculations result in a set of $2 \cdot (|C| + 1)$ aggregated cost values, two for each reserved contract and two for the scenario in which excess demand is covered with on-demand instances. At this point in the algorithm, two decisions are taken:

1. Is it cost-beneficial to add the most cost-effective reserved contract, taking into account the resulting costs for the period $[t_{cur}, t_{cur} + \gamma]$?
2. Is it cost-beneficial to add that reserved contract when considering only period $[t_{cur}, t_{cur} + PLP]$?

For making Decision 1, the total cost when adding the most cost-effective contract $c \in C$ is compared to the status quo scenario. If a (long-term) cost reduction is achieved when purchasing c, the impact of the purchase is evaluated for the shorter-term in $[t_{cur}, t_{cur} + PLP]$. This two-stage decision process avoids premature purchases; if c does not yield an expected cost reduction for the short-term, it is better to postpone the acquisition, thereby increasing the accuracy of the load predictions.

If acquiring c yields a cost gain both short- and long-term, the corresponding contract $\bar{c}$ with $\bar{c}_{ptime} = t_{cur}$ is added to the set of purchase suggestions and to R. Next, the algorithm iterates to evaluate whether additional contract purchases lead to further cost reductions. If the answer to Decision 1 or Decision 2 is negative, the PMA finishes and returns the set of purchase suggestions. Note that it is possible to optimize compute times of the algorithm by skipping the $Cost_{total}$ calculations if the most cost-beneficial plan up to PLP is od, because the answer to Decision 2 is negative then anyway.

Cost Calculation per time step

When calculating the cost per time step, dealing with a contract with the *Every Hour* charging policy is different from dealing with an *As-You-Go* policy. With the *Every Hour* policy, the purchase decision comprises a commitment in terms of cost of both the upfront fee and the hourly fee for every hour in the contract's period. In our cost calculations, we update the values of c_h and c_u of contracts with $c_p = EH$ to reflect the monetary commitment the purchase decision includes. For these contracts, c_h is set to 0, as running an instance is free once the purchase is made, and c_u is set to $c_u + c_l \cdot c_h$, which represents the minimal cost due when purchasing the reserved contract, regardless of its utilization.

In order to calculate the $CostPerTimestep(t, R, \hat{l}, od)$ at time t, we need to determine which of the already purchased contracts are in force. Therefore, the

algorithm turns the set of contracts $\bar{c} \in R$ that are in effect at time t into a sequence sorted by the contracts' hourly cost $\bar{c}_h$. This *current contract state* (CCS_t) is defined in Equations 8.3 and 8.4. This allows for an efficient calculation of the cost of running $\hat{l}_t$ instances at time t by summation of the first $\hat{l}_t$ elements in the sequence. If $|CCS_t| < \hat{l}_t$, the sum is augmented with $\hat{l}_t - |CCS_t|$ on-demand hourly contract fees od_h prior to the summation.

$$CCS_t = \langle \bar{c}_h \mid \forall\ \bar{c} \in R : \bar{c}_{ptime} \leq t < \bar{c}_{ptime} + \bar{c}_l \rangle \tag{8.3}$$

$$\forall i : CCS_t(i) \leq CCS_t(i+1) \tag{8.4}$$

$$CostPerTimestep(t, R, \hat{l}, od) = \begin{cases} \sum_{i=1}^{\hat{l}_t} CCS_t(i) & : \quad |CCS_t| \geq \hat{l}_t \\ od_h \cdot (\hat{l}_t - |CCS_t|) + \sum CCS_t & : \quad |CCS_t| < \hat{l}_t \end{cases} \tag{8.5}$$

The algorithm presented in Section 8.2 greedily takes purchase decisions at time t_{cur}, thereby neglecting the renewal of contracts ending in the future. This ensures an efficient heuristic for an otherwise difficult and computationally hard packing problem, but may result in suboptimal decisions and, consequently, a higher overall cost. An example of such a suboptimal decision is illustrated in Figure 8.1. A single contract (in blue) is in effect at time t_{cur}, but expires before $t_{cur} + \gamma$. If a reserved contract with an optimal utilization level greater than 50% existed, the –unmodified– algorithm would suggest to purchase an additional contract to cover the grey load (which is not currently covered by any reserved contract). It however is a better idea to renew the existing blue contract, and use an on-demand instance to handle the short-term load spike.

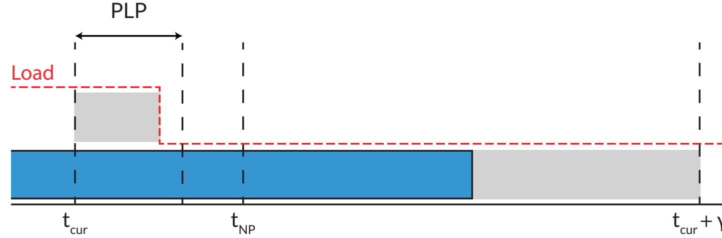


Figure 8.1: Illustration of renewal problem

In order to deal with this shortcoming, we can assume that contracts in the distant future are renewed automatically. In the construction of CCS_t (Equation 8.3), contracts with $\bar{c}_{ptime} + \bar{c}_l > t_{NP}$ are treated as if $\bar{c}_l = \infty$, and are added to every CCS_t. This way, the decision to renew a contract is only made in the iteration of the algorithm just before the contract's expiration. Figure 8.2 illustrates this approach with two contracts. Contract 1 has $\bar{c}_p = EH$, and therefore $\bar{c}_h = 0$. It ends before t_{NP}. Contract 2 has $\bar{c}_p = AYG$ with $\bar{c}_h = 0.034$, and ends after t_{NP}.

Intuitively, this "Infinite Contract" policy solves the aforementioned problem and will likely be beneficial when the predicted load remains constant or increases. However, in the previous chapter (Chapter 7) we presented a simple version of this algorithm and illustrated its operation using a case study. Our results showed that

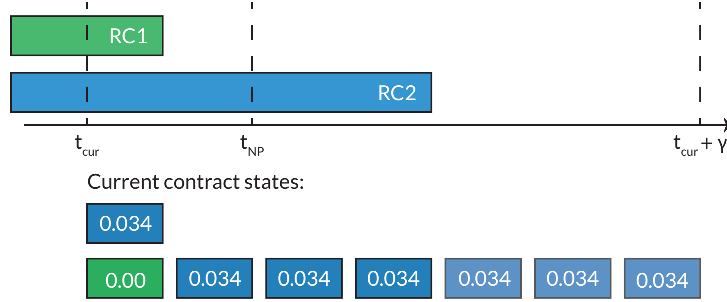


Figure 8.2: Example of the “Infinite Contracts” policy.

the “Infinite Contract” policy is not the most cost-beneficial method in case of a declining load. In fact, the view in which contracts run infinitely makes the algorithm overestimate the amount of available contracts in the future. As a result, when the load decreases below the number of active contracts, some of these contracts will appear unused for the next period, and the algorithm will consequently buy or renew fewer contracts than necessary.

We tackle this problem with the addition of two additional *contract renewal policies* (CRPs), leading to four CRPs:

- **Finite Contract** (FC): In this view, contracts are added to the CCS_t only in the time steps between $t \in [t_{cur}, \bar{c}_{ptime} + \bar{c}_l]$, as formulated in Equations 8.3, 8.4 and 8.5.
- **Infinite Contract** (IC): Here, contracts expiring after t_{NP} are assumed to renew automatically, and have a duration $\bar{c}_l = \infty$. The $\bar{c}_h$ value of reserved contracts expiring before t_{NP} are added only to CCS_t each time step $t \in [t_{cur}, \bar{c}_{ptime} + \bar{c}_l]$. This corresponds to the scenario illustrated in Figure 8.2.
- **Ensemble**: Switch between the FC and IC policy depending on the slope of the predicted load $\hat{l}$. If the least squares linear fit to $\hat{l}$ has a slope ≥ 0, the IC policy is applied. If not, FC is selected.
- **Two-pass**: Instead of approximating the best renewal behavior by selecting a renewal approach based on the general slope of the future load and treating all contracts in the same way, the renewal of existing contracts is done for each contract individually in a separate step. At the beginning of each iteration of the PMA, the CCS are constructed by completing the following three steps:
    1. Add all existing contracts to the CCS in the time steps between $t \in [t_{cur}, \bar{c}_{ptime} + \bar{c}_l]$, identical to the FC policy.
    2. Iterate through the existing contracts in ascending order of end time $(\bar{c}_{ptime} + \bar{c}_l)$, and decide for each of these contracts whether to renew them or not, based on the predicted load $\hat{l}$. A contract $\bar{c}$ is renewed, and thus added to every CCS between $\bar{c}_{ptime} + \bar{c}_l$ and $\min(\bar{c}_{ptime} + 2\bar{c}_l, t_{cur} + \gamma)$, if the cost of running with the existing contracts in the CCS is higher than the cost of running if the current contract is renewed. The upfront

cost included in the renewal cost is prorated to the period up to γ, as expressed in Equation 8.6:

$$Cost_{renewal}(\bar{c}, R, l_{hist}, od) = \frac{\gamma - \bar{c}_{ptime} - \bar{c}_l}{\bar{c}_l} \cdot \bar{c}_u + \sum_{t=\bar{c}_{ptime}+\bar{c}_l}^{\min(\bar{c}_{ptime}+2\bar{c}_l, t_{cur}+\gamma)} CostPerTimestep(t, R \cup \{c\}, \hat{l}, od) \quad (8.6)$$

3. The renewed contracts expiring before t_{NP} are removed again from the CCS, to allow the algorithm to exchange the original contract for a contract of a different type, if necessary.

8.3 Workload Prediction

The added value of a prediction depends on whether it is able to capture the general trend of the organization's load, as well as its ability to incorporate seasonal behavior, such as weekly or yearly recurring patterns. The trend is important to determine the order of magnitude of the number of running instances, as well as its likely increase or decrease over time. The size and the frequency of the fluctuations, on the other hand, is important to differentiate between reserved contracts with a higher or lower optimal utilization level.

A sequence of load observations over time is a typical example of a time series [118]. *Time series analysis and forecasting* comprises the use of statistical models to forecast values in the future based on past observations. In using time series models for prediction purposes, the model selection, parameter estimation and forecasting are complex and time-consuming. However, progress has been made in the accuracy of automated forecasting methods [120, 121, 122]. In addition to commercial products[1], implementations for automated model and parameter selection for ARIMA and various exponential smoothing methods are available in the *Forecast* package for R [125, 120]. These methods are used in this chapter, and are discussed here.

