ELASTIC CLOUD COMPUTING FOR QOS-AWARE DATA PROCESSING

Shigeru Imai

Submitted in Partial Fullfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Approved by:
Dr. Carlos A. Varela, Chair
Dr. Stacy Patterson
Dr. Mohammed Zaki
Dr. Rajkumar Buyya

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[May 2018]
Submitted April 2018

CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

ACKNOWLEDGMENT x

ABSTRACT xii

1. INTRODUCTION 1

1.1 Background of Cloud Elasticity 1

1.1.1 Data Processing Models 1

1.1.2 Cloud Service Models 2

1.1.3 Cloud Deployment Types 4

1.1.4 QoS-Aware Elastic Resource Allocation 5

1.1.5 Scheduling Techniques 7

1.2 Contributions 8

1.3 Outline 10

2. ELASTIC BATCH DATA PROCESSING 11

2.1 Auto-Scaling Using Application-Level Migration 11

2.1.1 Introduction 11

2.1.2 Cloud Operating System 12

2.1.3 Experiments 13

2.1.4 Summary 16

2.2 Cost-Optimal Heterogeneous Virtual Machine Scheduling over Hybrid Clouds 17

2.2.1 Introduction 17

2.2.2 Workload-Tailored Elastic Compute Unit 18

2.2.3 VM Scheduling Algorithm 20

2.2.4 Experiments 26

2.2.5 Summary 28

3. ELASTIC MICRO-BATCH DATA PROCESSING 30

3.1 Introduction 30

3.2 Air Traffic Management Problem 32

3.2.1 Problem Formulation 32

3.2.2 Lagrangean Decomposition 34
3.3 Elastic Air Traffic Management Middleware 36
3.3.1 Background 36
3.3.2 Application Implementation 37
3.3.3 Middleware Architecture 37
3.4 Virtual Machine Scheduling 38
3.4.1 Performance Characterization of ILP Optimization 38
3.4.2 Resource Prediction Model 41
3.4.3 Elastic Scheduling Algorithms 42
3.4.3.1 Baseline Scheduling 42
3.4.3.2 Speculative Scheduling 44
3.4.3.3 VM Allocation Policy 45
3.5 Evaluation 45
3.5.1 Experimental Settings 45
3.5.2 Elastic Behavior Confirmation 46
3.5.2.1 Nationwide Dataset 46
3.5.2.2 Dallas Dataset 48
3.5.3 Comparison with Static Scheduling 48
3.5.4 Comparison with Auto Scaling 51
3.6 Summary 52
4. SUSTAINABLE ELASTIC STREAM DATA PROCESSING 53
4.1 Introduction 53
4.2 Background of Elastic Stream Processing 57
4.2.1 Distributed Stream Processing Systems 57
4.2.1.1 Comparison of Distributed Stream Processing Systems . . . 57
4.2.1.2 Scaling Stream Processing Applications 60
4.2.2 Elastic Stream Processing Systems 61
4.2.2.1 Performance Metrics 61
4.2.2.2 Summary of Elastic Stream Processing Systems 63
4.3 A Framework for Sustainable Elastic Stream Data Processing 64
4.3.1 Maximum Sustainable Throughput 64
4.3.2 Sustainable Elastic Stream Data Processing Framework 66
4.4 Summary 67

5. MAXIMUM SUSTAINABLE THROUGHPUT PREDICTION 68
5.1 Introduction 68
5.2 Related Work 69
5.3 MST Prediction Framework 70
5.3.1 Linear Regression 71
5.3.2 Framework Overview 71
5.3.3 MST Prediction Models 72
5.3.4 Phase 1: VM Subset Selection 75
5.3.4.1 VM Subset Selection Method 75
5.3.4.2 VM Subset Selection Results 76
5.3.5 Phase 2: Model Training & Selection 78
5.4 Evaluation of MST Prediction 81
5.5 Discussion 84
5.6 Summary 86
6. UNCERTAINTY-AWARE ELASTIC VIRTUAL MACHINE SCHEDULING . . . 87
6.1 Introduction 87
6.2 Formulation of VM Scheduling Problem 88
6.3 VM Scheduling Techniques 89
6.3.1 Baseline MST 89
6.3.2 Online Model Learning 90
6.3.3 Uncertainty-Awareness for MST 90
6.3.4 Uncertainty-Awareness for Workload Forecasting 92
6.3.4.1 Workload Forecasting 92
6.3.4.2 Highest Workload Estimation 93
6.3.4.3 VM Scheduling 94
6.3.5 Uncertainty-Awareness for both MST and Workload Forecasting . . . 95
6.4 Evaluation Setup 96
6.4.1 Test Applications and Workloads 96
6.4.2 Offline MST Model Training 98
6.4.3 Workload Forecasting Model Training 99
6.5 Evaluation 99
6.5.1 Common Experimental Settings 99
6.5.1.1 Simulation Time Horizon 99
6.5.1.2 Workloads and Test Applications 100

6.5.1.3 Hypothetical Ground Truth MST . . . 101
6.5.1.4 Baseline Scheduling . . . 101
6.5.1.5 Evaluation Metrics . . . 102
6.5.2 Evaluation: Scheduling Policy vs. QoS & Cost . . . 102
6.5.2.1 Experimental Settings . . . 103
6.5.2.2 Scheduling Results . . . 104
6.5.3 Evaluation: QoS Satisfaction Target vs. QoS & Cost . . . 106
6.5.4 VM Allocation Sequence . . . 107
6.6 Related Work . . . 107
6.7 Summary . . . 109
7. CONCLUSION AND FUTURE DIRECTIONS . . . 110
7.1 Chapter Summary . . . 110
7.2 Future Directions . . . 112
7.2.1 Future Elastic Resource Allocation Framework . . . 112
7.2.2 Elastic Resource Allocation for Serverless Computing . . . 114
7.2.3 Improvements to Presented Techniques . . . 115
REFERENCES . . . 117
APPENDICES
A. Peak Calculations for Model 1 . . . 128

LIST OF TABLES

1.1 Different aspects of elastic resource allocation addressed in this thesis. 9

2.1 Test VM configurations for fixed VMs vs. COS. 14

2.2 Amazon EC2 on-demand instances (as of September 2013). 19

2.3 ECU and WECU values for the trap application (1 WECU = 2758.54 tasks/sec). 20

3.1 Amazon EC2 VM instance types used in experiments (information as of November 2015). 40

3.2 Non-linear transforms used for linear regression. 42

3.3 Key notations used in scheduling algorithms. 42

3.4 VM hours, cost, and latency violations for elastic and static scheduling algorithms (Nationwide dataset). 50

3.5 VM hours, cost, and latency violations for elastic and static scheduling algorithms (Dallas dataset). 50

4.1 Comparison of representative distributed stream processing systems (as of 2018). 58

4.2 Summary of recent elastic stream processing systems. 64

5.1 Best VM subsets $\mathcal{S}^*$ and prediction errors in RMSE for variable maximum VM counts ($M_{\text{train}} \in 5, ..., 32$). 78

5.2 Weights of Models 1 and 2 after training. 82

6.1 Selected ARMA models for the test workloads. 99

6.2 Workloads and test applications (MST models) used for evaluations. 100

6.3 VM scheduling policies for evaluation. 103

LIST OF FIGURES

1.1 Layered architecture of cloud services. 3

1.2 Cloud deployment types explored in this thesis. 4

1.3 High-level view of QoS-aware elastic resource allocation. 6

1.4 Chapter organization of the thesis. 10

2.1 System architecture of the Cloud Operating System. 13

2.2 VM CPU utilization from Heat Diffusion problem running on the Cloud Operating System. 15

2.3 Throughput of the Heat Diffusion problem running on the Cloud Operating System. 15

2.4 Comparison of throughput for the Heat Diffusion problem between fixed VMs and the Cloud Operating System. 16

2.5 Workload scalability over a hybrid cloud. 18

2.6 Relative performance of Amazon EC2 instances and private physical nodes (node A and node B). the performance of `m1.small` is 1.0. 20

2.7 Cost-optimal configurations among other possible configurations. 24

2.8 Runtime results for (a) ECU-based and (b) WECU-based resource configurations. 28

3.1 Number of U.S. commercial flights on January 18th, 2014 (created from data available on [65]). 31

3.2 Example of the simplified air traffic management problem. 33

3.3 Architecture of the elastic air traffic management middleware framework. . . . 39

3.4 Characteristics of the ILP problem execution time. 41

3.5 Experimental results for the Nationwide dataset. 47

3.6 VM allocation sequence for the speculative scheduling algorithm created from the Nationwide dataset. 48

3.7 Experimental results for the Dallas dataset. 49

3.8 CPU utilization and VM allocation by a threshold-based auto scaling. 52

4.1 Common real-time stream processing environment. 54

4.2 Example time series of input data rates and processing throughput. 55

4.3 Examples of scaling an application topology with three processing units ($u = 3$) to three and five machines ($m = 3$ and 5), respectively. Each machine has two vCPUs ($\alpha = 2$) and one vCPU is assigned per thread ($\beta = 1$). 60

4.4 Maximum sustainable throughput measurement environment. 65

4.5 Convergence of throughput for a web access log processing stream application. 66

4.6 Proposed sustainable elastic stream data processing framework. 67

5.1 Overview of the MST prediction framework. 72

5.2 MST prediction results using the best VMs subset: $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$. X-axis: number of VMs. Y-axis: MST [Mbytes/sec]. 79

5.3 MST prediction results for typical use-case benchmarks: (a) Grep, (b) Rolling Count, (c) Unique Visitor, (d) Page View, and (e) Data Clean, and a machine learning application: (f) VHT, using $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$ for Models 1 and 2. X-axis: number of VMs. Y-axis: MST [Mbytes/sec]. 82

5.4 Validation error (RMSE) and selected models for the typical use-case benchmarks. Models are trained with $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$. 83

5.5 Prediction error (MAPE) for the typical use-case benchmarks using mean value of actual MST samples and prediction made by the proposed framework. . . . 84

6.1 Probability density function for normal distribution $\mathcal{N}(\hat{\tau}(m) - \lambda(t), \hat{\sigma}_\tau^2)$. The shaded area corresponds to the value of $\Pr[\tau(m) - \lambda(t) \geq 0]$. 91

6.2 Finding the highest expected workload in the scheduling cycle $[t, t + C)$. 94

6.3 Workloads used for evaluation. Each workload has one week of data: (a) World Cup 98 (6/29/1998-7/5/1998, time in UTC), (b) Tweets (4/10/2016-4/16/2016, time in EDT), and (c) ADS-B (8/13/2017-8/19/2017, time in UTC). 97

6.4 Selected MST prediction models after training. Models from (a) Grep to (g) Rolling Hashtag Count are trained with samples obtained from up to 24 VMs in $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$, whereas (h) Rolling Flight Distances is trained with samples obtained from up to 14 VMs in $\mathcal{S}_{\text{flight}} = \{1, 2, 3, 5, 6, 8, 9, 11, 12, 14\}$. 98

6.5 Example of hypothetical ground truth probability distribution created from the measured MST samples for the Grep benchmark. 101

6.6 Average (a) QoS satisfaction rates and (b) Relative costs across the all applications for different scheduling policies in Table 6.3. QoS satisfaction target: $\rho = 0.95$. Error bars show $\pm$ 1 standard deviation. 105

6.7 QoS satisfaction rates and relative costs for the #0, #1, #5, and #7 scheduling policies in Table 6.3. QoS satisfaction target: $\rho = 0.95$. Error bars show $\pm$ 1 standard deviation. 106

6.8 QoS satisfaction target ρ vs. actual QoS satisfaction rates and relative costs. Error bars show $\pm$ 1 standard deviation. 107

6.9 Scheduling results for the Grep application with the FIFA world cup 1998 website access workload (6/29/1998-7/5/1998): (a) Input workload and allocated MST, (b) Allocated number of VMs, and (c) Backlogged data. 108

ACKNOWLEDGMENT

I have been fortunate to have two wonderful advisors: Prof. Carlos A. Varela and Prof. Stacy Patterson. I would like to express the deepest appreciation to Prof. Carlos A. Varela. Throughout my time at RPI, he has been supporting me with great enthusiasm and patience. He has always motivated me with inspirational questions and given me great freedom in my research. I am also deeply grateful to Prof. Stacy Patterson for her thoughtful and practical guidance. Her advice on distributed systems and optimization problems has been instrumental to this dissertation. It is an honor to be one of her first PhD students at RPI. This dissertation would not have been possible without their support.

My gratitude extends to my dissertation committee members, Prof. Zaki and Prof. Buyya, for their insightful comments and questions to my dissertation. Their feedback inspired me to envision a greater view of my work.

I would also like to thank the current and former Worldwide Computing Laboratory members, especially Paul, Freddie, Sida, Alessandro, Wennan, Carlos Gomez, Rory, Richard, Ping, Qingling, Yousaf, and Wei, for interesting technical discussions and enjoyable moments at the lab. Thanks to Teruhiko, Juntao, Xiaohui, Yao, Ning, Antwane, Will, Jamal, Saori, Colin, Yurie, Steve, Chikako, Aoi, Toru, and friends from SUNY Albany for their friendship. I will never forget countless fun moments I shared with them. Thanks to the faculty and staff of the Computer Science Department, especially Chris Coonrad and late Terry Hayden, for always being there to help me and making me feel at home.

I would like to express my gratitude to Yamada Corporation Fellowship, NSF, and Air Force Scientific Office of Research for the financial support I received over the years. Amazon Web Services Cloud Credits for Research and Google Cloud Platform Credit Award generously granted me to use their cloud resources.

Finally, I would like to thank my family for their love and moral support. Most of all,

I appreciate my beloved wife Miki for her everyday support, understanding, patience, and encouragement during my long PhD journey. Thank you.

ABSTRACT

Infrastructure-as-a-Service (IaaS) clouds such as Amazon EC2 offer various types of virtual machines (VMs) through pay-per-use pricing. *Elastic resource allocation* allows us to allocate and release VMs as computing demand changes while satisfying Quality-of-Service (QoS) requirements. In this thesis, we explore QoS-aware elastic resource allocation for three different data processing models: batch, micro-batch, and streaming.

First, we present two frameworks for elastic batch data processing. The first elastic batch data processing framework supports autonomous VM scaling using application-level migration. It does not require any prior knowledge about the target application, but dynamically reconfigures the application to keep the CPU utilization within a certain range. The second framework uses *Workload-tailored Elastic Compute Units* as a measure of computing resources analogous to Amazon EC2's ECUs. Given a deadline, our framework finds the cost-optimal resource configuration of heterogeneous VMs to satisfy the required throughput.

Next, we propose an elastic micro-batch data processing framework for continuous air traffic optimization. Air traffic optimization is commonly formulated as an integer linear programming (ILP) problem. For continuous optimization, we periodically solve ILP problems with regular intervals, where each problem is a micro-batch data processing job. Since the fluctuating number of flights creates dynamically changing computational demand, our framework predicts future workload and proactively schedules VMs to solve the ILP problems in a timely manner.

Finally, we propose a framework for sustainable elastic stream processing based on the concept of *Maximum Sustainable Throughput (MST)*. It is the maximum processing throughput a streaming application can process indefinitely for a number of VMs. Stream processing is sustainable if the system's MST is always greater than the input data rates of incoming workload. Using MST and future workload prediction models, our framework proactively

schedules VMs to keep the stream processing sustainable. It explicitly incorporates uncertainties in both MST and workload prediction models, and estimates the number of VMs to satisfy a certain probability criteria.

Our studies show that QoS-aware elastic data processing is effective for these processing models in both performance scalability and cost savings. For batch processing, elastic resource scheduling helps achieve the target QoS metrics such as CPU utilization and job completion time. For both micro-batch and stream processing with fluctuating workloads, QoS-aware elastic scheduling saves up to 49% cost compared to a static scheduling that covers the peak workload to achieve a similar level of throughput QoS satisfaction. These results show potential for future fully automated cloud computing resource management systems that efficiently enable truly elastic and scalable general-purpose workload.

CHAPTER 1
INTRODUCTION

Cloud computing has changed the way people use computing resources through its pay-per-use cost model. Public cloud providers such as Amazon Web Services and Google Cloud Platform offer various types of computing services with different costs and Quality-of-Service (QoS) policies, enabling us to use the power of cloud computing to process a huge amount of data in a cost-efficient manner. One of the key aspects of cloud computing is *elasticity* [1]. *Elastic resource allocation* allows us to allocate and release virtual machines (VMs) dynamically as computing demand changes. However, to decide when and how many VMs to allocate and release is not a trivial task. Moreover, depending on the use case, there are different ways to exploit cloud elasticity. In this chapter, we first introduce the background of cloud elasticity in Section 1.1. In Section 1.2, we present how we address different aspects of elastic resource allocation in our contributions. In Section 1.3, we show the outline of this thesis.

1.1 Background of Cloud Elasticity

1.1.1 Data Processing Models

We show some of the representative distributed data processing models as follows.

- **Batch processing** is used to process stored datasets all at once with high throughput. Depending on the size of the dataset, processing latency can be very long up to hours or even days. One of the typical use cases is Extract, Transform, Load (ETL) process, which involves data extraction from multiple sources, data transformation (*e.g.*, clean, join), and data loading to a target data storage (*e.g.*, relational database). Popular frameworks include Hadoop [2] and Spark [3].

- **Stream processing** is used to process an unbounded stream of events in near real-time. Unlike batch processing, stream processing digests one event at a time or multiple events collected in a time window. Thus, processing latency is expected to be much lower than batch processing. There are many applications that require low latency, for example, anomaly detection [4], [5], Twitter trend analysis [6], [7], and taxi traffic analysis [8]. Distributed scalable stream processing systems such as Storm [6], Flink [9], and Samza [10] have been actively used and developed.

- **Micro-batch processing** repeats batch processing with smaller datasets in a short time interval. Spark Streaming [11] takes a micro-batch approach and is built on top of Spark to take advantage of Spark's fault tolerance and high throughput. Alternatively, we can manually submit small batch jobs to a batch data processing system periodically. One of the benefits of this approach is that we can use batch processing algorithms while generating outputs periodically just as stream processing.

1.1.2 Cloud Service Models

As National Institute of Standards and Technology (NIST) defines, cloud computing has several types of service models including: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) [12]. The relationship between these models is shown in Figure 1.1. In the IaaS model, the cloud service provider only offers bare virtual machines (VMs)—abstractions of physical machines (PMs)—with various prices depending on the quality of service. Therefore, application developers need to either manually implement non-functional concerns such as security, scalability, and fault-tolerance or use middleware to address them. IaaS affords great flexibility for developing general-purpose applications and installing almost any software on VMs. In the PaaS model, the service provider offers an application development framework that takes care of most non-functional concerns. Even though the framework is restricted to a particular class of applications, application developers can benefit from the framework and concentrate on their own application

models. In the SaaS model, the service provider offers complete application software from the cloud to the end user. For example, to build an SaaS service, an SaaS developer can use PaaS, IaaS, or a bare metal physical infrastructure with increasing complexity.

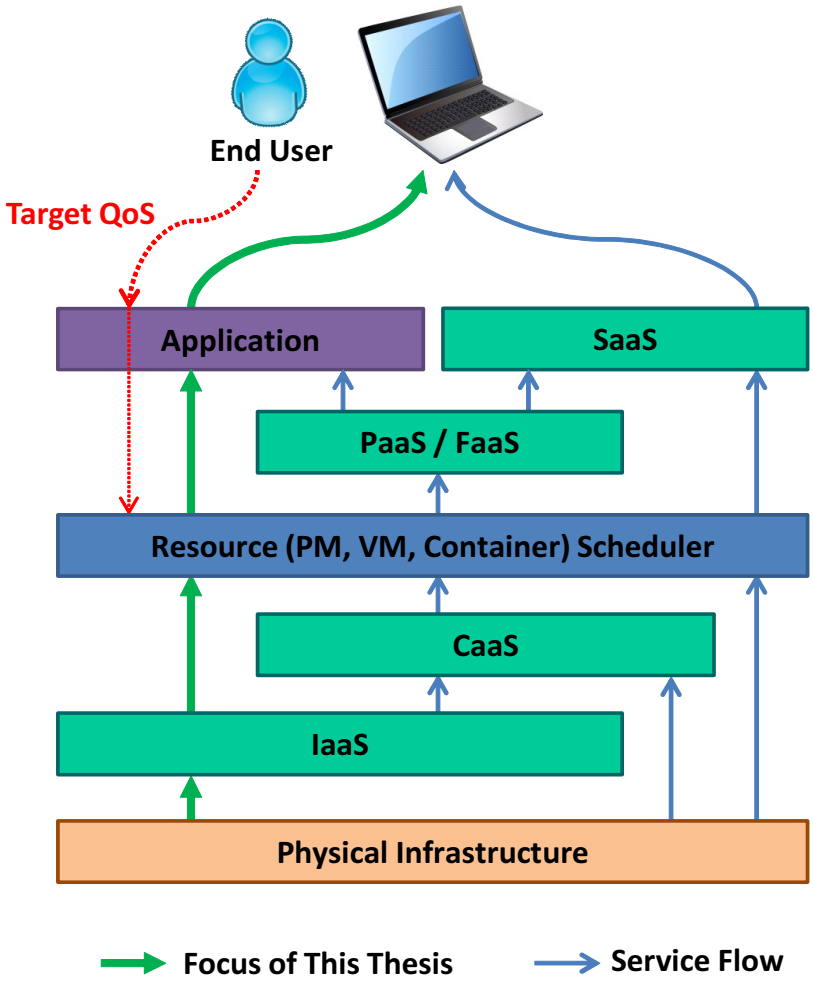


Figure 1.1. Layered architecture of cloud services.

In addition to these standard models, Container-as-a-Service (CaaS) and Function-as-a-Service (FaaS) are emerging as shown in Figure 1.1. In the CaaS model, the cloud service provider offers *container* as the unit of resource allocation. Unlike VMs offer hardware-level virtualization, containers offer operating system-level virtualization based on two key features provided by the Linux kernel [13]: 1) kernel namespaces to isolate the container from the host (*e.g.*, a process inside the container cannot see other processes outside the

container) and 2) control groups (known as cgroups) to limit resources (*e.g.*, CPU, memory, etc.) used by a container. Since containers are light-weight compared to VMs, they start up quickly in a few seconds. In the FaaS model, the cloud service provider enables *serverless* application development [14]. The user of FaaS service does not need to rent VMs, but provide event-driven application logic (*i.e.*, function) and pay only for the compute time consumed by functions. The functions are invoked upon the reception of events and the FaaS service provider automatically scales up resources as the number of events grows.

Regardless of which cloud service model we choose, to realize automated elasticity from the end user perspective, there must be a resource scheduler to control computational resources: PMs, VMs, or containers. In this thesis, we focus on the IaaS model and study on elastic VM scheduling.

1.1.3 Cloud Deployment Types

The cloud service models shown in Section 1.1.2 can be provided through the deployment types as shown in Figure 1.2. The following three deployment types are recognized as standard by NIST [12].

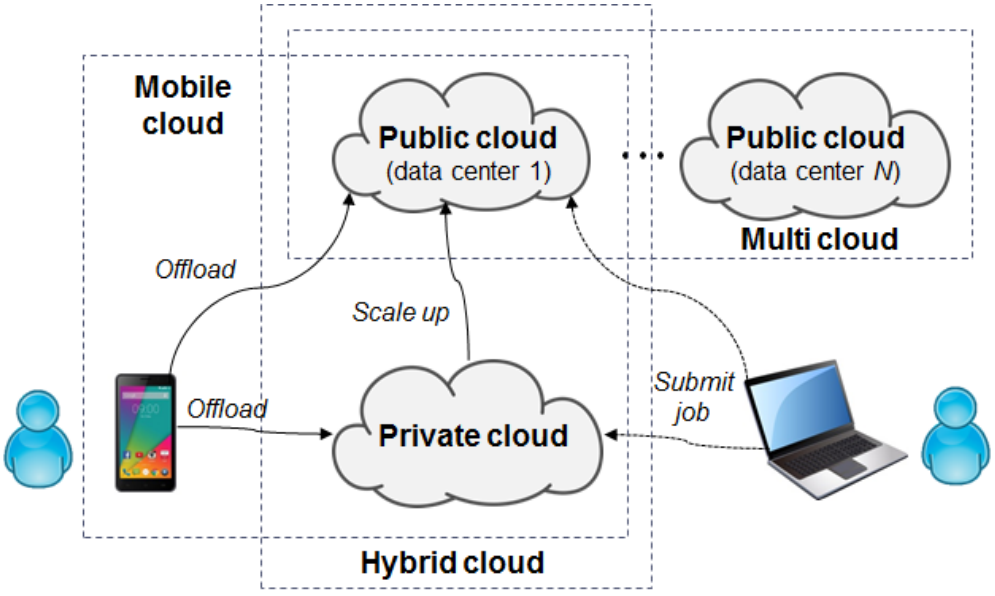


Figure 1.2. Cloud deployment types explored in this thesis.

- **Private cloud**: The cloud infrastructure is owned and used by a single organization.
- **Public cloud**: The cloud infrastructure is owned by a commercial service provider and is open for the public use.
- **Hybrid cloud**: The cloud infrastructure consists of two or more different types of clouds. Typically, public clouds are used to process temporary workload spikes.

In addition to the above standard deployment types, the following relatively new deployment types are emerging.

- **Multi-cloud**: The cloud infrastructure consists of multiple clouds (thus, hybrid cloud is a type of multi-cloud). Unlike the intercloud model [15], [16] requires an agreement between multiple cloud providers to give users transparent access to the cloud, the multi-cloud model does not need such agreement. That is, multi-cloud model is more client-centric, and the interoperability between multiple clouds is maintained by the user using libraries (*e.g.*, Apache Libcloud [17]) or brokers [18]. There are various benefits of using multi-cloud. For example, processing geo-distributed datasets using distributed data centers to take advantage of data locality, storing backup data across different cloud service providers to increase the data redundancy, and avoiding to depend on only one cloud service provider (*i.e.*, vendor lock-in).
- **Mobile-cloud**: To augment the limited computing resources of mobile devices, clouds are used through mobile communication network or WiFi [19]. To avoid potential large latency caused by accessing public clouds through Wide Area Network, proximate computing infrastructure called *cloudlet* has also been proposed [20]. By offloading resource-intensive tasks from the mobile device to clouds, we can increase the mobile device's battery life and enhance user experience [21].

1.1.4 QoS-Aware Elastic Resource Allocation

Figure 1.3 shows a high-level view of QoS-aware elastic resource allocation (often referred to as *auto-scaling* [22]) in IaaS clouds. It repeats the following steps to satisfy a

Quality-of-Service (QoS) metric requested by the user.

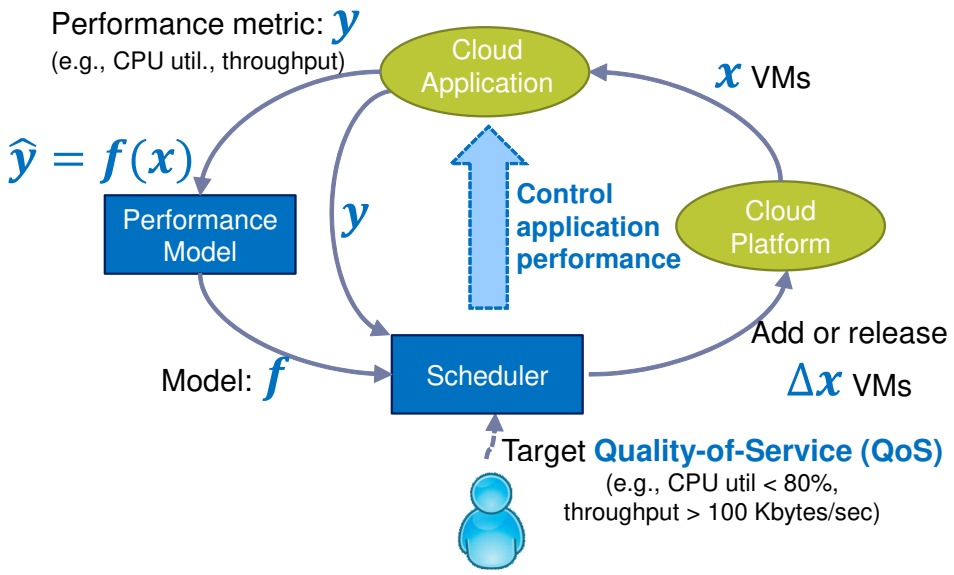


Figure 1.3. High-level view of QoS-aware elastic resource allocation.

1. The cloud platform such as Amazon EC2 provides x VMs to the cloud application.

2. The cloud application runs on the VMs and produces some performance metric y such as CPU utilization or throughput.

3. (Optional) The performance model $\hat{y} = f(x)$ is updated using a sample (x, y) to estimate the performance produced by x VMs.

4. The scheduler takes one of the following steps depending on the scheduling approach (see Section 1.1.5 for details):

   (a) *Model-based* approach: Using the performance model f, the scheduler estimates the required number of VMs x' to satisfy the target QoS. It then requests allocation or deallocation of Δx $(= |x - x'|)$ VMs.

   (b) *Model-free* approach: The scheduler checks if the monitored performance metric y satisfies the target QoS. If not, it requests allocation or deallocation of Δx VMs (Δx is constant).

In the cloud services mentioned in Section 1.1.2, QoS is typically used in *Service-Level Agreements (SLAs)*. A SLA is a contract between the service provider and customer that includes the level of service guarantees defined by QoS metrics such as availability and response time. If the service provider violates a service guarantee, they pay some penalty as agreed to in the SLA. For example, the SLA for Amazon EC2 says that Amazon makes reasonable efforts to maintain at least 99.95% of service availability, but in case the availability becomes less than 99.95%, it gives back 10% of the total charges paid by the customer [23]. Commercial cloud providers mainly focus on service availability for their SLAs; they do not offer any performance guarantees [24].

In academic research, QoS-aware elastic VM allocation commonly optimizes cost while meeting a performance QoS. For example, Buyya et al. propose an architecture for SLA-oriented resource provisioning [25] and illustrate an implementation of deadline-constrained resource provisioning with Aneka platform [26]. Example application areas for cost-efficient resource allocation include Bag-of-Tasks [27]–[30], MapReduce [31]–[33], data streaming [34]–[37], and workflow [38]–[41].

1.1.5 Scheduling Techniques

We categorize existing scheduling techniques used in elastic resource allocation from two aspects: 1) whether the scheduling is based on a model; and 2) whether the scheduling involves future workload prediction.

Model-based vs. Model-free : In the *model-based* scheduling, the scheduler decides the quantity of computational resources x (*e.g.*, number of VMs, CPUs, parallelism) using a model $f(x)$ that estimates a metric of interest (*e.g.*, latency, throughput). If the scheduler is to meet certain QoS constraints, an estimate of x is obtained as follows:

$$\hat{x} = \underset{x}{\operatorname{argmin}} \, [\textit{QoS constraints with } f(x)]. \tag{1.1}$$

Examples of common QoS constraints include:

$$\text{Latency constraint} \quad : \quad f(x) \leq \textit{target latency}, \tag{1.2}$$

$$\text{Throughput constraint} \quad : \quad f(x) \geq \textit{input workload}, \tag{1.3}$$

$$\text{CPU utilization constraint} \quad : \quad f(x) \leq \textit{target upper CPU utilization}. \tag{1.4}$$

Unlike the model-based scheduling, *model-free* scheduling adds or removes fixed amount of resources without using any models until QoS constraints are satisfied. Thus, $f(x)$ in constraints (1.2)-(1.4) can be replaced with an observed metric y in the model-free scheduling.

Proactive vs. Reactive : To implement an elastic resource scheduler for fluctuating workloads (*i.e.*, micro-batch and streaming), we can either *reactively* adjust the number of VMs at run-time in response to resource utilization metric changes (*e.g.*, CPU, memory, network), or *proactively* estimate the number of VMs using prediction models for application performance and future workload. The former includes a threshold-based approach, as used in AWS AutoScaling (Step Scaling) [42], and reinforcement learning [43], [44]. In the latter approach, various types of application performance models are proposed as we describe in Section 4.2.2.1. Auto-regression and ARMA [45] are widely used to predict time series and have been successfully applied to elastic cloud systems [46], [47]. If the system foresees a growing demand in advance, it can allocate new VMs and make them ready before the spike actually occurs so that we can avoid the performance degradation.

1.2 Contributions

In this thesis, we focus on the perspective of IaaS users and explore QoS-aware elastic resource allocation from several different aspects of cloud computing as shown in Table 1.1. The contributions of this thesis are summarized as follows:

- An elastic middleware framework based on application-level migration as the reconfig-

Table 1.1. Different aspects of elastic resource allocation addressed in this thesis.

	Processing Model			
	Batch		Micro-batch	Streaming
Cloud Deployment	Private	Hybrid	Public	Public
Scheduling	Reactive, Model-free	Reactive, Model-based	Proactive, Model-based	Proactive, Model-based
Performance Metric	CPU utilization	Job finish time	Job finish time	MST[1]
Performance Model	None	Linear	Non-linear	Non-linear

uration strategy [48].

- The notion of *Workload-tailored Elastic Compute Unit (WECU)* as a unit of computing power analogous to Amazon EC2's ECUs, but customized for a specific workload. A dynamic programming-based VM scheduling algorithm to obtain the cost-optimal VM configuration [28].

- A micro-batch elastic middleware framework that is designed to solve integer linear programming problems periodically [49].

- A framework for sustainable elastic stream processing [50] that consists of the following sub contributions:
  - A cost-efficient maximum throughput prediction framework for stream processing applications, which statistically determines the best set of VM configurations to train prediction models [51].
  - A robust VM scheduling method that incorporates uncertainties in both application performance and workload prediction models [37].

[1]Maximum Sustainable Throughput, see Section 4.3.1 for details.

1.3 Outline

The rest of the proposal is organized as shown in Figure 1.4. Chapters are organized according to the processing models. Chapter 2 presents two elastic *batch* data processing frameworks: Section 2.1 shows an auto-scaling framework using application-level migration, and then Section 2.2 shows a cost-optimal scheduling framework over hybrid clouds. Chapter 3 shows an elastic *micro-batch* data processing framework for an air traffic optimization problem. Chapters 4, 5, and 6 are about sustainable elastic *stream* data processing. Chapters 4 first introduces background and a framework for sustainable elastic stream processing. Chapter 5 shows a framework for maximum sustainable throughput prediction, and Chapter 6 presents an uncertainty-aware VM scheduling framework that is designed to consider uncertainties from both application performance and workload prediction models. Chapter 7 summarizes results from each chapter and discusses potential future directions.

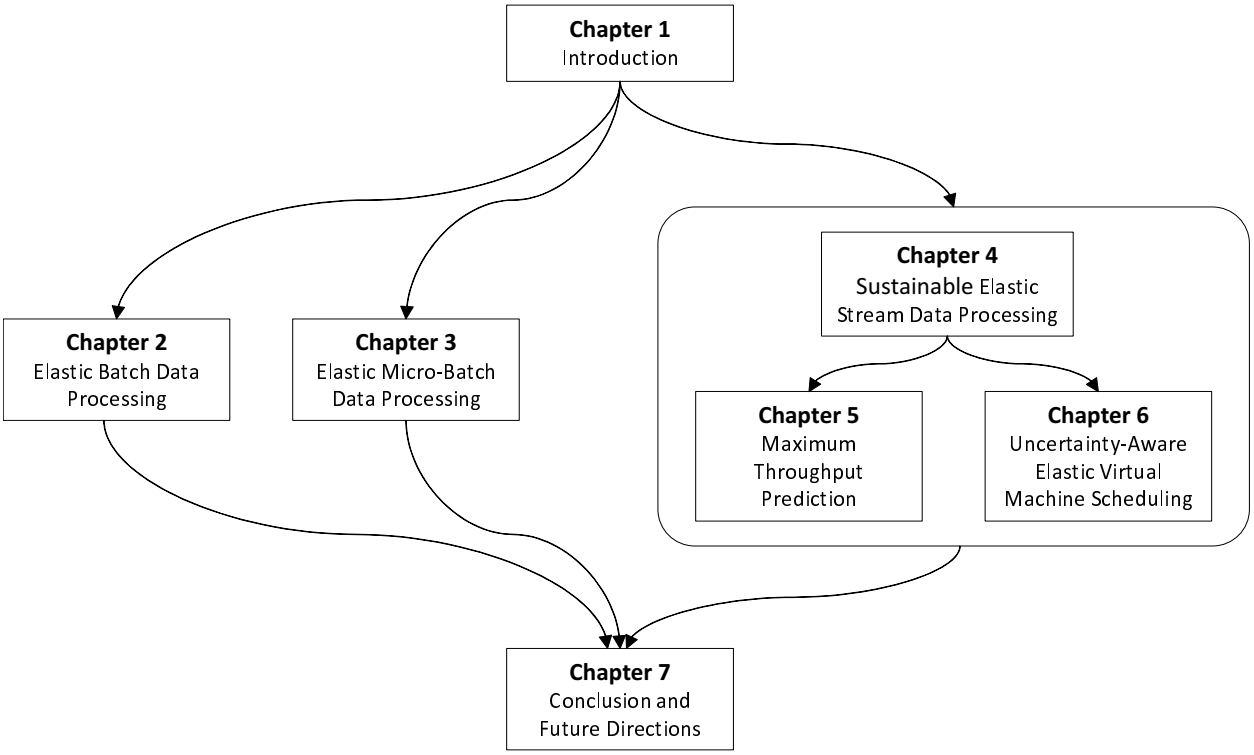


Figure 1.4. Chapter organization of the thesis.

CHAPTER 2
ELASTIC BATCH DATA PROCESSING

2.1 Auto-Scaling Using Application-Level Migration

2.1.1 Introduction

One way to exploit scalability afforded by cloud computing is to use VM migration [52], [53], [54], [55]. For instance, Sandpiper [56] monitors CPU, network, and memory usage, and then predicts the future usage of these resources based on the profile created from the monitored information. If a physical machine gets high resource utilization, Sandpiper tries to migrate a VM with higher resource utilization to another physical machine with lower resource utilization and balances the load between physical machines. As a result, the migrated VM gets better resource availability and therefore succeeds to scale up its computation.

On the other hand, it is also possible to achieve load balancing and scalability with finer granularity using application-level migration. Due to its smaller footprint compared to the VM's, application-level migration takes less time and uses less network resources. Also, it can achieve better load balancing and scaling that cannot be done by the VM-level coarse granularity. However, migration or coordination of distributed execution is not transparent for application developers in general. One solution is using an application platform with transparent application migration capabilities. Simple Actor Language System and Architecture (SALSA) [57] is an actor-oriented programming language that simplifies

Portion of this chapter previously appeared in: Shigeru Imai, Thomas Chestna, and Carlos A. Varela, "Elastic scalable cloud computing using application-level migration", in *Proc. IEEE/ACM Int'l Conf. on Utility and Cloud Computing (UCC 12)*, 2012, pp. 91-98.

Portion of this chapter previously appeared in: Shigeru Imai, Thomas Chestna, and Carlos A. Varela, "Accurate resource prediction for hybrid IaaS clouds using workload-tailored elastic compute units," in *Proc. IEEE/ACM Int'l Conf. on Utility and Cloud Computing (UCC 13)*, 2013, pp.171-178.

dynamic reconfiguration for mobile and distributed computing through its features such as universal naming and transparent migration over the Internet. Applications composed of SALSA actors can be easily reconfigured at runtime through actor migration.

In this section, we present the Cloud Operating System (COS), a middleware framework that is based on SALSA. It supports 1) opportunistic creation and removal of VMs over private clouds and 2) autonomous actor migration on VMs to enable cloud computing applications to effectively scale up and down. Using COS, application programmers can focus on their problem of interest and leave resource management to COS. Actors on COS autonomously migrate between VMs if it is beneficial in terms of resource availability in the target node, communication cost with other actors, and the cost for migration.

2.1.2 Cloud Operating System

The system architecture of COS is shown in Figure 2.1. Whereas the COS manager makes decisions whether it allocates or deallocates VMs, Internet Operating System (IOS) [58] works as a load-balancer. It tries to balance the workload among the VMs by autonomously migrating SALSA actors. To avoid thrashing behavior due to frequent VM creation and termination, the following heuristics is implemented in the COS manager. We choose this threshold approach because it is simple yet effective to avoid thrashing behavior as we have seen in [58].

- The COS manager decides to create a new VM only if it has received high CPU utilization events from all of the running nodes within a certain time range (*e.g.*, 10 seconds). The reason is that there may still remain under-utilized VMs and IOS agents may be able to balance the load and decrease the overall CPU utilization of VMs.

- Some workloads alternate between high and low CPU usage. If criteria of terminating VMs are too strict, this type of workload suffers very much. Let the constants to detect low and high CPU utilization be k_LOW and k_HIGH respectively for checking past n

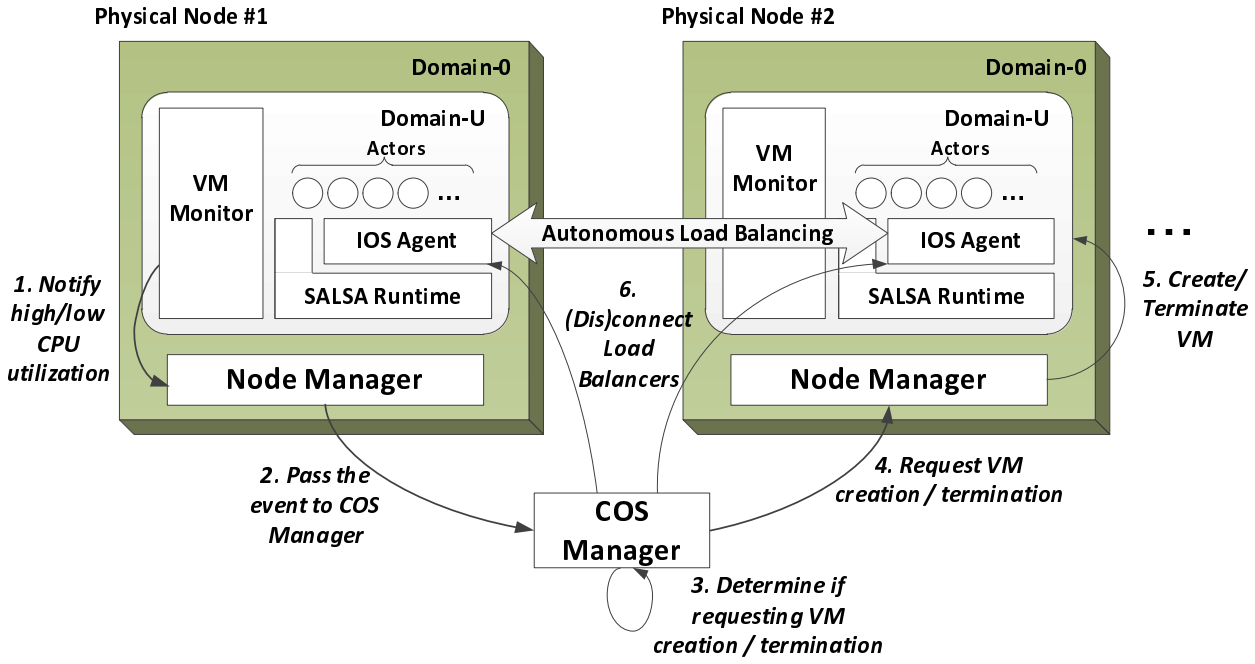


Figure 2.1. System architecture of the Cloud Operating System.

utilization. We set a higher value on k_LOW than k_HIGH to make VM termination more difficult than creation.

- A VM Monitor sends a low CPU utilization event to a Node Manager only after it detects persistent low CPU utilization since it takes some time until an IOS agent gets some workload from other IOS agents.

2.1.3 Experiments

We conducted experiments to confirm how COS automatically scales up and finds an appropriate VM configuration for the application. We also compared the performance between COS and fixed VM cases.

Experimental settings The workload is a Heat Diffusion problem, a communication-intensive workload. It simulates heat transfer in a two-dimensional grid in an iterative fashion. At each iteration, the temperature of a single cell is computed by averaging the temperatures of neighboring cells. We ran the problem for 300 iterations. It is developed in

Table 2.1. Test VM configurations for fixed VMs vs. COS.

Test Case	vCPU/VM	PM	VM	Total vCPUs	Memory [MB/VM]
Fixed VM	4	1	1	4	1024
	4	2	2	8	1024
	4	3	3	12	1024
COS	4	dynamically determined at runtime			1024

SALSA using actors to perform the distributed computation. Each actor is responsible for computing a sub-block of the grid and the actors have to communicate with each other to get the temperatures on boundary cells.

We use three physical machines and create only one VM per physical machine. Each of the physical machines has quad core Opteron processors running at 2.27 GHz, and they were connected via 1-Gbit Ethernet. Ubuntu Linux 12.04 LTS with Xen hypervisor version 3.4.1 was installed on all the machines. While COS dynamically changes the number of VMs at runtime, the fixed VM cases use one, two, and three VMs respectively from the start to the end of the Heat simulation. Table 2.1 summarizes the test VM configurations.

Results As shown in Figure 2.2, COS successfully reacted to the high CPU utilization and created new VMs. First it started with the first VM (VM1) only, and then created the second VM (VM2) at around 21 seconds. Once both VM1 and VM2 got highly utilized, COS created the third VM (VM3) at around 92 seconds. Since the actors were gradually migrated by IOS agents from VM1 to VM3, not from VM2, the CPU utilization of VM3 increased whereas the CPU utilization of VM1 decreased almost by half. COS successfully created VMs when extra computing power was demanded. Figure 2.3 shows the throughput of the Heat Diffusion simulation. It is clear that the throughput gets higher as the number of VMs increases.

Figure 2.4 shows the relationship between the number of vCPUs and the execution time. COS uses 9.002 vCPUs on average, whereas the fixed VMs cases use 4, 8, and 12 vCPUs for 1, 2, and 3 VMs, respectively. It is clear that more use of vCPUs makes the

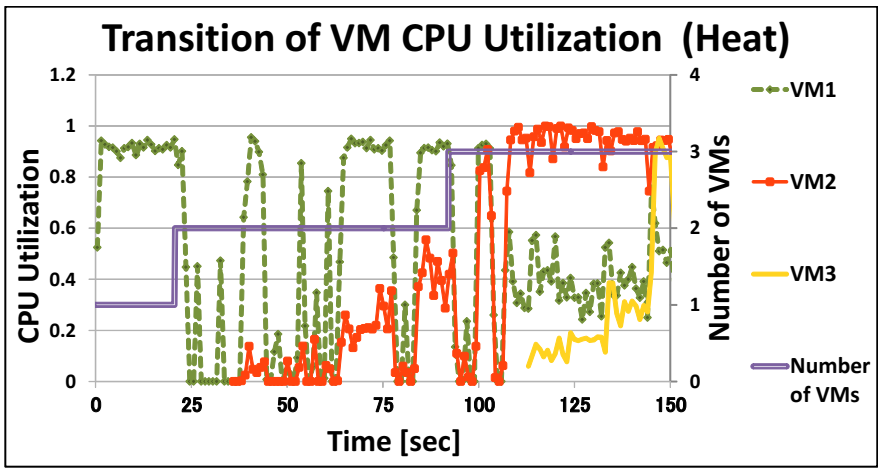


Figure 2.2. VM CPU utilization from Heat Diffusion problem running on the Cloud Operating System.

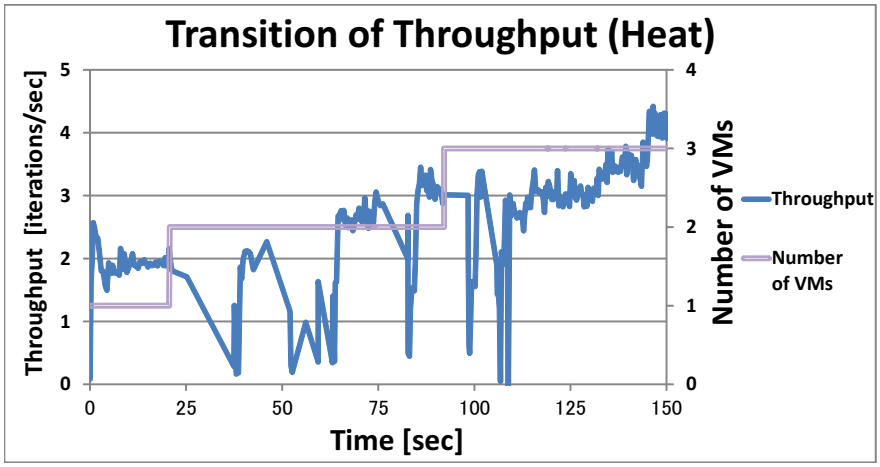


Figure 2.3. Throughput of the Heat Diffusion problem running on the Cloud Operating System.

execution times faster. Also, the graph suggests that relationship between the number of vCPUs and the execution time is linear. Suppose the cost per time is proportional to the number of vCPUs, the total cost, the product of the number of vCPUs and the execution time, is almost constant. Therefore, using 3 VMs is the best way to go for the Heat Diffusion problem; however, COS does better than the cases of 1 VM and 2 VMs. It performs well if we consider the fact that COS does not have any knowledge about the complexity of the problem and the number of vCPUs is dynamically determined at runtime.

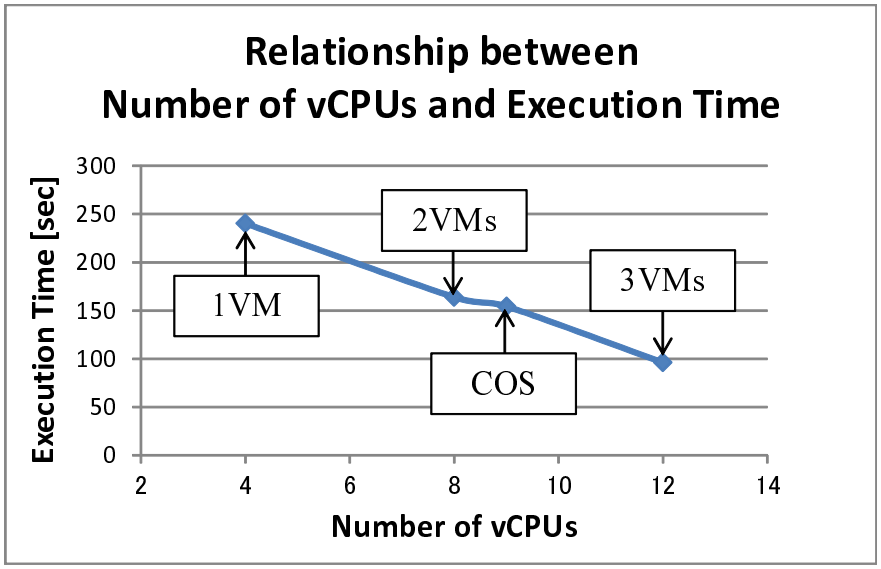


Figure 2.4. Comparison of throughput for the Heat Diffusion problem between fixed VMs and the Cloud Operating System.

2.1.4 Summary

In this section, we introduced a middleware framework, the Cloud Operating System, that supports autonomous cloud computing workload elasticity and scalability based on application-level migration. To scale a workload up, COS creates new VMs and migrates application actors to these new VMs. To scale a workload down, COS consolidates actors into fewer VMs and terminates idle VMs. COS only requires cloud applications to contain migratable components and does not impose any further restrictions on workloads. Using application-level migration, workloads can be elastic and scalable autonomously through middleware-driven dynamic reconfigurations.

Future work includes exploring automating more fine-grained resource management in COS by dynamically reconfiguring the number of vCPUs and the memory size of VMs (see [59] for the potential performance impact of these reconfigurations performed manually.) Finally, we also plan to connect mobile device applications to hybrid clouds (*e.g.*, see [60]) using application-level migration to gain elasticity, scalability, and efficiency, in a general-purpose programming framework.

2.2 Cost-Optimal Heterogeneous Virtual Machine Scheduling over Hybrid Clouds

2.2.1 Introduction

Growing demand for "Big Data" analytics applications requires a corresponding growth in "big computing" resources, making hybrid clouds an increasingly attractive infrastructure. A hybrid cloud enables users who have access to their own private cloud to scale out to public computing resources only occasionally. Especially for massively-parallel applications, for which the "cost-associativity" quality of the cloud computing model holds [1], we can get results faster by paying more money for public cloud resources; however, orchestrating distributed VM instances in hybrid clouds in a cost-efficient manner is a non-trivial task.

Amazon created the *Elastic Compute Unit (ECU)* as a measure of computing power of their various VM instance types depending on the price and QoS they offer to help users choose appropriate virtual machines. Their explanation of ECU is as follows: "We use several benchmarks and tests to manage the consistency and predictability of the performance of an EC2 Compute Unit. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor" [61]. ECU helps as a relative performance measure for a wide range of workloads, but it cannot be accurate for all types of workloads. Moreover, in the hybrid cloud model, ECU is not defined for private computing resources. Therefore, to accurately schedule resources not only on the public cloud, but also on the private cloud, the computational power of private computing resources must be quantified for better hybrid cloud performance prediction. To get accurate performance predictions and associated cost reduction over hybrid clouds, we introduce the notion of *Workload-tailored Elastic Compute Unit (WECU)* as a unit of computing power for a specific type of workload on a specific virtual or physical machine type. WECU is obtained by actually running the target workloads on available VM instances on the target machine for a short period.

In addition to improving the performance prediction based on WECU, we also consider application-level migration to support dynamic workload scalability. As we have shown in

Section 2.1, when dynamically scaling up the computation of long-running farmer-worker programs, migrating the workers from a private cloud to a public cloud is an effective approach because it is light-weight compared to VM-level migration and is able to let the workers continuously run without restarting [48]. Figure 2.5 illustrates the workload scalability over a hybrid cloud.

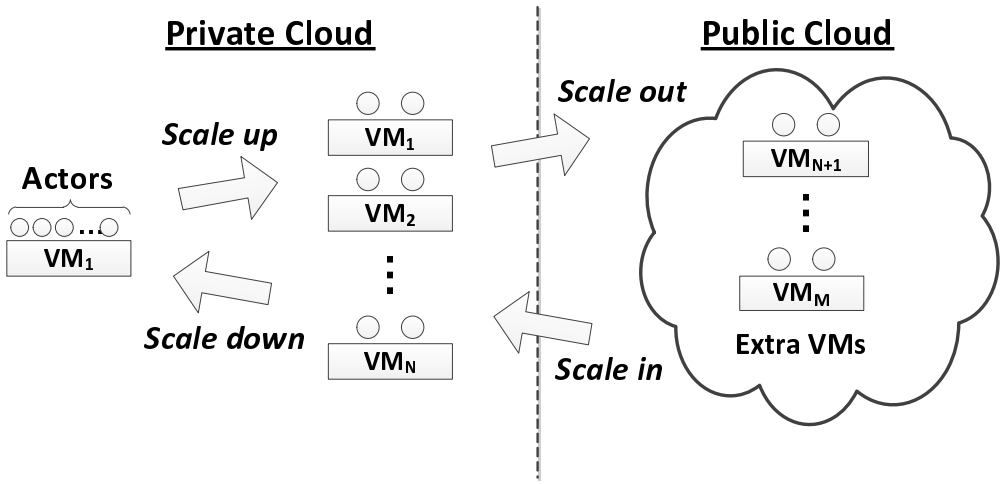


Figure 2.5. Workload scalability over a hybrid cloud.

In this section, we present the notion of WECU and propose a cost-optimal VM scheduling algorithm for heterogeneous VM types based on dynamic programming. We also show WECU's performance predictability for a massively-parallel application written in SALSA.

2.2.2 Workload-Tailored Elastic Compute Unit

There are quite a few VM instance types available in Amazon EC2 associated with various ECUs and prices. some of the instance types are shown in Table 2.2. Amazon provides ECUs as indicators of performance, but it may not be precise depending on applications. To find out the actual performance difference between instance types, we did a preliminary experiment evaluating a simple benchmark application called Trap written in SALSA on each instance type as well as on two other nodes, A and B, in our private cloud. Trap is a massively parallel application and computes a trapezoidal numerical integration for a given

function with a given interval (see Section 5 for details). The task size is defined as the number of trapezoid.

Table 2.2. Amazon EC2 on-demand instances (as of September 2013).

instance type	vCPU	ECU	price[USD]	price/ECU
m1.small	1	1	0.06	0.06
m1.medium	1	2	0.12	0.06
m1.large	2	4	0.24	0.06
m1.xlarge	4	8	0.48	0.06
m3.xlarge	4	13	0.5	0.038
m3.2xlarge	8	26	1	0.038
c1.medium	2	5	0.145	0.029
c1.xlarge	8	20	0.58	0.029

For task sizes ranging from 100,000 to 1,600,000, we observe relative runtime performance results in Figure 2.6. in this Figure, the base performance is defined as the runtime for the `m1.small` instance and the other relative performances are divided by the `m1.small`'s runtime. as the graph clearly shows, ECU does not always show the exact performance differences for the Trap application. for example, `c1.medium` has five ECUs while its relative performance is only about four, also `c1.xlarge` has 26 ECUs while its relative performance is ranging from 12 to a little over 16.

These results tell us that ECUs are not very accurate in predicting the performance of the Trap application, and thus we need to define a new performance measurement unit called *Workload-tailored Elastic Compute Unit (WECU)*. WECU is defined as follows:

Definition of WECU : *One WECU corresponds to the throughput of a one-ECU instance on a specific workload. General WECU values associated with arbitrary instance types processing the same workload are defined by dividing their throughput by the base throughput.*

In short, it is a relative performance unit based on the actual workload throughput. For the Trap application and instances used in the preliminary experiment, we define the base throughput as `m1.small`'s throughput, that is, 1 WECU = 2758.54 tasks/sec. All obtained

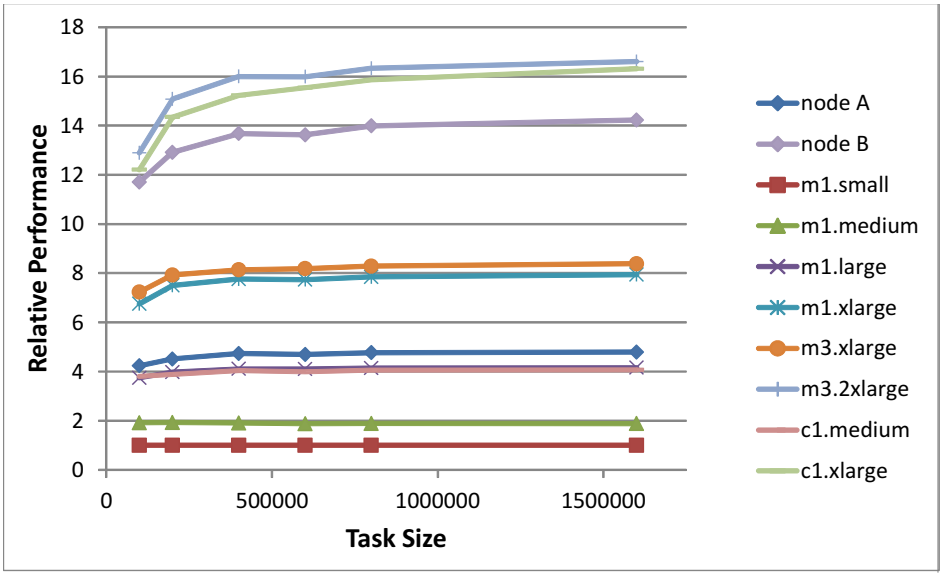


Figure 2.6. Relative performance of Amazon EC2 instances and private physical nodes (node A and node B). the performance of `m1.small` is 1.0.

values of WECUs are shown in Table 2.3. Note these values are computed after the performance converges. Since we will use WECU for long-running workloads, we want to use the throughput in steady-state conditions.

Table 2.3. ECU and WECU values for the trap application (1 WECU = 2758.54 tasks/sec).

Instance Type	ECU	WECU	Price[USD]/WECU
m1.small	1	1.0	0.06
m1.medium	2	1.882	0.0632
m1.large	4	4.156	0.0594
m1.xlarge	8	7.940	0.0617
m3.xlarge	13	8.380	0.0610
m3.2xlarge	26	16.604	0.0615
c1.medium	5	4.070	0.0359
c1.xlarge	20	16.316	0.0366
node A	N/A	4.795	N/A
node B	N/A	14.223	N/A

2.2.3 VM Scheduling Algorithm

The VM scheduling algorithm consists of four steps as shown below:

- **Step 1.** Compute target throughput and set a corresponding computational power requirement.
- **Step 2.** Check if the private cloud can satisfy the computational power requirement.
- **Step 3.** If Step 2 cannot satisfy the requirement, use extra resources in the public cloud.
- **Step 4.** Assign workers to the instances allocated in Steps 2 and 3.

The details of each step are explained below in order.

Step 1: Target Throughput and Computational Requirement

We are given the following inputs:

- $t_{deadline}$: deadline to finish the tasks,
- t_{curr}: current time,
- $tasks$: total number of tasks to process,
- $tasks_{done}$: total number of completed tasks,
- λ: throughput per WECU,
- Δ_{curr}: current throughput, and
- τ: throughput threshold for activating reconfiguration.

The target throughput Δ_{target} is computed as follows:

$$\Delta_{target} = (tasks - tasks_{done})/(t_{deadline} - t_{curr}).$$

For non-terminating workloads, we assume a target throughput is given in tasks/second.

If $|\frac{\Delta_{target}-\Delta_{curr}}{\Delta_{target}}| < \tau$, the algorithm does nothing and quits, otherwise it computes a required computational power η_{target} as follows:

$$\eta_{target} = \Delta_{target}/\lambda. \tag{2.1}$$

Note that η_{target} is in WECU units, which is defined in Section 2.2.2. We need to achieve η_{target} with computing instances collectively allocated from the private and public clouds so that the target throughput Δ_{target} can be maintained.

Step 2: Private Cloud Resource Configuration

We are given the following inputs:

- $R_{priv} = \{(type_1, \eta_1, cpu_1, num_1), ..., (type_N, \eta_N, cpu_N, num_N)\}$: N types of VM instances available in the private cloud, where $type_i$ is the VM instance type, η_i is the computational power, cpu_i is the number of virtual CPUs, and num_i is the available number of the i-th instance type respectively; and
- η_{target}: target computational power computed in Eq. (2.1).

Algorithm 1 outputs the following:

- A_{priv} : a set of instances to be allocated in the private cloud. ith element is a 4-tuple $(type_i, \eta_i, cpu_i, num_i)$,
- σ_{priv} : total computational power provided by A_{priv}, and
- η_{remain} : remaining computational power needed to satisfy target throughput.

Algorithm 1 allocates resources giving priority to instance types in the order they appear in R_{priv}. It simply deducts available computing power from η_{remain}. If $\eta_{remain} \leq 0$, then the algorithm outputs A_{priv} and goes to Step 4 to assign workers, otherwise it proceeds to Step 3 to further allocate more instances from the public cloud.

input : R_{priv}, η_{target}
output: $A_{priv}, \sigma_{priv}, \eta_{remain}$
$A_{priv} = \emptyset$; $\sigma_{priv} = 0.0$; $\eta_{remain} = \eta_{target}$; $i = 1$;
while $i \leq N$ and $0 < \eta_{remain}$ **do**
  if $\lceil \eta_{remain}/\eta_i \rceil \leq num_i$ **then**
    $num = \lceil \eta_{remain}/\eta_i \rceil$;
  end
  else
    $num = num_i$;
  end
  $A_{priv} = A_{priv} \cup \{(type_i, \eta_i, cpu_i, num)\}$;
  $\sigma_{priv} = \sigma_{priv} + num \times \eta_i$;
  $\eta_{remain} = \eta_{remain} - num \times \eta_i$;
  $i = i + 1$;
end
return A_{priv}, η_{remain};

Algorithm 1. Private cloud resource configuration.

Step 3: Cost Optimal Public Cloud Resource Configuration

We are given the following inputs:

- $R_{pub} = \{(type_1, \eta_1, cpu_1, price_1), ..., (type_M, \eta_M, cpu_M, price_M)\}$: M types of VM instances available in the public cloud, where $type_i$ is the VM instance type, η_i is the computational power, cpu_i is the number of virtual CPUs, and $price_i$ is the hourly price of the ith instance type respectively; and
- η_{target}: η_{remain} obtained in Step 2.

The minimal cost that satisfies arbitrary computational power η is given as follows:

$$COST(\eta) = \min_{1 \leq i \leq M} \begin{cases} price_i & (\eta \leq \eta_i) \\ price_i + COST(\eta - \eta_i) & (\text{otherwise}) \end{cases} \tag{2.2}$$

It compares the cost of choosing instance i out of M instance types recursively until it finds an instance which has larger computational power than the required η. Using a dynamic programming algorithm directly derived from Eq. (2.2), the following outputs are obtained:

- A_{pub}: instances to be allocated from the public cloud,
- $cost_{pub}$: total cost of A_{pub}, and
- σ_{pub}: total computational power of the instances in A_{pub},

where

$$\begin{aligned} A_{pub} &= \{(type_1, \eta_1, price_1, num_1), ..., (type_L, \eta_L, price_L, num_L)\}, \\ cost_{pub} &= \sum_{i\text{-th instance type} \in A_{pub}} price_i \times num_i, \\ \sigma_{pub} &= \sum_{i\text{-th instance type} \in A_{pub}} \eta_i \times num_i. \end{aligned}$$

Our dynamic programming algorithm's running time is $O(\eta M)$. Cost-optimality of Eq. (2.2) can be visually confirmed by Figure 2.7. As the Figure shows, obtained resource configurations produce cost-optimal configurations among other possible ones.

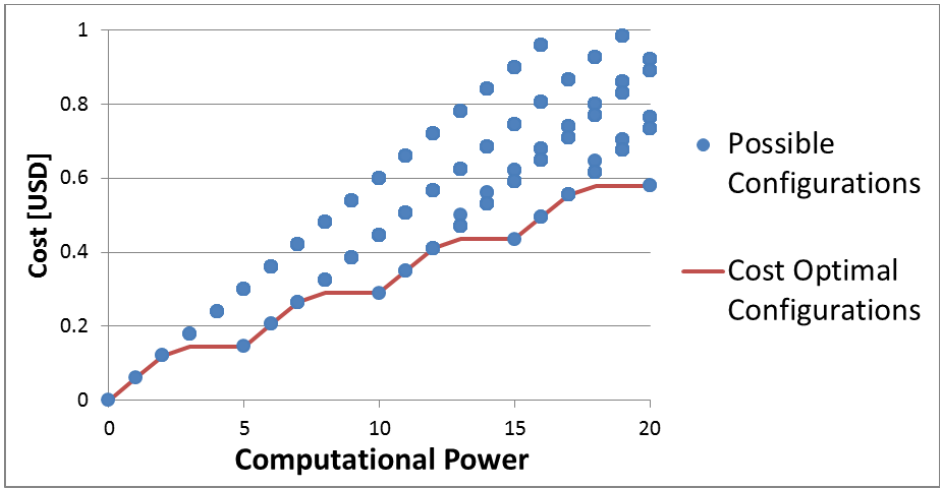


Figure 2.7. Cost-optimal configurations among other possible configurations.