ARIMA

The *auto-regressive integrated moving average* model (ARIMA) is a generalization of the auto-regressive moving average (ARMA) model [126, 118], in which the data is differenced first in order to isolate eventual non-stationarity in the data. The ARMA model itself consists of two parts, an autoregressive (AR) part and a moving average (MA) part. The ARIMA model is referred to as $ARIMA(p,d,q)$, where p, d and q are positive integers that refer to the order of the auto-regressive (AR), integrated (I) and moving average (MA) parts of the model. The general ARIMA model is generally unable to cope with seasonal behavior. However, seasonal ARIMA models that incorporate additional seasonal terms into the non-seasonal ARIMA models exist. A seasonal ARIMA model is written as $ARIMA(p,d,q)(P,D,Q)_m$, in

[1] Forecast Pro and AutoBox, among others.

which m represents the number of periods per season. The determination of the parameters is usually found to be difficult and subjective. The `auto.arima` function in the R Forecast package can automatically select the best ARIMA model for a given time series, based on a number of tests as described in [120].

Exponential Smoothing

In its simplest form, an *exponential smoothing* model uses a weighted average with exponentially decreasing weights in order to forecast the next value. In order to predict more than one step ahead and take into account a trend, a slope component is added which itself is updated by exponential smoothing [124, 119]. This is known as Holt's method. This method can be extended to time series with seasonality, in which case the method is known as the Holt-Winters method.

In [127, 128], an approach for automated forecasting is presented based on an extended range of exponential smoothing methods, with and without seasonality. The automatic forecasting strategy, which is used as one of the forecasting methods in this chapter, is implemented as `ets` in the R Forecast package.

DSHW

It is plausible that a data set contains more than one seasonal component, when for example both weekly and monthly recurring patterns are present. An extension of the Holt-Winters method adds a second seasonal component to the exponential smoothing model, as described in [123]. This method is referred to as Double-seasonal Holt-Winters (DSHW), and is implemented in the R Forecast package.

Robust DSHW

Exponential smoothing and Holt-Winters methods are known to be sensitive to outliers in the input data. According to [129], outliers affect the forecasting methods because they are part of the past series of values and thus affect the smoothed values. The selection of the input parameters –which regulate the degree of smoothing– is impacted by the outliers as well. Gelper et al. [129] present a robust version of the Holt-Winters smoothing algorithm, in which the input values y_t are replaced in advance by a cleaned version y_t^*. The cleaned version is obtained by replacing unexpectedly high or low values in the series by a boundary value, based on an estimated scale $\hat{\sigma}_t$ and a positive constant k. In practice, a value is considered to be an outlier if the absolute difference between the observed value y_t and its predicted value $\hat{y}_{t|t-1}$ at $t-1$ is larger than $k \cdot \hat{\sigma}_t$. This is done by applying the Huber ψ function outlined in Equation 8.7 on the normalized error, as illustrated in Equation 8.8.

$$\psi(x) = \begin{cases} x & \text{if } |x| < k \\ sign(x)k & \text{otherwise.} \end{cases} \tag{8.7}$$

$$y_t^* = \psi\left(\frac{y_t - \hat{y}_{t|t-1}}{\hat{\sigma}_t}\right)\hat{\sigma}_t + \hat{y}_{t|t-1} \tag{8.8}$$

At the time of writing, an implementation of robust exponential smoothing is not available in the R forecast package. In order to incorporate a robust prediction model in our contract procurement evaluation, we implement robust double-seasonal

Holt-Winters model using the approach presented in [129]. As suggested in that contribution, the value of k is fixed at 2. The estimated scale is calculated using the Median Absolute Deviation (MAD, Equation 8.9) of the errors.

$$\hat{\sigma}_t = \underset{1\leq s\leq t}{\mathrm{MAD}}(r_s) = \underset{1\leq s\leq t}{\mathrm{median}}(|r_s - \underset{1\leq s\leq t}{\mathrm{median}}(r_s)|). \tag{8.9}$$

Previous Period

Instead of using mathematical models to predict the future load based on the past load pattern, a simple approach could be to adapt the reserved contract portfolio *reactively*, based on the load in the previous period. The *previous period* method extrapolates the most recent period of δ over a period of γ by simply repeating the data points. If δ is chosen wisely, the resulting load pattern includes simple seasonal patterns. When $\delta = 7$ days, for example, difference in load between week days and weekends would be permeated over γ, and would thus be taken into account when estimating contract utilization levels.

This technique corresponds to the manual approach in which one calculates the past utilization in period δ, and purchases contracts accordingly. This is a more conservative approach, appropriate to quantify the difference between a *proactive* prediction-based method and a *reactive* method based on past observations.

8.4 Experiment Setup

In order to evaluate the impact of the presented algorithm, simulation-based experiments are performed. These focus on the deployment of web workloads that correspond to real-world load traces, on Amazon EC2.

Simulation

We use a simulation environment in order to evaluate the impact the algorithm presented in this chapter in combination with each of the Contract Renewal Policies (CRP) and each of the automated forecasting models. This allows a thorough evaluation for different parameters and workloads.

Our Python-based simulator executes scenarios in which contracts are bought according to the suggestions made by the purchase algorithm, server instances are allocated based on a server load trace and the total cost for running this scenario is calculated based on the contract portfolio available at every moment in time between the start date and end date of the simulation.

Prediction

In order to evaluate the cost impact of different prediction techniques, a clairvoyant “Full Knowledge” (FK) predictor is used as a point of reference. This predictor uses the simulation’s workload traces to return completely accurate predictions. The real prediction techniques are configured automatically by the software used, except for the seasonal periods: predictors with a single seasonal period are configured with a weekly seasonality, the double-seasonal models use a seasonality of 1 week and 1 year.

Workload

To evaluate PMA, we require a number of real-world load traces with diverse characteristics; with and without trends and seasonal behavior, and of different magnitude with respect to server load. The traces have to be of significant length as a load history of two years is required. Unfortunately, long-term cloud usage traces are hard to find, as they are not published by cloud providers or organizations. Some cluster and grid job traces are available at the Parallel Workloads Archive (PWA) [115], the Grid Workloads Archive (GWA) [82] and the Google Cluster Trace (GTC) [116]. These job traces do not fit our needs, because they are relatively short –only three traces from the PWA are longer than 3 years– or they are upwardly limited by the available resources in the cluster or grid.

The best suited traces for our purpose can be found at Quantcast[2], an advertising company that tracks page views on a large number of websites. These page view reports are available for free on their website. Our experiments use these reports to build a server instance history, that serves as input for the simulator. In the list of the top-200 visited websites, all directly measured reports –excluding reports with discontinuities (such as unmeasured periods) or insufficient data (less than 3 years)– are converted into a list of server load patterns. This results in an extensive list of 51 server load patterns to be used for our evaluation. To the best of our knowledge, we are the first to evaluate the performance of a cloud scheduling algorithm using such an extensive set of application load traces. The list of load patterns is shown in Table 8.1 for reference.

The page view reports contain daily aggregated data. We therefore assume the load is constant throughout the day, as the addition of artificial day-night patterns would not impact our results because of the different timescales involved. We further assume that the web application is deployed on instances of a single type and adopt the `m1.xlarge` instance type, which is commonly used for web servers. Given an average number of HTTP requests per page view (RPP) and a number of requests a single `m1.xlarge` server can handle (RPS), the number of page views p_t at time t can be converted to a number of server instances l_t following Equation 8.10.

$$l_t = \frac{p_t \cdot RPP}{RPS} \tag{8.10}$$

According to an extensive statistical study over 4.2 billion web pages performed by Google [117], an average page view results in an RPP value of 44.56 GET requests. The number of requests per second (RPS) a server instance is able to handle is subject to a number of application-specific parameters, which are hard to determine without application knowledge or benchmarking abilities. For example, the mix of static versus dynamic content, the number of database requests and the caching techniques used by an application are unknown. As this number has no influence on our comparative results and only affects the absolute cost figures, we assume that for one `m1.xlarge` server $RPS = 50$.

Contract Types

The on-demand prices and reserved contracts for an EC2 `m1.xlarge` instance are configured according to their actual prices. In our evaluation, we focus on reserved

[2] `http://www.quantcast.com`

Website	Start date	End date	Days	# servers	Avg. # servers
answers.com	Apr 01, 2007	Dec 31, 2013	2466	[19, 1040]	145.15
bleacherreport.com	Jan 11, 2009	Dec 31, 2013	1815	[1, 277]	52.14
break.com	Oct 12, 2007	Dec 31, 2013	2272	[16, 286]	83.70
chacha.com	Apr 22, 2009	Dec 31, 2013	1714	[2, 251]	43.90
cnbc.com	Aug 19, 2008	Dec 31, 2013	1960	[1, 339]	115.42
complex.com	Aug 18, 2007	Dec 31, 2013	2327	[1, 79]	10.58
csmonitor.com	Jan 10, 2009	Dec 31, 2013	1816	[1, 50]	9.13
deadspin.com	Apr 01, 2007	Dec 31, 2013	2466	[1, 84]	11.05
drudgereport.com	Apr 17, 2008	Dec 31, 2013	2084	[131, 754]	267.11
ew.com	Apr 14, 2008	Dec 31, 2013	2087	[4, 81]	20.01
examiner.com	Nov 19, 2008	Dec 31, 2013	1868	[1, 65]	18.41
fool.com	Jul 03, 2008	Dec 31, 2013	2007	[1, 29]	11.23
gawker.com	Apr 01, 2007	Dec 31, 2013	2466	[1, 103]	17.29
gizmodo.com	Apr 01, 2007	Dec 31, 2013	2466	[1, 230]	30.03
goodreads.com	Dec 31, 2008	Dec 31, 2013	1826	[6, 120]	39.74
grindtv.com	Apr 02, 2007	Dec 31, 2013	2465	[1, 74]	4.65
hollywoodlife.com	Aug 12, 2009	Dec 31, 2013	1602	[1, 107]	12.55
hubpages.com	Mar 03, 2008	Dec 31, 2013	2129	[4, 57]	21.36
huffingtonpost.com	Apr 01, 2007	Dec 31, 2013	2466	[1, 534]	151.65
instructables.com	Aug 31, 2007	Dec 31, 2013	2314	[3, 37]	19.28
jezebel.com	Apr 13, 2007	Dec 31, 2013	2454	[1, 95]	13.81
legacy.com	Sep 22, 2009	Dec 31, 2013	1561	[1, 110]	69.15
lifehacker.com	Apr 01, 2007	Dec 31, 2013	2466	[1, 47]	15.14
linkedin.com	Nov 06, 2008	Dec 31, 2013	1881	[63, 1626]	690.05
macrumors.com	Oct 11, 2007	Dec 31, 2013	2273	[1, 105]	21.15
merriam-webster.com	Nov 12, 2008	Dec 31, 2013	1875	[1, 84]	28.92
nationalgeographic.com	May 08, 2008	Dec 31, 2013	2063	[10, 65]	21.42
nbc.com	Feb 28, 2008	Dec 31, 2013	2133	[1, 155]	26.59
people.com	Apr 07, 2008	Dec 31, 2013	2094	[1, 566]	112.87
quizlet.com	Mar 02, 2008	Dec 31, 2013	2130	[1, 113]	15.57
rantsports.com	Mar 26, 2010	Dec 31, 2013	1376	[1, 137]	13.74
rottentomatoes.com	Jan 12, 2010	Dec 31, 2013	1449	[14, 52]	31.07
sbnation.com	Jan 07, 2009	Dec 31, 2013	1819	[1, 44]	6.00
simplyhired.com	Apr 15, 2008	Dec 31, 2013	2086	[1, 178]	35.55
squidoo.com	Jan 26, 2008	Dec 31, 2013	2166	[1, 48]	15.01
stackoverflow.com	Jun 11, 2009	Dec 31, 2013	1664	[4, 200]	57.21
thedailybeast.com	Nov 16, 2008	Dec 31, 2013	1871	[2, 65]	11.50
theonion.com	Jan 31, 2009	Dec 31, 2013	1795	[1, 73]	13.16
time.com	May 13, 2008	Dec 31, 2013	2058	[3, 239]	32.15
tmz.com	Oct 27, 2009	Dec 31, 2013	1526	[1, 152]	43.29
topix.com	Oct 27, 2007	Dec 31, 2013	2257	[2, 64]	33.78
typepad.com	Nov 07, 2007	Dec 31, 2013	2246	[1, 85]	29.55
urbandictionary.com	May 03, 2007	Dec 31, 2013	2434	[1, 458]	28.41
usmagazine.com	Sep 29, 2008	Dec 31, 2013	1919	[17, 339]	72.50
washingtontimes.com	Aug 05, 2009	Dec 31, 2013	1609	[2, 36]	5.63
wattpad.com	Sep 30, 2009	Dec 31, 2013	1553	[3, 271]	74.13
whitepages.com	Sep 23, 2009	Dec 31, 2013	1560	[1, 164]	69.83
womensforum.com	Sep 15, 2010	Dec 31, 2013	1203	[1, 57]	7.73
wordpress.com	Apr 01, 2007	Dec 31, 2013	2466	[49, 1142]	383.27
wunderground.com	Apr 01, 2007	Dec 31, 2013	2466	[1, 193]	60.27
zimbio.com	May 27, 2007	Dec 31, 2013	2410	[1, 211]	56.38