Step 4: Workers Assignment

Given $tasks, A_{priv}, \sigma_{priv}, A_{pub}$ and σ_{pub} from the previous steps, Algorithm 2 outputs the following worker assignment:

- W_{priv}: contains a set of tuples $(worker_i, tasks_i)$ for instance type i, where $worker_i$ is the number of workers and $tasks_i$ is the number of tasks to be assigned to instances of type i.
- W_{pub}: contains the same information as above for the public cloud.
- $tasks_{per\_worker}$: the number of tasks per worker.

input : $tasks, A_{priv}, \sigma_{priv}, A_{pub}, \sigma_{pub}$
output: $W_{priv}, W_{pub}, tasks_{per\_worker}$
$W_{priv} = \emptyset$; $W_{pub} = \emptyset$;
foreach instance type i in A_{priv} **do**
  $tasks_i = tasks \times \frac{\eta_i}{\sigma_{priv}+\sigma_{pub}} \times \frac{1}{num_i}$;
end
foreach instance type j in A_{pub} **do**
  $tasks_j = tasks \times \frac{\eta_j}{\sigma_{priv}+\sigma_{pub}} \times \frac{1}{num_j}$;
end
$tasks_{per\_worker} = \min_{1 \le i \le N, 1 \le j \le M} \{\frac{tasks_i}{cpu_i \times 2}, \frac{tasks_j}{cpu_j \times 2}\}$;
foreach instance type i in A_{priv} **do**
  $worker_i = \lceil \frac{tasks_i}{tasks_{per\_worker}} \rceil$;
  $W_{priv} = W_{priv} \cup \{(worker_i, tasks_i)\}$;
end
foreach instance type j in A_{pub} **do**
  $worker_j = \lceil \frac{tasks_j}{tasks_{per\_worker}} \rceil$;
  $W_{pub} = W_{pub} \cup \{(worker_j, tasks_j)\}$;
end
return $W_{priv}, W_{pub}, tasks_{per\_worker}$;

Algorithm 2. Workers assignment.

Algorithm 2 assigns tasks in proportion to an instance type's computational power while keeping the number of tasks for each worker constant. By assigning tasks to workers this way, we are able to balance the load between heterogeneous VM instances just by migrating workers. Also, $tasks_{per\_worker}$ is determined in connection to the number of virtual CPUs. This is because it is known that the granularity of workers has an impact on the workload performance on a given number of processors [62]. To be in a region of high

performance, $tasks_{per\_worker}$ is chosen so that the number of workers is at least twice as many as the number of virtual CPUs.

2.2.4 Experiments

We run the resource algorithm shown in Section 3.4.3 using both WECU and Amazon's ECU, and compare the performance predictability of both approaches.

Experimental Settings

- *Workload: Trapezoidal Numerical Integration*: The Trap application written in SALSA computes an approximated value of $\int_a^b f(x)dx$. The farmer actor breaks the interval $[a, b]$ into n trapezoids and assigns a certain number of trapezoids to each actor program. After all the worker actors compute the value of f for given trapezoids, the farmer sums up all the partial trapezoid values returned from workers. Since the computation of each trapezoid does not depend on others, each worker can work without communicating with other workers.

- *Hybrid Cloud Environment*: We have two host machines, node A and B, as the private cloud hosts. Node A has AMD Opteron 848 processor (4 cores) running at 2.2 GHz and 15 Gbytes of memory. Node B has Intel Xeon CPU E31220 processor (4 cores) running at 3.1 GHz and 6 Gbytes of memory. We dedicate node A to the name service required by SALSA and the farmer actor while node B is used to run worker actors. As the public cloud, we use Amazon EC2 with VM instance types presented in Table 2.2. Also, in the following experiments, information shown in Table 2.3 is used as computational power in ECU and WECU units.

- *ECU-based Approach*: The resource configuration algorithm presented in Section 3.4.3 is also applicable for ECU, if ECU is used as a unit of computational power η. Therefore, we use ECU values depicted in Table 2.3 and compare the same algorithm with

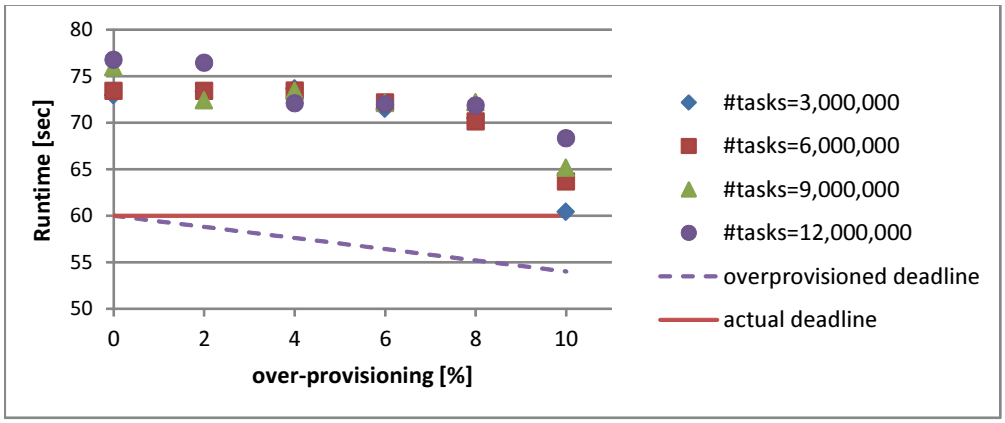
WECU and ECU units respectively. Note that for nodes A and B, 4 and 14 are used as computational power η for the ECU-based approach.

- *Parameters*: The Trap application is tested in the hybrid cloud with the following conditions:
  - $t_{deadline}$: 60 seconds,
  - $Tasks$ (the number of trapezoids): $\{3 \times 10^6, 6 \times 10^6, 9 \times 10^6, 12 \times 10^6\}$, and
  - Over-provisioning rate [%]: $\{0, 2, 4, 6, 8, 10\}$.

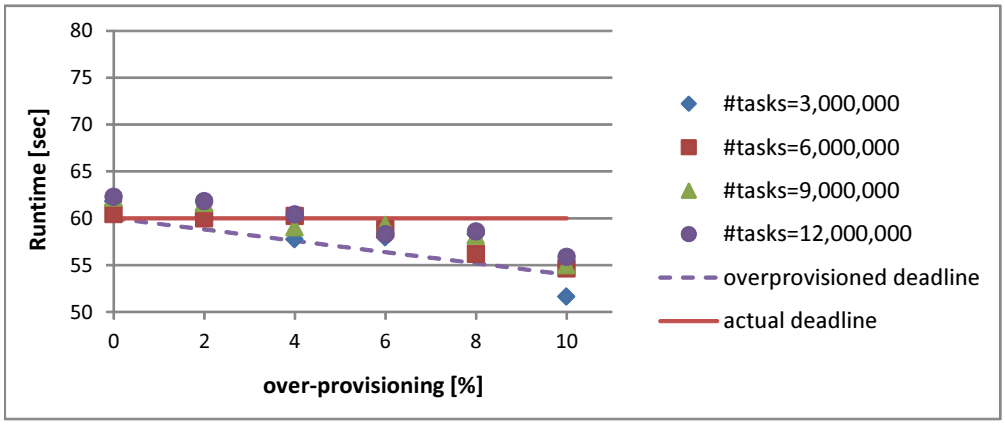
  Over-provisioning is typically used to account for the lack of accuracy in predicting workload performance. For example, if the over-provisioning rate is 10%, then the over-provisioned deadline will be 54 seconds and the algorithm takes this value as the new deadline $t_{deadline}$.

Results The runtime results for (a) ECU-based and (b) WECU-based approaches are shown in Figure 2.8. From Figure 2.8, we can clearly tell that the WECU-based resource configuration better predicts the actual performance compared to the ECU-based approach. The average differences between initial predicted runtime and the actual runtime are: 4.59% for WECU and 29.25% for ECU. Even though initial worker distribution overhead is not accounted, WECU-based approach succeeded to meet the deadline for over-provisioning rates larger than 6%.

We noticed our dynamic programming based algorithm is in favor of choosing {`m1.small, c1.medium, c1.xlarge`} over {`m1.medium, m1.large, m1.xlarge, m3.xlarge, m3.2xlarge`} instance types. This happens because, as shown in Table 2.3, most of the former group of instance types are better in Price/WECU ratio than the latter. If a workload is memory or I/O-intensive and `m1` and `m3` instance types perform better in such workloads, we expect that this will be reflected in the WECU relative performance, and price/performance ratio will make them favorable.



(a) ECU-provisioned Runtime



(b) WECU-provisioned Runtime

Figure 2.8. Runtime results for (a) ECU-based and (b) WECU-based resource configurations.

2.2.5 Summary

We introduced the concept of Workload-tailored Elastic Compute Unit (WECU) as a measure of computing resources analogous to Amazon EC2's ECUs. We present a dynamic programming-based scheduling algorithm to select resources which satisfy the desired throughput in a cost-optimal way. Using a loosely-coupled benchmark running on a hybrid cloud environment, we confirmed WECUs have 24% better runtime prediction ability than ECUs on average.

Through the experiments, it is suggested the ECU-based approach needs a significantly higher over-provisioning rate to satisfy service demands compared to the one needed by the

WECU-based approach. This means that the ECU-based approach has a wider search space of over-provisioning rates than the WECU-based approach does, therefore it is not easy for the ECU-based to find the right over-provisioning rate.

Future work includes evaluating our middleware's adaptability with real applications that have dynamically changing workloads and also implementing a budget-constrained resource management algorithm. Also, our current algorithm is cost-optimal for a given computational power, but it is not Pareto-optimal. We plan to modify our resource management algorithm to produce Pareto-optimal resource configurations.

CHAPTER 3
ELASTIC MICRO-BATCH DATA PROCESSING

3.1 Introduction

The number of flight passengers is expected to reach 7.3 billion by 2034 globally, which requires a 4.1% average growth in flight capacity in every year from 2014 on [63]. Air traffic optimization is crucial to enhance flight capacity and also alleviate human controllers' workload. Air traffic management problems are commonly formulated as integer linear programming (ILP), which are known to be NP-hard [64]. However, we can still obtain approximate solutions and use them to help control air traffic. To keep up with ever changing traffic and continuously produce air traffic management solutions, we can solve ILP problems in a *micro-batch* manner. That is, we create a single ILP problem from data observed in a certain time window as a batch data processing job and repeatedly solve created problems as we obtain new data.

However, since the number of flights fluctuates a lot in practice, computational demands for air traffic optimization also change dynamically. For example, Figure 3.1 shows how the number of commercial flights in the U.S. changed over 24 hours from 4am EST on January 18th, 2014. Once air traffic hits the peak at around 1pm, it gradually drops and eventually reaches 200 at around 3am. To keep up with the fluctuating computing demands in a cost-efficient way, we can dynamically allocate and deallocate VMs from Infrastructure-as-a-Service (IaaS) cloud computing providers. The challenge is dynamically choosing the right number of VMs that satisfies computational demands at the lowest possible cost.

Portion of this chapter previously appeared in: Shigeru Imai, Stacy Patterson, and Carlos A. Varela, "Elastic virtual machine scheduling for continuous air traffic optimization," in *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid 16)*, 2016, pp. 183–186.

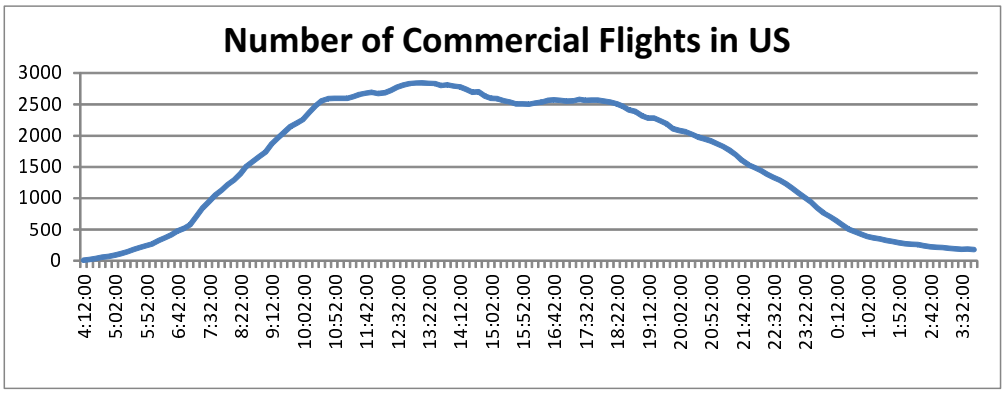


Figure 3.1. Number of U.S. commercial flights on January 18th, 2014 (created from data available on [65]).

To elastically allocate or deallocate VMs, we can either adaptively adjust the number of VMs at run-time without any prior knowledge of the application or proactively predict the number of VMs using a resource prediction model. The former includes a threshold-based approach, as used in Amazon's Auto Scaling [42], and reinforcement learning [43], [44]. We take the latter approach to improve the resource utilization, cost, and latency violations. Moreover, we use an auto-regressive time series prediction model to decide when to schedule VMs in a speculative manner. In this chapter, we present the following:

- An elastic middleware framework to solve ILP problems created from continuously incoming air traffic streams (Section 3.2) over IaaS clouds. The framework obtains an approximate solution to ILP problems using a two-level optimization technique based on Lagrangean decomposition [66].

- VM scheduling algorithms that are specifically designed to solve ILP problems generated from continuous air traffic streams (Section 3.4). We use a time series prediction model to decide when to allocate VMs and we also use a resource prediction model to estimate how many VMs to allocate. The resource prediction model estimates required VM resources given the number of flight routes and target processing latency by using linear regression.

- Experimental results that show our speculative VM scheduling algorithm can achieve a similar performance to a static schedule while using 49% less VM hours for a smoothly changing air traffic. Our algorithm is able to adapt dynamically to potentially unforeseen fluctuating demand with a reasonable prediction accuracy.

3.2 Air Traffic Management Problem

3.2.1 Problem Formulation

The Link Transmission Model (LTM) [67] is an air traffic flow management model that optimizes nationwide air traffic by formulating it as an ILP problem. Cao and Sun decompose the original LTM problem into multiple sub-problems using Lagrangean decomposition and use MapReduce [68] to approximate the solution to large scale LTM problems in parallel [69].

Our work is inspired by their approach. We formulate a simplified version of the LTM problem that captures the computationally intensive nature of the original LTM problem. Figure 3.2 shows an example of the simplified air traffic management problem. A *route* connects a departure airport and an arrival airport, and it consists of multiple *links* that are distributed over multiple *sectors*. As illustrated in the example, the same sector at the center of the grid is shared by multiple routes, therefore congestion must be controlled.

We can formalize the simplified air traffic management problem as an ILP problem as follows:

$$\text{minimize} \quad \sum_{i=1}^{K} \mathbf{c}_i^{\intercal} \mathbf{x}_i \tag{3.1}$$

$$\text{subject to} \quad \sum_{i=1}^{K} A_i \mathbf{x}_i \leq \mathbf{b} \tag{3.2}$$

$$\mathbf{0} \leq \mathbf{x}_i \;\; \forall i \in [1, K], \tag{3.3}$$

$$\sum_{j=1}^{N_i} x_i^j \leq d_i \;\; \forall i \in [1, K], \tag{3.4}$$

$$\text{where} \quad x_i^j \in \mathbb{Z} \geq 0, \;\; A_i \in \{0,1\}^{S,N_i}, \;\; \mathbf{b} \in \mathbb{Z}^S \geq 0,$$

$$\mathbf{c}_i \in \mathbb{R}^{N_i} \geq 0, \;\; d_i \in \mathbb{Z} \geq 0.$$

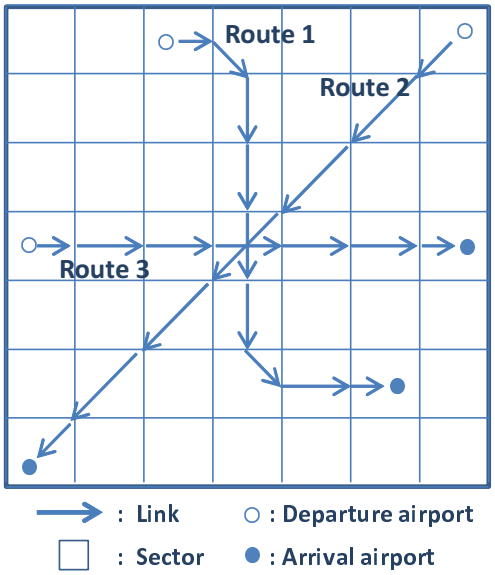


Figure 3.2. Example of the simplified air traffic management problem.

The objective of this optimization problem is to assign an ideal number of flights to each sector so that we can maximize the air traffic capacity while satisfying capacity of each sector. Given constants are: the number of routes K, the number of links of the i-th route $N_i (i = 1, ..., K)$, and the number of sectors S. The number of flights for a route i is expressed as a vector $\mathbf{x}_i = [x_i^1, x_i^2, ..., x_i^{N_i}]^\intercal$, where $x_i^j \in \mathbb{Z} \geq 0$ is the number of flights at link j of the route i. $\mathbf{x}_i (i = 1, ..., K)$ are the variables to be optimized subject to the following capacity constraints:

- **Sector capacity:** Total number of flights in a sector s must be less than or equal to $b_s \in \mathbb{Z} \geq 0$ $(s = 1, ..., S)$ (Inequality (3.2)).
- **Route capacity:** Total number of flights on a route i must be less than or equal to $d_i \in \mathbb{Z} \geq 0$ $(i = 1, ..., K)$ (Inequality (3.4)).

The objective function $\sum_{i=1}^{K} \mathbf{c}_i^\intercal \mathbf{x}_i$ is defined with vectors $\mathbf{c}_i = [c_i^1, c_i^2, ..., c_i^{N_i}]^\intercal (i = 1, ..., K)$, where $c_i^j \in \mathbb{R} \geq 0$ determines the degree of preference of assigning flights on link j of route i. We can set higher values for less congested sectors and lower values for highly congested sectors. The sector capacity constraint is defined by Inequality (3.2) using $S \times N_i$ matrices

$A_i (i = 1, ..., K)$ and a vector $\mathbf{b} = [b_1, b_2, ..., b_S]$, where $b_s \in \mathbb{Z} \geq 0$. For a route i, the mapping of links to sectors naturally dictates the construction of A_i; each element a_{sj} takes the value of 1 if link j is on sector s, otherwise 0. Each element of $\mathbf{b}$ determines the sector capacity of a corresponding sector.

A solution to this problem captures the two important properties of the original LTM problem that affect the computational workload. First, adding a new route increases the number of variables in proportion to the number of links on the route. Second, each A_i matrix is extremely sparse, which significantly affects the difficulty of satisfying constraints because there are very few number of variables in each constraint.

3.2.2 Lagrangean Decomposition

ILP is NP-hard. Thus, it is common to use algorithms that find an approximate solution in a reasonable amount of time. Lagrangean decomposition is a popular technique to obtain an approximate solution to ILP problems, and it was used for LTM in [69]. Lagrangean decomposition offers a way to split a larger linear integer problem into multiple smaller sub-problems by relaxing *complicating constraints*. For our air traffic management problem described in Section 3.2.1, the complicating constraint is the sector capacity constraint (Inequality (3.2)), which prohibits us from separating the original problem into K sub-problems with Inequality (3.3) and (3.4). By constructing a Lagrangean relaxation of the original problem, we can bring the complicating constraints to the objective function as a penalty term as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{K} \mathbf{c}_i^{\intercal} \mathbf{x}_i - \boldsymbol{\lambda}^{\intercal} \Big(\sum_{\mathrm{i}=1}^{K} A_i \mathbf{x}_i - \mathbf{b} \Big) \\ \text{subject to} \quad & \mathbf{0} \leq \mathbf{x}_i, \ \sum_{j=1}^{N_i} x_i^j \leq d_i \ \ \forall i, \end{aligned} \tag{3.5}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^S \geq \mathbf{0}$ is a vector of Lagrange multipliers. Now, we can decompose the problem (3.5) into a smaller sub-problems, one for each route i:

$$\begin{aligned} \text{maximize} \quad & \Big(\mathbf{c}_i^\intercal - \boldsymbol{\lambda}^\intercal A_i\Big)\mathbf{x}_i \\ \text{subject to} \quad & \mathbf{0} \leq \mathbf{x}_i, \ \sum_{j=1}^{N_i} x_i^j \leq d_i. \end{aligned} \tag{3.6}$$

Next, we define the master dual problem of the Lagrangean (3.5), which is responsible for updating $\boldsymbol{\lambda}$:

$$\begin{aligned} \text{minimize} \quad & g(\boldsymbol{\lambda}) = \sum_{i=1}^{K} (\mathbf{c}_i^\intercal - \boldsymbol{\lambda}^\intercal A_i)\mathbf{x}_i^* + \boldsymbol{\lambda}^\intercal \mathbf{b} \\ \text{subject to} \quad & \mathbf{0} \leq \boldsymbol{\lambda}, \end{aligned} \tag{3.7}$$

where $\mathbf{x}_i^*$ is the optimal solution to the sub-problem (3.6) for a route i. To solve $\boldsymbol{\lambda}$ for the dual problem (3.7), we use the gradient method:

$$\frac{\partial g}{\partial \boldsymbol{\lambda}} = \mathbf{b} - \sum_{i=1}^{K} A_i \mathbf{x}_i^* \tag{3.8}$$

$$\boldsymbol{\lambda}(t+1) = \boldsymbol{\lambda}(t) - \alpha \frac{\partial g}{\partial \boldsymbol{\lambda}}, \tag{3.9}$$

where α is a small positive step-size.

As shown in Algorithm 3, we iteratively solve the K sub ILP problems to find optimal $\mathbf{x}_i (i = 1, ..., K)$ for a specific $\boldsymbol{\lambda}$ (Line 8) and update $\boldsymbol{\lambda}$ for the master dual problem (Line 17). We keep track of the minimum value of objective ($minObj$), and if it is not improved by more than $\delta\%$ for I iterations, we go out from the while loop and return the final results.

input : $A_i, \mathbf{c}_i, d_i (i = 1, ..., K), \mathbf{b}, \boldsymbol{\lambda}_{\text{init}}, \delta, I$
output: $\mathbf{x}_i (i = 1, ..., K), minObj$

$t \leftarrow 0$;
$\boldsymbol{\lambda}(t) \leftarrow \boldsymbol{\lambda}_{\text{init}}$;
$minObj \leftarrow Double.MAX\_VALUE$;
while *Iterations minObj not improved more than* $\delta\% < I$ **do**
    `// Solve K sub-problems`
    for $i = 1$ **to** K **do**
        $\mathbf{x}_i \leftarrow solveILP(A_i, \mathbf{c}_i, d_i, \boldsymbol{\lambda}(t))$;
    end
    `// Update master objective`
    $obj \leftarrow compObj(A_1, ..., A_K, \mathbf{b}, \mathbf{c}_1, ..., \mathbf{c}_K, \mathbf{x}_1, ..., \mathbf{x}_K, \boldsymbol{\lambda})$;
    if $obj < minObj$ **then**
        $minObj \leftarrow obj$;
    end
    `// Update` λ `for the next iteration`
    $\alpha = \frac{1}{t}$;
    $\boldsymbol{\lambda}(t+1) \leftarrow \boldsymbol{\lambda}(t) - \alpha \cdot gradient(A_i, \mathbf{b}, \mathbf{x}_i)$;
    $t \leftarrow t + 1$;
end
return $\mathbf{x}_i (i = 1, ..., K), minObj$;

Algorithm 3. Two-level ILP optimization.

3.3 Elastic Air Traffic Management Middleware

3.3.1 Background

System Interaction We assume that the user of the middleware is a human air traffic controller who uses output of our middleware for air traffic control activity. We also assume that some flight information providers (*e.g.*, FlightAware [70]) or airplanes directly send the latest flight status information to the middleware (see Figure 3.3). Since air traffic management is time critical, the middleware tries to schedule VMs so that the optimization result can be used by the user in a timely manner. Hence, the user can configure *latency* to request how quickly the application should return the result.

Cloud Deployment The middleware is designed to work on an IaaS cloud. The IaaS cloud can be private, public, or hybrid; however, the scheduling algorithm presented in Section 3.4

is optimized for public IaaS clouds due to its *billing cycle* aware scheduling. The billing cycle is the unit of monetary charge (*e.g.*, 1 hour for Amazon EC2 [71] as of November 2016). The scheduler only terminates VMs just before their billing cycle so that the application can use the VMs' computing power until the last minute.

3.3.2 Application Implementation

We use Spark 1.5.1 [72], a general cluster computing engine, to implement Algorithm 3. Spark's high-level abstractions for distributed programming and in-memory data processing features are suitable for the iterative ILP problem solving process. Spark applications run on a cluster consisting of a master node and multiple worker nodes. In Algorithm 3, *executors* running on the worker nodes execute Line 7 to solve K sub-problems in parallel, and the rest of the code is executed on the master node. While Spark allows us to cache parameters $A_i, \mathbf{c}_i, d_i$ for sub-problems on each worker node, the master needs to broadcast the updated value of $\boldsymbol{\lambda}$ to the workers in each iteration.

When executors solve the sub-problems, we use lp_solve [73] since it is open-source and thread-safe. Since Spark runs multiple threads in one executor process in parallel, thread safety is a required property for the ILP problem solver.

3.3.3 Middleware Architecture

Figure 3.3 illustrates the architecture of the proposed middleware framework. We describe how the middleware works, step by step, as follows:

- **Step 1**: The *Controller* periodically pulls (*e.g.*, every 5 minutes) flight status information in the queue such as airplane positions and flights' departure and arrivals.
- **Step 2**: The Controller creates an ILP problem instance from the obtained flight status information and then pushes it to the *VM Scheduler* with requested processing latency (*e.g.*, 4 minutes).

- **Step 3**: The VM Scheduler uses a time series prediction model and a resource prediction model to estimate the required number of VMs to finish the optimization within the requested processing latency.

- **Step 4**: The VM Scheduler allocates or deallocates VMs accordingly by calling cloud APIs such as Apache Libcloud [17].

- **Step 5**: The Controller requests the *Application Launcher* to run the ILP application.

Even though flight status information flows into the middleware continuously, the middleware processes the information collected within a sliding time window. We can see this as a micro-batch processing model just as used in Spark Streaming [72].

3.4 Virtual Machine Scheduling

First, we confirm how the ILP optimization application works on actual VMs through a preliminary experiment in Section 3.4.1. Next, we describe a resource prediction model created using linear regression in Section 3.4.2. Finally, in Section 3.4.3, based on observations from the preliminary experiment and the resource prediction model, we present two VM scheduling algorithms: *baseScheduler* and *specScheduler*.

3.4.1 Performance Characterization of ILP Optimization

To understand how the Spark application works, we conducted a preliminary experiment with the following settings:

- Number of links per route: generated from a Gaussian distribution (average = 19, variance = 9).

- Number of sectors: 1024 made of a 32 by 32 grid just as shown in Figure 3.2.

- Convergence criterion: the value of master dual objective in Eq. (3.7) does not improve more than 1% for 1000 iterations (*i.e.*, $\delta = 1, I = 1000$ in Algorithm 3).

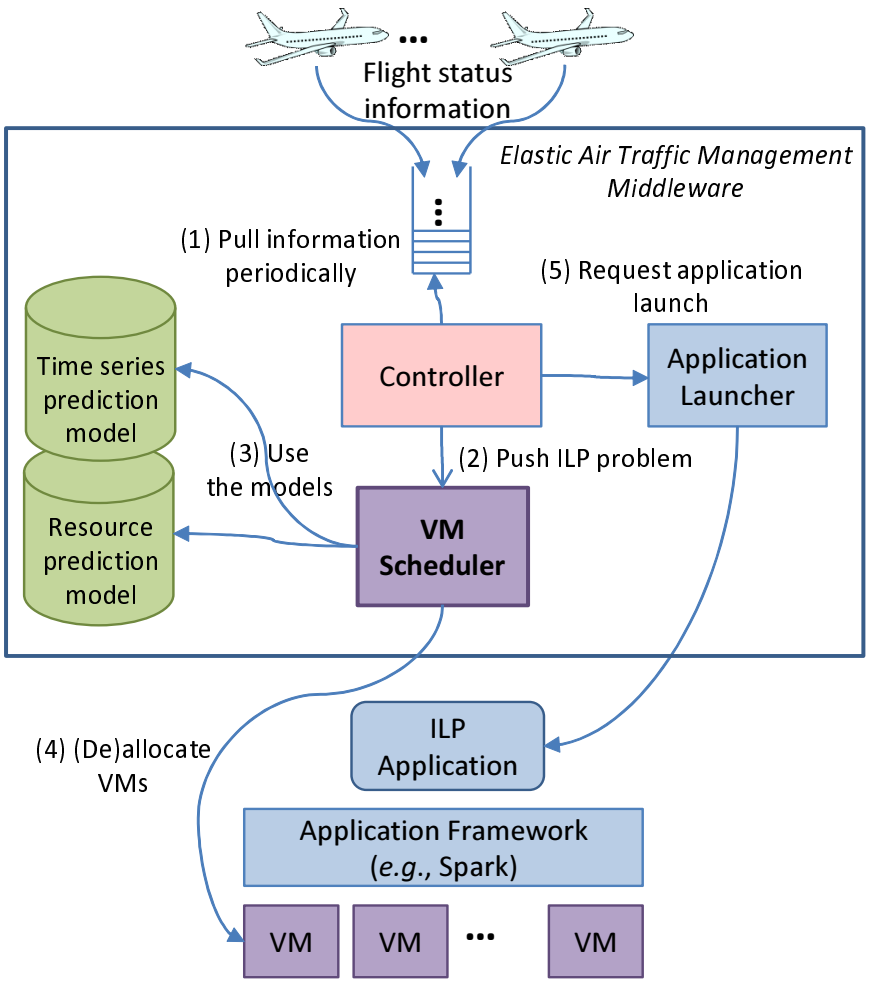


Figure 3.3. Architecture of the elastic air traffic management middleware framework.

- Spark setting: 1 executor per core.

We tested 50 application runs with randomly selected VM instances and number of routes from the following options:

- VM instances for Spark worker nodes: {c4.large, c4.xlarge, c4.2xlarge} instance types available from Amazon EC2 (see Table 3.1). Up to five instances can be created for each instance type.

- Number of routes: {128, 256, 512, 1024}

Table 3.1. Amazon EC2 VM instance types used in experiments (information as of November 2015).

Name	vCPU cores	Cost [USD/hr]	Instance limits
c4.large	2	0.11	5
c4.xlarge	4	0.22	5
c4.2xlarge	8	0.441	5

Figure 3.4 presents the relationship between the total number of cores used by the VMs and the application execution time. First, we observe that the performance variance is relatively small (at most 7%) regardless of VM configurations as long as we use the same number of cores for the same number of routes. This is due to the fact that we assign one thread per core, and therefore, we end up using the same number of threads even for different VM configurations as long as they have the same number of cores. Second, as we can clearly see from the graph, the application execution time does not improve significantly from around 18 to 20 cores for all numbers of routes. This behavior is consistent with a performance analysis of a K-means Spark application reported in [74], in which the performance converges at around 15 threads. They concluded that multi-threaded computation overhead (*i.e.*, work time inflation [75]) and load imbalance caused the scalability bottleneck. Since our application and K-means have a similar synchronization pattern (*i.e.*, both are iterative and synchronize all workers between every iteration), this analysis applies to our case as well. These observations lead to the following decisions for the resource allocation method design:

- The unit of resource (de)allocation is the number of cores. Since the performance and cost per core is equal among {c4.large, c4.xlarge, c4.2xlarge}, we do not distinguish one VM instance type from another.

- We set the upper limit for the number of cores that we allocate to match the application's inherent scalability limitations.

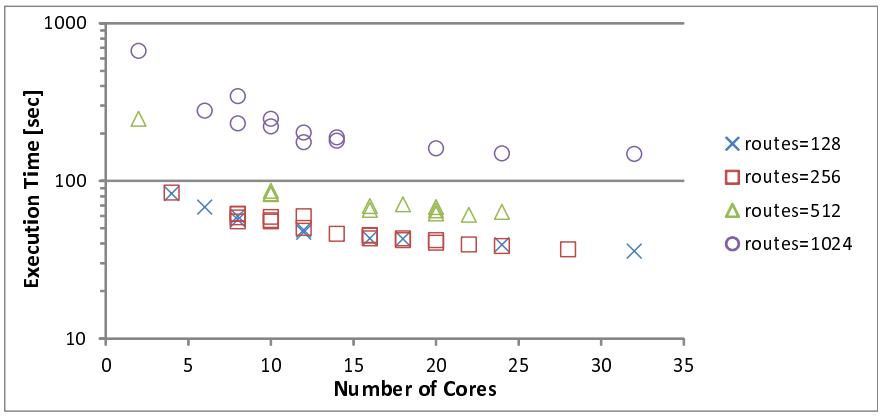


Figure 3.4. Characteristics of the ILP problem execution time.

3.4.2 Resource Prediction Model

The VM Scheduler introduced in Section 3.3.3 needs a model to determine how many resources it should allocate to achieve the target processing latency. We use linear regression with a non-linear transform to model the relationship between the two input parameters: processing latency l and number of routes r, and the output: number of cores c. We sampled 50 application runs using the same experimental settings as the preliminary experiment in Section 3.4.1, but this time with uniformly random numbers of routes between 100 and 1200. Subsequently, we tested five non-linear transforms to determine which one best fits the sampled training data as shown in Table 3.2. As the transformation becomes more complex, correlation between the model and the training data improves. Based on this result, we have identified $\Phi_{\text{-2:2}}$ to be the most correlated with the data. We thus use this model in our algorithms. The number of cores can be obtained as follows:

$$f(l, r) = \mathbf{w}^{\mathsf{T}} \cdot \Phi_{\text{-2:2}}(l, r), \tag{3.10}$$

where $\mathbf{w} = [w_1, w_2, ..., w_{11}]$ is a weight vector acquired from linear regression of the number of cores given the latency and the number of routes.

Table 3.2. Non-linear transforms used for linear regression.

Name	Transformation vector	Correlation
$\Phi_{\text{-}1}$	$[1/r, 1/l, 1]^\intercal$	0.615
Φ_2	$[1, r, l, r^2, rl, l^2]^\intercal$	0.765
$\Phi_{\text{-}2}$	$[1/r^2, 1/rl, 1/l^2, 1/r, 1/l, 1]^\intercal$	0.870
$\Phi_{\text{-}2:1}$	$[1/r^2, 1/rl, 1/l^2, 1/r, 1/l, 1, r, l]^\intercal$	0.873
$\Phi_{\text{-}2:2}$	$[1/r^2, 1/rl, 1/l^2, 1/r, 1/l, 1, r, l, r^2, rl, l^2]^\intercal$	0.890

3.4.3 Elastic Scheduling Algorithms

The VM Scheduler periodically calls one of the scheduling algorithms to keep the processing latency consistent. We describe two scheduling algorithms for the VM Scheduler that use the model presented in Section 3.4.2. Key notations used in the algorithms are summarized in Table 3.3.

Table 3.3. Key notations used in scheduling algorithms.

Name	Description
l_{req}	Requested processing latency.
r_t	Number of routes to process at time t.
t_{up}	VM startup time.
$f(l, r)$	Resource prediction model that predicts the number of cores to satisfy latency l when processing r routes.
V_{act}	Set of VMs that are actively used by the application.
V_{idle}	Set of VMs to be removed at the end of next billing cycle.
V_{spec}	Set of speculative VMs to be allocated.
V	Set of currently allocated VMs. $V = V_{\text{act}} \cup V_{\text{idle}}$.
$t_{\text{bc}}(v)$	Next billing cycle time of a VM $v \in V$.
$c(v)$	Number of cores of a VM $v \in V$.

3.4.3.1 Baseline Scheduling

The baseline scheduling algorithm (*baseScheduler*) is shown in Algorithm 4. This algorithm responds to increasing computational demand by creating new VMs, while at the same time, tries to take advantage of existing VMs even when they are not needed to achieve required processing latency. First, we compare the available number of cores c_{all} with required cores c_{req} estimated from the resource prediction model f (Line 2-4). If there

```
1  input  : l_req, r_t, t_up, V
   output: V_act, V_idle
2  c_req ← ⌈f(l_req, r_t)⌉;
3  c_all ← Σ_{v∈V} c(v);
4  if c_req ≤ c_all then
5  |  // There are enough cores
6  |  Sort v ∈ V in descending order of t_bc(v);
7  |  V_act ← selectVMs(c_req, V);
8  |  V_idle ← V − V_act;
9  |  V_extra = {v | v ∈ V_idle,  t + l_req < t_bc(v)};
10 |  if V_extra ≠ ∅ then
11 |  |  V_act ← V_act ∪ V_extra;
12 |  |  V_idle ← V_idle − V_extra;
13 |  end
14 else
15 |  // Not enough cores, allocate VMs
16 |  c_alloc ← ⌈f(l_req − t_up, r_t)⌉ − c_all;
17 |  V_act ← V ∪ allocVMs(c_alloc);
18 |  V_idle ← ∅;
19 end
20 return V_act, V_idle;
```

Algorithm 4. Baseline VM scheduling algorithm (*baseScheduler*).

are enough cores, we sort V in descending order of next billing cycles (the VM with the latest billing cycle comes first) (Line 6). Then, we let *selectVMs* select at least c_{req} worth of VMs from the sorted V and put them to V_{act} and the rest of VMs to V_{idle} (Lines 7 and 8). Further, if there still remains VMs in V_{idle} such that their billing cycles come after the time that we expect the application to finish, we utilize those VMs V_{extra} too (Line 10-12). At this point, VMs in V_{idle} are expected to end their billing cycles before the application finishes, and therefore they will be deallocated. If there are not enough cores to satisfy the requested latency, we have to allocate new VMs to satisfy the requested latency. Since newly allocated VMs take t_{up} time before they become fully operational, we can only start running the application after t_{up} time has passed. Therefore, we set a tighter deadline $l_{\text{req}} - t_{\text{up}}$ and estimate the number of cores to allocate c_{alloc} again (Line 16). Then, we call the *allocVMs* sub-routine to allocate at least c_{alloc} worth of VMs and update V_{act} and V_{idle} accordingly.

3.4.3.2 Speculative Scheduling

The speculative scheduling algorithm (*specScheduler*) is shown in Algorithm 5. This algorithm takes advantage of future computational demand prediction and tries to allocate VMs before they are actually needed. We predict the number of routes for time $t+1$ using a slope computed from r_t and r_{t-1} as follows.

$$r_{t+1} = \frac{r_t - r_{t-1}}{t-(t-1)} + r_t = 2r_t - r_{t-1}. \tag{3.11}$$

This prediction model is equivalent to an auto-regressive model (*i.e.*, AR(2)) just as used in [46], [47].

```
1  input  : l_req, r_t, r_{t-1}, t_up, V
   output: V_act, V_idle, V_spec
2  // Obtain a baseline configuration first
3  (V_act, V_idle) ← baseScheduler(l_req, r_t, t_up, V);
4  c_all ← Σ_{v∈V} c(v);
5  // Speculative VM allocation
6  r̂_{t+1} ← predictNumRoutes(r_t, r_{t-1});
7  ĉ_req ← ⌈f(l_req, r̂_{t+1})⌉;
8  V_spec ← ∅;
9  if c_all < ĉ_req then
10 |  // Schedule to finish launching V_spec VMs before the next time step
11 |  V_spec ← scheduleAllocVMs(ĉ_req − c_all);
12 end
13 return V_act, V_idle, V_spec;
```

Algorithm 5. Speculative VM scheduling algorithm (*specScheduler*).

The *specScheduler* first obtains a baseline configuration using the *baseScheduler* and computes all of available number of cores in c_{all} (Line 3-4). Next, *specScheduler* predicts the number of routes for the next step in $\hat{r}_{t+1}$ by using the prediction model of Eq. (3.11) (Line 6). Using $\hat{r}_{t+1}$ and the resource prediction model f, we estimate a speculative required cores $\hat{c}_{\text{req}}$. Finally, if we need more cores at next time step than what we currently have ($c_{\text{all}} < \hat{c}_{\text{req}}$), then we schedule to launch VMs that are worth $\hat{c}_{\text{req}} - c_{\text{all}}$ cores just before the next time step (Lines 9 and 11).

3.4.3.3 VM Allocation Policy

Given the number of cores, we allocate VMs from a limited pool of VMs when executing *allocVMs* (Line 17, Algorithm 4) and *scheduleAllocVMs* (Line 11, Algorithm 5). When selecting VMs, we try to allocate a VM type with smaller number of cores. If a VM type reaches its instance creation limit, then we try to allocate VM types with bigger number of cores until at least the requested number of cores is allocated. In case of Amazon EC2, we try to allocate c4.large instances first, and then c4.xlarge followed by c4.2xlarge. The reason that we give priority to smaller instances is because they have finer core granularity. That is, we would have a higher chance of allocating exact number of cores so that we can avoid over provisioning of VMs.

3.5 Evaluation

We first introduce simulation based experimental settings in Section 3.5.1. Next, we evaluate the proposed algorithms' elastic behavior in Section 3.5.2. Then, we compare the proposed algorithms' performance with static VM scheduling and threshold-based auto scaling in Sections 3.5.3 and 3.5.4 respectively.

3.5.1 Experimental Settings

The experiments are simulation-based. We develop a simulator that executes the proposed VM scheduling algorithms. Using the generated schedules by the simulator, we manually allocate and deallocate VMs on Amazon EC2 cloud and run the Spark application to evaluate used VM hours, cost, and latency violations based on actual execution time. We use two 3-hour route datasets for testing: the first one is called *Nationwide* that we create from the 24-hour real nationwide flights shown in Figure 3.1, and the second one is called *Dallas* that we create based on a simulated flights over Dallas/Fort Worth area [76]. Both have almost the same peak number of routes, about 1200, but the patterns of fluctuation are different. While Nationwide has a smooth curve, Dallas has steep spikes, as shown in

Figure 3.7.

We use the following test parameters for evaluation:

- Scheduling interval: 5 minutes (36 scheduling problems over 3 hours).
- Requested processing latency (l_{req}) : 4 minutes.
- VM startup time (t_{up}): 90 seconds.
- VM instances for Spark's worker nodes: {c4.large, c4.xlarge, c4.2xlarge} (see Table 3.1 for details). Up to five instances can be created for each instance type.
- Billing cycle: 1 hour (Amazon EC2's default).

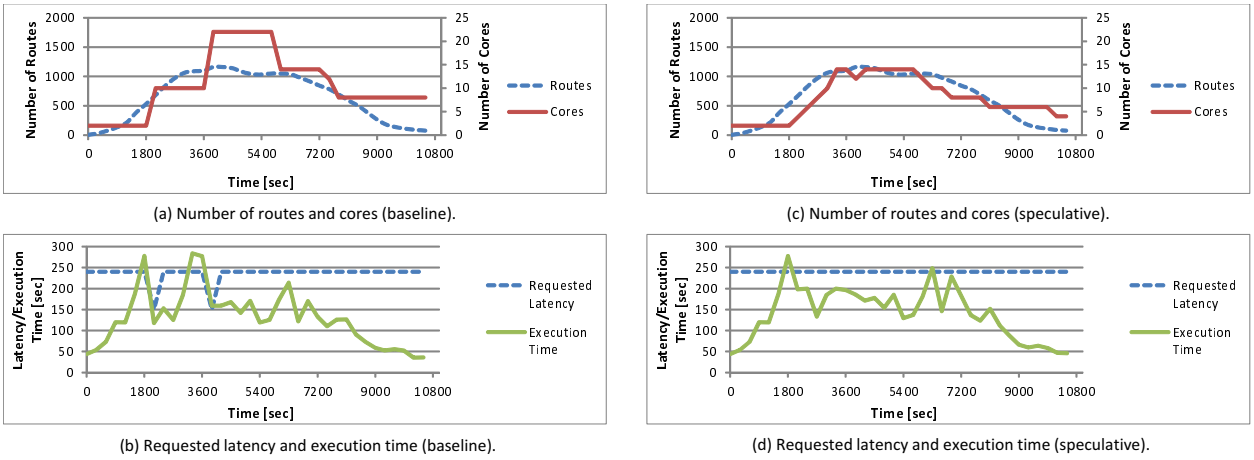
3.5.2 Elastic Behavior Confirmation

3.5.2.1 Nationwide Dataset

Results for *baseScheduler* and *specScheduler* for the Nationwide dataset are shown in Figure 3.5(a)-(d). From Figure 3.5(a), we see that the baseline scheduler allocates VMs, initially for 8 cores at 1900 seconds and then for 12 cores at 3900 seconds. Looking at Figure 3.5(b), we notice that there are two "dips" in requested latency at the same time as the scheduler allocates new VMs. This lower latency corresponds to the value of $l_{\text{req}} - t_{\text{up}}$ (= 150 seconds) at Line 15 of Algorithm 4. To account for the VM startup time, we intentionally set a tighter latency. Therefore, the scheduling algorithm has to allocate relatively large number of cores. Since these cores are more than enough to satisfy the regular required latency l_{req} (= 240 seconds), there are periods (2100 to 2700 seconds, 3900 seconds to the end) when execution time stays lower than required, that is, resources are over-provisioned during these periods. This is a limitation of the reactive approach. There are four latency violations occurring at 1800, 3300, 3600, and 3900 seconds respectively. At these times, the prediction model underestimated the number of cores needed to satisfy the requested latency. The average prediction error of execution time for the four violations is 12%. This result suggests that the accuracy of resource prediction is limited and we may need to over-provision VMs

intentionally. The total cost for the base scheduler is $1.72 and latency violations are 4 out of 36.

From Figure 3.5(c), we can visually confirm that the speculative scheduler gradually allocates smaller numbers of cores, unlike the baseline scheduler which abruptly allocates larger numbers of cores. This is a direct effect of the speculative VM allocation. In fact, all the allocated VMs are launched by *scheduleAllocVMs* at Line 10 in Algorithm 5. Since there are already enough cores by the time *baseScheduler* at Line 2 tries to schedule, it does not need to create any new VMs. As a result, there are no latency drops in Figure 3.5(d). The total cost for the speculative scheduler is $1.01 and latency violations are 2 out 36. The cost is a 41% improvement compared to the baseline scheduler.



(a) Number of routes and cores (baseline).

(c) Number of routes and cores (speculative).

(b) Requested latency and execution time (baseline).

(d) Requested latency and execution time (speculative).

Figure 3.5. Experimental results for the Nationwide dataset.

Figure 3.6 shows a VM allocation sequence scheduled by the speculative scheduler created for the Nationwide dataset. We can see that five c4.large instances (ID = 0 to 4) are allocated by 2910 seconds, and then a c4.xlarge instance (ID = 5) is allocated at 3210 seconds. Interestingly, at 9900 seconds, the scheduler chooses to keep the c4.xlarge instance instead of the c4.large (ID = 4) instance even though the c4.large can also satisfy the requested processing latency. This is because the c4.xlarge will have the billing cycle later than the c4.large does; however, in a truly continuous optimization scenario, it may be less critical because wasted

VM hours will be negligible compared with the application execution time.

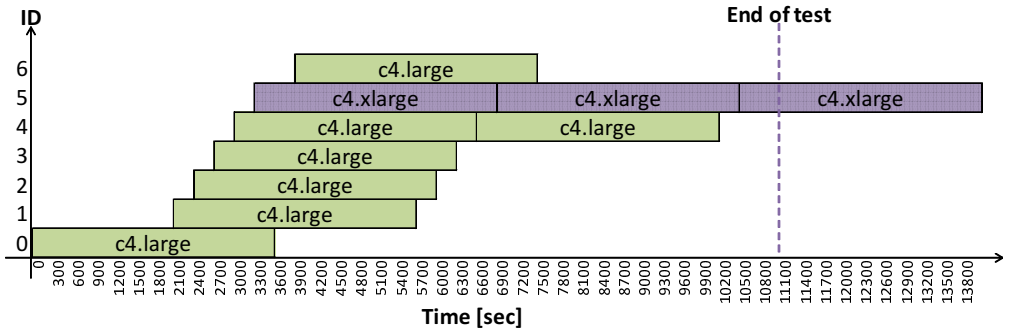


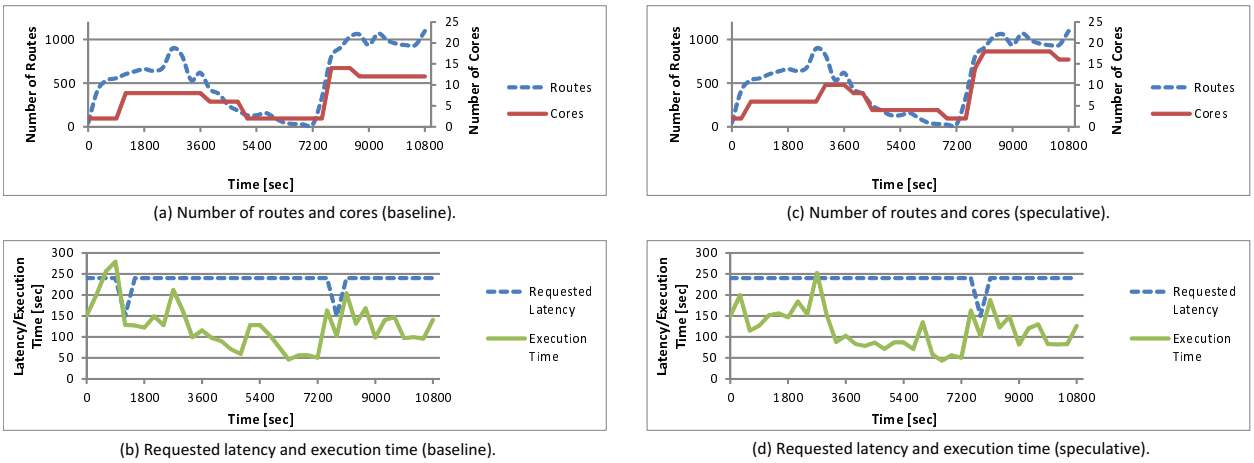
Figure 3.6. VM allocation sequence for the speculative scheduling algorithm created from the Nationwide dataset.

3.5.2.2 Dallas Dataset

Results for *baseScheduler* and *specScheduler* for the Dallas dataset are shown in Figure 3.7(a)-(d). From Figure 3.7(a), we can confirm that the baseline scheduler allocates VMs for 6 cores at 1200 seconds and then for 12 cores at 7800 seconds. In Figure 3.7(c), the speculative scheduler follows changes of the routes smoothly for the first stage of the sequence; however, at 7500 seconds, it fails to predict the number of routes correctly and ends up allocating less VMs than actually needed at 7800 seconds. It allocates two more VMs at 7800 seconds and that is the reason why we see a requested latency drop at 7500 seconds of Figure 3.7(d). The current slope-based time series predictor cannot keep up with sudden route changes. Apart from the time series prediction failure of the speculative scheduler, both schedulers are able to adapt to the spike and allocate/deallocate VMs successfully. For the base scheduler, the total cost is $0.83 and latency violations are 2 out of 36. For the speculative scheduler, they are $1.38 and 1 out 36 respectively.

3.5.3 Comparison with Static Scheduling

Elastic scheduling can adapt to unforeseen fluctuating demand whereas static scheduling cannot. We compare static scheduling against our proposed elastic algorithms to confirm



(a) Number of routes and cores (baseline).

(c) Number of routes and cores (speculative).

(b) Requested latency and execution time (baseline).

(d) Requested latency and execution time (speculative).

Figure 3.7. Experimental results for the Dallas dataset.

the effectiveness of our approach's adaptivity. Experimental settings are the same as Section 3.5.2, and we use both Nationwide and Dallas datasets. For the static scheduling, we test VM configurations with cores $= \{2, 4, 8, 10, 12, 14, 16\}$ for Nationwide and $\{2, 6, 8, 10, 12\}$ for Dallas. Comparison of VM hours, cost, and the percentage of latency violations are shown in Tables 3.4 and 3.5, respectively, for the Nationwide and Dallas datasets. Since static schedules do not waste any VM hours at all (*i.e.*, they do not visit a situation as in Figure 3.6), we compute the cost for our elastic schedulers in proportion to the execution time for fairness. In the tables, VM hours means the net number of cores used over 3 hours of the experiments. The percentage of latency violations is computed out of 36 scheduling problems over 3 hours.

For the Nationwide dataset, the speculative scheduler successfully improves over the baseline scheduler in terms of latency violations by 50% with 41% less VM hours and cost. The performance of the static schedule with 12 cores is comparable to the speculative scheduler (0% vs. 5.56% violations). When comparing the two, the speculative scheduler achieves a similar performance with 49% less VM hours and cost. For the Dallas dataset, the base scheduler improves latency violations over the closest static allocation approach (6 cores) by 66% despite it uses 17% less VM hours. When comparing to the static schedule with 8 cores, the speculative scheduler slightly over-provisions due to inaccuracy of both time series and

Table 3.5. VM hours, cost, and latency violations for elastic and static scheduling algorithms (Dallas dataset).

Policy	Cores	VM hours [core·hour]	Cost [USD]	Violations [%]
Static	2	6	0.33	63.89
	4	12	0.66	44.44
	8	24	1.32	19.44
	10	30	1.65	13.89
	12	36	1.98	0
	14	42	2.31	0
	16	48	2.64	0
Auto Scaling	2 to 8	15.96	0.88	25
Elastic (base.)	2 to 22	31.33	1.72	11.11
Elastic (spec.)	2 to 14	18.33	1.01	5.56

Policy	Cores	VM hours [core·hour]	Cost [USD]	Violations [%]
Static	2	6	0.33	61.11
	6	18	0.99	16.67
	8	24	1.32	2.78
	10	30	1.65	2.78
	12	36	1.98	0
Elastic (base.)	2 to 14	15	0.83	5.56
Elastic (spec.)	2 to 18	25.15	1.38	2.78

resource predictions. That is, it spends 5% more VM hours and cost, but equally performs as the static scheduler with 8 cores in terms latency violations.

While our elastic scheduling policy exhibits a small percentage of latency violations, we note that any static scheduler, other than a very highly provisioned one, will not be able to guarantee zero latency violations. For any static VM allocation, there is a possibility that it will not be sufficient for some level of demand. Our elastic schedulers, on the other hand, successfully adapt to unforeseen computational demand changes and scale VMs accordingly with reasonably low cost.

3.5.4 Comparison with Auto Scaling

Since threshold-based auto scaling is commonly used as an application-agnostic scaling technique, we test it against our application aware approach. We use the same experimental settings as Section 3.5.2. We implement the following rules that are compatible to Amazon Auto Scaling [42]:

- VM instance type: c4.large.
- VM allocation: minimum 1, maximum 5 instances.
- Rule to scale up: if the average CPU utilization of allocated VMs is consistently above 70% for 2 minutes, add one VM.
- Rule to scale down: if the average CPU utilization of allocated VMs is consistently below 30% for 2 minutes, reduce one VM.
- Cooldown period: once scaling decision is made, no new scaling activity is performed for 300 seconds.
- VM termination: the instance that is closest to the next billing cycle is chosen to terminate.

Figure 3.8 shows average CPU utilization and the number of VMs over the 3 hour experiment period. The auto scaler successfully increases VMs up to 4, and then decreases them to 1. The results are summarized in Table 3.4. Since the threshold-based auto scaler is not aware of the application performance requirement (*i.e.*, 240 seconds latency) at all, it under-provisions the VMs and ends up producing relatively many latency violations compared to our elastic schedulers.

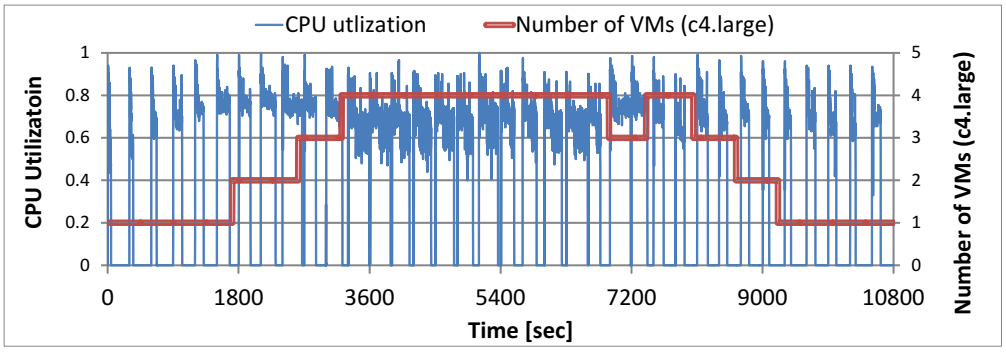


Figure 3.8. CPU utilization and VM allocation by a threshold-based auto scaling.

3.6 Summary

In this chapter, we presented an elastic middleware framework that is specifically designed to solve ILP problems generated from continuous air traffic streams over an IaaS cloud. We proposed a speculative VM scheduling algorithm with time series and resource prediction models. Experiments show that our speculative VM scheduling algorithm can achieve a similar performance to a static schedule while using 49% less VM hours for a smoothly changing air traffic. However, for a sharply changing air traffic, our speculative VM scheduling algorithm costs slightly more VM hours to achieve the same performance. Our algorithm is able to adapt dynamically to potentially unforeseen fluctuating demand with a reasonable prediction accuracy. We plan to improve model prediction accuracy especially for time series using a more complex model (*e.g.*, ARMA).

We have several potential directions for future work. First, we would like to apply our middleware to other application areas since the concept of solving large scale ILP problems created from continuous data stream is widely applicable. Candidate application areas include: public transportation routing [77], investment portfolio optimization [78], and marketing budget optimization [79], [80]. Second, we plan to explore other modeling techniques for predicting resource allocation. Finally, we plan to extend our framework to support other optimization policies such as budget constrained and deadline constrained policies.

CHAPTER 4
SUSTAINABLE ELASTIC STREAM DATA PROCESSING

4.1 Introduction

The need for real-time stream data processing is ever increasing as we are facing an unprecedented amount of data generated at high velocity. Upon the arrival of a stream event, we want to process it as quickly as possible to timely react to anomalies such as aircraft airspeed sensor failure [4] or unusually high CPU usage in data centers [5]. Traffic management [49], [81] and sensor data processing from Internet-of-Things (IoT) devices [82]–[84] are also common real-time stream processing applications. To process these fast data streams in a scalable and reliable manner, a new generation of stream processing systems has emerged: systems such as Storm [6], [85], Flink [9], [86], and Spark Streaming [11], [72] have been actively used and developed in recent years. Cloud computing offers on-demand elasticity to these systems through its pay-per-use cost model for VMs. Using cloud computing, *elastic stream data processing* autonomously allocates or deallocates VMs on-demand to match fluctuating application workload [87]. It helps save cost while maintaining required QoS objectives.

Figure 4.1 shows a common real-time stream data processing environment. This stream processing system environment consists of a data producer, a message broker (*e.g.*, Kafka), a stream processing system, and a data store. The data producer sends events at the input data rate of $\lambda(t)$ Mbytes/sec at time t, and they are appended to message queues in the

Portions of this chapter previously appeared as: Shigeru Imai, Stacy Patterson, and Carlos A. Varela, "Maximum sustainable throughput prediction for data stream processing over public clouds" in *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid 17)*, 2017, pp. 504–513.

Portions of this chapter are to appear in: Shigeru Imai, Stacy Patterson, and Carlos A. Varela, "Uncertainty-aware elastic virtual machine scheduling for stream processing systems" in *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid 18)*, 2018.

message broker. The stream processing system pulls data out of the message broker as quickly as possible at the throughput of $\mu(t)$ Mbytes/sec using $m(t)$ VMs. After the stream processing system processes events, it optionally stores results in the data store (*e.g.*, file system or database). The message broker acts as a buffer between the data producer and the stream processing system, enabling them to work asynchronously.

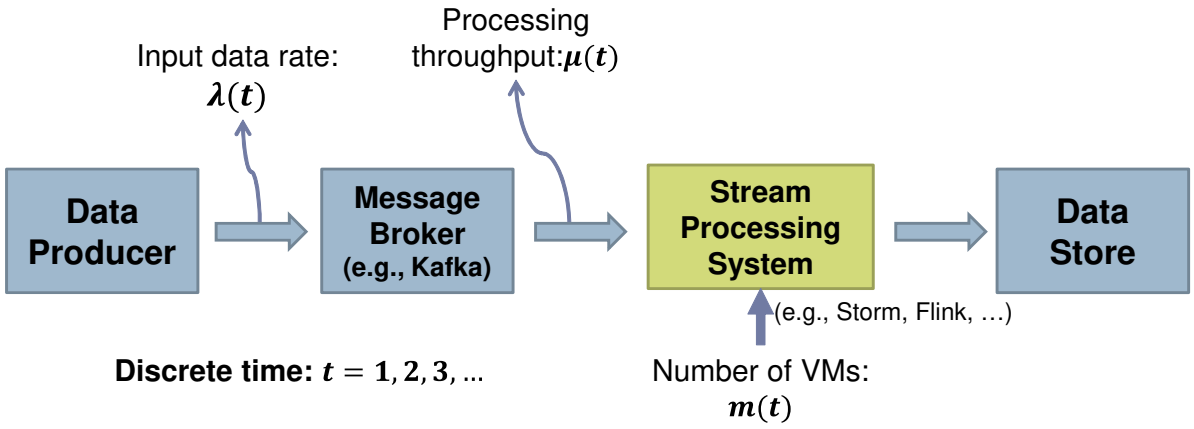


Figure 4.1. Common real-time stream processing environment.

One concern in the stream processing environment shown in Figure 4.1 is that when the input data rate $\lambda(t)$ is consistently higher than the processing throughput $\mu(t)$, backlogged-data in the message broker accumulates, eventually making the whole system inoperable. Figure 4.2 shows an example time series of increasing input data rates and data processing throughput measured in the stream processing environment in Figure 4.1. The data processing throughput $\mu(t)$ starts diverging from the increasing input data rate $\lambda(t)$ around 140 seconds when the input data rate is about 2.5 Mbytes/sec. This means that the stream processing system cannot keep up with the input data rate, and the incoming events starts accumulating in the message broker. If this condition persists, data processing is *unsustainable* as the accumulated events will eventually exceed the capacity of available storage in the message broker. On the other hand, if the input date rate is less than 2.5 Mbytes/sec, data processing is *sustainable*.

For sustainable data processing, it is critical to control the processing throughput $\mu(t)$

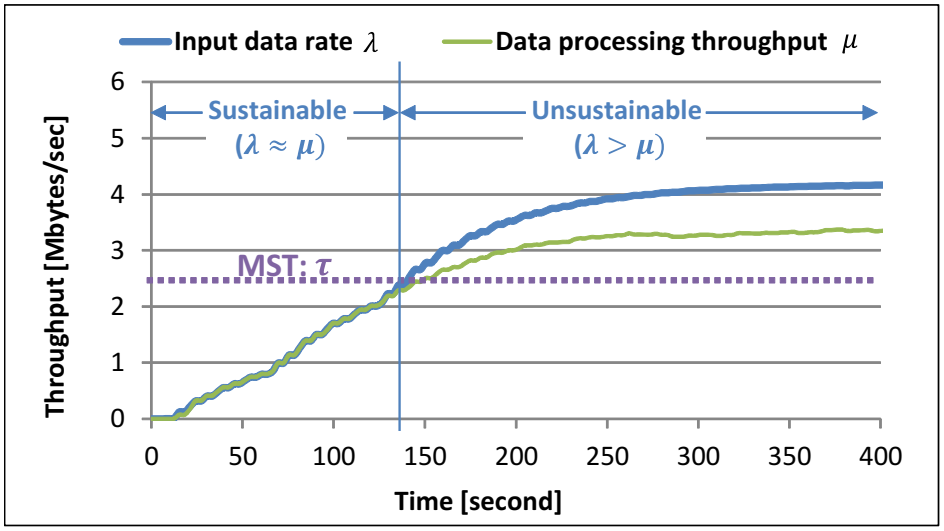


Figure 4.2. Example time series of input data rates and processing throughput.

through $m(t)$ so that $\mu(t)$ can always match the the input data rate $\lambda(t)$. To help estimate how many VMs we should allocate to achieve sustainable data processing, we define the *Maximum Sustainable Throughput* as follows.

Definition of MST : *Maximum sustainable throughput (MST) is the maximum throughput that a stream processing application can process indefinitely with a given number of VMs.*

Due to the application's performance variability and measurement noise, observed MST is a random variable affected by the number of VMs. Thus, we assume it follows a probability distribution $\Pr(\tau|m)$, where τ is an MST value for a number of VMs m. As long as the system's MST is greater than the data input rate $\lambda(t)$, data is immediately consumed and the queue size in the message broker remains short. We call this condition *throughput-QoS* and express it by the following relationship:

$$\forall t : \lambda(t) \leq \tau(m(t)), \tag{4.1}$$

where $\tau(m(t))$ is a true MST value at time t that follows the probability distribution $\Pr(\tau|m(t))$. If this condition is true for all t, data processing is sustainable. However,

since this probability distribution is unknown, we cannot directly predict the exact value of $\tau(m(t))$. Thus, we estimate the true MST value using an estimator function $\hat{\tau}(m)$ and try to find a sequence of numbers of VMs $m(t)$ $(t = 1, 2, ...)$ that satisfies the following condition:

$$\forall t : \lambda(t) \leq \hat{\tau}(m(t)). \tag{4.2}$$

QoS objectives used by recent elastic stream data processing systems include keeping resource utilization within certain range [42], [88], guaranteeing latency [7], [36], [89], and minimizing latency spikes [35]. One thing in common among these works is that these metrics are concerned about stream application performance only, but not about sustainability of the entire stream processing environment including message brokers. This thesis focus on the sustainability of data stream processing by ensuring the throughput-QoS condition in (4.1). More specifically, we tackle the following research questions.

Predicting MST : To satisfy the condition shown in (4.2), we need a prediction model $\hat{\tau}(m)$ to estimate the MST for a given number of VMs. Due to the complex nature of distributed data processing, some applications scale linearly while some shows scale non-linearly, as we will show later in Section 5.4. To model such complex behavior, it is not feasible to model application performance analytically. Thus, we collect performance metrics sample from actual application runs and train prediction models using regression. Research questions we pose in MST prediction are as follows:

1. How can we model non-linear scaling behavior of stream data processing systems?
2. How to save cost and time to collect training samples while achieving high prediction accuracy?

VM Scheduling : One way to classify elastic VM scheduling techniques is by whether they are *reactive* or *proactive*. Reactive scheduling allocates or deallocates VMs in response to some resource usage metric changes (*e.g.*, CPU, memory, network), such as is done in

AWS AutoScaling [42]. On the other hand, proactive scheduling makes scaling decisions using predictions of application performance and future workload, to scale-up the number of VMs before performance degrades or to scale-down the number of VMs to reduce cost while maintaining throughput-QoS. Proactive scheduling approaches that incorporate measures of application performance and workloads have the potential to achieve better throughput-QoS at lower cost, when compared to reactive approaches. However, their cost-effectiveness and QoS depend on the accuracy of the performance and workload prediction mechanisms. We have the following research questions for proactive VM scheduling:

1. Can a VM scheduling algorithm be aware of inaccuracy of prediction models and still produce robust and cost-effective VM configurations for sustainable data stream processing?

2. How does online learning and workload prediction abilities improve throughput-QoS satisfaction?

For the rest of the chapter, we show technical background of the elastic sustainable stream data processing in Section 4.2, and show our elastic stream data processing framework including MST sampling method in Section 4.3.

4.2 Background of Elastic Stream Processing

We review representative distributed stream processing systems in Section 4.2.1 and recent research works of elastic stream processing in Section 4.2.2.

4.2.1 Distributed Stream Processing Systems

4.2.1.1 Comparison of Distributed Stream Processing Systems

Table 4.1 shows a summary of four representative distributed data stream processing systems from the Apache foundation: Storm [6], [85], Samza [10], [90], Flink [9], [86], and Spark Streaming [11], [91].

Table 4.1. Comparison of representative distributed stream processing systems (as of 2018).

	Storm	Samza	Flink	Spark Streaming
Version	1.2.1	0.14	1.4.1	2.2.1
Processing Model	stream	stream (batch)	stream (batch)	micro-batch (batch)
Fault-tolerant Mechanism	tracking ack messages	checkpointing	checkpointing	checkpointing
Processing Guarantee	at least once / at most once	at least once	exactly once	exactly once
Kafka Support	yes	yes (required)	yes	yes
Supported Resource Manager	YARN (unofficial) / Mesos	YARN	YARN	YARN / Mesos
Run-time Reconfiguration	yes (no state recovery)	no	yes (not elastic)	yes (elastic)

Unlike Storm only supports pure stream processing, Samza, Flink, and Spark Streaming support both batch and stream (or micro-batch for Spark Streaming) processing in a single processing engine. Spark Streaming's micro batching generates high throughput with a price of high latency.