Table 8.1: The list of server load traces.

contracts with a duration of one year, because of the unavailability of real-world trace data for a longer period. We also believe that making predictions for a period of three years would call for the incorporation of additional domain and business-specific knowledge; an avenue for future work. The set of reserved contract types C used in the evaluation can be found in Table 8.2.

c_n	c_u	c_h	c_l	c_p
Heavy	\$1352	\$0.112	1 year	EH
Medium	\$1108	\$0.168	1 year	AYG
Light	\$486	\$0.271	1 year	AYG
On Demand	\$0.0	\$0.480	1 year	AYG

Table 8.2: Available contract types for `m1.xlarge`.

Scenario Setup

Every combination of a purchase algorithm, a workload trace and a prediction technique constitutes a *scenario*. In every scenario, the algorithm is executed on a regular basis (daily, weekly or monthly) over the *simulation period*. The simulation period starts two years after the beginning of each trace, the first two years serve as the load history input for the algorithm. The simulation period ends one year before the end of the trace, which follows from the data requirements of the clairvoyant FK predictor.

The load traces' non-zero starting point require an *initial contract portfolio* (ICP). This portfolio is a cost-optimized set of contracts: 7 days before the start of the simulation, the optimal set of reserved contracts for the last 90 days is calculated. We distribute the start times of the contracts in the ICP uniformly between 7 days and 371 days prior to the simulation's start time, in order to avoid that all contracts end at the same time. This would overemphasize the importance of the load prediction and procurement choices at that particular time, skewing the results.

In order to evaluate the cost impact of the PMA, a *Renewal Only* (RO) algorithm is introduced that serves as a reference scenario. This simple algorithm renews all expiring contracts at the appropriate moment in time, but does not take into account predictions and does not purchase new reserved contracts other than the existing ones. As both the PMA and the RO have the same ICP, the differences in cost between both algorithms will be caused by changes made in the contract portfolio by the PMA.

Evaluation Metrics

In order to reliably compare the cost-effectiveness of different scenarios, the aggregated cost has to be carefully calculated, given the limited timespan of the simulation. If the organization acquires a new reserved contract on –for example– the last day of the simulation, attributing the full up-front cost to the scenario's total cost would result in a negative bias, as the purchase only becomes profitable in the time period after the end of the simulation. Consequently, we introduce a cost calculation method that spreads the up-front cost for each purchase over the whole period in

which it is valid. For all contracts in which the period between $\bar{c}_{ptime}$ and $\bar{c}_{ptime} + \bar{c}_l$ does not overlap entirely with the simulation period, the up-front cost $\bar{c}_u$ and, in case of $\bar{c}_p = EH$, the hourly rates $\bar{c}_h$ are only accounted for in proportion to the time in which the contract period and simulation period overlap.

Even using this cost calculation method, very small irregularities are possible if the utilization level of a contract is not uniformly distributed over the entire contract length. These are side effects of the use of simulation for which the impact on our results was found to be negligible.

8.5 Results

In this section, we present and discuss the results of a number of experiments with regard to the performance of the algorithm with different CRPs, the performance of the proactive and reactive prediction techniques and the influence of the algorithm's configuration parameters. Finally, we discuss the significance of the presented results as well as the computational complexity of the algorithm.

Box-and-whisker plots are used to show the distribution of the metrics over the different workload traces, with the whiskers representing the 5th and 95th percentile of the results. Outliers are not plotted for clarity reasons, but mentioned in the text if they are relevant for the discussion.

Algorithm

In the first experiment, we evaluate the cost level achieved by PMA with each of the Contract Renewal Policies (CRPs) relative to the cost of the baseline RO scenario. We first configure the PMA with the Full Knowledge predictor, so it operates under "best case" conditions, and compare the results of the PMA with each of the CRPs. Figure 8.3 shows the cost reduction of PMA relative to the cost of the RO algorithm.

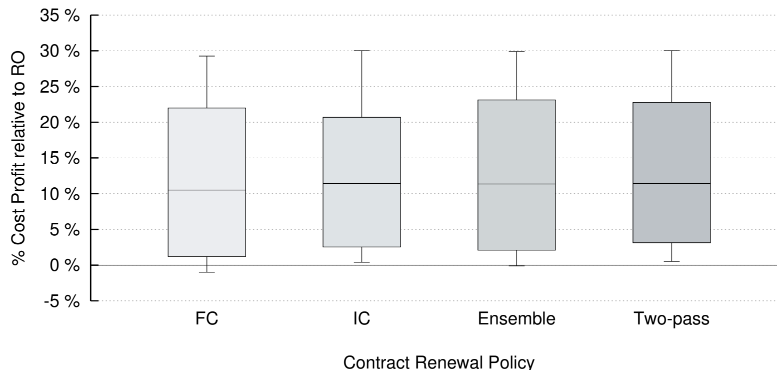
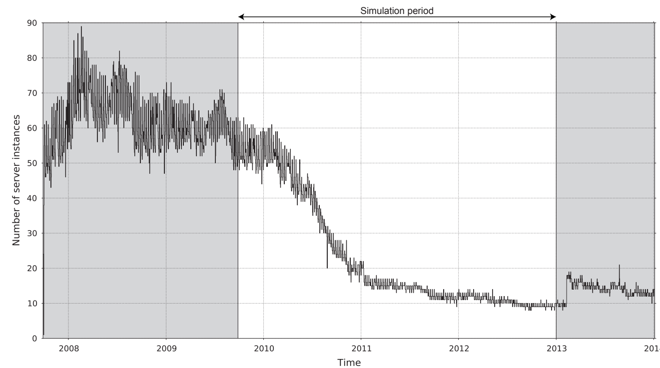


Figure 8.3: PMA vs. RO (Iteration interval: weekly, Purchase Lookahead Period: 30 days)

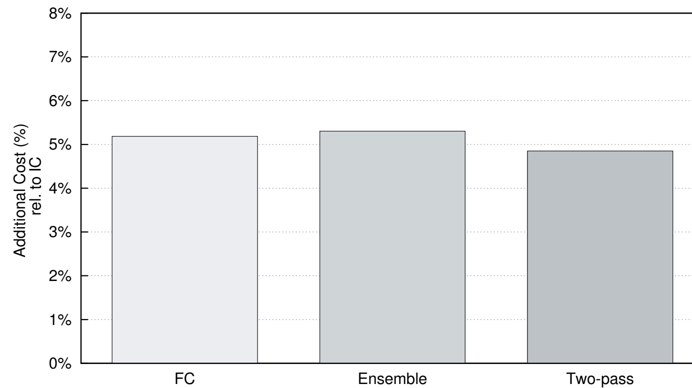
In our set of load traces, the PMA obtains a cost reduction up to 30.7% compared to the Renewal Only scenario. Averages range between 11.6% and 12.9%, depending on the CRP. Note that every scenario is initialized with a set of contracts optimized

for the period just before the start of the simulation, and that the achieved cost reductions are the result of adaptations in the contract portfolio made based on the future load.

If we compare the performance of the CRPs, the two-pass policy outperforms the other CRPs on average with 1.5% (FC), 0.2% (IC) and 0.4% (Ensemble). It therefore seems to be the best overall choice, although the differences between the policies is small and –in this set of workload traces– even almost negligible. In a few cases, however, the differences are significant. For *tinypic.com* for example, which is not included in this chapter's list of server load traces because it is outside of Quantcast's top-200, we showed in Chapter 7 that IC performs significantly worse than FC. This is due to the decrease in load (see Figure 8.4a), which in turn causes the PMA with the IC contract renewal policy to overestimate the future load and purchase an insufficient amount of contracts. In that case, both the ensemble and two-pass policies succeed in keeping the cost down. The cost results are presented in Figure 8.4b. Compared to IC, the cost can be reduced with up to 5.3% using the Ensemble CRP, the Two-pass CRP also manages to lower costs with 4.9%.



(a) Load pattern



(b) Additional cost relative to IC

Figure 8.4: tinypic.com Case

As the two-pass policy shows the best overall performance in our test set and

does well in specific cases where IC underperforms, it is used in all subsequent experiments in this section.

Prediction Models

Next, we compare the PMA with different prediction techniques for all workload traces to the PMA scenario with the full knowledge (FK) predictor, and express their expenditure increases relative to the latter scenario.

In this experiment, the purchase algorithms are executed on a weekly basis, the Purchase Lookahead Period is set to 30 days and the two-pass renewal policy is applied. The influence of these input parameters on the cost is discussed later on in Section 8.5. The results of the six prediction techniques and 51 workload traces are presented in Figure 8.5 as the distribution of the cost increase relative to PMA+FK.