Different fault-tolerant mechanisms lead to different processing guarantees. In Storm, each source processing unit (*i.e.*, spout) keeps tracks of acknowledgment messages from its succeeding processing units (*i.e.*, bolts) for all the events the spout transmitted. The spout retains an event until it receives the acknowledgment. When the wait for acknowledgment times out, the spout replays the corresponding event. In case an event is just fully processed and timed-out simultaneously, the spout can replay the same event twice. Because of this mechanism, Storm processing guarantee remains *at-least-once*. When the acknowledgment mechanism is turned off, Storm becomes failure-prone and only supports *at-most-once*. Samza is designed to work with Kafka as the message broker and its fault-tolerance is also based on Kafka. Samza's StreamTasks pull data from Kafka in parallel while each task keeps track of a current offset in a Kafka partition. To record where to resume reading in case of failure, Samza periodically checkpoints the current offset for each task. When an failure

occurs, it is possible that Samza replays events that are already processed since the last checkpoint. Therefore Samza's processing guarantee is also at-least-once. The Storm and Samza's at least once processing guarantees can become problematic for applications that are not idempotent. For example, when failures occur for a typical WordCount application, resulting counts may be incorrect due to duplicated events. Flink and Spark Streaming offer *exactly-once* processing guarantees, which is a more desirable property compared to at-least-once. Flink periodically takes a global snapshot as a set of all task states in a distributed manner similar to Chandy-Lamport algorithm [92]. In case of failure, it loads checkpointed states to the tasks and replays events from the time when the checkpoint is taken. This recovers task states at the time of failure, and thus the tasks can resume processing without making duplicate outputs. Spark Streaming takes a similar approach by checkpointing states for the Spark's resilient distributed datasets (RDDs).

All systems can ingest events from Kafka [93] and especially Samza is tightly coupled with it. YARN [94] and Mesos [95] are general resource management frameworks for shared clusters supported by many data processing frameworks. Except that Samza requires YARN as a resource management framework, all the other frameworks have their own default standalone cluster manager. Unlike YARN is supported by all four frameworks (support for Storm is unofficial), Mesos is only supported by Storm and Spark Streaming.

The levels of run-time reconfiguration support largely vary depending on each framework. Storm exposes reconfiguration functionalities without any state recovery. Since Samza's configuration is completely immutable once a job starts, it prevents any dynamic run-time reconfiguration [96]. Flink supports a run-time reconfiguration feature with state recovery to change the parallelism, but elastic scaling (dynamic addition or removal of computational resources) is not supported. Spark Streaming has the most complete dynamic scaling feature among the four frameworks. If the task scheduling is delayed more than a pre-configured threshold period, it automatically requests more resources from the resource manager and scales up gracefully with load rebalancing and state recovery. It was initially

supported in v1.6.1 with some issues and has been fixed since v2.0.0 [97].

4.2.1.2 Scaling Stream Processing Applications

In typical stream processing systems such as Storm, the user writes an application in the form of *topology* connected by multiple *processing units*. As shown in Figure 4.3, the topology can be scaled up with arbitrary parallelism of threads when they are deployed on the cluster. As illustrated in Figure 4.3, one way to determine the parallelism of a topology

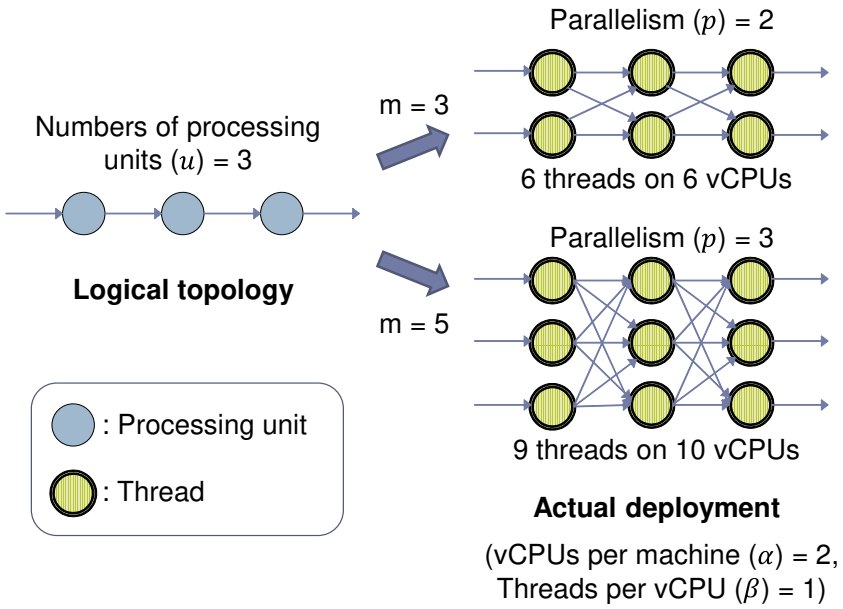


Figure 4.3. Examples of scaling an application topology with three processing units ($u = 3$) to three and five machines ($m = 3$ and 5), respectively. Each machine has two vCPUs ($\alpha = 2$) and one vCPU is assigned per thread ($\beta = 1$).

is as follows. Given the following parameters,

- u: Number of processing units
- m: Number of virtual machines
- α: Number of vCPUs per virtual machine
- β: Number of threads per vCPU

the ratio between the number of vCPUs and threads can be expressed with parallelism p (*i.e.*, the number of threads per processing unit) as follows:

$$\#\text{vCPUs} : \#\text{threads} = 1 \ : \ \beta = m\alpha \ : \ up. \tag{4.3}$$

Solving (4.3) for p, we get $p = \frac{\alpha\beta m}{u}$. Since parallelism p must be integer ≥ 1, finally we obtain p as follows:

$$p = \max\Big(1, \Big\lfloor \frac{\alpha\beta m}{u} \Big\rfloor\Big). \tag{4.4}$$

This scaling method tries to assign β threads per vCPU. As shown in Figure 4.3, given $u = 3, \alpha = 2, \beta = 1$, when we deploy the topology to $m = 3$ machines, 6 threads are assigned on 6 vCPUs. In case of $m = 5$ machines, 9 threads are assigned on 10 vCPUs with one vCPU left unused.

4.2.2 Elastic Stream Processing Systems

In this section, we survey existing elastic stream processing systems. First, we show what metrics are used in these systems, and then compare their approaches from several aspects.

4.2.2.1 Performance Metrics

Various metrics have been used to make scaling decisions for elastic stream processing systems. We describe representative metrics: latency, throughput, traffic load, and resource utilization.

Latency Given a logical topology composed of multiple processing units as shown in Section 4.2.1.2, latency is the difference between the time when the topology receives a message from data source and the time when the topology finishes processing the message. Queueing theory [98], [99] is a mathematical theory about queuing systems (*e.g.*, customers wait in queue before receiving service). Given a distribution of inter-arrival of messages and a

distribution of service time, it can estimate the average queue length and average wait time after the system reaches its steady state. Kendall's notation is conveniently used to describe queuing models using three components $A/S/c$, where A is a probability distribution for inter-arrival time, S is a probability distribution of service time, and c is the number of servers. It has been used to model the processing latency of distributed data stream processing applications [7], [36]. DRS models each processing unit as an $M/M/c$ system (*i.e.*, inter-arrival: Poisson, service: exponential, number of servers: c) [7]. Potentially due to its inability to express pipe-lining effects between processing units, when the application is network intensive, estimated latency is significantly inaccurate (up to 10x different). Lohrmann et al. models each processing unit as a $GI/G/1$ system (*i.e.*, inter-arrival: unknown, service: unknown, number of servers: 1) [36]. It uses Kingman's formula [100] that is known to approximate the wait time accurately when the system is in heavy traffic. Since the model assumes the service rate is always greater than the message arrival rate, when the processing units cannot keep up with fast message arrival, their latency model becomes unusable.

Li et al. claim that queueing theory does not work for distributed stream processing systems since 1) assumptions required by queueing theory may not hold in complex distributed stream processing systems and 2) the behavior of multi-point to multi-point queueing network used in distributed stream processing systems is still unsolved in queuing theory [89]. Instead, they proposed a topology-aware latency prediction model trained with support vector regression and reported that the average prediction accuracy of 83.7%.

Throughput Throughput is a metric to quantify how many messages a stream processing system can process per unit time. ElasticStream [34] estimates maximum throughput for a given number of VMs using a linear model. Gedik et al. monitor throughput and improve it through a control algorithm [101].

CPU Utilization AWS AutoScaling [42] is a general-purpose auto-scaling service for Amazon EC2 cloud. It offers configurable automated scaling policies based on resource utilization

metrics. Their Target Tracking Scaling Policies allow the user to specify a target resource utilization (*e.g.*, 80% CPU utilization) and AutoScaling automatically adds or removes VMs to maintain the target utilization. Esc monitors CPU utilization for all computing nodes and detects an overloaded condition if all the nodes have CPU utilization greater than some threshold [88]. FUGU also uses upper/lower CPU utilization threshold values to trigger scale-up/down operations [35].

Other Resource Contention Metrics T-Storm monitors all the traffic between processing units and use it to minimize inter-node traffic when assigning processing units to computing nodes [102]. Gedik et al. compute congestion index as a metric to measure how busy communication channels between processing units are [101]. It is computed as a fraction of time when a send call is blocked because the receiver's message queue is full. Stela also detects congested conditions for a processing unit if the incoming data rate to the processing unit is greater than the outgoing throughput [103]. Spark Streaming's dynamic allocation monitors the duration for backlogged tasks [72].

4.2.2.2 Summary of Elastic Stream Processing Systems

A summary of recent elastic stream processing systems is shown in Figure 4.2. They are listed in the chronological order.

It is clear that model-free approaches are based on resource contention metrics such as CPU utilization and traffic congestion. They are all reactive and do not offer any guarantee on application performance. Model-based approaches are based on several performance metrics: resource utilization (AWS AutoScaling [42]), latency (FUGU [35], Nephele [36], DRS [7]), and throughput (ElasticStream [34], Imai et al. [37], [49]). AWS AutoScaling [42] recently started to support a model-based scaling service in their Target Tracking Scaling which automatically estimates the required number of VMs to satisfy pre-configured resource utilization. However, the details of their model remain undisclosed. Latency-aware approaches [7], [35], [36] are all reactive and try to maintain application latency consis-

Table 4.2. Summary of recent elastic stream processing systems.

Author (System)	Year	Performance Metric	Model	Proactive/Reactive
AWS AutoScaling [42] (Step Scaling)	2010	resource utilization	model-free	reactive
Ishii and Suzumura (ElasticStream) [34]	2011	throughput	model-based	proactive (1-step lookahead)
Satzger et al. (Esc) [88]	2011	CPU utilization	model-free	reactive
Heinze et al. (FUGU) [35]	2014	latency	model-based	reactive
Gedik et al. [101]	2014	congestion, throughput	model-free	reactive
Xu et al. (T-Storm) [102]	2014	inter-node traffic	model-based	reactive
Lohrmann et al. (Nephele) [36]	2015	latency	model-based	reactive
Fu et al. (DRS) [7]	2015	latency	model-based	reactive
Xu et al. (Stela) [103]	2016	congestion	model-based	reactive
Spark Streaming [97]	2016	backlogged duration	model-free	reactive
Imai et al. [49]	2016	throughput	model-based	proactive (1-step lookahead)
AWS AutoScaling [42] (Target Tracking)	2017	resource utilization	model-based	reactive
Imai et al. [37]	2018	throughput with uncertainty estimation	model-based	proactive (N-step lookahead)

tent. Since they do not concern the sustainability of stream data processing, our sustainable stream processing approach can be complementary to their approaches. It is evident that proactive scheduling has not been explored well in elastic stream processing systems. ElasticStream [34] presented 1-step lookahead proactive scheduling with a linear throughput prediction model, which was evaluated up to 4 VMs. In this thesis, we explore the sustainability of elastic data stream processing while achieving larger scalability (up to 128 VMs). We present non-linear throughput prediction models in Chapter 5 and VM scheduling with multi-step lookahead proactive scheduling in Chapter 6.

4.3 A Framework for Sustainable Elastic Stream Data Processing

4.3.1 Maximum Sustainable Throughput

As we have defined in Section 4.1, MST is a metric to quantify the maximum processing throughput for the stream processing system. In this section, we show how we measure the true MST of the stream processing system for a given number of VMs. In the common stream processing environment in Figure 4.1, when there is no backlog in the message broker, the

processing throughput $\mu(t)$ never exceeds the input data rate $\lambda(t)$ since the stream processing system cannot process data unless they are provided by the data producer. To obtain the true performance of the target stream processing system, we need to ensure that the condition $\lambda(t) \geq \mu(t)$ always holds so that the data is always available to process by the stream processing system. However, the data producer or the message broker can be a bottleneck and hinder the stream processing system from processing the data at its maximum speed. To avoid this issue, we pre-load enough data to the message broker as shown in Figure 4.4, and effectively simulate the condition $\lambda(t) \geq \mu(t)$.

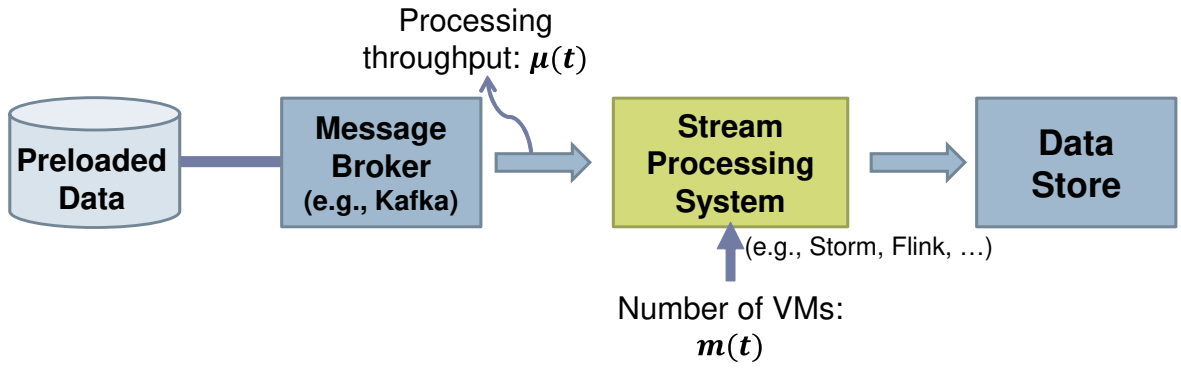


Figure 4.4. Maximum sustainable throughput measurement environment.

Figure 4.5 shows an example of throughput transition over time for a stream application that processes web access logs (*i.e.*, *Unique Visitor* in Section 5.4), observed in the environment shown in Figure 4.4. In this setting, the stream processing system pulls data from the message broker as quickly as possible. The throughput gradually grows until 100 seconds and then converges. After the convergence, we start sampling the throughput, and that is the MST we measure. In this example, MST is around 42 Mbytes/sec. To detect convergence of the throughput, we use the following *K out of N* method: keep monitoring the latest N samples and if K samples are within $\pm\delta\%$ from the previous samples, we determine that the throughput is converged. As long as enough data is pre-loaded in the message broker , the stream processing system is able to process the data indefinitely up to the rate of MST.

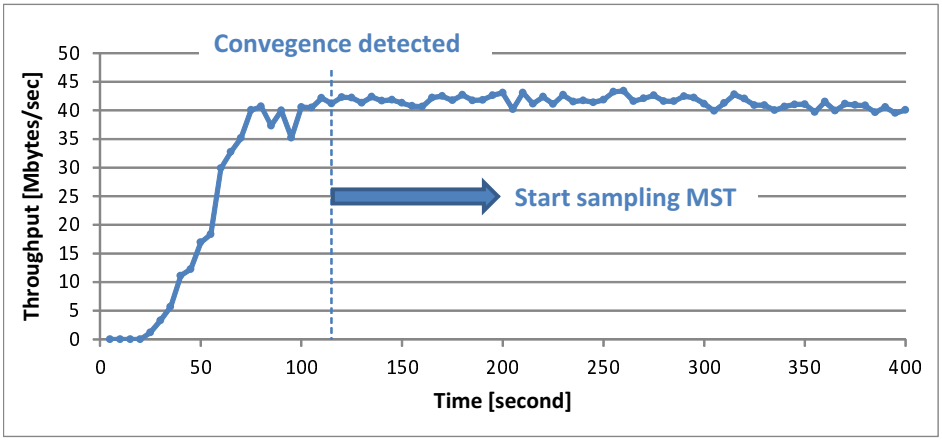


Figure 4.5. Convergence of throughput for a web access log processing stream application.

4.3.2 Sustainable Elastic Stream Data Processing Framework

Figure 4.6 shows the architecture of the proposed sustainable elastic stream data processing framework, which runs on a cloud computing environment. To satisfy the throughput-QoS in (4.1), the VM scheduler reconfigures the application and scales up or down VMs to match the input data rate. Reconfiguration operations are implemented in the stream processing framework using application-specific mechanisms, for example, the rebalance API in Apache Storm. The MST prediction model is trained with performance sample metrics obtained from the application monitor. We train the model offline as we describe in Chapter 5 as well as online. The VM scheduler makes scaling decisions with a fixed time interval using a MST prediction model and a workload prediction model. The workload predication model obtains workload information (*i.e.*, input data rates) from the workload monitor, and updates itself to predict future workloads. The MST model is used to predict the number of VMs that will be needed to process the forecasted workload. Since it takes tens of seconds to minutes to launch a VM, predicting future workloads and proactively allocating VMs to cover the predicted workloads can be effective in achieving a high QoS satisfaction rate.

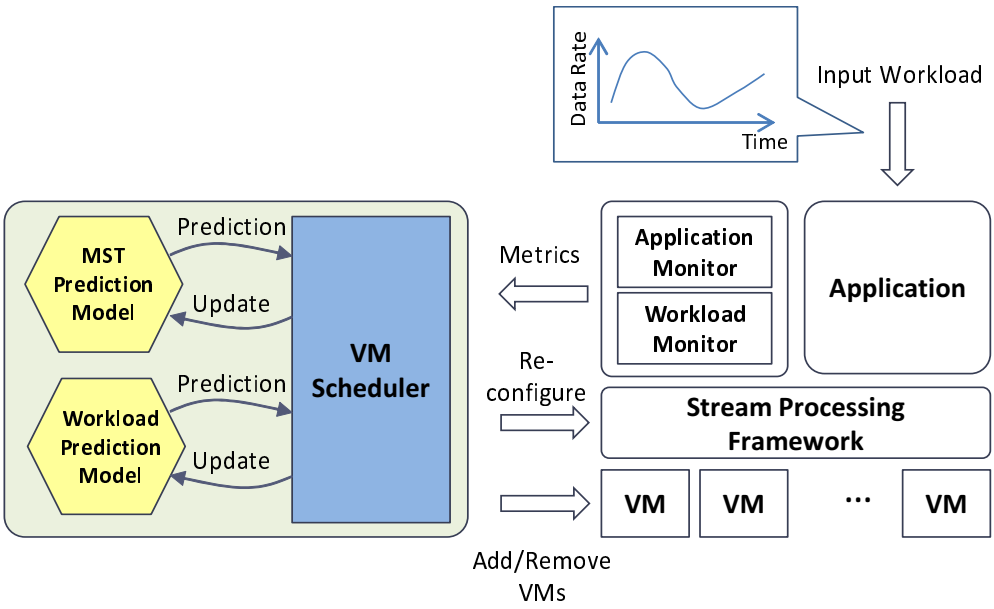


Figure 4.6. Proposed sustainable elastic stream data processing framework.

4.4 Summary

In this chapter, we described the background of elastic stream processing and introduced a framework for sustainable elastic stream data processing using maximum sustainable throughput (MST). Based on this framework, we show how we train MST prediction models in Chapter 5 and VM scheduling techniques in Chapter 6.

CHAPTER 5
MAXIMUM SUSTAINABLE THROUGHPUT PREDICTION

5.1 Introduction

To realize a MST-based elastic stream processing in a cost efficient manner, we need an MST prediction model for a given stream application and number of VMs. However, most recent elastic stream processing studies primarily focus on guaranteeing latency [7], [35], [36], [89]. ElasticStream [34] is the only elastic streaming system that estimates maximum throughput. It uses a model that is linear in the number of VMs, which is not realistic for all applications, as we show in this chapter.

Due to the complex nature of distributed data processing, it is not always feasible to model application performance analytically, without any observations of application performance. Recent works have used supervised learning to model the performance of distributed batch processing applications [31], [104], [105]. These works collect performance metric samples from actual application runs and train prediction models using regression. Among them, our work was inspired by Ernest [104], which models job completion time for Apache Spark's batch processing as a polynomial of the input data size and number of machines. It uses training samples obtained from a few machines to predict the performance for a larger number of machines. We take a similar approach to Ernest to predict MST values for stream processing applications. While Ernest uses a single prediction model, we note that there are cases where a single model cannot be trained to work well for multiple stream processing

Portions of this chapter previously appeared as: Shigeru Imai, Stacy Patterson, and Carlos A. Varela, "Maximum sustainable throughput prediction for data stream processing over public clouds" in *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid 17)*, 2017, pp. 504–513.

Portions of this chapter have been submitted to: Shigeru Imai, Stacy Patterson, and Carlos A. Varela, "Maximum sustainable throughput prediction for large-scale data streaming systems" in *IEEE Transactions on Cloud Computing*.

applications.

In this chapter, we propose a cost-effective framework to predict MST values for stream processing applications with various scalability characteristics. Since it may be difficult to find one prediction model that works well for all of the applications, we first train several models using linear regression. We then select the best-fitting model for the target application through the evaluation of extra MST samples. To save cost and time to collect MST samples while achieving high prediction accuracy, we statistically determine the most effective set of VMs within a budget. For evaluation, we use Intel's Storm benchmarks [106] running on Amazon EC2 cloud. Using up to 128 VMs, experiments show that the models trained by our framework accurately predict MST.

The rest of the chapter is organized as follows. In Section 5.2, we present related work on performance models used in elastic data processing. Section 5.3 presents our MST prediction framework. Section 5.4 shows the evaluation of our models' prediction accuracy, and Section 5.5 discusses the results of experiments. Finally, we conclude the chapter in Section 5.6.

5.2 Related Work

There have been a number of research projects on predicting job completion time for batch processing. There are prediction models specifically designed for MapReduce. AROMA [31] takes a purely data-driven approach, in which it combines clustering of resource usage profiles and regression with Hadoop MapReduce-specific variables. ARIA [107] shows an analytically-designed job completion time model based on the general map-reduce programming framework [68], Ernest [104] models job completion time for Spark's batch processing based on computation and communication topology. Ernest represents job completion time as a polynomial of the number of machines and the size of input data. Compared to AROMA and ARIA, the model used in Ernest only requires target scaling factors (*i.e.*, input data size and number of machines), and therefore it is more widely applicable. Our

approach is similar to Ernest as we use the number of machines as the only variable in our models. However, Ernest assumes a single model, whereas our framework uses multiple models and selects the model that is expected to give the least prediction error for each test application. This helps us predict performance for both linearly and non-linearly scaling applications, as we show later in this paper.

To safely scale up a cluster to process fluctuating workload, it is important to know the maximum processing capacity of the cluster. Metrics similar to MST has been used for web service applications [108], [109], but maximum throughput has received less attention compared to latency in stream data processing. To the best of our knowledge, ElasticStream [34] is the only elastic stream processing system that tries to maintain the cluster's maximum throughput to handle fluctuating input data rates through automated VM allocation. It uses a linear model to predict maximum throughput; however, there are applications for which maximum throughput is not linearly scalable as we show in Section 5.4. Unlike ElasticStream, we model MST for both linearly and non-linearly scalable applications.

5.3 MST Prediction Framework

In this section, we propose an MST prediction framework that is designed with the following considerations.

1. **Application as a Black Box**: We see stream applications as a black box so that our framework is generally applicable to a wide range of stream processing frameworks.

2. **Default Task Scheduler**: We use a default task scheduler for the stream processing system and do not control task scheduling.

3. **Homogeneous VM Type**: We only use a single VM type, namely m4.large on Amazon EC2. If different stream processing tasks have different resource usage requirements, there may be room for performance improvement by optimizing the task scheduling of these tasks with heterogeneous VM types (*e.g.*, maximizing resource uti-

lization or minimizing inter-machine communication). However, since we choose to use a default task scheduler and do not aim for performance optimization, homogeneous VM types are sufficient.

4. **Saving Time and Cost for Training**: We limit the maximum number of VMs used to collect training samples to save time and cost.

In the following sub-sections, we show the details of the proposed framework. We first explain linear regression in Section 5.3.1, give an overview in Section 5.3.2, and describe the the following building blocks of the framework in order: MST prediction models (Section 5.3.3), VM subset selection (Section 5.3.4), and model training and selection (Section 5.3.5).

5.3.1 Linear Regression

We use linear regression [110] as the method to model the relationship between MST (dependent variable: y) and the number of VMs m (independent variable: $\mathbf{x}$). Given a training dataset $\mathcal{D}_{\text{train}} = \{(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(n)}, y^{(n)})\}$, linear regression looks for the optimal weight vector $\mathbf{w}$ in a prediction model $h(\mathbf{x}) = \mathbf{w^T x}$ that minimizes the following mean square error:

$$E(h) = \frac{1}{n}\sum_{i=1}^{n}(h(\mathbf{x}^{(i)}) - y^{(i)})^2. \tag{5.1}$$

By taking the gradient of Eq. (5.1), we can analytically obtain the optimal $\mathbf{w}$. Training is the process to obtain the optimal $\mathbf{w}$ for the prediction model $h(\mathbf{x})$ given the training dataset $\mathcal{D}_{\text{train}}$, whereas testing is to evaluate the trained model on a new test dataset.

5.3.2 Framework Overview

Figure 5.1 shows an overview of the proposed framework, which consists of two phases as follows.

Phase 1 In this phase, we empirically determine the most effective set of VMs to obtain training samples. First, we collect MST samples from representative benchmark applications in terms of resource usage patterns. Next, we enumerate subsets of a candidate training VMs set $\mathcal{V}_{\text{train}}$ to create various training sets. We train the models using linear regression with each training set and select the best VM subset $\mathcal{S}^*$ with the lowest test error. We do this process once offline.

Phase 2 In this phase, we train a new test application that the user wants to predict the MST values. We collect training MST samples only for the VM subset $\mathcal{S}^*$ determined from Phase 1. After training the models with the collected samples, we obtain extra validation samples that are not included in $\mathcal{S}^*$ and select the model with lowest validation error. We run this process per test application.

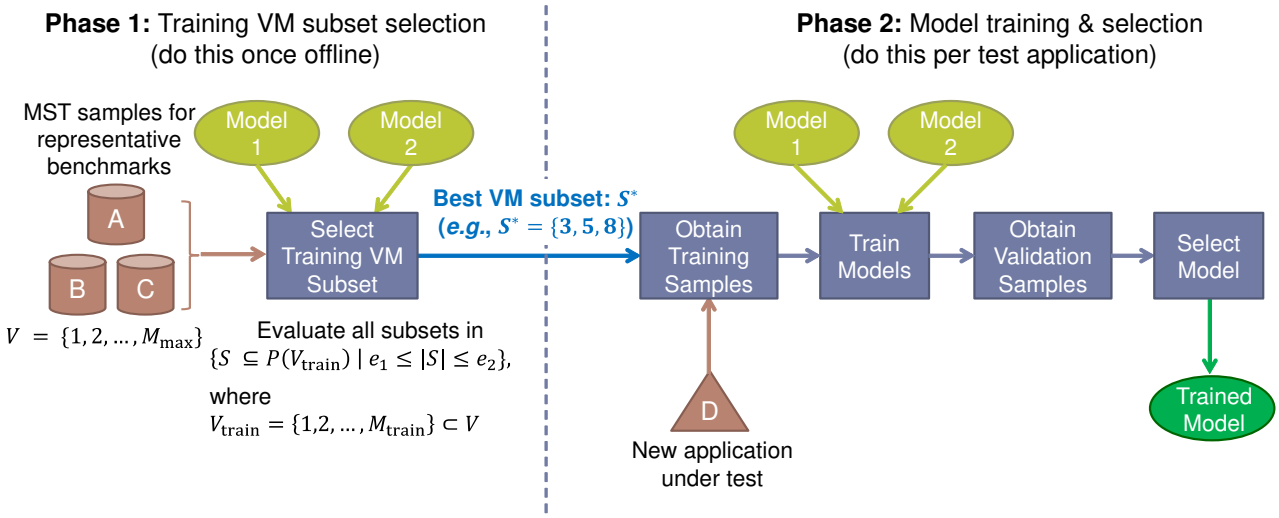


Figure 5.1. Overview of the MST prediction framework.

5.3.3 MST Prediction Models

We design two MST prediction models based on the stream processing environment we have shown in Section 4.1. We assume that a combination of the following factors determines the MST for a given number of VMs m.

1. *Parallel processing gain*: Performance improves as m increases.

2. *Input/output distribution overhead*: Performance decays linearly as m increases due to event transmissions from Kafka to the m worker nodes and also due to result transmissions from the m worker nodes to the data store.

3. *Inter-worker communication overhead*: Performance decays quadratically as m increases due to the $m(m-1)$ communication paths between m worker nodes.

Based on these factors, we create the following two models.

Model 1 : As shown in Eq. (5.2), This model predicts MST as a function of the number of VMs m. It is defined as the inverse of event processing time, which is represented as a polynomial of the number of VMs m. The terms have the following meanings: serial processing time (w_0), parallel processing time (w_1), input/output distribution time (w_2), and inter-worker communication time (w_3). Note that all the weights are restricted to be non-negative (*i.e.*, $w_i \geq 0, i = 0, ..., 3$).

$$f_1(m) = \frac{1}{Time(m)} = \frac{1}{w_0 + w_1 \cdot \frac{1}{m} + w_2 \cdot m + w_3 \cdot m^2}. \tag{5.2}$$

This model was inspired by Ernest [104], which models job completion time for Spark's batch processing jobs by considering computation and communication topology. Ernest represents job completion time as a polynomial of the number of machines and the size of input data. We take the inverse of processing time to model throughput.

Depending on the values of w_2 and w_3, there are cases where f_1 has a peak value at certain VMs $m = m^*$. On the other hand, if $w_2 = w_3 = 0$, f_1 monotonically increases and converges to $1/w_0$ as m goes infinity. If there is a peak, we can have multiple different VM counts to obtain a certain value of MST. Since it is not reasonable to choose the larger VM counts when the smaller one can achieve the same MST, we can effectively see that the predicted MST flattens out after its peak. Taking into this "peak effect" account, we define

the final form of Model 1 as shown in Eq. (5.3):

$$\hat{\tau}_1(m) = \begin{cases} f_1(m) & (m^* \text{ does not exist}), \\ f_1(m) & (m^* \text{ exists and } m \leq m^*), \\ f_1(m^*) & (m^* \text{ exists and } m > m^*). \end{cases} \tag{5.3}$$

If Model 1 has its peak value at m^*, it has a constant value of $f_1(m^*)$ after the peak. For more detailed analysis on when f_1 has its peak, see Appendix A.

Model 2 : This model is a simple polynomial equation as shown in Eq. (5.4). The terms have the following meanings: base throughput (w_0), parallel processing gain (w_1), inter-worker communication overhead (w_2). All the weights are restricted to non-negative (*i.e.*, $w_i \geq 0, i = 0, 1, 2$), but we add a minus sign for w_2 to account for negative impact of the inter-worker communication.

$$f_2(m) = w_0 + w_1 \cdot m - w_2 \cdot m^2. \tag{5.4}$$

To find when f_2 gets to its peak, we take the derivative of f_2 in Eq. (5.4):

$$\frac{\partial f_2}{\partial m} = w_1 - 2w_2 \cdot m. \tag{5.5}$$

Now we find when the slope (5.5) is zero:

$$w_1 - 2w_2 \cdot m = 0, \tag{5.6}$$

$$m^* = m = \frac{w_1}{2w_2}. \tag{5.7}$$

Since f_2 is a quadratic function and concave downward (*i.e.*, $-w_2 < 0$), if $w_2 \neq 0$, f_2 has the

peak value of $f_2(m^*)$ as follows:

$$f_2(m^*) = w_0 + w_1 \cdot \left(\frac{w_1}{2w_2}\right) - w_2 \cdot \left(\frac{w_1}{2w_2}\right)^2. \tag{5.8}$$

Similar to Model 1, we define Model 2 as shown in Eq. (5.9):

$$\hat{\tau}_2(m) = \begin{cases} f_2(m) & (w_2 = 0), \\ f_2(m) & (w_2 \neq 0, \text{ and } m \leq m^*), \\ f_2(m^*) & (w_2 \neq 0 \text{ and } m > m^*). \end{cases} \tag{5.9}$$

5.3.4 Phase 1: VM Subset Selection

In this phase, we statistically determine the most effective subset of VM counts for training in terms of prediction error by exhaustive search. We first describe a method to select such subset and then show the results of selected VM subsets.

5.3.4.1 VM Subset Selection Method

In Phase 1, we perform the following steps to find the best VM subset.

Step 1. Collecting MST Samples First, we run a set of benchmarks $\mathcal{B} = \{b_1, b_2, ...\}$ on each VM count in $\mathcal{V} = \{m_1, m_2, ..., M_{\max}\}$ for K times per application. $\mathcal{V}$ contains the numbers of VM instances up to $M_{\max}$, for which the user needs to predict the MST (*e.g.*, $M_{\max} = 128$). We collect MST samples using the MST measurement method in Section 4.3.1. The k-th MST sample for application b_i with m VMs is denoted by $\tau^{(k)}(b_i; m)$.

Step 2. Select the Best VM Subset Let $\mathcal{S}$ be a subset of $\mathcal{V}$ used to train MST prediction Models in $\mathcal{T} = \{\hat{\tau}_1, \hat{\tau}_2, ...\}$. Our goal in this step is to find the best subset $\mathcal{S}^*$ which gives the lowest average prediction error over all benchmarks and models. To collect training samples at a low cost, we limit the maximum VM counts that we can have in $\mathcal{S}$ and also constraint the number of elements in $\mathcal{S}$. Let $\mathcal{V}_{\text{train}} = \{m \in \mathcal{V} \mid m \leq M_{\text{train}}\}$ be the

candidate set of training VM counts containing up to M_{train}, we create a collection of VM count subsets in $\mathcal{C}$ as follows:

$$\mathcal{C} = \{\mathcal{S} \subseteq P(\mathcal{V}_{\text{train}}) \mid e_1 \leq |S| \leq e_2\}, \tag{5.10}$$

where $P(\mathcal{V}_{\text{train}})$ is a power set of $\mathcal{V}_{\text{train}}$, and e_1 and e_2 are the minimum and maximum number of elements in $\mathcal{S}$, respectively. For each training dataset corresponding to $\mathcal{S} \in \mathcal{C}$, we create trained models $\hat{\tau}_j(\mathcal{S}, b_i; m)$ for all $\hat{\tau}_j \in \mathcal{T}$ and $b_i \in \mathcal{B}$. Using samples not used for training (*i.e.*, samples obtained for VM counts in $\mathcal{V} - \mathcal{S}$), we compute the Root Mean Square Error (RMSE) for each $\mathcal{S} \in \mathcal{C}$ over all applications and models as shown in Equations (5.11)-(5.12). Finally, we pick the best subset $\mathcal{S}^*$ which gives the lowest RMSE as shown in Eq. (5.13).

$$SSE(\mathcal{S}) = \sum_{b_i \in \mathcal{B}} \sum_{\hat{\tau}_j \in \mathcal{T}} \sum_{k=1}^{K} \sum_{m_l \in \mathcal{V}-\mathcal{S}} \left(\tau^{(k)}(b_i; m_l) - \hat{\tau}_j(\mathcal{S}, b_i; m_l)\right)^2 \tag{5.11}$$

$$\mathcal{S}^* = \underset{\mathcal{S} \in \mathcal{C}}{\text{argmin}}\, RMSE(\mathcal{S}) \tag{5.12}$$

$$= \underset{\mathcal{S} \in \mathcal{C}}{\text{argmin}} \sqrt{\frac{SSE(\mathcal{S})}{|\mathcal{B}| \cdot |\mathcal{T}| \cdot K \cdot |\mathcal{V} - \mathcal{S}|}}. \tag{5.13}$$

5.3.4.2 VM Subset Selection Results

We perform experiments for both Steps 1 and 2 to select the best S^* as follows.

Experimental Settings for Step 1 We run the following three *simple resource benchmarks* with Apache Storm version 0.10.1 [85] on Amazon EC2.

- *Word Count*: CPU intensive, typical word count for text inputs.
- *SOL* (*i.e.*, Speed-Of-Light): Network intensive, received events are transferred to the next processing units immediately without any processing.
- *Rolling Sort*: Memory intensive, received events are accumulated in a ring buffer and are sorted every 60 seconds.

We choose these three benchmarks since they have representative orthogonal resource usage patterns (*i.e.*, CPU, network, and memory intensive), and thus, the training samples obtained from these benchmarks can be generalizable to other applications. We collected MST samples as described in Section 4.3.1. We pre-load data in Kafka and let Storm pull the data as quickly as possible. After we started a benchmark, we waited until throughput converged or until 90 seconds had passed. Subsequently, we monitored traffic going from Kafka to Storm slave nodes for 20 seconds and computed the throughput as MST. For traffic monitoring, we modified tcpdump [111] to enable monitoring network traffic between nodes. Kafka offers various metrics including outgoing throughput through the Java Management Extensions interface; however, since it only provides one-minute moving average values and takes longer to converge, we decided to monitor raw network traffic using tcpdump. After the sampling, we shutdown the benchmark and waited for 10 seconds for the next round of sampling. We repeated this process 6 times (*i.e.*, $K = 6$) for all three simple resource benchmarks with the following VM counts up to $M_{\max} = 128$ m4.large VMs.

$$\mathcal{V} = \{1, 2, 3, ..., 7, 8, 12, 16, 24, 32, 48, 64, 80, 96, 128\}. \tag{5.14}$$

Experiment Settings for Step 2 We applied the VM subset selection method to the collected MST samples with the following configurations:

- Models: Models 1 and 2.
- Constraints on the number of elements in $\mathcal{S}$: $e_1 = 3, e_2 = 5$.
- Maximum number of VMs to include in the training data set: $M_{\text{train}} \in \{5, 6, 7, 8, 12, 16, 24, 32\}$.

We could increase M_{train} up to 128 VMs and potentially achieve good prediction results; however, this would incur high time and cost penalties. Therefore, we limited M_{train} to 32 VMs (= 25% of 128 VMs).

Selected VM Subsets Results Table 5.1 shows the best VM subset $\mathcal{S}^*$ and prediction error in the RMSE for each M_{train}. We get a reasonably low error of 0.08, considering the fact that the range of MST values is $[0, 1]$ after normalization. Since enumerated subsets created from a higher M_{train} contains all the elements of the ones created from a lower M_{train}, errors monotonically decrease as we increase M_{train}. For the evaluation of Phase 2, we assume that the user have enough budget to run up to 24 VMs, and we choose the best subset $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$ when $M_{\text{train}} = 24$ or 32.

Figure 5.2 shows prediction results with $\mathcal{S}^*$. Model 1 fits better than Model 2 for Word Count, whereas Model 2 fits better than Model 1 for Rolling Sort. For SOL, both models fit equally well. These results show that a single model is not sufficient to capture the performance of all applications. Therefore, we need to choose a better-fitting model for each application from Models 1 and 2.

Table 5.1. Best VM subsets $\mathcal{S}^*$ and prediction errors in RMSE for variable maximum VM counts ($M_{\text{train}} \in 5, ..., 32$).

M_{train}	Best subset: $\mathcal{S}^*$	RMSE
5	$\{2, 4, 5\}$	0.2864
6	$\{2, 4, 5\}$	0.2864
7	$\{2, 4, 5, 7\}$	0.1394
8	$\{2, 4, 5, 7\}$	0.1394
12	$\{2, 4, 5, 7\}$	0.1394
16	$\{6, 8, 16\}$	0.1263
24	$\{3, 4, 6, 8, 24\}$	0.0813
32	$\{3, 4, 6, 8, 24\}$	0.0813

5.3.5 Phase 2: Model Training & Selection

In Phase 2, we first take a new test application from the user, which is subject to performance prediction. We obtain training MST samples only for a selected subset $\mathcal{S}^*$ from Phase 1, and we train both Models 1 and 2 using the the same set of training samples. Followed by the model training, we compare trained Models 1 and 2 using some extra validation samples to estimate which model is more accurate. Then, we select the model

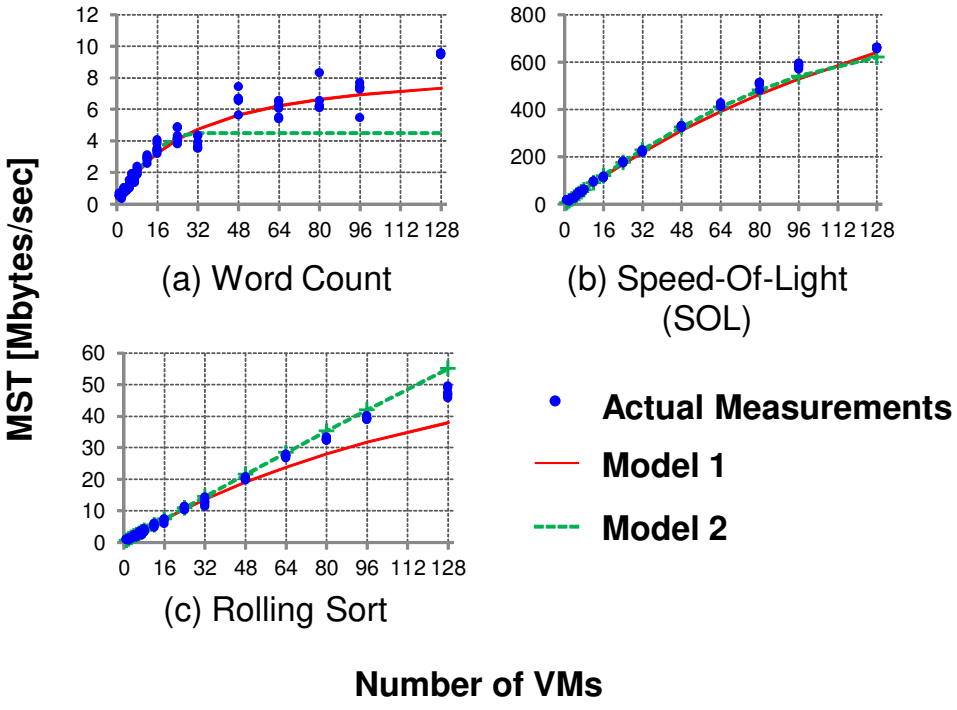


Figure 5.2. MST prediction results using the best VMs subset: $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$. X-axis: number of VMs. Y-axis: MST [Mbytes/sec].

with the lower validation error as the final output of Phase 2.

Validation Data Points Selection In Figure 5.2(a) and Figure 5.2(c), we can see the prediction results beyond the VMs subset $\mathcal{S}^*$ largely vary between Models 1 and 2. Both models' prediction results are very close up to $m = 32$, but they start to diverge after that. Thus, we collect extra validation MST samples outside $\mathcal{S}^*$ to decide which model to choose. We look at VM counts where their MST values from both models start to diverge.

If Model 1 $\hat{\tau}_1(m)$ and Model 2 $\hat{\tau}_2(m)$ intersect each other at one or more positive real numbers of m, we take the ceiling of these numbers and put them in a set of intersecting VM counts: $\mathcal{I} = \{m_1, m_2, ...\}$, $m_i < m_{i+1}$. If the models do not intersect, $\mathcal{I} = \emptyset$. We get these intersecting points analytically if possible. If not, we can numerically estimate these intersecting points by plugging in multiple integers for m. Assuming both models are well trained up to the maximum VM counts in $\mathcal{S}^*$, we should validate models with VMs

larger than any VM counts in $\mathcal{S}^*$. Also, to make sure there is some discrepancy between the two models, we introduce a discrepancy threshold κ. Based on these ideas, we determine validation VM counts as shown in Algorithm 6.

```
1  input : I: set of intersecting VM counts, S*: training VM counts, κ:
           discrepancy threshold
   output: S_val: set of VM counts for validation
2  // Filter I with the max VM counts in S*
3  if I ≠ ∅ then
4  |  I' = {m_i | m_i > max S*, m_i ∈ I};
5  end
6  else
7  |  I' = {max S* + 1};
8  end
9  S_val = ∅;
10 // Search for divergent VMs
11 for i = 1, ..., |I'| do
12 |  // m_i:  i-th element of I'
13 |  m_start = m_i;
14 |  if i = |I'| then
15 |  |  m_end = M_max;
16 |  end
17 |  else
18 |  |  // m_{i+1}:  (i+1)-th element of I'
19 |  |  m_end = m_{i+1} − 1;
20 |  end
21 |  for m = m_start, ..., m_end do
22 |  |  d(m) =|τ̂_1(m) − τ̂_2(m)| / min(τ̂_1(m), τ̂_2(m));
23 |  |  if d(m) > κ then
24 |  |  |  S_val = S_val ∪ {m};
25 |  |  |  break;
26 |  |  end
27 |  end
28 end
29 return S_val;
```

Algorithm 6. Selection of validation VM counts.

First, we filter the intersecting VMs set $\mathcal{I}$ with the maximum VM counts in $\mathcal{S}^*$ and store the filtered VMs set in $\mathcal{I}'$ (Line 4). Even if $\mathcal{I}$ is empty, we still want to search for the divergent VM counts starting from $\max \mathcal{S}^* + 1$ (Line 7). To search for one divergent

VM count per element in $\mathcal{I}'$, we loop through $\mathcal{I}'$ (Line 11–28). As shown in Eq. (5.15), we compute the following discrepancy ratio between the predicted values of Models 1 and 2 for VM counts m (Line 22).

$$d(m) = \frac{|\hat{\tau}_1(m) - \hat{\tau}_2(m)|}{\min(\hat{\tau}_1(m), \hat{\tau}_2(m))}. \tag{5.15}$$

If it is greater than the discrepancy threshold κ, we assume that there is enough discrepancy between the two models and add corresponding m to the validation set $\mathcal{S}_{\text{val}}$ (Line 24).

Model Selection Through Validation Error Once we get $\mathcal{S}_{\text{val}}$, we obtain MST values for a set of VM counts in $\mathcal{S}_{\text{val}}$ and compute validation errors in the RMSE for both models. Finally, we select the model with the lowest validation error. If $\mathcal{S}_{\text{val}}$ is empty, which means the discrepancy between the two models never goes above κ, we select the model with the lowest training error.

5.4 Evaluation of MST Prediction

We evaluate the MST prediction models that are trained and selected by our framework. We used the following *typical use-case benchmarks* from Intel Storm Benchmarks [106].

- *Grep*: Match a given regular expression with text inputs and count the number of matched lines of texts.
- *Rolling Count*: Count the number of words and output the word counts every 60 seconds.
- *Unique Visitor*: Count the number of unique visitors to websites from web access logs.
- *Page View*: Count the number of page views per website from web access logs.
- *Data Clean*: Filter the logs with 200 HTTP status code from web access logs.

In Figure 5.3, we plot actual and predicted MST values as functions of VM counts. Actual measurements of MST are shown in dots. Predicted values for Model 1 are shown in

Table 5.2. Weights of Models 1 and 2 after training.

Benchmark	Model 1				Model 2		
	w_0	w_1	w_2	w_3	w_0	w_1	w_2
Grep	0.01617	1.04913	0	0	0.63117	0.74233	0.00063
Rolling Count	0.11748	2.75500	0	0	0.16593	0.27683	0.00322
Unique Visitor	0.02050	1.67805	0	0	0.01615	0.56212	0.00160
Page View	0.03512	1.61983	0	0	0.27180	0.49439	0.00089
Data Clean	0.11039	1.62948	0	0.00004	0.12286	0.49693	0.01233
VHT	0.01958	0	0.00005	0	50.69306	0	0.00394

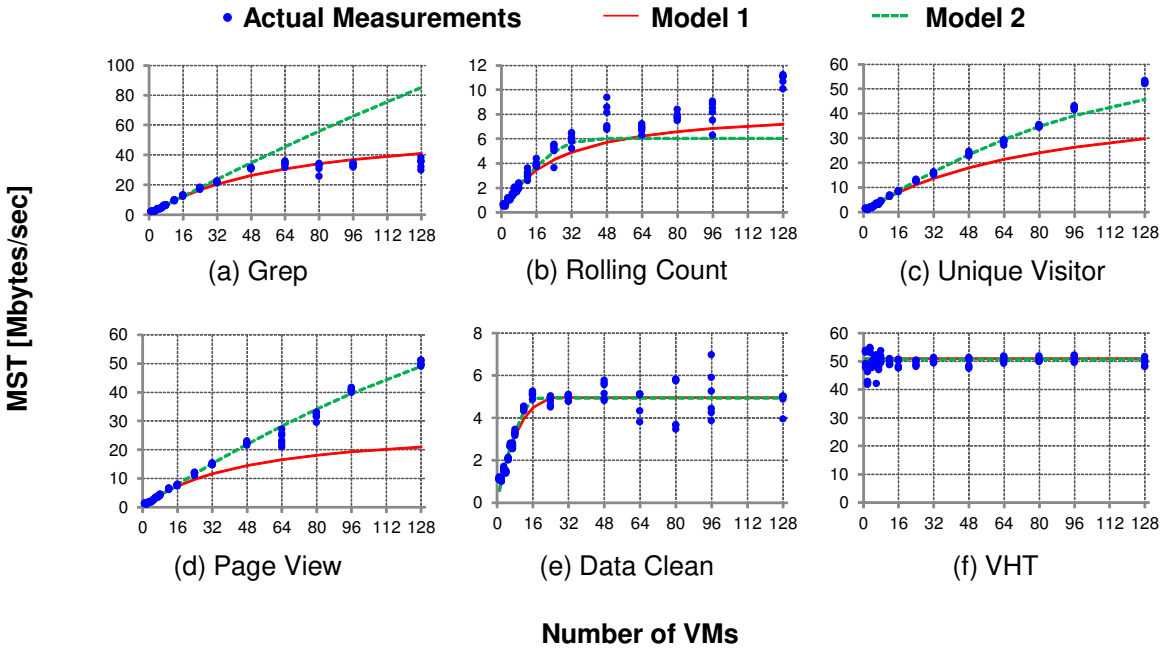


Figure 5.3. MST prediction results for typical use-case benchmarks: (a) Grep, (b) Rolling Count, (c) Unique Visitor, (d) Page View, and (e) Data Clean, and a machine learning application: (f) VHT, using $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$ for Models 1 and 2. X-axis: number of VMs. Y-axis: MST [Mbytes/sec].

solid lines and predicted values for Model 2 are shown in dotted lines. From the figure, we can see that the actual MST values either hit a bottleneck (*i.e.*, Grep, Rolling Count, Data Clean, and VHT) or show linear improvements (*i.e.*, Unique Visitor and Page View). Model 1 captures the bottlenecks for Grep and Rolling Count relatively well. Model 2 is trained almost as a linear function (*i.e.*, w_2 is at most 0.0016) for Unique Visitor and Page View, and therefore, it captures their linear behavior well. However, Model 2 fails to capture the

non-linear behavior of Grep. Both models are equally well fitted to Data Clean and VHT.

Figure 5.4 shows the validation error in RMSE for model selection. To obtain these results, we used the following validation samples selected by Algorithm 6: 26 VMs for Grep, 25 and 72 VMs for Rolling Count, 25 VMs for Unique Visitor, 25 VMs for Page View. For Data Clean and VHT, since both models produced very close predicted MST values, there were no discrepancies larger than the $\delta = 0.10$ threshold. Thus, the models for these two benchmarks were compared using training errors. As the result of validation, Model 1 is chosen for Grep, Rolling Count, and Model 2 is chosen for the rest of benchmarks. Looking

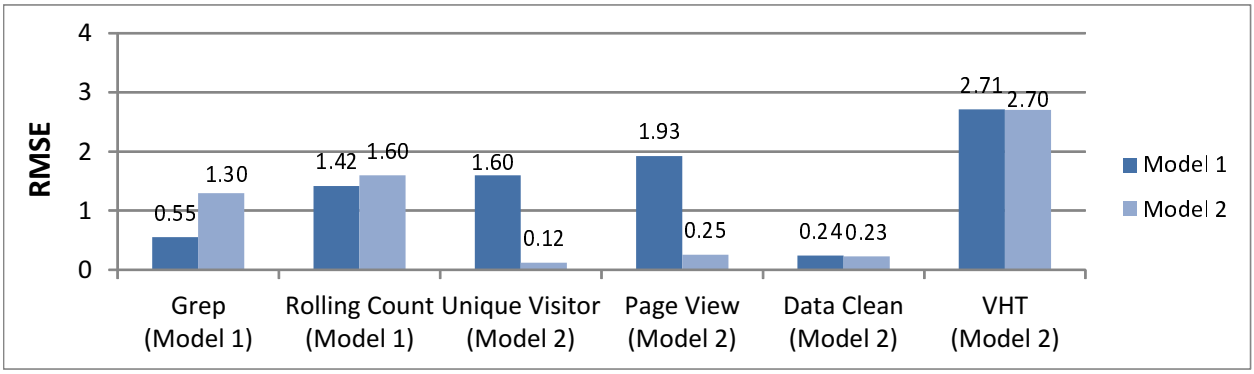


Figure 5.4. Validation error (RMSE) and selected models for the typical use-case benchmarks. Models are trained with $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$.

at Figure 5.3, these model selection results are visually convincing. Rolling Count has similar RMSE errors for Model 1 and Model 2. However, due to the second validation sample from 72 VMs, Model 2 is selected. For Data Clean and VHT, both models are visually very close and the training errors are almost identical.

Figure 5.5 shows prediction error in the Mean Absolute Percentage Error (MAPE) by the selected models. The MAPE is defined as:

$$MAPE = 100 \cdot \frac{1}{n} \sum_{i=1}^{n} \frac{|t^{(i)} - p^{(i)}|}{|t^{(i)}|}, \tag{5.16}$$

where n is the number of samples, $t^{(i)}$ is the i-th sample's true value, and $p^{(i)}$ is the i-th sample's predicted value. The figure plots the MAPE error computed from the selected

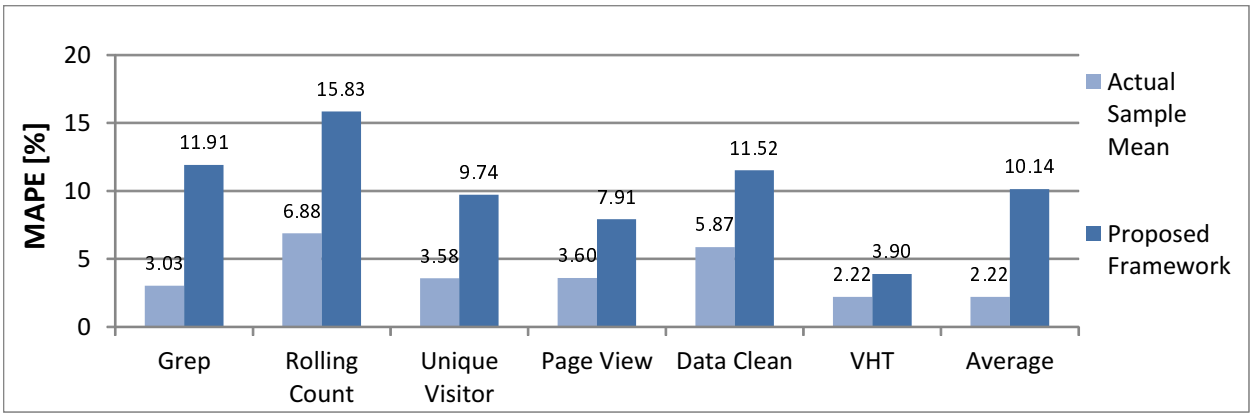


Figure 5.5. Prediction error (MAPE) for the typical use-case benchmarks using mean value of actual MST samples and prediction made by the proposed framework.

models and MAPE error computed from the mean value of actual MST samples. Since the mean value is known to minimize the sum of squared error $\sum_{i=1}^{n}(t^{(i)} - p^{(i)})^2$, even though it does not guarantee to minimize the MAPE, it shows low error in MAPE. Overall, the MAPE error for our prediction framework is up to 15.83% (average: 10.14%). Since there is some variance in the actual MST samples, even the prediction made from the mean has the MAPE error up to 6.88% (average: 2.22%).

5.5 Discussion

In this section, we discuss the results from the experiments we have done in Sections 5.4.

Likely Cause of the Bottlenecks From the experiments in Section 5.4, the scalability for Grep, Rolling Count, Data Clean, and VHT is limited. The reason seems to be load imbalance between workers. For the Grep benchmark, to compute the total count of matched patterns, the global counter is incremented by a single thread, and MST is bounded by the performance of that single thread. For Rolling Count, the bottleneck may be caused by imbalanced distributions of words. Just as the map-reduce programming framework [68], once the first layer of processing units splits texts into words, the next processing units are

determined by the hash value of a word. Thus, depending on the distribution of words, some nodes are more loaded than other nodes. Similar to Grep, maximum performance is bounded by the nodes that are assigned frequently appearing words. The Data Clean's performance limitation seems to be caused by a similar reason to Grep: URLs filtered by the 200 status code go to the same node. Since the objective of our prediction framework is to accurately predict MST for larger numbers of VMs, we are not concerned about application-specific bottlenecks; however, clarifying the mechanism behind these bottlenecks could help improve the accuracy of prediction models in future.

Time Complexity and Generalization of the Proposed Framework Due to the exhaustive search, the time complexity of the subset search method presented in Section 5.3.4.1 is $O(|\mathcal{V}_{\text{train}}|^{e_2} \cdot |\mathcal{B}|)$. With an implementation using octave, the search with $M_{\text{train}} = 32$ took only a few minutes to find the best VM subset $\mathcal{S}^*$ on a laptop PC with Intel Core i5 CPU. As we increase the size of $\mathcal{V}_{\text{train}}$ through M_{train} and/or e_2, the runtime is expected to grow exponentially. One way to keep the runtime manageable is to use random sampling to identify $\mathcal{S}^*$, rather than performing an exhaustive search over all subsets. We should be able to find the right balance between the runtime and the likelihood of finding the optimal subset. Currently when we determine data points for validation in Algorithm 6, we assume the framework only uses the two models we presented in Section 5.3.3. We can generalize this algorithm to support n models, for example, by defining the average pairwise discrepancies between n models as follows:

$$d(m) = \frac{1}{\binom{n}{2}} \cdot \sum_{i,j} \frac{|\hat{\tau}_i(m) - \hat{\tau}_j(m)|}{\min(\hat{\tau}_i(m), \hat{\tau}_j(m))}. \tag{5.17}$$

With this change, the entire framework will be compatible to n models.

5.6 Summary

In this chapter, we have presented a framework to predict the maximum sustainable throughput (MST) for cloud-based stream processing applications. We identified a common data processing environment used by modern stream processing systems and presented two models for MST prediction. We statistically determine the best subset of VM counts in terms of prediction error to collect training samples. For each new application, we train the framework models using this subset. The framework takes several trained models and selects the model that is expected to predict MST values for the target application with the lowest error. We evaluated our framework on streaming applications in Apache Storm, using up to 128 VMs. Experiments showed that our framework can predict MST with up to 10.14% average prediction error.

In future work, we plan to apply the proposed prediction framework to other stream data processing engines such as Flink to confirm the applicability of our approach. Other interesting future directions include online learning to improve the performance prediction model accuracy over time, the use of meta-algorithms such as ensemble learning to construct a prediction model from multiple weak models, and even larger-scale performance simulation using a cloud environment simulator such as CloudSim [112].

CHAPTER 6
UNCERTAINTY-AWARE ELASTIC VIRTUAL MACHINE SCHEDULING

6.1 Introduction

Challenges to accurate performance prediction arise from factors such as skewed data distribution [113] and resource contention with other applications sharing the same VMs [114]. It has been shown that if a VM scheduler incorporates an application's performance variability, it can achieve higher QoS objective satisfaction compared to a scheduler that is agnostic of the performance variability [41], [115]. It is also common to use manual over-provisioning to compensate for potential inaccuracies in performance models [116]–[118].

Workload prediction, such as ARIMA [45], has been used in elastic VM scheduling [47], [116]–[119]. The VM schedulers in these works select the number of VMs based on the prediction value with the minimum expected error, but they do not consider the variability of workloads. Even though real-world time series often have recurring patterns, they can be affected by irregular events, resulting in larger variance. Despite the evidence of the impact of uncertainties in both application performance and future workload, existing works in VM scheduling have not addressed these uncertainties in a holistic manner.

This work proposes a framework for proactive elastic VM scheduling for stream processing systems in cloud computing environments that explicitly models uncertainties in 1) VM performance, 2) application performance, and 3) workload variability. We quantify the uncertainty from #1 and #2 through the variance of an application performance model and

Portions of this chapter are to appear in: Shigeru Imai, Stacy Patterson, and Carlos A. Varela, "Uncertainty-aware elastic virtual machine scheduling for stream processing systems" in *Proc. IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid 18)*, 2018.

#3 through the variance of a workload prediction model. By incorporating uncertainties from both prediction models into scaling decisions, our elastic stream processing framework achieves high QoS objective satisfaction rates without human intervention at far lower cost than a static over-provisioning approach. We focus on the application performance metric of *Maximum Sustainable Throughput* (MST) [118]. MST represents the maximum data input rate a stream processing application can process without backlogging any input data. Using workload and MST prediction models, our VM scheduler proactively schedules VMs to satisfy the throughput-QoS condition (4.1): to always maintain an MST that is greater than the data rate of input workloads. Through simulations following real-world workload distributions, we show that our scheduling framework achieves 98.62% of QoS satisfaction rate, at up to 48% lower cost compared to static scheduling that covers the peak workload.

The rest of the chapter is organized as follows. In Section 6.2, we present the formulation of VM scheduling problem we solve. Section 6.3 presents the proposed scheduling techniques. Section 6.4 describes our workloads and evaluation setup, and Section 6.5 shows the evaluation results. We present related work in Section 6.6 and conclude the paper in Section 6.7.

6.2 Formulation of VM Scheduling Problem

We model the workload changes $\lambda(t)$ over time as a discrete sequence, with timesteps $t = 1, 2, \ldots$. Every scheduling cycle of C timesteps, we make a scaling decision whether to allocate or shut down VMs. We use $s = 1, 2, \ldots$ to index scheduling cycles. A scheduling cycle s corresponds to $t = (s-1)C + 1, (s-1)C + 2, ..., sC$ timesteps. This means that we have a constant VM configuration for the duration of C timesteps, whereas the workload changes every timestep.

When we start a new VM, we assume it takes $U(< C)$ timesteps until it becomes available. For example, we use $C = 10$ minutes and $U = 2$ minutes.

The problem is to find a cost-minimal sequence of VM allocations such that the MST

is greater than or equal to the input data rate $\lambda(t)$ for all t. Thus, for each scheduling cycle s, we must determine $m(s)$ such that,

$$\bar{\lambda} = \max_{t=(s-1)C+1,\ldots,sC} \lambda(t), \tag{6.1}$$

$$m(s) = \operatorname*{argmin}_{m\in\{1,2,\ldots,M_{\max}\}} \tau(m) \geq \bar{\lambda}, \tag{6.2}$$

where $M_{\max}$ is the maximum number of VMs available for scaling.

6.3 VM Scheduling Techniques

In this section, we describe our methods for estimating the number of VMs the scheduler should allocate at each update interval. A novel feature of our approach is that we explicitly incorporate uncertainty into our models; this includes uncertainty in the estimation process as well as uncertainty about the change in workloads. By incorporating uncertainty, our framework is better able to meet QoS requirements, as demonstrated in Section 6.5.

We first present our baseline VM scheduling approach. We then describe three methods to incorporate uncertainty. These methods can be used independently or in combination. We evaluate the effectiveness of these policies in Section 6.5.

6.3.1 Baseline MST

This technique schedules VMs based on a MST model without workload forecasting. We are given an MST prediction model $\hat{\tau}(m)$ trained offline and selected by the method presented in Chapter 5 (*i.e.*, $\hat{\tau}(m)$ is either Model 1 or Model 2).

At the beginning of every scheduling cycle s, we make a scheduling decision at time $t = (s-1)C + 1$. We estimate the minimum number of VMs to cover the given workload $\lambda(t)$ as follows:

$$m(t) = \operatorname*{argmin}_{m\in\{1,2,\ldots,M_{\max}\}} \hat{\tau}(m) \geq \lambda(t). \tag{6.3}$$

When scaling up, new VMs will become ready after the VM startup delay of U timesteps.

On the other hand, when scaling down, VMs will be immediately shut down. Since we do not forecast future workload at all in this baseline technique, this method is purely reactive.

6.3.2 Online Model Learning

Online learning can be used to update MST models as we obtain new training samples. These training samples can reduce uncertainty in the performance models that are trained offline. We assume the given MST model $\hat{\tau}(m)$ is trained with the following training data set that contains n sample pairs:

$$\mathcal{D}_{\text{train}} = \{(m^{(1)}, \tau^{(1)}), (m^{(2)}, \tau^{(2)},), ..., (m^{(n)}, \tau^{(n)})\}, \tag{6.4}$$

where $(m^{(i)}, \tau^{(i)})$ is the i-th training sample pair of VM count and corresponding MST value. By definition, MST can be obtained when excessive workloads saturate the system capacity. Thus, when the scheduler under-provisions VMs for the application workload (*i.e.*, when the throughput-QoS objective is violated), the application monitor can obtain a new training sample pair (m, τ). This sample pair is added to the training data set $\mathcal{D}_{\text{train}}$ to re-train the selected MST model. Note that the formulation (6.3) does not change even if we apply online learning to the MST model $\hat{\tau}(m)$.

6.3.3 Uncertainty-Awareness for MST

We quantify uncertainty from VM and application performance through the variance of an MST model. Let $\hat{\tau}(m)$ be the MST model trained using linear regression with $\mathcal{D}_{\text{train}}$ shown in Eq. (6.4). For the i-th pair $(m^{(i)}, \tau^{(i)})$ in $\mathcal{D}_{\text{train}}$, we assume the following relationship holds between the measured MST $\tau^{(i)}$ and predicted MST values $\hat{\tau}(m^{(i)})$ for $i = 1, 2, ..., n$, where n is the size of the training data set:

$$\tau^{(i)} = \hat{\tau}(m^{(i)}) + \varepsilon^{(i)}, \tag{6.5}$$

where $\varepsilon^{(i)} \sim \mathcal{N}(0, \sigma_\tau^2)$. We can estimate the true variance σ_τ^2 by the following $\hat{\sigma}_\tau$ [45]:

$$\hat{\sigma}_\tau^2 = \frac{\sum_{i=1}^{n} (\tau^{(i)} - \hat{\tau}(m^{(i)}))^2}{n - k}, \tag{6.6}$$

where k is the number of regression coefficients in the MST model (*i.e.*, $k = 4$ for Model 1 and $k = 3$ for Model 2).