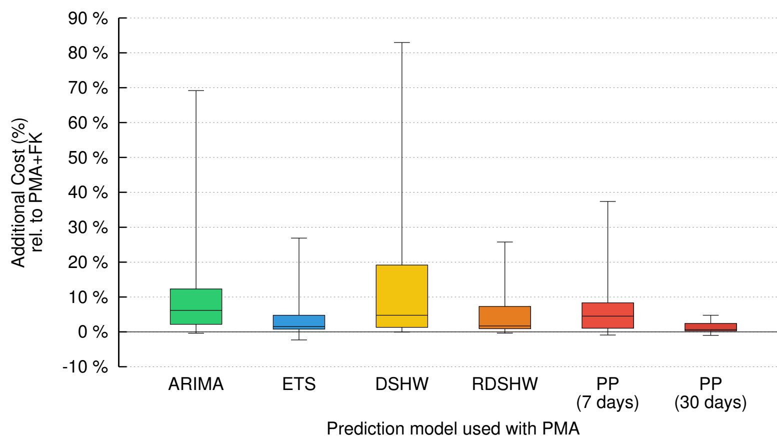


Figure 8.5: Prediction Model evaluation (Iteration interval: weekly, Purchase Lookahead Period: 30 days)

In all prediction techniques, except for the previous period with $\delta = 30$, a significant amount of workload traces suffer from a surplus in cost up to 82%, with outliers (not displayed) up to a 134% for ARIMA and 1010% for one scenario using DSHW. This is a major downside to an automated approach, as the odds are pretty high that, without intervention, the results of the PMA are unreliable if used in combination with one of these prediction techniques.

DSHW and, to a lesser extent, ARIMA show to be the most susceptible to this problem. While the robustness measures taken by the RDSHW approach seems to partially fix the problem, in one RDSHW scenario (womensforum.com, +93.7%) the cost still nearly doubles. A closer inspection shows that these scenarios are workloads with a highly irregular behavior that is difficult to predict. If we drop the annual seasonality and use exponential smoothing with only a 7-day seasonal pattern (the ETS scenario), that behavior seems to occur less because the weekly pattern occurs more frequently in the predictor's input data and the irregularities in the data are of less importance to the prediction model.

Using a previous period predictor with a period δ of 30 days, finally, the cost rise is limited to at most 10.2% with an average cost increase of only 1.6%. PP-30

performs better than RDSHW in 80.4% of the cases, and appears to be the best prediction technique in the experiment. With $\delta = 7$, the workload changes are picked up more quickly, with a lag of at most one week, and the technique performs considerably worse.

As the results show high sensitivity with respect to δ, Figure 8.6 illustrates the impact of a δ value ranging between 7 days and 1 year. The results indicate that the highest cost reductions can be achieved at $\delta = 30$, with an average additional cost of +1.6%. For lower and higher values of δ, the spread and average of the additional cost increases.

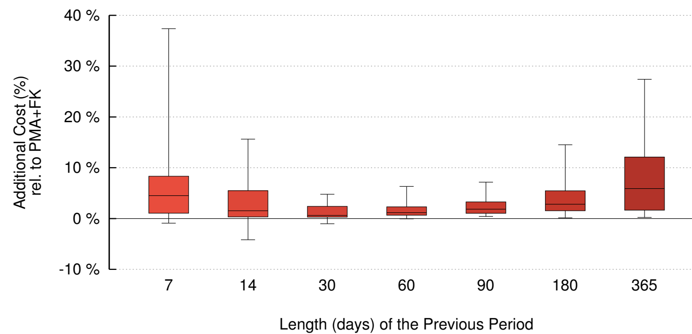


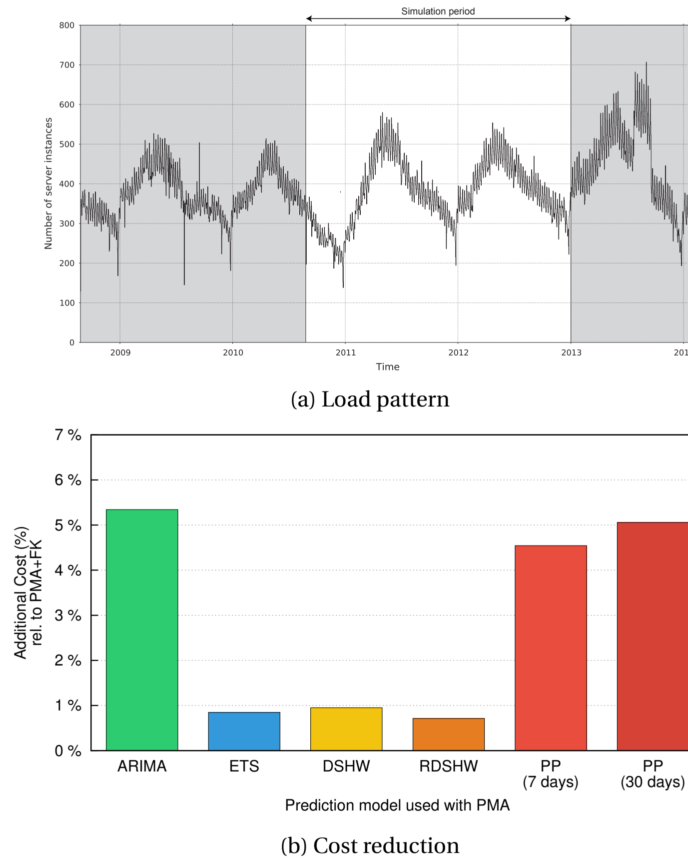
Figure 8.6: Influence of δ on PP. (Iteration interval: weekly, Purchase Lookahead Period: 30 days)

It appears that, for most of the load patterns in our test set, the use of PMA in combination with a reactive approach in which the load of the last 30 days is considered significant for the upcoming year, suffices to obtain good results. This can be explained by the fact that an overestimation has a much higher impact on the resulting cost than an underestimation, as overestimation will cause any procurement algorithm to purchase too much and too expensive contracts, while an underestimation can easily be taken care of by utilizing additional on-demand instances and buying the required contracts in the next iteration of the algorithm.

In specific cases, however, it remains beneficial to use more advanced prediction techniques. An example is *gardenweb.com*, a web site which has a significant increase in visitors during spring and summer, and a decrease during the fall and winter season. The load pattern is shown in Figure 8.7a. Figure 8.7b shows the additional cost for each prediction technique relative to the full knowledge predictor. The results show a substantial cost difference between the prediction techniques based on exponential smoothing (ETS, DSHW and RDSHW) and the previous period techniques. This is caused by the non-trivial seasonal pattern, which is picked up by the seasonal predictors but causes the other prediction techniques to purchase too much reserved contracts during spring.

Influence of iteration interval and Purchase Lookahead Period

Up until now, the PMA's iteration interval was configured to run every week, and PLP was set to 30 days. In the following, we evaluate the influence of both parameters on the achieved cost reduction. For each workload trace, the cost is calculated for every



(a) Load pattern

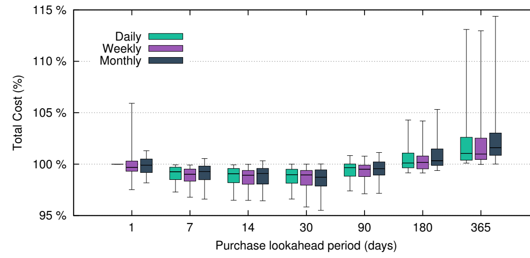
(b) Cost reduction

Figure 8.7: gardenweb.com Case

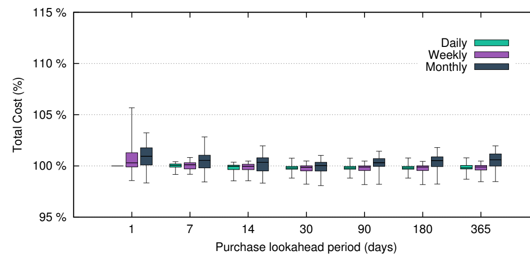
combination of the Iteration Interval and PLP, using both FK, PP-30 and RDSHW predictors. The cost for each scenario is calculated relative to the scenario in which the algorithm is run daily with PLP = 1 day. In Figure 8.8, the distribution of these relative cost values for each combination are shown.

In case of FK, the iteration interval seems to be of little importance with regard to the total cost. Changes in the PLP on the other hand cause the total cost to decrease to on average 98.5% (weekly, PLP = 30) of the reference scenario and further increase up to on average 105.5% (daily, PLP = 365) with outliers up to 244% (not plotted).

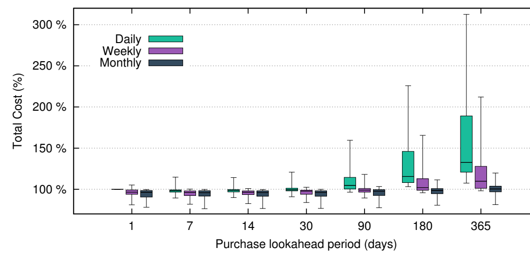
The scenario in which PLP = 365 is equivalent to an algorithm that does not take into account this parameter, as the PMA will purchase any contracts that appear beneficial over the next year without considering whether this is also the case over a shorter time period. The use of such a short-term evaluation period next to the contract length is beneficial only when there is a significant trend over the next year. In a load pattern without a trend, the difference between the average utilization for the PLP and the entire contract length is small, and the outcome of Decision 1 and Decision 2 (Section 8.2) will be identical. This is demonstrated when using the Previous Period predictor in Figure 8.8b. This prediction approach assumes the load is stationary, and repeats past values. Due to this stationarity, the PLP parameter has



(a) Full Knowledge



(b) Previous Period - 30 days



(c) Robust DSHW

Figure 8.8: Influence of iteration interval and Purchase Lookahead Period

no influence (less than 0.01% on average) if it is larger than the repeated period of 30 days. Because of the lag with which the algorithm reacts on changes in the load, the monthly scenarios perform slightly worse than the daily and weekly varieties. The difference between the latter two is negligible.

With the RDSHW predictor, finally, the variance for increasing values of PLP is more pronounced due to the predictor's tendency to reinforce and sometimes overestimate the trend. As there is no limit to the number of contracts purchased in one time step, a decreased iteration interval causes the likelihood of errors. This results to an average increase of 60.5% with outliers up to 574% (not plotted) for running the algorithm on a daily basis without taking into account the PLP (= 365). Good results are achieved using RDSHW for a weekly or monthly iteration interval in combination with a PLP up to 14 days.

In conclusion, the introduction of a short-term purchase lookahead period (PLP) pays off when the algorithm is used in tandem with a prediction technique that is able to make non-stationary predictions, and the daily procurement of contracts offers little advantage over the weekly evaluation of the contract portfolio.

Significance of the results

The reported results depend on the start times of the contracts in the *initial contract portfolio*, which are generated according to a uniform distribution. In order to evaluate the significance of the presented results, we ran 100 iterations with a different random seed for every scenario with the *linkedin.com* workload trace in the experiment in which PMA with different prediction techniques is compared to PMA+FK. Relative standard deviations for the calculated cost of all prediction techniques were below 0.071%, the standard deviations of the additional cost relative to PMA+FK –which is the reported metric in Figure 8.5– is at most 0.079%.

Computational Complexity

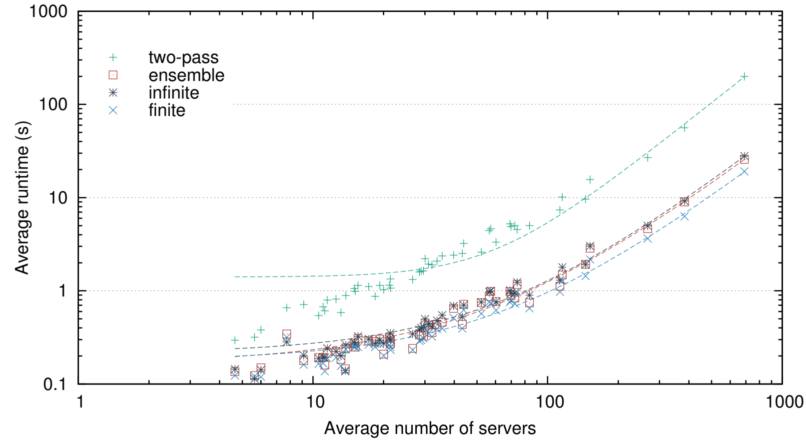
We executed the experiments on a machine with four 12-core AMD Opteron 6234 processors and 196 GB of memory. The average amount of CPU time necessary for running the PMA purchase algorithm and for making one prediction is calculated for each scenario.

The runtime of PMA is directly linked to the number of servers on which it operates. Figure 8.9a shows the correlation between the average number of servers in a workload trace and the average runtime of one run of the PMA+FK with different CRPs. Points are plotted individually, the plotted lines represent the quadratic functions that are fitted to the points. This $O(n^2)$ complexity is due to the cost calculation method: for each additional server the cost for the next year is calculated by summing the hourly rates for the load at each hour. On average, running an iteration of PMA with the two-pass CRP is 4.7 times slower than with the ensemble CRP. A quadratic runtime correlation combined with the orders of magnitude of the achieved runtimes indicates that the algorithm offers a tractable solution to the concerned procurement problem.

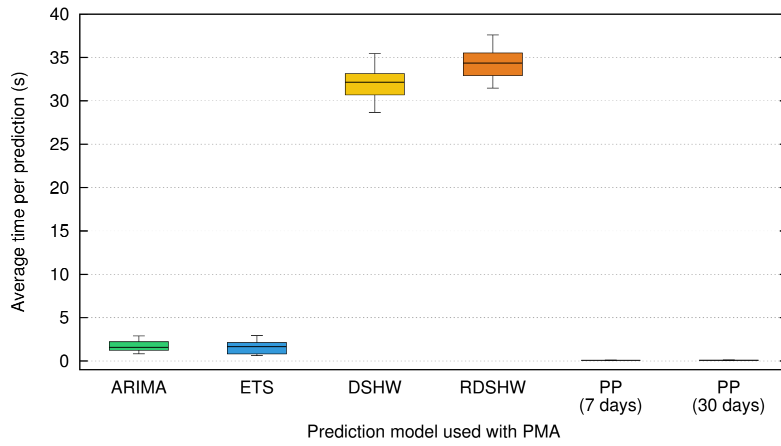
Figure 8.9b shows the distribution of the average prediction runtimes over the different workload traces. Outliers are omitted for clarity reasons. Prediction runtimes for the double-seasonal models are over ten times the value of the ARIMA and ETS models. The previous period runtime is negligible, as this technique simple repeats past values without manipulating the data.

8.6 Conclusion

In this chapter, we have presented an algorithm to automate the procurement of reserved IaaS contracts. We also evaluated whether the use of an automated approach to load forecasting pays off in terms of reduced costs when purchasing reserved IaaS contracts. The presented *purchase management algorithm* iteratively produces contract procurement suggestions based on server load predictions while taking into account a current contract portfolio. Using simulation, we evaluated the cost reduction the algorithm achieves compared to a stationary approach using a wide range of web application load traces. Further, we quantified the cost differences between (a) full knowledge predictions, (b) an automated forecasting approach using ARIMA and exponential smoothing models and (c) a reactive approach that extrapolates past observations. Our results show that the algorithm has a significant potential for cost reduction. The used seasonal time-series models however perform



(a) PMA Decision Runtimes



(b) Predictor runtimes

Figure 8.9: Average runtimes

worse in terms of cost than a stationary "previous period" method, except for specific cases. This is caused by the imbalance between the penalties for overestimating versus underestimating the future load. The former is penalized by a high up-front cost, while the latter can be easily accommodated using the flexibility of on-demand instances with only a limited additional cost. Our future work includes the incorporation of Quality-of-Service differences between reserved and on-demand contracts, such as unavailability of or upper limits to the number of on-demand instances, as well as the expansion of this approach to a hybrid cloud setting in which the procurement of physical servers is added as an additional long-term commitment.

Conclusions and Future Research

CHAPTER 9

Conclusions and Future Research

In the current era, the use of computing to support organizations' business processes and research projects is still expanding firmly. In this thesis, we identified a number of areas in which additional research was necessary to support and ease this process.

Firstly, the benefits of using a fine-grained value-based scheduling algorithm were compared to a coarse-grained scheduling technique. Our contributions in this field included:

- A thorough evaluation of the conditions under which a fine-grained scheduler, implemented as an online auction-based mechanism with static bids and without preemption, can outperform a priority queue system.
- An evaluation of whether these conditions are common in real-world workload traces.
- An explanation for the differences between our findings and results presented in other contributions in the field.

Our findings showed that a priority queue approach with a limited number of queues closely approximates the performance of the auction-based approach. This is important, because a fine-grained value-based scheduling approach brings about increased complexity for both end user and system administrator. The *parallelization degree* and *load* of the cluster's workload appeared determinative for the performance of a fine-grained algorithm. These parameters must increase beyond levels that are currently found in publicly available workload traces for the difference between both techniques to become significant.

Secondly, the problem of dynamically extending private infrastructure with resources from a public cloud provider was considered, and efforts were made in

the scope of cost-efficiently scheduling batch-jobs with a hard deadline constraint on hybrid clouds. Our contributions were:

- A Binary Integer Program for cost-optimal scheduling BoT applications with hard deadline constraints on multiple public and private cloud providers. The program considered computational and network costs.
- An online scheduling algorithm that tackles cost-efficient placement of those applications, taking into account the transfer times of the necessary data sets to the cloud providers.
- A study of the effect of errors in the user-provided runtime estimates on the presented results.

Integer programming appeared to be feasible for taking scheduling decisions concerning placement of BoT applications on public cloud providers, but high solve time variances prevented it from being an attainable solution when scheduling in a hybrid cloud setting. The presented heuristics resolve that issue and allow for additional cost gains to be achieved. Our results demonstrated that adopting a job queue with an EDF policy in combination with a cost-based approach for migrating applications to the public cloud results in significant cost reductions. The proposed algorithms proved to be able to schedule a large number of applications within a practical timeframe. In addition, the queue-based policies showed to significantly increase robustness with respect to runtime estimation errors in exchange for an increased turnaround time.

Lastly, we elaborated on automating the process of reducing costs when using cloud resources by incorporating reserved contracts. Time series forecasting techniques were used to generate load predictions without application knowledge. Those were fed to an algorithm that produced procurement suggestions for those reserved contacts while taking into account an organization's current contract portfolio. The main contributions were:

- An algorithm for automated IaaS contract procurement of quadratic complexity in the number of server instances, that takes into account an organization's existing contract portfolio.
- An illustration of the suitability of Genetic Programming-based symbolic regression for time series forecasting in this context.
- The integration and evaluation of different time series forecasting techniques (including both seasonal and non-seasonal models) to guide contract acquisition through workload prediction. We analyze in which cases such forecasting techniques can prove valuable and present their limitations with respect to acquiring current real-world IaaS contracts.
- An illustration of the operation of the algorithm by means of a case study.
- An extensive empirical evaluation of the proposed algorithm and forecasting techniques using a large dataset of real-world web traffic workloads.

Our results showed that the algorithm's potential for cost reduction is significant, with cost reductions up to 30% compared to a stagnant scenario, but that the applied prediction models perform worse in terms of cost than a stationary "previous period" method except for specific cases. In the current reserved pricing landscape and given the evaluated web application workloads, the use of prediction models to make better procurement decisions is only beneficial when the workload shows non-trivial but predictable seasonal patterns.

After analyzing the approaches and contributions discussed in this dissertation, new questions arise and some questions remain unanswered. Some of these possible avenues for future work are discussed in the remaining of this chapter.

- Chapter 2 reviewed fine- and coarse-grained value-based scheduling approaches to regulate access to an organization's shared compute resources. Regardless of which variant is used, an incentive system is needed to encourage users to trade in value in return for less tight deadlines. When using a priority queue system, one way to deal with this problem could be to introduce dynamic prices that take into account the current load of the infrastructure. The design of such a price setting mechanism is an interesting challenge, especially when taking into account more complex QoS constraints such as soft deadlines.

- Part II of this thesis discussed the scheduling problem of BoT applications with a hard deadline in a hybrid cloud setup. Often, however, users do not have a single deadline in mind when it comes to running their applications. Trading off cost and QoS constraints such as a deadline is a complex task for the owner of the application. Users need assistance in determining the valuation of their workload and selecting the right options from the numerous options available. A decision support system could provide users with sufficient information on the (expected) costs that a workload's execution will generate and how the cost level is expected to change with the provided level of QoS.

- In this thesis, instance types were either fixed or the selection of a suitable instance type for an application was performed using a simplified model based on Amdahl's law. One aspect of the flexibility of the current cloud offerings is the ability to quickly change the instance type of the infrastructure, given the application's requirements. This is called *vertical scaling* (as opposed to *horizontal scaling*, i.e. adding more instances). Selecting the most cost-efficient instance type for a specific application, however, is still an unresolved issue. Given performance figures on cloud providers and instance types and information on the performance and speedup of an organization's applications, a cost-minimizing scheduling approach could deal flexibly with an application's instance type based on the current load and deadlines of that or other applications.

- When dealing with reserved contracts to minimize the overall cost for an application, the addition of load prediction techniques proved beneficial in only a limited number of cases. In this study, the higher Quality-of-Service offered by these reserved contracts was not taken into account when evaluating the

benefit of such an automated approach. The impact of incorporating these Quality-of-Service differences between reserved and on-demand contracts has not yet been quantified. In addition, the expansion of the automated procurement approach to a hybrid cloud setting in which the procurement of physical servers is added as an additional long-term commitment is also a remaining challenge.

- Next to on-demand and reserved rates, some cloud providers offer an additional *spot market* model in which prices fluctuate based on the current load in their data centers. Although this spot market and its behavior has been studied already in related work in the field, its cost-efficient and reliable use as an extension of an organization's private compute infrastructure still remains an open problem.