Based on Eq. (6.5), we assume that MST follows the normal distribution:

$$\tau(m) \sim \mathcal{N}(\hat{\tau}(m), \hat{\sigma}_\tau^2). \tag{6.7}$$

Given an input data rate $\lambda(t)$ at time t, we introduce a new random variable $\delta = \tau(m) - \lambda(t)$. Since $\lambda(t)$ is constant for a time period of $[t, t+1)$ (*i.e.*, one minute in this work), δ follows $\mathcal{N}(\hat{\tau}(m) - \lambda(t), \hat{\sigma}_\tau^2)$. We can then estimate the probability of $\delta \geq 0$:

$$\Pr[\delta \geq 0] \;\; = \;\; \int_0^\infty f(\delta \mid \hat{\tau}(m) - \lambda(t), \hat{\sigma}_\tau^2)\, d\delta,$$

where $f(\delta | \hat{\tau}(m) - \lambda(t), \hat{\sigma}_\tau^2)$ is the probability density function for the normal distribution (6.7). Figure 6.1 shows this probability density function. In the figure, the shaded area corresponds to the value of $\Pr[\tau(m) - \lambda(t) \geq 0]$. We estimate the minimum number of VMs that are

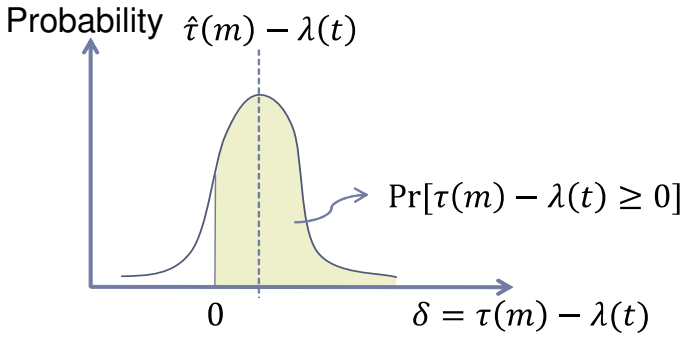


Figure 6.1. Probability density function for normal distribution $\mathcal{N}(\hat{\tau}(m) - \lambda(t), \hat{\sigma}_\tau^2)$. The shaded area corresponds to the value of $\Pr[\tau(m) - \lambda(t) \geq 0]$.

required to satisfy the throughput-QoS objective $\tau(m) \geq \lambda(t)$ as follows:

$$m(t) = \underset{m \in \{1,2,...,M_{\max}\}}{\operatorname{argmin}} \Pr[\tau(m) - \lambda(t) \geq 0] \geq \rho, \tag{6.8}$$

where $\rho \in [0.5, 1.0)$ is a *QoS satisfaction target*.

In this formulation, uncertainty is quantified in terms of variance, as shown in Eq. (6.6), and larger variance leads to require a larger number of VMs to satisfy the constraint in Eq. (6.8). When $\rho \approx 1.0$, we expect δ to be large to prevent QoS violations, which means that the mean of the normal distribution in Figure 6.1 is far away from zero. When $\rho = 0.5$, however, we expect δ to be very close to zero. When $\rho = 0.5$, this is equivalent to finding the smallest $\hat{m}$ that satisfies $\hat{\tau}(m) \geq \lambda(t)$ (*i.e.*, it is equivalent to the baseline MST in Section 6.3.1).

6.3.4 Uncertainty-Awareness for Workload Forecasting

In this section, we add uncertainty-awareness for workload forecasting to the baseline MST shown in Section 6.3.1. We make multiple forecasts into the next scheduling cycle and determine the highest expected workload we should cover to satisfy the QoS objective.

6.3.4.1 Workload Forecasting

We use Auto-regressive-moving-average (ARMA) models [45] to forecast future workloads. An ARMA(p, q) model consists of an auto-regressive (AR) term of order p and a moving-average (MA) term of order q as follows:

$$\begin{aligned} \lambda(t) &= \phi_1\lambda(t-1) + \cdots + \phi_p\lambda(t-p) + \\ &\quad \varepsilon(t) + \theta_1\varepsilon(t-1) + \cdots + \theta_q\varepsilon(t-q), \end{aligned} \tag{6.9}$$

where $\lambda(t), \lambda(t-1), ..., \lambda(t-p)$ is a series of observed input data rates; $\varepsilon(t), \varepsilon(t-1), ..., \varepsilon(t-q)$ is a series of error terms $\varepsilon(t) \sim \mathcal{N}(0, \sigma_\lambda^2)$; and $\phi = (\phi_1, ..., \phi_p)$ and $\theta = (\theta_1, ..., \theta_q)$ are model

parameters. Suppose we have a series of n input data rates $\lambda(t), \lambda(t-1), ..., \lambda(t-n-1)$; maximum likelihood estimation is used to estimate ϕ and θ. Once we estimate ϕ and θ, we calculate a k-step ahead forecast $\hat{\lambda}(t+k)$ by recursively applying Eq. (6.9) as follows:

$$\begin{aligned} \hat{\lambda}(t+k) &= \phi_1\lambda(t+k-1) + \cdots + \phi_p\lambda(t+k-p) + \\ &\quad \varepsilon(t+k) + \theta_1\varepsilon(t+k-1) + \cdots + \theta_q\varepsilon(t+k-q). \end{aligned} \tag{6.10}$$

An estimate of σ_λ^2 is given by:

$$\hat{\sigma}_\lambda^2 = \frac{\phi_1^2 + \cdots + \phi_p^2 + \theta_1^2 + \cdots + \theta_q^2}{n}. \tag{6.11}$$

Further, we can define a new set of parameters $\psi = (\psi_1, \psi_2, ..., \psi_j)$ using ϕ and θ as follows:

$$\begin{aligned} \psi_1 &= \phi_1 - \theta_1 \\ \psi_2 &= \phi_1\psi_1 + \phi_2 - \theta_2 \\ &\vdots \\ \psi_j &= \phi_1\psi_{j-1} + \cdots + \phi_p\psi_{j-p} - \theta_j, \end{aligned} \tag{6.12}$$

where $\theta_j = 0$ for $j > q$. Using $\hat{\sigma}_\lambda^2$ and ψ, we can estimate the variance for the k-step ahead forecast $\hat{\sigma}_{\lambda|t+k}^2$ as follows [45]:

$$\hat{\sigma}_{\lambda|t+k}^2 = \left\{1 + \sum_{j=1}^{k} \psi_j^2\right\}\hat{\sigma}_\lambda^2. \tag{6.13}$$

6.3.4.2 Highest Workload Estimation

Figure 6.2 shows how we determine the highest expected workload in the next scheduling cycle. First, we forecast input data rates and variances for the next scheduling cycle in $\hat{\lambda}(t+k)$ and $\hat{\sigma}_{\lambda|t+k}^2$ for $k \in [1, S)$ using Eqs. (6.10) and (6.13), respectively. We then

determine the timestep h with the highest expected workload at $t + h$:

$$h = \underset{k \in [1,S)}{\operatorname{argmax}} \ \hat{\lambda}(t+k) + 2\hat{\sigma}_{\lambda|t+k}. \tag{6.14}$$

We add $2\hat{\sigma}_{\lambda|t+k}$ to the forecasted value in Eq. (6.14) to yield a conservative VM allocation.

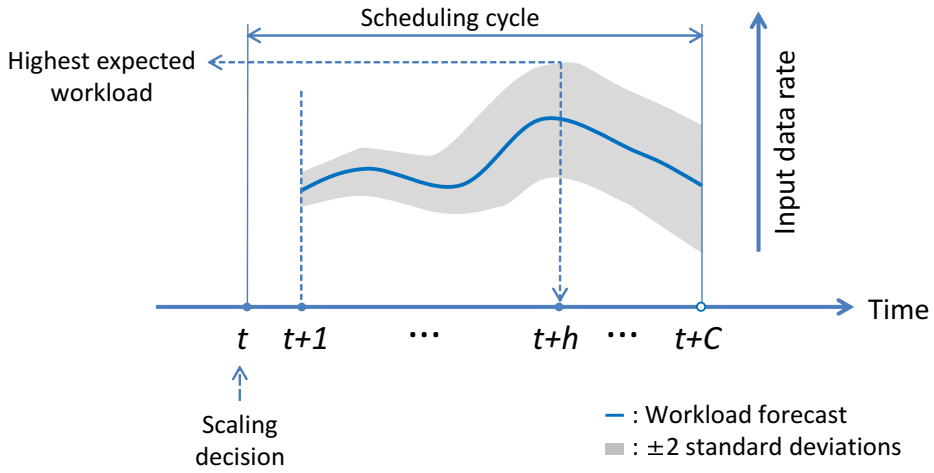


Figure 6.2. Finding the highest expected workload in the scheduling cycle $[t, t + C)$.

6.3.4.3 VM Scheduling

Once we determine h from Eq. (6.14), we then estimate the minimum number of VMs needed to cover the workload $\hat{\lambda}(t + h)$ using the method described in Section 6.3.3 to incorporate the predicted workload variance. Assuming the h-step ahead forecast follows the normal distribution:

$$\lambda(t+h) \sim \mathcal{N}(\hat{\lambda}(t+h), \hat{\sigma}^2_{\lambda|t+h}), \tag{6.15}$$

the difference between MST and the $\lambda(t+h)$, $\delta = \tau(m) - \lambda(t+h)$, is also expected to follow a normal distribution:

$$\delta \sim \mathcal{N}(\hat{\tau}(m) - \hat{\lambda}(t+h), \hat{\sigma}^2_{\lambda|t+h}). \tag{6.16}$$

Similar to Eq. (6.8), we estimate the probability of $\delta \geq 0$:

$$\Pr[\delta \geq 0] = \int_0^\infty f(\delta|\hat{\tau}(m) - \hat{\lambda}(t+h), \hat{\sigma}^2_{\lambda|t+h})\, d\delta. \tag{6.17}$$

We estimate the minimum number of VMs $\hat{m}$ required to satisfy the throughput-QoS objective $\tau(m) \geq \lambda(t+h)$ as:

$$m(t) = \underset{m \in \{1,2,\ldots,M_{\max}\}}{\operatorname{argmin}} \Pr[\tau(m) - \lambda(t+h) \geq 0] \geq \rho. \tag{6.18}$$

6.3.5 Uncertainty-Awareness for both MST and Workload Forecasting

We now show how to incorporate uncertainties from both MST and workload forecasting into the VM scheduling. Following the method in Section 6.3.4.2, we first determine the future timestep $t+h$ that is expected to give the highest workload value in the next scheduling cycle. Assuming that the MST and workloads are independent, the difference between MST and the h-step ahead workload, $\delta = \tau(m) - \lambda(t+h)$, also follows a normal distribution:

$$\delta \sim \mathcal{N}(\hat{\tau}(m) - \hat{\lambda}_{t+h}, \hat{\sigma}^2_\tau + \hat{\sigma}^2_{\lambda|t+h}). \tag{6.19}$$

Similar to Eqs. (6.8) and (6.17), we estimate the probability of $\delta \geq 0$ as follows:

$$\Pr[\delta > 0] = \int_0^\infty f(\delta|\hat{\tau}(m) - \hat{\lambda}(t+h), \hat{\sigma}^2_\tau + \hat{\sigma}^2_{\lambda|t+h})\, d\delta,$$

where $f(\delta \mid \hat{\tau}(m) - \hat{\lambda}(t+k), \hat{\sigma}^2_\tau + \hat{\sigma}^2_{\lambda|t+k})$ is the probability density function for the normal distribution (6.19). We then estimate the minimum number of VMs $\hat{m}$ required to satisfy the throughput-QoS objective $\tau(m) \geq \lambda(t+h)$ as:

$$m(t) = \underset{m \in \{1,2,\ldots,M_{\max}\}}{\operatorname{argmin}} \Pr[\tau(m) - \lambda(t+h) \geq 0] \geq \rho. \tag{6.20}$$

6.4 Evaluation Setup

In this section, we describe the test applications and workloads used in our evaluations. We also describe the training phase and parameterizations for our application performance and workload models.

6.4.1 Test Applications and Workloads

We evaluate our proposed framework using the following applications:

- *Grep, Rolling Count, Unique Visitor, Page View*, and *Data Clean*: Typical use-case benchmarks from Intel Storm Benchmarks [106].
- *Vertical Hoeffding Tree (VHT)*: Stream machine learning application from Apache SAMOA [120].
- *Rolling Hashtag Count*: Typical word count-like application which counts the number of Twitter hashtags. It outputs hashtag counts every five seconds for the results computed in the 60 seconds moving window.
- *Rolling Flight Distances*: This application computes distances between all flights in the near future. It outputs flight pairs which distance is less than a threshold every five seconds for the results computed in the 60 second moving window.

We use the three workloads as shown in Figure 6.3. All workloads are time series of data rates in Kbytes/sec over one week of time. These workloads were created as follows:

- *World Cup 98* in Figure 6.3(a) was created from the FIFA World Cup 1998 website access logs [121] (duration: 6/29/1998-7/5/1998, time in UTC). The workload spikes corresponds to the days when there were popular matches. For example, there were two matches of round of 16 on June 30 and two quarter finals on July 3.
- *Tweets* in Figure 6.3(b) was created from tweets downloaded from twitter.com using Twitter APIs [122] (duration: 4/10/2016 to 4/16/2016, time in EDT). Downloaded

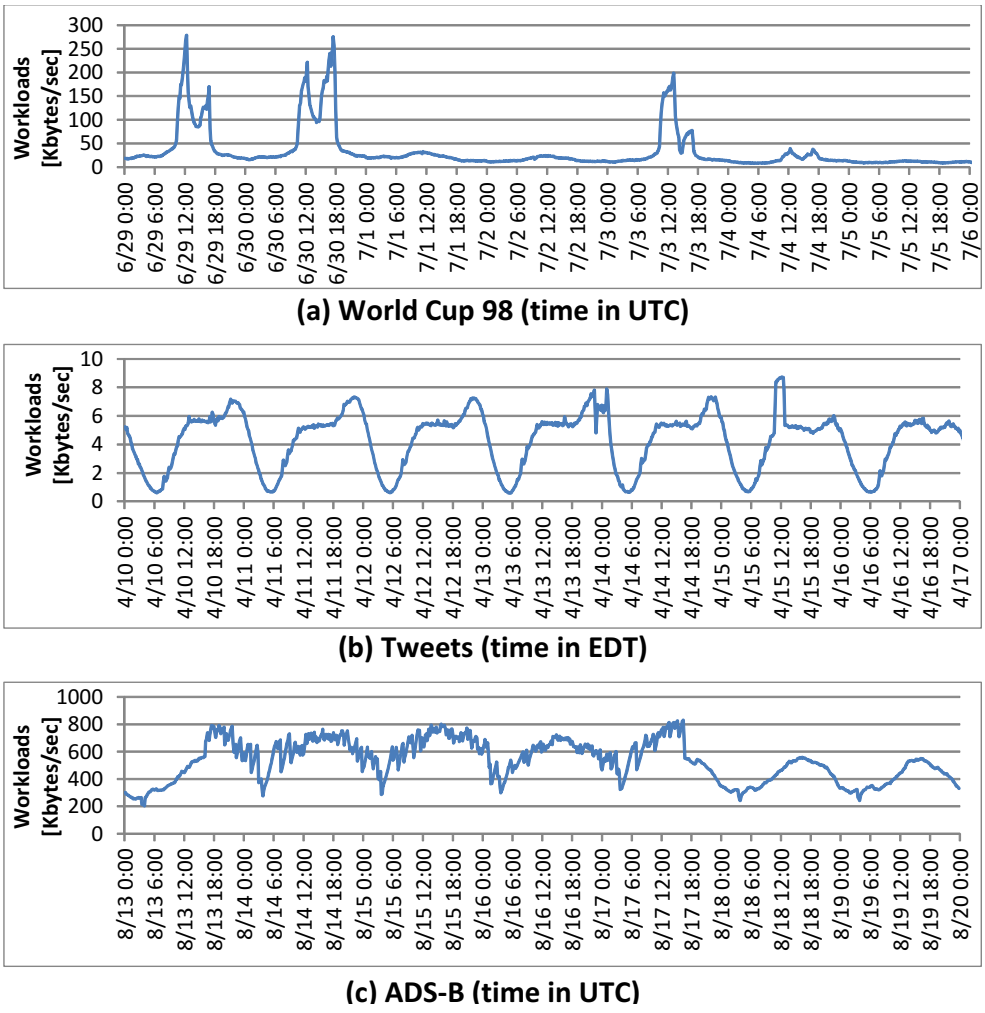


Figure 6.3. Workloads used for evaluation. Each workload has one week of data: (a) World Cup 98 (6/29/1998-7/5/1998, time in UTC), (b) Tweets (4/10/2016-4/16/2016, time in EDT), and (c) ADS-B (8/13/2017-8/19/2017, time in UTC).

tweets were those in English from the U.S. contiguous 48 states. Even though we were only allowed to obtain a fraction of all tweets from that time period, the time series shows clear daily recurring patterns.

- *ADS-B* in Figure 6.3(c) was created from the Automatic Dependent Surveillance-Broadcast (ADS-B) data from ADS-B Exchange [123] (duration: 8/13/2017-8/19/2017, time in UTC). The data are provided by worldwide community of volunteers and contains flight data from all over the world. Just as the Tweets workload, it also shows

daily recurring patterns.

6.4.2 Offline MST Model Training

We obtain MST samples as described in Section 4.3.1. All the test applications re implemented with Apache Storm v0.10.0 and are configured to process stream data from Kafka v2.11. Both Storm and Kafka are running on Amazon EC2, where we use the m4.large VM instance type for all the worker nodes of Storm. As we have explained in Section 6.3.1, we collect MST samples for each test application for the pre-determined set of VMs counts $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$ to predict MST for up to 128 VMs. We then train both Models 1 and 2 and select the better-fitting model for each application. For Rolling Flight Distances, we use the following $\mathcal{S}_{\text{flight}} = \{1, 2, 3, 5, 6, 8, 9, 11, 12, 14\}$ since it is designed to process the actual ADS-B workloads in Figure 6.3(c) and the MST of 14 VMs is sufficient to process the peak workload of 800 Kbytes/sec. Figure 6.4 shows the selected MST models. Predicted MST values are plotted together with actual MST measurements for up to 128 VMs except for Rolling Flight Distances, which shows measurements up to 14 VMs.

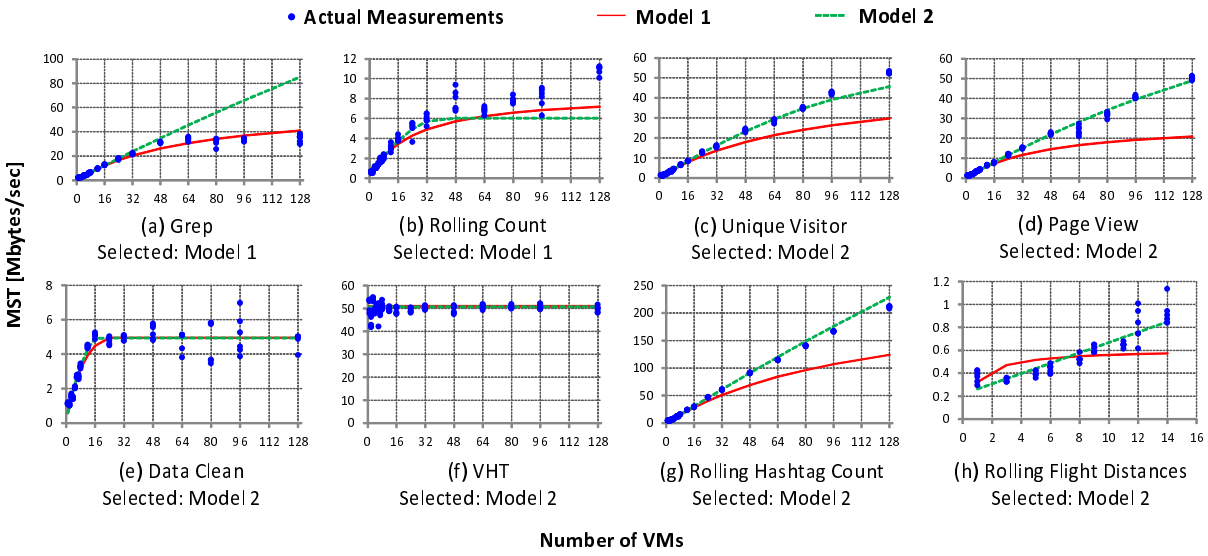


Figure 6.4. Selected MST prediction models after training. Models from (a) Grep to (g) Rolling Hashtag Count are trained with samples obtained from up to 24 VMs in $\mathcal{S}^* = \{3, 4, 6, 8, 24\}$, whereas (h) Rolling Flight Distances is trained with samples obtained from up to 14 VMs in $\mathcal{S}_{\text{flight}} = \{1, 2, 3, 5, 6, 8, 9, 11, 12, 14\}$.

6.4.3 Workload Forecasting Model Training

As shown in Eq. (6.9), an ARMA(p, q) model consists of an auto-regressive (AR) term of order p and a moving-average (MA) term of order q. We determine the values for p and q using Akaike information criterion (AIC), which is commonly used in time series analysis . AIC evaluates the fitness of a model to data as well as the simplicity of the model. Using the first 100 values from each test workload, we perform a grid search to find the (p, q) pair that gives the lowest AIC: we compute AIC for all possible pairs of $p = 0, 1, 2, 3$ and $q = 0, 1, 2, 3$ except for $(p, q) = (0, 0)$. As the result of the search, we obtained models as shown in Table 6.1. For online update of the model, we use a sliding window of the past 100 workload values to estimate ϕ and θ in Eq. (6.9) by maximum likelihood estimation.

Table 6.1. Selected ARMA models for the test workloads.

Workload	Model
World Cup 98	ARMA(1, 2)
Tweets	ARMA(1, 1)
ADS-B	ARMA(1, 3)

6.5 Evaluation

We evaluate the VM scheduling techniques presented in Section 6.3 with the real-world workloads shown in Figure 6.3. Evaluations are simulation-based, however, MST prediction models are trained with real measurements as described in Section 6.4.2. We first describe the experiment settings that are common to all evaluations. We then give the experimental results.

6.5.1 Common Experimental Settings

6.5.1.1 Simulation Time Horizon

We perform simulations over discrete timesteps $t = 1, 2, ..., T$, where a workload value $\lambda(t)$ is given for each t. The workloads change every minute and have duration of one week, so $T = 7 \times 24 \times 60 = 10,080$ for all three workloads. Our VM scheduler performs VM

allocation or deallocation every C timesteps and consider U timesteps of VM startup time. In the following simulations, we use $C = 10$ and $U = 2$, which correspond to 10 and 2 minutes of physical time, respectively.

6.5.1.2 Workloads and Test Applications

We test the workloads against the MST models created for the test applications as shown in Table 6.2. The World Cup 98 workload is used for Grep, Rolling Count, Unique Visitor, Page View, Data Clean, and VHT to evaluate applications with different scaling patterns with a consistent real-world workload. For Rolling Hashtag Count and Rolling Flight Distances, we use the Tweets and ADS-B workloads that are actually processed by these two applications to create their MST models, respectively. Since data rates in the test workloads and throughput generated from the trained MST models are not directly comparable to the World Cup 98 and ADS-B workloads, we artificially create more workload data proportionally keeping workload distribution patterns to reach a maximum throughput as shown in the "Peak Throughput" column in Table 6.2. For the ADS-B workload, actual peak of the workload and the MST model for Rolling Flight Distances match, so there is no need to recreate the workload.

Table 6.2. Workloads and test applications (MST models) used for evaluations.

Workload	Test Application (MST models)	Peak Throughput [Mbytes/sec]
World Cup 98	Grep	32
	Rolling Count	10
	Unique Visitor	48
	Page View	46
	Data Clean	5
	VHT	49
Tweets	Rolling Hashtag Count	200
ADS-B	Rolling Flight Distances	None

6.5.1.3 Hypothetical Ground Truth MST

To evaluate a series of allocated VM counts $m(t)$ for $t = 1, 2, ..., T$, we need to know the ground truth MST for $m(t)$ for a given application. We create a hypothetical ground truth MST model $\hat{\tau}_{\text{tru}}(m)$ from actual MST samples. Given a VM count m, $\hat{\tau}_{\text{tru}}(m)$ returns an MST value drawn from a normal distribution $\mathcal{N}(\mu_m, \sigma_m^2)$. The mean μ_m and variance σ_m^2 are created from pairwise linear interpolation of the mean and variance of the MST for measured VMs m_1 and m_2, where m_1 and m_2 are the two closest VM counts to m, with $m_1 < m < m_2$. Figure 6.5 shows an example of hypothetical ground truth probability distribution created from the measured MST samples for the Grep benchmark. We can see that the mean and 95% confidence interval are linearly interpolated.

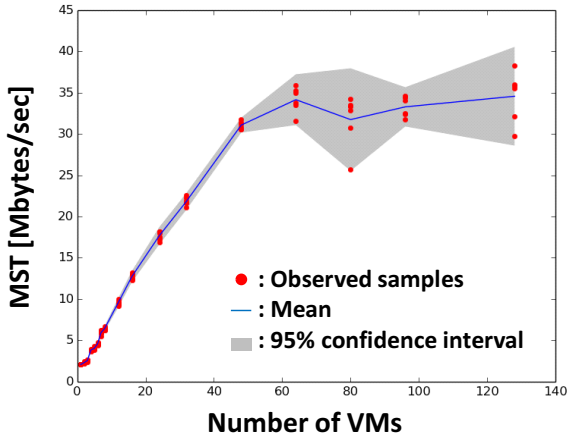


Figure 6.5. Example of hypothetical ground truth probability distribution created from the measured MST samples for the Grep benchmark.

6.5.1.4 Baseline Scheduling

We use the following scheduling policies as the baseline.

Ground Truth: Optimal scaling policy based on the actual MST values. It allocates the minimum number of VMs that generates a larger MST than a given workload.

Static Peak: Static VM allocation policy that covers the peak workload. The peaks are equal to the normalization factors in Table 6.2 (for Rolling Flight Distances, the peak is 830

Kbytes/sec).

6.5.1.5 Evaluation Metrics

We use the following metrics to evaluate scheduling results.

QoS Satisfaction Rate

$$QoS\ Satisfaction\ Rate[\%] = \frac{100}{T}\sum_{t=1}^{T}\mathcal{I}(\hat{\tau}_{\text{tru}}(m(t)) \geq \lambda(t)), \tag{6.21}$$

where T is the total timesteps (=10,080) of a simulation, $m(t)$ is the allocated VM counts using our VM scheduling technique at time t, $\lambda(t)$ is the workload at time t, and $\mathcal{I}(\cdot)$ is the indicator function that returns 1 if the argument is true and 0 otherwise. Note that since the scheduling cycle is $C = 10$ timesteps, we will compare ten workload values against a single MST value.

Relative VM costs Since total VM cost depends on a sequence of workload values, we cannot directly compare absolute costs obtained from different workload values. Thus, we compute VM allocation cost relative to the cost obtained from the ground truth scheduling and the cost obtained from the static scheduling as follows:

$$RelativeCost_{\text{tru}} = \frac{\sum_{t=1}^{T} m(t)}{\sum_{t=1}^{T} m_{\text{tru}}(t)}, \tag{6.22}$$

$$RelativeCost_{\text{static}} = \frac{\sum_{t=1}^{T} m(t)}{T \cdot \max_t m_{\text{tru}}(t)}, \tag{6.23}$$

where $m(t)$ is the number of allocated VM at time t by the prediction framework and $m_{\text{tru}}(t)$ is the true required number of VMs obtained by the ground truth scheduling at time t. Since we use homogeneous VMs, a relative cost is the ratio of the numbers of VMs.

6.5.2 Evaluation: Scheduling Policy vs. QoS & Cost

We first evaluate the impact of the different scheduling policies on QoS and Cost.

6.5.2.1 Experimental Settings

Depending on how we incorporate uncertainty awareness of MST, workload forecasting, and online learning techniques into the VM scheduler, we have multiple scheduling policies, as shown in Table 6.3. Each column has the following meaning:

UA: This option enables uncertainty-awareness for MST.

OL: If this option is chosen, online learning for the MST model is performed as described in Section 6.3.2, otherwise the MST model is fixed during the simulation.

ARMA: If this option is chosen, workload forecasting with an ARMA model is enabled and the scheduling is uncertainty-aware for workload forecasting and scaling, otherwise workload forecasting is not used.

Eq.: Depending on the combination of the three options, it shows a reference to the equation used for scheduling.

Table 6.3. VM scheduling policies for evaluation.

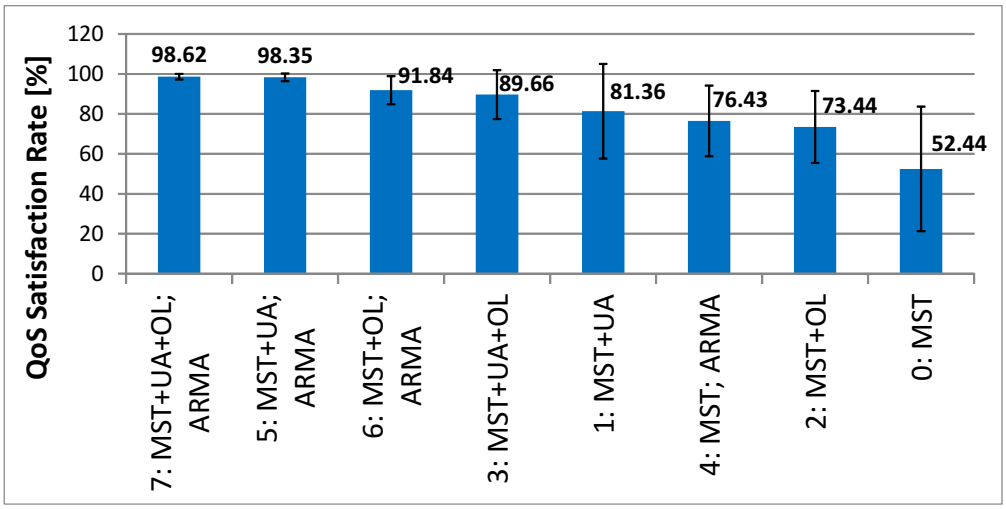
Policy ID	MST		**ARMA**	**Eq.**
	UA	**OL**		
0: MST				(6.3)
1: MST + UA	✓			(6.8)
2: MST + OL		✓		(6.3)
3: MST + UA + OL	✓	✓		(6.8)
4: MST ; ARMA			✓	(6.18)
5: MST + UA ; ARMA	✓		✓	(6.20)
6: MST + OL ; ARMA		✓	✓	(6.18)
7: MST + UA + OL ; ARMA	✓	✓	✓	(6.20)

Policy IDs are named after the corresponding scheduling options. We ran simulations five times per application per policy. Since there are eight applications, the number of simulation runs per policy is $8 \times 5 = 40$. For each policy, we took the average of 40 runs for QoS satisfaction rates and relative costs against the ground truth and static peak scheduling policies. The QoS satisfaction target ρ is 0.95.

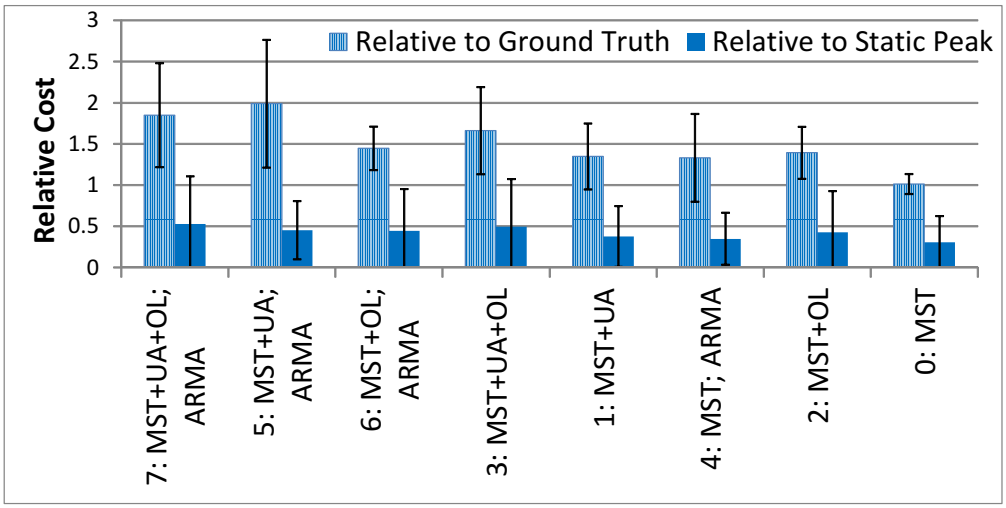
6.5.2.2 Scheduling Results

Figure 6.6 shows average (a) QoS satisfaction rates and (b) relative costs across all the applications for the scheduling policies in Table 6.3, in descending order of the QoS satisfaction rate. Error bars show ± 1 standard deviation. We can see the overall trend where the QoS satisfaction rate improves with increasing cost as the policy becomes more complex. The most complex #7 policy achieved 98.62% QoS satisfaction rate. While it cost 84% more than the ground truth scheduling, it was 48% less when compared to static scheduling that covers the peak workload. We also notice that each of the three scheduling options significantly improved QoS satisfaction rates from 52% to 73-81% QoS satisfaction rates compared to the base #0 policy. Policies #7 and #5 show very close QoS satisfaction rates: 98.62% vs. 98.35%; however, in terms of the relative cost to ground truth, #7 costs 7% less than #5 (#7: 1.85 vs. #5: 1.99). This result suggests that the online learning technique contributes to lower the cost. Regardless of using the online learning technique, when we consider uncertainties from both MST and workload forecasting models in #7 and #5, we successfully achieved high QoS satisfaction rates. These satisfaction rates are also close to the QoS satisfaction target of $\rho = 0.95$.

Figure 6.7 shows application-wise QoS satisfaction rates and relative costs for four scheduling policies, #0, #1, #5, and #7. These policies were chosen so that a new scheduling technique was incrementally added. From #0 to #1, UA was added; from #1 to #5, ARMA was added; and from #5 to #7, OL was added. Overall, we can see that as we add a new technique, the QoS satisfaction rates improve. The first two rates improvements (*i.e.*, #0 to #1 and #1 to #5) can be explained in terms of the scheduling techniques: we add variance for MST in (6.7) and then add another variance for workload forecasting incrementally in (6.19). Since larger variance leads to more conservative VM allocation, improvement in QoS satisfaction rates can be expected. For the third improvement (*i.e.*, #5 to #7), Unique Visitor, Page View, and Rolling Hashtag Count show at least 20% of QoS rate increase. This is most likely due to the relationship between initial MST models after offline training



(a) QoS Satisfaction Rates



(b) Relative Costs

Figure 6.6. Average (a) QoS satisfaction rates and (b) Relative costs across the all applications for different scheduling policies in Table 6.3. QoS satisfaction target: $\rho = 0.95$. Error bars show $\pm$ 1 standard deviation.

and actual MST sample values. As shown in Figure 6.4, these three models have some VM counts where predicted values exceed actual sample values. This means that the models over-estimate their MST, and thus they tend to under-provision VMs. By updating their MST models online, the models become more accurate, which can lead to better QoS satisfaction rates.