BIJLAGE A

Samenvatting

Kostenbewust Beheer van Computerinfrastructuur in Clusters en Clouds

Organisaties en bedrijven gebruiken steeds vaker applicaties die ingezet worden om bedrijfskritische taken te vervullen. Vaak worden deze applicaties uitgevoerd op zogenaamde "clusters", een aantal door een netwerk met elkaar verbonden computers waartoe verschillende gebruikers in de organisatie gelijktijdig toegang hebben. Wanneer meerdere gebruikers hun applicaties gelijktijdig op dezelfde computerbronnen willen uitvoeren, stelt er zich een planningsprobleem en is er een manier nodig om aan die applicaties prioriteiten toe te kennen. De softwarecomponent die instaat voor het plannen van applicaties in een context met meerdere gebruikers heet een *scheduler*, het planningsproces wordt *scheduling* genoemd.

De computerinfrastructuur waarop die applicaties uitgevoerd worden, evolueerde de voorbije jaren ook enorm. Daarbij wordt deze infrastructuur steeds vaker beschouwd als een nutsvoorziening, die toelaat om geconsumeerd te worden wanneer nodig, waarvoor betaald wordt op basis van het gemeten verbruik en waarbij de drempel voor het gebruik erg laag is. Dat is de basis van "cloud computing", een verzamelnaam voor diensten waarbij toegang tot computerinfrastructuur en -applicaties gehuurd kan worden. Belangrijk aspect daarbij is *flexibiliteit*: de toegang tot een nieuwe server wordt quasi onmiddellijk verschaft, en wordt afgerekend per gebruikt uur. Die flexibiliteit opent deuren voor organisaties: niet langer is de hoeveelheid servers die ze zelf in hun bezit hebben een belemmering voor de uit te voeren applicaties. Dat voegt aan het eerder genoemde planningsprobleem een aspect van kostenefficiëntie toe: aangezien elk uur rekentijd in de cloud verbonden is met een kost, is het belangrijk dat de gebruikers en applicaties in zo'n organisatie daar bewust mee omgaan.

Het onderzoek dat in dit proefwerk gepresenteerd wordt, situeert zich in het domein van gedistribueerde computerinfrastructuur, en probeert een antwoord te bieden op onderzoeksvragen in het kader van het kostenbewust beheren van computerinfrastructuur in clusters en clouds. Dit proefschrift is onderverdeeld in drie delen.

Deel 1 beschouwt het plannen van applicaties in cluster-systemen gebaseerd op de waarde die de eigenaars van die applicaties hechten aan de correcte en intijdse

uitvoering van hun applicatie. Daarbij wordt gekeken naar gerelateerde wetenschappelijke publicaties in het gebied van waarde-gebaseerd plannen, en wordt de toegevoegde waarde van zo'n waarde-gebaseerd planningsmechanisme vergeleken met een eenvoudiger systeem gebaseerd op een aantal vaste wachtrijen. Uit onze resultaten blijkt dat zo'n wachtrij-gebaseerd systeem met een beperkt aantal wachtrijen de prestaties van een waarde-gebaseerd systeem benadert op vlak van gegenereerde waarde.

In Deel 2 van dit werk wordt het planningsprobleem onderzocht dat zich stelt wanneer een private cluster in een organisatie uitgebreid wordt met de flexibele infrastructuur van één of meerdere cloud providers. We beperken ons daarbij tot het planningsprobleem met betrekking tot applicaties van het type "batch jobs" die voor een bepaalde deadline uitgevoerd moeten worden. Het probleem wordt benaderd vanuit het oogpunt van kostenefficiëntie, en er wordt bij de voorgestelde oplossingsmethodologieën opeenvolgend gebruik gemaakt van lineair programmeren en heuristieken. De aanpak die gebruik maakt van een lineair programmeringsformulering blijkt uit onze resultaten onhaalbaar vanuit computationeel standpunt wanneer private en publieke cloud bronnen gecombineerd worden. De voorgestelde heuristieken lossen die kwestie op en tonen aan dat significantie kostenwinsten geboekt kunnen worden op dat vlak. Bovendien zijn ze robuust ten aanzien van fouten in de afgeschatte duurtijd van de taken van een applicatie.

Deel 3 van deze thesis neemt het kosten-optimalisatieprobleem verder onder de loep. Door gebruik te maken van goedkopere cloud contracten waarvoor een voorafbetaling nodig is, kunnen cloud gebruikers hun infrastructuurkosten terugdrijven zonder zwaar in te moeten boeten aan flexibiliteit. Om optimaal gebruik te maken van zo'n contracten is echter inzicht nodig in hoe de benodigde hoeveelheid computerbronnen van een gebruiker of organisatie zich in de toekomst zal gedragen. Wanneer er te veel contracten aangekocht worden, kunnen de kosten immers snel oplopen, tot zelfs boven het niveau waarbij er geen gebruik wordt gemaakt van de goedkopere contracten. Onze aanpak in dit proefwerk maakt gebruik van voorspellingstechnieken om de toekomstige belasting te voorspellen, waarna het voorgestelde algoritme automatisch voorstellen genereert omtrent de hoeveelheid contracten en de types ervan die aangekocht moeten worden. Onze resultaten tonen aan dat het algoritme potentieel heeft om de kosten voor cloud infrastructuur met tot 30% terug te drijven, maar dat het gebruik van geavanceerde voorspellingstechnieken slechts een significant kostenvoordeel biedt wanneer de belasting niet-triviale maar wel-voorspelbare seizoenspatronen vertoont.

APPENDIX B

Other Publications

This chapter provides a short summary of the publications that were not incorporated in the main dissertation.

Evaluating the Divisible Load Assumption in the context of Economic Grid Scheduling with deadline-based QoS guarantees

W. Depoorter, R. Van den Bossche, K. Vanmechelen, J. Broeckhove
Proceedings of CCGrid 2009, May 18-21 2009, Shangai, China, pp. 452–459.

Abstract

The efficient scheduling of jobs is an essential part of any grid resource management system. At its core, it involves finding a solution to a problem which is NP-complete by reduction to the *knapsack problem.* Consequently, this problem is often tackled by using heuristics to derive a more pragmatic solution. Other than the use of heuristics, simplifications and abstractions of the workload model may also be employed to increase the tractability of the scheduling problem. A possible abstraction in this context is the use of *Divisible Load Theory* (DLT), in which it is assumed that an application consists of an *arbitrarily divisible load* (ADL). Many applications however, are composed of a number of atomic tasks and are only modularly divisible. In this paper we evaluate the consequences of the ADL assumption on the performance of economic scheduling approaches for grids, in the context of CPU-bound modularly divisible applications with hard deadlines. Our goal is to evaluate to what extent DLT can still serve as a useful workload abstraction for obtaining tractable scheduling algorithms in this setting. The focus of our evaluation is on the recently proposed tsfGrid heuristic for economic scheduling of grid workloads which operates under the assumptions of ADL. We demonstrate the effect of the ADL assumption on the actual instantiation of schedules and on the user value realized by the RMS. In addition we describe how the usage of a DLT heuristic in a high-level admission controller for a mechanism which does take into account the atomicity of individual tasks, can significantly reduce communication and computational overhead.

Evaluating Nested Virtualization Support

S. Verboven, R. Van den Bossche, O. Berghmans K. Vanmechelen, J. Broeckhove
Proceedings of PDCN 2011, February 15-17 2011, Innsbruck, Austria.

Abstract

Recent evolutions in the hard- and software used to virtualize x86 architectures have led to a rising popularity for numerous virtualization products and services. Virtualization has become a common layer between an operating system and physical hardware. Virtualized systems have already transparently replaced many physical server setups while virtual machines themselves are increasingly popular as a means to package, distribute and rapidly deploy software. Nested virtualization –running a virtual machine inside another virtual machine– seems a logical next step. Presently, however, there is little to no information available on nested virtualization support by widely used virtualization applications such as VMware, VirtualBox, Xen or KVM. In this contribution, we evaluate the feasibility of nested virtualization using a range of currently available hard- and software products. We briefly explain the different techniques behind the tested virtualization software and an analysis is made whether particular nested virtualization setups should be feasible. All theoretically possible options are explored and the results are presented and analyzed. We conclude with a presentation of the results of some initial performance experiments. Our results show that nested virtualization is currently possible in selected combinations with promising results regarding recent evolutions in hardware assisted virtualization.

APPENDIX C

Scientific Resume

Education

- 2003–2005, Bachelor Computer Science (Kandidaat in de Informatica)
  University of Antwerp, Belgium
- 2005–2007, Master Computer Science (Licentiaat in de Informatica)
  University of Antwerp, Belgium
- 2007–2008, Master Business Economics (Master Bedrijfseconomie)
  Ghent University, Belgium
- 2008–2014, PhD Candidate Computer Science
  University of Antwerp, Belgium

Teaching

Computer Science, University of Antwerp

- Introduction to Programming (BSc), 2008–2012
- Computer Systems and Architecture (BSc), 2008, 2010–2012
- Introduction to C++ (BSc), 2008–2012
- Bachelor Thesis (BSc), 2012–2014
- Distributed Computing (MSc), 2012–2014
- Topics in Distributed Computing (MSc), 2012–2013

Mathematics and Chemistry, University of Antwerp

- Introduction to Programming (BSc), 2011–2014

Biological and Bio-engineering Science, University of Antwerp

- Computer Skills (BSc), 2009–2014

Publications

- **Evaluating the Divisible Load Assumption in the context of Economic Grid Scheduling with deadline-based QoS guarantees**
  W. Depoorter, R. Van den Bossche, K. Vanmechelen, J. Broeckhove
  Proceedings of CCGrid 2009, May 18-21 2009, Shangai, China, pp. 452–459.

- **An evaluation of the benefits of fine-grained value-based scheduling on general purpose clusters**
  R. Van den Bossche, K. Vanmechelen, J. Broeckhove
  Proceedings of CCGrid 2010, May 17-20 2010, Melbourne, Australia, pp. 196–204.

- **An evaluation of the benefits of fine-grained value-based scheduling on general purpose clusters**
  R. Van den Bossche, K. Vanmechelen, J. Broeckhove
  Future Generation Computer Systems 27, 2011, pp. 1-9.

- **Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads**
  R. Van den Bossche, K. Vanmechelen, J. Broeckhove
  Proceedings of Cloud 2010, July 5-10 2010, Miami, USA, pp. 228-235.

- **Evaluating Nested Virtualization Support**
  S. Verboven, R. Van den Bossche, O. Berghmans K. Vanmechelen, J. Broeckhove
  Proceedings of PDCN 2011, February 15-17 2011, Innsbruck, Austria.

- **Cost-Efficient Scheduling Heuristics for Deadline Constrained Workloads on Hybrid Clouds**
  R. Van den Bossche, K. Vanmechelen, J. Broeckhove
  Proceedings of CloudCom 2011, November 29-December 1 2011, Athens, Greece, pp. 320-327.

- **Online Cost-Efficient Scheduling of Deadline-Constrained Workloads on Hybrid Clouds**
  R. Van den Bossche, K. Vanmechelen, J. Broeckhove
  Future Generation Computer Systems 29, 2013, pp. 973-985.

- **Optimizing a Cloud contract portfolio using Genetic Programming-based Load Models**
  S. Stijven, R. Van den Bossche, K. Vladislavleva, K. Vanmechelen, J. Broeckhove, M. Kotanchek
  Genetic Programming Theory and Practice XI, 2014.

- **Optimizing IaaS Reserved Contract Procurement using Load Prediction**
  R. Van den Bossche, K. Vanmechelen, J. Broeckhove
  Accepted for publication in *Proceedings of CLOUD 2014.*

- **IaaS Reserved Contract Procurement Optimisation with Load Prediction**
  R. Van den Bossche, K. Vanmechelen, J. Broeckhove
  Under review for publication in *Future Generation Computer Systems.*

Bibliography

[1] Michael A Rappa. The utility business model and the future of computing services. *IBM Systems Journal*, 43(1):32–42, 2004.

[2] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. An evaluation of the benefits of fine-grained value-based scheduling on general purpose clusters. *Future Generation Computer Systems*, 27(1):1–9, 2010.

[3] Chee Shin Yeo and Rajkumar Buyya. Pricing for Utility-driven Resource Management and Allocation in Clusters. *International Journal of High Performance Computing Applications*, 21(4):405–418, November 2007.

[4] Cynthia B. Lee and Allan E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 107–116, New York, NY, USA, 2007. ACM.

[5] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 22–30. IEEE Computer Society, 2002.

[6] Alvin AuYoung, Brent N. Chun, Chaki Ng, David Parkes, Amin Vahdat, and Alex C. Snoeren. Practical market-based resource allocation. Technical Report CS2007-0901, University Of California, San Diego, 2007.

[7] K. Lai, B. A. Huberman, and L. Fine. Tycoon: A distributed, market-based resource allocation system. Technical Report cs.DC/0412038, HP Labs, December 2004.

[8] Gunther Stuer, Kurt Vanmechelen, and Jan Broeckhove. A commodity market algorithm for pricing substitutable grid resources. *Future Generation Computer Systems*, 23(5):688–701, 2007.

[9] Kurt Vanmechelen, Wim Depoorter, and Jan Broeckhove. Economic grid resource management for cpu bound applications with hard deadlines. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 258–266, Washington, DC, USA, 2008. IEEE Computer Society.

[10] Kurt Vanmechelen and Jan Broeckhove. A comparative analysis of single-unit Vickrey auctions and commodity markets for realizing grid economies with dynamic pricing. In J. Altmann and D.J. Veit, editors, *Proceedings of the 4th International Workshop on Grid Economics and Business Models*, volume 4685 of *Lecture Notes in Computer Science*, pages 98–111, Heidelberg, 2007. Springer-Verlag.

[11] Chien-Min Wang, Hsi-Min Chen, Chun-Chen Hsu, and Jonathan Lee. Dynamic resource selection heuristics for a non-reserved bidding-based grid environment. *Future Generation Computer Systems*, 26(2):183 – 197, 2010.

[12] C. S. Yeo and Rajkumar Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Software Practice and Experience*, 36(13):1381–1419, nov 2006. ISSN = "0038-0644".

[13] Nicolas Dubé. *SuperComputing Futures: the Next Sharing Paradigm for HPC Resources.* PhD thesis, Université Laval, 2008.

[14] Kurt Vanmechelen. *Economic Grid Resource Management using Spot and Futures Markets.* PhD thesis, University of Antwerp, 2009.

[15] Donald Francis Ferguson. *The Application of Microeconomics to the Design of Resource Allocation and Control Algorithms.* PhD thesis, COLUMBIA UNIVERSITY, 1989.

[16] Rajkumar Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing.* PhD thesis, Monash University, Australia, 2002.

[17] Hesam Izakian, Ajith Abraham, and Behrouz Tork Ladani. An auction method for resource allocation in computational grids. *Future Generation Computer Systems*, 26(2):228 – 235, 2010.

[18] Leonid Hurwicz. On informationally decentralized systems. In C. B. McGuire and R. Radner, editors, *Decision and Organization: a Volume in Honor of Jacob Marshak*, volume 12 of *Studies in mathematical and managerial economics*, chapter 14, pages 297–336. North-Holland, 1972.

[19] J. Gomoluch and M. Schroeder. Market-based resource allocation for grid computing: A model and simulation. In M. Endler and D. Schmidt, editors, *Proceedings of the First International workshop on Middleware for Grid Computing*, pages 211–218, Rio De Janeiro, RJ, 2003. PUC-Rio.

[20] Florentina I. Popovici and John Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 36, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Kevin L. Mills and Christopher Dabrowski. Can economics-based resource allocation prove effective in a computation marketplace? *Journal of Grid Computing*, 6(3):291–311, 2008.

[22] I. Stoica and A. Pothen. A robust and flexible microeconomic scheduler for parallel computers, 1996.

[23] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the Maui scheduler. In *Proceedings of the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102, London, UK, 2001. Springer-Verlag.

[24] Peter Cramton, Yoav Shoham, and Richard Steinberg, editors. *Combinatorial Auctions*. MIT Press, Cambridge, MA, 2006.

[25] D. Lehmann, R. Maller, and T. Sandholm. The winner determination problem. In P. Cramton, Y. Shoham, and R. Steinberg, editors, *Combinatorial Auctions*, chapter 12, pages 297–317. MIT Press, Cambridge, MA, 2006.

[26] Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/, 2010.

[27] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and Rajkumar Buyya. Libra: a computational economy-based job scheduling system for clusters. *Software: Practice and Experience*, 34(6):573–590, 2004.

[28] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16(1):8–37, 1961.

[29] Björn Schnizler. *Resource Allocation in the Grid – A Market Engineering Approach*. PhD thesis, University of Karlsruhe, 2007.

[30] Cynthia Bailey Lee and Allan Snavely. On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions. *International Journal of High Performance Computing Applications*, 20(4):495–506, 2006.

[31] R. Wolski, J.S. Plank, J. Brevik, and T. Bryan. Analysing market-based resource allocation strategies for the computational grid. *International Journal of High-Performance Computing Applications*, 15:258–281, 2001.

[32] Zhu Tan and J.R. Gurd. Market-based grid resource allocation using a stable continuous double auction. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 283–290, Sept. 2007.

[33] U. Kant and D. Grosu. Double auction protocols for resource allocation in grids. In *Proceedings of the IEEE International Conference on Information Technology: Coding and Computing*, pages 366–371. IEEE Computer Society, 2005.

[34] Marek Wieczorek, Stefan Podlipnig, Radu Prodan, and Thomas Fahringer. Applying double auctions for scheduling of workflows on the grid. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[35] Dror G. Feitelson and A. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, pages 542–546, Washington, DC, USA, 1998. IEEE Computer Society.

[36] Sam Verboven, Peter Hellinckx, Frans Arickx, and Jan Broeckhove. Runtime prediction based grid scheduling of parameter sweep jobs. *Journal of Internet and Technology*, 11:47–54, 2010.

[37] Michael A. Iverson, Füsun Özgüner, and Gregory J. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, pages 263–270. IEEE Computer Society, 1996.

[38] Michael A. Iverson, Füsun Özgüner, and Lee C. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Transactions on Computers*, 48(12):1374–1379, 1999.

[39] Farrukh Nadeem, Muhammad Murtaza Yousaf, Radu Prodan, and Thomas Fahringer. Soft benchmarks-based application performance prediction using a minimum training set. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 71, Washington, DC, USA, 2006. IEEE Computer Society.

[40] Yuanyuan Zhang, Wei Sun, and Yasushi Inoguchi. Predict task running time in grid environments based on cpu load predictions. *Future Generation Computer Systems*, 24(6):489 – 497, 2008.

[41] Rajkumar Buyya, D. Abramson, and J. Giddy. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14:1507–1542, 2002.

[42] Lior Amar, Ahuva MuAlem, and Jochen Stösser. The power of preemption in economic online markets. In *Proceedings of the 5th international workshop on Grid Economics and Business Models*, pages 41–57, Berlin, Heidelberg, 2008. Springer-Verlag.

[43] Kurt Vanmechelen, Wim Depoorter, and Jan Broeckhove. A simulation framework for studying economic resource management in grids. In *Proceedings of the 8th International Conference on Computational Science*, pages 226–235, Krakow, Poland, 23-25 june 2008.

[44] W. Keith Edwards. *Core Jini*. 1999.

[45] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 2004.

[46] Marcos Dias de Assuncao, Alexandre di Costanzo, and Rajkumar Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 141–150, New York, NY, USA, 2009. ACM.

[47] Rajiv Ranjan, Aaron Harwood, and Rajkumar Buyya. A case for cooperative and incentive-based federation of distributed clusters. *Future Generation Computer Systems*, 24(4):280 – 295, 2008.

[48] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky Computing. *IEEE Internet Computing*, 13:43–51, 2009.

[49] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[50] Peter Mell and Tim Grace. The nist definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Information Technology Laboratory, Computer Security Division, September 2011.

[51] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[52] Marcos Dias Assunção, Alexandre Costanzo, and Rajkumar Buyya. A cost-benefit analysis of using cloud computing to extend the capacity of clusters. *Cluster Computing*, 13:335–347, 2010. 10.1007/s10586-010-0131-x.

[53] Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crăciun. Portable cloud applications – from theory to practice. *Future Generation Computer Systems*, 29(6):1417–1430, 2012.

[54] A. Edmonds, T. Metsch, A. Papaspyrou, and A. Richardson. Toward an open cloud standard. *Internet Computing, IEEE*, 16(4):15–25, july-aug. 2012.

[55] The Apache Software Foundation. libcloud, a unified interface to the cloud, http://libcloud.apache.org/, 2012.

[56] Apache Software Foundation. Deltacloud API, http://deltacloud.apache.org/, 2012.

[57] Johan Tordsson, Rubén S. Montero, Rafael Moreno-Vozmediano, and Ignacio M. Llorente. Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future Generation Computer Systems*, 28(2):358–367, 2012.

[58] Wubin Li, Johan Tordsson, and Erik Elmroth. Modeling for dynamic cloud scheduling via migration of virtual machines. In *Proceedings of the IEEE Third International Conference on Cloud Computing Technology and Science*, pages 163 –171, 29 2011-dec. 1 2011.

[59] José Luis Lucas-Simarro, Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Scheduling strategies for optimal service deployment across multiple clouds. *Future Generation Computer Systems*, 29(6):1431–1441, 2013.

[60] David Breitgand, Alessandro Maraschini, and Johan Tordsson. Policy-driven service placement optimization in federated clouds. Technical report, IBM Research Division, 2011.

[61] Jörg Strebel and Alexander Stage. An economic decision model for business software application deployment on hybrid cloud environments. In *Proceedings of Multikonferenz Wirtschaftsinformatik*, pages 195 – 206, Göttingen, 2010. Universitätsverlag Göttingen.

[62] A. Andrzejak, D. Kondo, and Sangho Yi. Decision model for cloud computing under sla constraints. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, pages 257–266, 2010.

[63] Sriram Kailasam, Nathan Gnanasambandam, Janakiram Dharanipragada, and Naveen Sharma. Optimizing service level agreements for autonomic cloud bursting schedulers. In *Proceedings of the 39th International Conference on Parallel Processing Workshops*, pages 285–294, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[64] Ulrich Lampe, Melanie Siebenhaar, Dieter Schuller, and Ralf Steinmetz. A cloud-oriented broker for cost-minimal software service distribution. In Wolfgang Ziegler Rosa M. Badia, editor, *Proceedings of the Second Optimising Cloud Services Workshop*, Oct 2011.

[65] Bahman Javadi, Jemal Abawajy, and Rajkumar Buyya. Failure-aware resource provisioning for hybrid cloud infrastructure. *Journal of Parallel and Distributed Computing*, 72(10):1318–1331, 2012.

[66] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag, Berlin Heidelberg, 2004.

[67] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. Parallel application characterization for multiprocessor scheduling policy design. In Dror G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 105–118. Springer-Verlag, 1996.

[68] Allen B Downey. A model for speedup of parallel programs. Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley, Jan 1997.

[69] Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. *SIG METRICS : Performance Evaluation Review*, 22:33–44, May 1994.

[70] Lawrence W. Dowdy. On the partitioning of multi-processor systems. Technical Report 88-06, Vanderbilt University, March 1988.

[71] Bibo Yang, Joseph Geunes, and William J. O'brien. Resource-constrained project scheduling: Past work and new directions. Technical report, Department of Industrial and Systems Engineering, University of Florida, 2001.

[72] Oumar Koné, Christian Artigues, Pierre Lopez, and Marcel Mongeau. Event-based milp models for resource-constrained project scheduling problems. *Computers & Operations Research*, 2009.

[73] José R. Correa and Michael R. Wagner. *LP-Based Online Scheduling: From Single to Parallel Machines*, volume 3509 of *Lecture Notes in Computer Science*, pages 196–209. Springer, 2005.

[74] Jean Damay, Alain Quilliot, and Eric Sanlaville. Linear programming based algorithms for preemptive and non-preemptive rcpsp. *European Journal of Operational Research*, 182:1012–1022, 2007.

[75] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009.

[76] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, volume 0, pages 124–131, Washington, DC, USA, May 2009. IEEE.

[77] Kavitha Ranganathan and Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 352–358, Washington, DC, USA, 2002. IEEE Computer Society.

[78] Ming Tang, Bu-Sung Lee, Xueyan Tang, and Chai-Kiat Yeo. The impact of data replication on job scheduling performance in the data grid. *Future Generation Computer Systems*, 22(3):254 – 268, 2006.

[79] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187 – 200, 2000.

[80] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer Berlin Heidelberg, 1997.

[81] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling – a status report. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2005.

[82] Alexandru Iosup, Ozan Sonmez, Shanny Anoep, and Dick Epema. The performance of bags-of-tasks in large-scale distributed systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 97–108, New York, NY, USA, 2008. ACM.

[83] Alvin AuYoung, Amin Vahdat, and Alex Snoeren. Evaluating the impact of inaccurate information in utility-based scheduling. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 38:1–38:12, New York, NY, USA, 2009. ACM.

[84] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

[85] Ahuva W. Mualem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12:529–543, 2001.

[86] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate? In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 253–263. Springer Berlin Heidelberg, 2005.

[87] Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 103–127. Springer Berlin Heidelberg, 2002.

[88] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove. Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads. In *Proceedings of the IEEE 3rd International Conference on Cloud Computing*, pages 228–235, 2010.

[89] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2003.

[90] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Cost-efficient scheduling heuristics for deadline constrained workloads on hybrid clouds. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, pages 320–327. IEEE Computer Society, 2011.

[91] Darin England and Jon B. Weissman. Costs and benefits of load sharing in the computational grid. In *Proceedings of the 10th international conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'04, pages 160–175, Berlin, Heidelberg, 2005. Springer-Verlag.

[92] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. *Future Generation Computer Systems*, 29(4):973–985, June 2013.

[93] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE Network*, 17(6):27–35, nov.-dec. 2003.

[94] Alok Shriram, Margaret Murray, Young Hyun, Nevil Brownlee, Andre Broido, Marina Fomenkov, and kc claffy. Comparison of public end-to-end bandwidth estimation tools on high-speed links. In Constantinos Dovrolis, editor, *Passive and Active Network Measurement*, volume 3431 of *Lecture Notes in Computer Science*, pages 306–320. Springer Berlin Heidelberg, 2005.

[95] Sung-Ju Lee, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Rodrigo Fonseca. Measuring bandwidth between planetlab nodes. In Constantinos Dovrolis, editor, *Passive and Active Network Measurement*, volume 3431 of *Lecture Notes in Computer Science*, pages 292–305. Springer Berlin Heidelberg, 2005.

[96] Ningning Hu and P. Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE Journal on Selected Areas in Communications*, 21(6):879–894, September 2006.

[97] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of Resource Provisioning Cost in Cloud Computing. *IEEE Transactions on Services Computing*, 5(2):164–177, April 2012.

[98] Siqi Shen, Kefeng Deng, Alexandru Iosup, and Dick Epema. Scheduling Jobs in the Cloud Using On-demand and Reserved Instances. In *Proceedings of the 19th International conference Euro-Par Parallel Processing*, pages 242–254. Springer-Verlag, 2013.

[99] Yu-Ju Hong, Jiachen Xue, and Mithuna Thottethodi. Selective commitment and selective margin: Techniques to minimize cost in an IaaS cloud. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, pages 99–109. Ieee, April 2012.

[100] Cheng Tian, Ying Wang, Feng Qi, and Bo Yin. Decision model for provisioning virtual resources in Amazon EC2. In *Proceedings of the International Conference on Network and Service Management*, pages 159–163, 2012.

[101] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimal virtual machine placement across multiple cloud providers. In *Proceedings of the IEEE Asia-Pacific Services Computing Conference*, pages 103–110, 2009.

[102] José Luis Lucas-Simarro, Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Dynamic placement of virtual machines for cost optimization in multi-cloud environments. In *Proceedings of the International Conference on High Performance Computing and Simulation*, pages 1–7, 2011.

[103] Sean Stijven, Ruben Van den Bossche, Ekaterina Vladislavleva, Kurt Vanmechelen, Jan Broeckhove, and Mark Kotanchek. Optimizing a cloud contract portfolio using genetic programming-based load models. In Rick Riolo, Jason H. Moore, and Mark Kotanchek, editors, *Genetic Programming Theory and Practice XI*, Genetic and Evolutionary Computation, pages 47–63. Springer New York, 2014.

[104] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[105] Tina Yu, Shu-Heng Chen, and Tzu-Wen Kuo. Discovering financial technical trading rules using genetic programming with lambda abstraction. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 2, pages 11–30. Springer, Ann Arbor, 13-15 May 2004.

[106] N. Nikolaev and H. Iba. Genetic programming of polynomial harmonic models using the discrete fourier transform. In *Proceedings of the 2001 Congress on Evolutionary Computation*, pages 902–909, 2001.

[107] Massimo Santini and Andrea Tettamanzi. Genetic programming for financial time series prediction. In *Proceedings of the 4th European Conference on Genetic Programming*, EuroGP '01, pages 361–370, London, UK, UK, 2001. Springer-Verlag.

[108] Witthaya Panyaworayan and Georg Wuetschner. Time series prediction using a recursive algorithm of a combination of genetic programming and constant optimization. *Facta Universitatis. Series: Electronics and Energetics*, 15(2):265–279, August 2002.

[109] Alexandros Agapitos, Matthew Dyson, Jenya Kovalchuk, and Simon Mark Lucas. On the genetic programming of time-series predictors for supply chain management. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1163–1170, Atlanta, GA, USA, 12-16 July 2008. ACM.

[110] Katya Rodriguez-Vazquez and Peter J. Fleming. Genetic programming for dynamic chaotic systems modelling. In *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 22–28, Mayflower Hotel, Washington D.C., USA, 6-9 July 1999. IEEE Press.

[111] Roy Schwaerzel and Tom Bylander. Predicting currency exchange rates by genetic programming with trigonometric functions and high-order statistics. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 955–956, New York, NY, USA, 2006. ACM.

[112] Evolved Analytics LLC. *DataModeler Release 8.0 Documentation.* Evolved Analytics LLC, 2011.

[113] N. Wagner, Z. Michalewicz, M. Khouja, and R.R. McGregor. Time series forecasting for dynamic environments: The dyfor genetic program model. *IEEE Transactions on Evolutionary Computation*, 11(4):433–452, 2007.

[114] Rob J. Hyndman and Andrey V. Kostenko. Minimum Sample Size Requirements For Seasonal Forecasting Models. *Foresight: The International Journal of Applied Forecasting*, 6:12–15, 2007.

[115] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with the parallel workloads archive. Technical report, 2012.

[116] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, November 2011.

[117] Sreeram Ramachandran. Web metrics: Size and number of resources, 2010.

[118] George E P Box, Gwylim M Jenkins, and Gregory C Reinsel. Time Series Analysis: Forecasting and Control. *Journal of Time Series Analysis*, 3:746, 2013.

[119] Charles C Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20(1):5–10, 2004.

[120] Rob J Hyndman and Yeasmin Khandakar. Automatic Time Series Forecasting: The forecast Package for R. *Journal Of Statistical Software*, 27:1–22, 2008.

[121] Leonard J Tashman and Michael L Leach. Automatic forecasting software: A survey and evaluation. *International Journal of Forecasting*, 7(2):209–230, 1991.

[122] Ulrich Küsters, B D McCullough, and Michael Bell. Forecasting software: Past, present and future. *International Journal of Forecasting*, 22(3):599–615, 2006.

[123] J W Taylor. Short-term electricity demand forecasting using double seasonal exponential smoothing. 54:799–805, 2003.

[124] Robert G. Brown. Exponential Smoothing for Predicting Demand. Technical report, 1956.

[125] R Development Core Team. R: A language and environment for statistical computing, 2008.

[126] Peter Whitle. *Hypothesis Testing in Time Series Analysis*. PhD thesis, 1951.

[127] Rob J Hyndman, Anne B Koehler, Ralph D Snyder, and Simone Grose. A state space framework for automatic forecasting using exponential smoothing methods. *International Journal of Forecasting*, 18(3):439–454, July 2002.

[128] R. J. Hyndman, A. B. Koehler, J. K. Ord, and R. D. Snyder. *Forecasting with exponential smoothing: the state space approach*. 2008.

[129] Sarah Gelper, Roland Fried, and Christophe Croux. Robust forecasting with exponential and Holt-Winters smoothing. *Journal of Forecasting*, 29(3):285–300, 2010.