For all the applications except for VHT, if we spend more cost, we are rewarded with

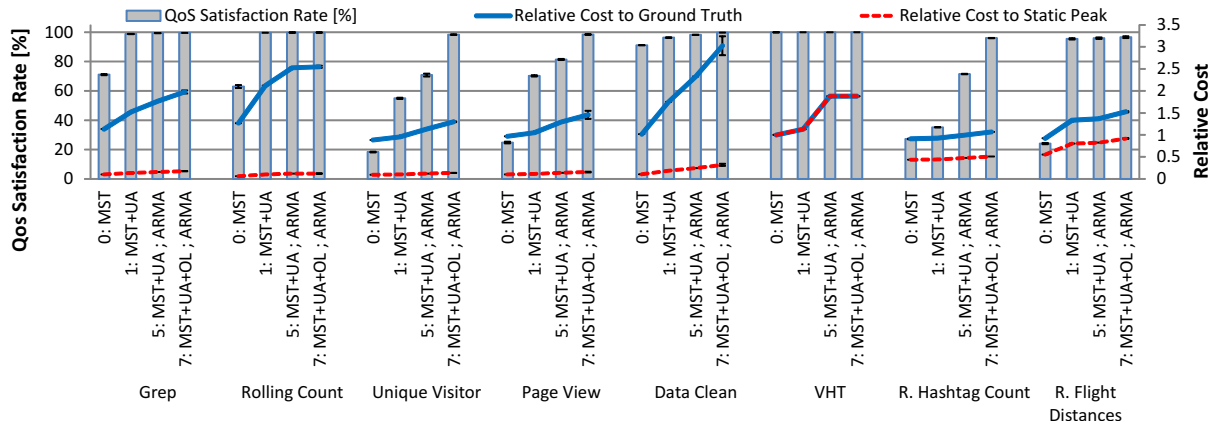


Figure 6.7. QoS satisfaction rates and relative costs for the #0, #1, #5, and #7 scheduling policies in Table 6.3. QoS satisfaction target: $\rho = 0.95$. Error bars show $\pm$ 1 standard deviation.

higher QoS satisfaction rates. However, VHT shows 100% QoS satisfaction rates even with the the most basic #0 policy. This is due to the fact that VHT does not scale at all as shown in Figure 6.4. To achieve a 100% QoS satisfaction rate, both static peak and ground truth policies needed 1 VM for the entire simulation period. However, our scheduling techniques with ARMA unnecessarily over-provision especially when spikes occur.

6.5.3 Evaluation: QoS Satisfaction Target vs. QoS & Cost

To investigate how the QoS satisfaction target ρ affects actual QoS and cost, we tested $\rho = \{0.5, 0.7, 0.9, 0.95\}$ with the "#7:MST+UA+OL;ARMA" policy. We ran simulations five times per application per ρ and took the average QoS satisfaction rate and relative cost across all the applications. Figure 6.8 shows the results. We can see that ρ and QoS satisfaction rate and both relative costs positively correlate. In fact, Pearson's correlation coefficient between ρ and the QoS satisfaction rate was 0.652. Looking at the slope from $\rho = 0.9$ to $\rho = 0.95$ on the relative cost to ground truth, it is steeper than that from $\rho = 0.7$ to $\rho = 0.90$. This result matches the following fact: since ρ is a constraint on the probability of a normal distribution, when the value of ρ approaches 1.0, it will be required to allocate infinitely many VMs to satisfy the constraint.

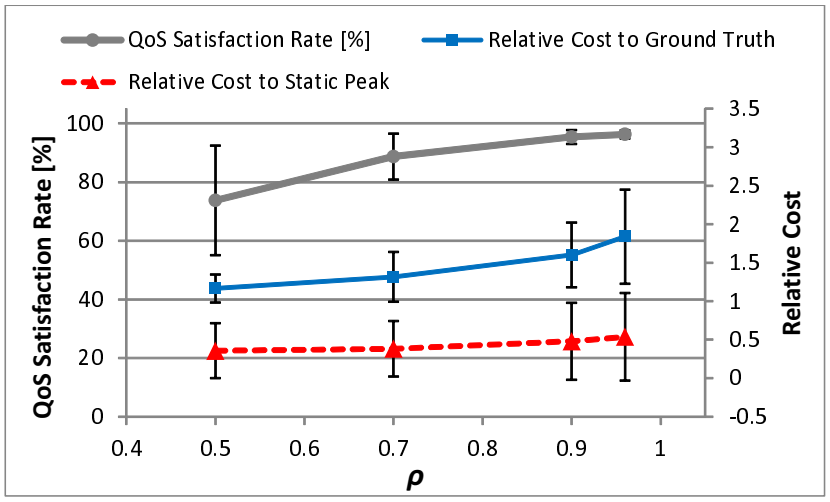


Figure 6.8. QoS satisfaction target ρ vs. actual QoS satisfaction rates and relative costs. Error bars show $\pm$ 1 standard deviation.

6.5.4 VM Allocation Sequence

To confirm the effectiveness of scheduling policies on VM allocation sequences, we evaluated the same four scheduling policies (#0, #1, #5, and #7) used in Figure 6.9 with $\rho = 0.95$ and the Grep application. We plot (a) Input workload and allocated MST, (b) allocated number of VMs, and (c) backlogged data. Since the difference is hardly visible, in Figures 6.9(a) and (b), we only plot #0 and #7 together with the ground truth and static policies. From Figure 6.9(a), it appears #0 closely follows the input workload; however, from Figure 6.9(c), we can confirm that it accumulates up to 12.8 Gbytes of backlogged data when the input workload spikes. Since #0 does not have any uncertainty consideration or workload forecasting, this result is unavoidable. Unlike #0, #5 and #7 proactively react to the workload spikes and successfully avoid backlogging.

6.6 Related Work

Workload forecasting is widely used in proactive elastic VM scheduling. Roy et al. [47] used an ARIMA model to forecast the World Cup 98 workload, which is also used in this work. Hu et al. proposed KSwSVR, a workload forecasting method based on Kalman filters and Support Vector Regression and applied it to elastic resource provisioning [116]. Jian et al.

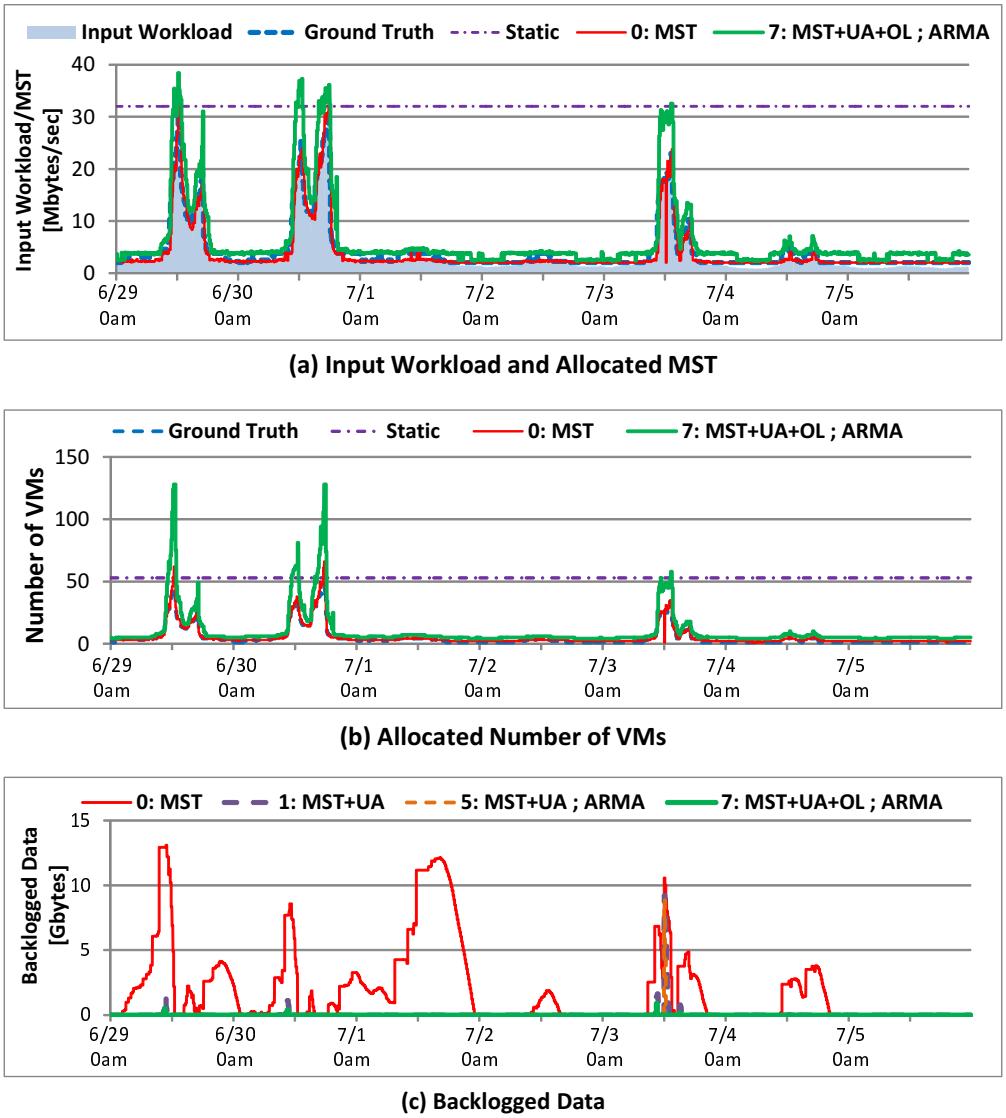


Figure 6.9. Scheduling results for the Grep application with the FIFA world cup 1998 website access workload (6/29/1998-7/5/1998): (a) Input workload and allocated MST, (b) Allocated number of VMs, and (c) Backlogged data.

used a simple linear model to forecast future workloads, where parameters are estimated by linear regression, and to proactively allocate VMs estimated by a M/M/m queuing theory model [117]. ProRenaTa, developed by Liu et al. [119], combines proactive and reactive approaches for elastic scaling on distributed storage systems. In each scheduling cycle, the proactive scheduler first predicts workload using an ARIMA model and makes an initial scaling decision. To address inaccurate workload prediction, after the reactive controller observes the actual workload, it allocates extra VMs if needed. One of the major differences

between our work and these works is that we explicitly account for estimated uncertainty from workload forecasting and use that information to achieve high QoS satisfaction rate in a cost efficient manner.

There are several previous works that require manual parameter configurations to achieve high QoS. Rodriguez and Buyya [41] considered performance variation of VMs for workflow applications with a performance degradation parameter defined for each VM type. This parameter effectively adjusts the over-provisioning rate per VM type. Even though KSwSVR [116] showed good prediction accuracy, they needed to specify a parameter for over-provisioning to reduce QoS violations. Similarly, in our own previous work [118], we introduced over-provisioning rate parameters to account for the inaccuracy of application performance prediction models. Our scheduling framework also provides ρ, a parameter to control QoS satisfaction rate; however, due to the reasonably positive correlation of 0.652 to the actual QoS satisfaction rate and also due to the fact that it is bounded by 1.0, the cost for finding the right ρ value can be significantly lower than the cost for parameter tuning in these existing works.

6.7 Summary

We have proposed a framework for proactive elastic VM scheduling for stream processing systems in cloud computing environments that explicitly incorporates uncertainty in application performance and workloads. The framework estimates the required number of VMs that satisfies a certain probability criteria for the QoS objective, using expected variances from application performance and workload prediction models. We have shown that our scheduling framework can achieve 98.62% of QoS satisfaction in simulations using real-world workloads. Further, the scheduler achieves up to 48% lower cost compared to a static scheduling that covers the peak workload.

CHAPTER 7
CONCLUSION AND FUTURE DIRECTIONS

In this thesis, we explored QoS-aware elastic data processing for IaaS clouds from three different processing models: batch, micro-batch, and streaming. Our studies show that QoS-aware elastic data processing is effective for these processing models in both performance scalability and cost savings. For batch processing, elastic resource scheduling helped achieve the target QoS metrics such as CPU utilization and job completion time. For both micro-batch and stream processing with fluctuating workloads, QoS-aware elastic scheduling saved up to 49% cost compared to a static scheduling that covers the peak workload to achieve a similar level of throughput QoS satisfaction. These results show potential for future fully automated cloud computing resource management systems that efficiently enable truly elastic and scalable general-purpose workload.

7.1 Chapter Summary

We summarize the results from each chapter as follows:

- In Chapter 2, we presented two frameworks for elastic batch processing: auto-scaling using application migration as a reconfiguration strategy in Section 2.1 and cost-optimal heterogeneous VM scheduling using Workload-tailored Elastic Compute Units (WECUs) in Section 2.2. The former approach did not use any prior knowledge about the target application, but gradually scaled up the application with its light-weight application migration. It achieved 36% shorter execution time compared to a static scheduling with the same initial condition (*i.e.*, 4 vCPUs). Unlike the former, the latter approach fully used performance metrics of the target application through WECUs. It generated cost-optimal VM configurations with 4.59% average execution time prediction error.

- In Chapter 3, we presented an elastic middleware framework that is specifically designed to solve ILP problems generated from continuous air traffic streams. We proposed a speculative VM scheduling algorithm with time series and resource prediction models. It achieved a similar performance to a static scheduling that covers the peak workload while using 49% less VM hours for a smoothly changing air traffic. However, for a sharply changing air traffic, our scheduling algorithm allocated slightly more VM hours to achieve the same performance. Our algorithm is able to adapt dynamically to potentially unforeseen fluctuating demand with a reasonable prediction accuracy.

- In Chapter 4, we outlined a framework for sustainable elastic stream processing and presented the concept of Maximum Sustainable Throughput (MST) that we further explored in Chapters 5 and 6.

- In Chapter 5, we presented a cost-effective MST prediction framework for stream processing applications with various scalability characteristics. We statistically determine the best subset of VM counts in terms of prediction error from up to 32 VMs to collect training samples. For each new application, we train the two MST prediction models using this subset. The framework takes several trained models and selects the model that is expected to predict MST values for the target application with the lowest error. We evaluated our framework on streaming applications, using up to 128 VMs. Experiments showed that our framework can predict MST with up to 15.8% average prediction error.

- In Chapter 6, we proposed a framework for proactive elastic VM scheduling for stream processing systems that explicitly incorporates uncertainty in application performance and workloads. The framework estimates the required number of VMs that satisfies a certain probability criteria for the QoS objective, using expected variances from application performance and workload prediction models. We have shown that our scheduling framework can achieve 98.62% of QoS satisfaction in simulations using real-

world workloads. Further, the scheduler achieved up to 48% lower cost compared to a static scheduling that covers the peak workload.

7.2 Future Directions

7.2.1 Future Elastic Resource Allocation Framework

Inspired by A Manifesto for Future Generation Cloud Computing [124], we envision a future QoS-aware elastic data processing framework that can cover a wide range of use-cases beyond data processing in single data centers. We enumerate possible features for such framework as follows:

- **Geo-distributed Data Processing**: The framework processes geo-distributed data sets as well as centralized data sets.
- **Resource Allocation Beyond Clouds**: The framework assumes computational resources are available not only from existing cloud computing services, but also from volunteer computing [125], computing platforms incentivised by crytpocurrency such as Gridcoin [126], and emerging fog computing nodes [127].
- **Flexible Processing Models**: The framework supports both batch and stream processing models just as supported in Flink [9].
- **QoS**: The framework accepts QoS criteria from the user in terms of end-to-end processing latency or throughput.

Geo-spatial data processing has been studied in the contexts of minimizing response times [128], using volunteer resources [129], and trading timeliness for accuracy in streaming [130]. However, QoS-aware elastic geo-spatial data processing is not fully explored. Potential research challenges include the following:

Application Performance Modeling It is challenging to create performance models for geo-distributed data processing applications running on a highly dynamic environment

posed by volunteer and fog computing nodes. Due to the heterogeneity of the nodes, it is not realistic to run benchmarks for all the nodes in advance. However, it is reasonable to assume that we can obtain general specifications of the nodes (*e.g.*, memory size, CPU speed) in advance and estimate the target application's performance using these general specifications. Another challenge is estimating latency and dynamically changing bandwidth for arbitrary application communication topology over wide-area networks. To keep up with dynamically changing network and resource environments, online machine learning techniques will be useful to maintain the accuracy of application performance models.

Adaptive Application Reconfiguration Application performance can be improved by dynamically splitting or merging application tasks [131], or migrating some of the tasks to idle resources [48], [58]. Since performing these operations over wide-area network can incur significant performance overhead, we will need to prevent thrashing behaviors (*e.g.*, repeating splitting and merging operations). Further, depending on the reliability of computing nodes (*e.g.*, cloud: 99%, volunteer: 60%), we will also need to replicate the same application tasks and data over multiple distributed locations for better fault-tolerance.

Elastic Resource Allocation To obtain the optimal resource configuration constrained by a required QoS, it is typical to formulate a combinatorial optimization problem. However, the number of combinations for multiple heterogeneous node types and application communication topologies over wide-area networks can be explosive. Thus, solving the optimization problem by a single central scheduler may take too long time even to obtain approximate solutions. Moreover, we need to keep solving new optimization problems repeatedly to adapt dynamically changing incoming workloads and network bandwidth. We have multiple strategies to tackle this problem as follows:

- Develop a heuristic algorithm for the central scheduler that gives us approximate solutions reasonably quickly.

- Develop multiple independent decentralized schedulers, where each scheduler makes resource allocation decisions using local information only.
- Develop cooperative decentralized schedulers that communicate with each other to exchange their local information.
- Develop a hierarchical scheduler, where a central scheduler interacts with distributed sub-schedulers and gives them high-level resource allocation objectives.

Since optimal strategy may change depending on the size of the optimization problem and also the rate of workload changes, we will need to explore these elastic resource allocation strategies in various data/resource distribution scenarios.

7.2.2 Elastic Resource Allocation for Serverless Computing

As mentioned in Section 1.1.2, the serverless computing service model (*i.e.*, FaaS) frees users from managing servers. It instead executes a function provided by the FaaS user upon the reception of an event. FaaS users are charged based on the number of function calls and the duration of code execution.

Current FaaS providers offer limited resource configuration options to the FaaS user: AWS Lambda [132] and Microsoft Azure Functions [133] offer different memory sizes (*e.g.*, 300 Mbytes to 3 Gbytes for AWS Lambda); and Google Cloud Functions [134] offers four different resource configurations from 128 Mbytes memory and 200 MHz CPU to 2 Gbytes memory and 2.4 GHz CPU. Thus, the FaaS user's ability to control application performance is limited. However, we can still create an application execution time model for several different resource configurations and use the model to dynamically adapt the resource configuration to fluctuating workloads.

FaaS providers implement an elastic resource scheduling framework underneath the FaaS layer as shown in Figure 1.1. From the FaaS provider perspective, the FaaS model gives an opportunity to consolidate more workloads into fewer physical machines since each

function call is independent and more fine-grained compared to VMs. There are research opportunities to efficiently consolidate parallel function calls requested by multiple users, where each function has different resource usage requirements and execution time.

7.2.3 Improvements to Presented Techniques

Reconfiguration Costs-Aware Scheduling VM scheduling simulation results for elastic data stream processing in Figure 6.9 showed that the proposed scheduling method successfully kept the amount of backlog low. The simulation took VM startup time into account, but not the cost for reconfiguration. In practice, however, when reconfiguring a stream application, stream processing systems need to stop consuming data [6], [34], [35]. Thus, throughput can be significantly decreased during reconfiguration. Depending on the data rate of input streams, backlog may be accumulated quickly. For example, if we stop consuming stream data of 10 Mbytes/sec for one minute, it takes 600 Mbytes of storage space in the message broker. Heinze et al. proposed a scheduling technique which tries to minimize latency spikes caused by reconfigurations [35]. However, their technique remains reactive. If the scheduler reacts to short-term input data rate spikes, it is possible to cause thrashing behavior of reconfigurations. For more robust VM scheduling, the scheduler should be aware of backlog (or latency) and make proactive scheduling decisions that follow long-term trends of workloads.

Uncertainty Quantification In Chapter 6, we considered uncertainty for MST prediction models. As the result of using linear regression, we estimated a single variance value from all the MST samples that we used to train a model (see Eq. (6.6)). However, variance of actual MST samples varies depending on the number of VMs as we can confirm in Figure 6.4. Since our proposed uncertainty-aware VM scheduling is generally applicable even if different variance is defined for each VM count, using different variance for each VM count can better reflect the true performance of applications and lead to a better QoS satisfaction. To quantify uncertainty for each different VM count, we can use techniques such as Gaussian Process

Regression or Kriging, where we can predict MST values while continuously interpolate uncertainty.

REFERENCES

[1] M. Armbrust *et al.*, "Above the clouds: A Berkeley view of cloud computing," EECS Department, University of California, Berkeley, CA, Tech. Rep. UCB/EECS-2009-28, 2009, [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf, Accessed on: April 18, 2018.

[2] Apache Software Foundation. *Apache Hadoop.* (2014) [Online]. Available: http://hadoop.apache.org/, Accessed on: March 8, 2018.

[3] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Symp. on Networked Syst. Design and Implementation (NSDI 12)*, 2012, pp. 15–28.

[4] S. Imai, R. Klockowski, and C. A. Varela, "Self-healing spatio-temporal data streams using error signatures," in *Proc. Int. Conf. on Big Data Science and Eng. (BDSE 13)*, 2013, pp. 957–964.

[5] M. Solaimani, M. Iftekhar, L. Khan, B. Thuraisingham, and J. B. Ingram, "Spark-based anomaly detection over multi-source vmware performance data in real-time," in *Proc. IEEE Symp. on Computational Intell. in Cyber Security (CICS 14)*, 2014, pp. 1–8.

[6] A. Toshniwal *et al.*, "Storm@twitter," in *Proc. ACM SIGMOD Int. Conf. on Manage. of Data*, 2014, pp. 147–156.

[7] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: dynamic resource scheduling for real-time analytics over fast streams," in *Proc. IEEE Int. Conf. on Distrib. Comput. Syst. (ICDCS 15)*, 2015, pp. 411–420.

[8] A. Martin, A. Brito, and C. Fetzer, "DEBS grand challenge: Real time data analysis of taxi rides using StreamMine3G," in *Proc. ACM Int. Conf. on Distrib. Event-Based Syst. (DEBS 15)*, 2015, pp. 269–276.

[9] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[10] Apache Software Foundation. *Apache Samza* [Online]. Available: http://samza.apache.org/, Accessed on: March 8, 2018.

[11] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. ACM Symp. on Operating Syst. Principles (SOSP 13)*, 2013, pp. 423–438.

[12] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. SP-800-145, 2011, [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf, Accessed on: April 18, 2018.

[13] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, no. 239, 2014.

[14] M. Roberts, "Serverless architectures," 2016, [Online]. Available: https://martinfowler.com/articles/serverless.html, Accessed on: March 8, 2018.

[15] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud - Protocols and formats for cloud computing interoperability," in *Proc. Int. Conf. on Internet and Web Appl. and Services (ICIW 09)*, 2009, pp. 328–336.

[16] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments," *ACM Computing Surveys*, vol. 47, no. 212, pp. 1–47, 2014.

[17] Apache Software Foundation. *Apache Libcloud*. (2017) [Online]. Available: https://libcloud.apache.org/, Accessed on: March 8, 2018.

[18] D. Petcu, "Multi-Cloud: Expectations and current approaches," in *Proc. Int. Workshop on Multi-cloud Appl. and Federated Clouds (MultiCloud 13)*, 2013, pp. 1–6.

[19] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya, "Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges," *IEEE Commun. Surveys Tut.*, vol. 16, no. 1, pp. 337–368, 2014.

[20] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[21] T. Justino and R. Buyya, "Outsourcing resource-intensive tasks from mobile apps to clouds: Android and aneka integration," in *Proc. IEEE Int. Conf. on Cloud Computing in Emerging Markets (CCEM 14)*, 2014, pp. 1–8.

[22] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[23] Amazon Web Services, "Amazon EC2 service level agreement," 2018, [Online]. Available: https://aws.amazon.com/ec2/sla/, Accessed on: March 8, 2018.

[24] S. A. Baset, "Cloud SLAs: present and future," *ACM SIGOPS Operating Syst. Rev.*, vol. 46, no. 2, pp. 57–66, 2012.

[25] R. Buyya, S. K. Garg, and R. N. Calheiros, "SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions," in *Proc. Int. Conf. on Cloud and Service Computing (CSC 11)*, 2011, pp. 1–10.

[26] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid clouds," *Future Generation Comput. Syst.*, vol. 28, no. 6, pp. 861–870, 2012.

[27] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads," in *Proc. Int. Conf. on Cloud Computing (CLOUD 10)*, 2010, pp. 228–235.

[28] S. Imai, T. Chestna, and C. A. Varela, "Accurate resource prediction for hybrid IaaS clouds using workload-tailored elastic compute units," in *Proc. IEEE/ACM Int. Conf. on Utility and Cloud Computing (UCC 13)*, 2013, pp. 171–178.

[29] R. N. Calheiros and R. Buyya, "Cost-effective provisioning and scheduling of deadline-constrained applications in hybrid clouds," in *Web Inform. Syst. Eng. (WISE 2012)*. Charm, Switzerland: Springer, 2012, pp. 171–184.

[30] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Scheduling strategies for optimal service deployment across multiple clouds," *Future Generation Comput. Syst.*, vol. 29, no. 6, pp. 1431–1441, 2013.

[31] P. Lama and X. Zhou, "AROMA: Automated resource allocation and configuration of mapreduce environment in the cloud," in *Proc. ACM Int. Conf. on Autonomic Computing (ICAC 12)*, 2012, pp. 63–72.

[32] K. Chen, J. Powers, S. Guo, and F. Tian, "CRESP: Towards optimal resource provisioning for MapReduce computing in public clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1403–1412, 2014.

[33] T. Bicer, D. Chiu, and G. Agrawal, "Time and cost sensitive data-intensive computing on hybrid clouds," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 12)*, 2012, pp. 636–643.

[34] A. Ishii and T. Suzumura, "Elastic stream computing with clouds," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD 11)*, 2011, pp. 195–202.

[35] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proc. ACM Int. Conf. on Distrib. Event-Based Syst. (DEBS 14)*, 2014, pp. 13–22.

[36] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proc. IEEE Int. Conf. on Distrib. Comput. Syst. (ICDCS 15)*, 2015, pp. 399–410.

[37] S. Imai, S. Patterson, and C. A. Varela, "Uncertainty-aware elastic virtual machine scheduling for stream processing systems," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 18)*, 2018, to appear.

[38] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Comput. Syst.*, vol. 29, no. 1, pp. 158–169, 2013.

[39] L. F. Bittencourt and E. R. M. Madeira, "HCOC: A cost optimization algorithm for workflow scheduling in hybrid clouds," *J. Internet Services and Appl.*, vol. 2, no. 3, pp. 207–227, 2011.

[40] T. A. Genez, L. F. Bittencourt, and E. R. Madeira, "Workflow scheduling for SaaS/PaaS cloud providers considering two SLA levels," in *Proc. IEEE Netw. Operations and Manage. Symp. (NOMS 12)*, 2012, pp. 906–912.

[41] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Trans. Cloud Comput.*, vol. 2, no. 2, pp. 222–235, 2014.

[42] Amazon Web Services. *AWS Auto Scaling*. (2018) [Online]. Available: https://aws.amazon.com/autoscaling/, Accessed on: March 8, 2018.

[43] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow," in *Proc. Int. Conf. on Autonomic and Autonomous Syst. (ICAS 11)*, 2011, pp. 67–74.

[44] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[45] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*. New York, NY: John Wiley & Sons, 2015.

[46] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao *et al.*, "Energy-aware server provisioning and load dispatching for connection-intensive internet services." in *Proc. USENIX Symp. on Networked Syst. Design and Implementation (NSDI 08)*, 2008, pp. 337–350.

[47] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD 11)*, 2011, pp. 500–507.

[48] S. Imai, T. Chestna, and C. A. Varela, "Elastic scalable cloud computing using application-level migration," in *Proc. IEEE/ACM Int. Conf. on Utility and Cloud Computing (UCC 12)*, 2012, pp. 91–98.

[49] S. Imai, S. Patterson, and C. A. Varela, "Elastic virtual machine scheduling for continuous air traffic optimization," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 16)*, 2016, pp. 183–186.

[50] ——, "Cost-efficient elastic stream processing using application-agnostic performance prediction," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 16): Doctoral Symp.*, 2016, pp. 604–607, Best doctoroal symposium paper award.

[51] ——, "Maximum sustainable throughput prediction for large-scale data streaming systems," Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, Tech. Rep., 2017, Extended journal version of [118] in review. [Online]. Available: http://wcl.cs.rpi.edu/papers/mst2017.pdf, Accessed on: April 18, 2018.

[52] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," in *Proc. ACM Symp. on Operating Syst. Design and Implementation (OSDI 02)*, 2002, pp. 377–390.

[53] C. Clark *et al.*, "Live migration of virtual machines," in *Proc. USENIX Symp. on Networked Syst. Design and Implementation (NSDI 05)*, 2005, pp. 273–286.

[54] Hansen, J. Gorm, and E. Jul, "Self-migration of operating systems," in *Proc. 11th ACM SIGOPS European Workshop (EW 11)*, 2004, p. 23.

[55] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proc. USENIX Annu. Tech. Conf. (ATEC 05)*, 2005, pp. 391–394.

[56] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines." *Computer Networks*, vol. 53, no. 17, pp. 2923–2938, 2009.

[57] C. A. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Tech. Track Proc.*, vol. 36, no. 12, pp. 20–34, 2001.

[58] K. E. Maghraoui, T. Desell, B. K. Szymanski, and C. A. Varela, "The Internet Operating System: Middleware for adaptive distributed computing," *Int. J. High Performance Computing Appl., Special Issue on Scheduling Techniques for Large-Scale Distrib. Platforms*, vol. 20, no. 4, pp. 467–480, 2006.

[59] Q. Wang and C. A. Varela, "Impact of cloud computing virtualization strategies on workloads' performance," in *Proc. IEEE/ACM Int. Conf. on Utility and Cloud Computing (UCC 11)*, 2011, pp. 130–137.

[60] S. Imai and C. A. Varela, "Light-weight adaptive task offloading from smartphones to nearby computational resources," in *Proc. ACM Research in Applied Computation Symp. (RACS 11)*, 2011.

[61] Amazon Web Services, "Amazon EC2 FAQ," 2018, [Online]. Available: http://aws.amazon.com/ec2/faqs/, Accessed on: March 8, 2018.

[62] T. Desell, K. E. Maghraoui, and C. A. Varela, "Malleable applications for scalable high performance computing," *Cluster Computing*, vol. 10, no. 3, pp. 323–337, 2007.

[63] International Air Transport Association (IATA), "New IATA Passenger Forecast Reveals Fast-Growing Markets of the Future," [Online]. Available: http://www.iata.org/pressroom/pr/pages/2014-10-16-01.aspx, Accessed on: April 10, 2018.

[64] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Upper Saddle River, NJ: Prentice-Hall, 1982.

[65] GE, "GE Flight Quest Challenge 2," 2018, [Online]. Available: http://www.gequest.com/c/flight2-final/data, Accessed on: March 8, 2018.

[66] D. P. Palomar and M. Chiang, "A tutorial on decomposition methods for network utility maximization," *IEEE J. Selected Areas in Commun.*, vol. 24, no. 8, pp. 1439–1451, 2006.

[67] Y. Cao and D. Sun, "A link transmission model for air traffic flow management," *J. Guidance, Control, and Dynamics*, vol. 34, no. 5, pp. 1342–1351, 2011.

[68] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[69] Y. Cao and D. Sun, "Migrating large-scale air traffic modeling to the cloud," *J. Aerospace Inform. Syst.*, vol. 12, no. 2, pp. 257–266, 2015.

[70] FlightAware, "FlightAware," 2018, [Online]. Available: http://flightaware.com/, Accessed on: March 8, 2018.

[71] Amazon Web Services, "Amazon EC2," 2018, [Online]. Available: https://aws.amazon.com/ec2/, Accessed on: March 8, 2018.

[72] Apache Software Foundation. *Apache Spark*. (2018) [Online]. Available: http://spark.apache.org/, Accessed on: March 8, 2018.

[73] LGPL open source project. *lp_solve: Linear Integer Programming Solver* [Online]. Available: http://lpsolve.sourceforge.net/, Accessed on: March 8, 2018.

[74] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Performance characterization of in-memory data analytics on a modern cloud server," in *Proc. Int. Conf. on Big Data and Cloud Computing (BDCloud 15)*, 2015, pp. 1–8.

[75] S. L. Olivier, B. R. De Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," *Scientific Programming*, vol. 21, no. 3-4, pp. 123–136, 2013.

[76] Yiyuan Zhao and Joachim K. Hochwarth and Adrianne A. Hersrud, "Comprehensive dynamic air traffic system simulation (ComDATSS)," [Online]. Available: https://www.aem.umn.edu/research/atc/projects/ComDATSS/, Accessed on: March 8, 2018.

[77] J.-F. Cordeau, P. Toth, and D. Vigo, "A survey of optimization models for train routing and scheduling," *Transportation Science*, vol. 32, no. 4, pp. 380–404, 1998.

[78] C. Papahristodoulou and E. Dotzauer, "Optimal portfolios using linear programming models," *J. Operational Research Society*, vol. 55, no. 11, pp. 1169–1177, 2004.

[79] T. Lu and C. Boutilier, "Dynamic segmentation for large-scale marketing optimization," in *Workshop on Customer Life-Time Value Optimization in Digital Marketing*, 2014, pp. 21–26.

[80] Z. Abrams, O. Mendelevitch, and J. Tomlin, "Optimal delivery of sponsored search advertisements subject to budget constraints," in *Proc. ACM Conf. on Electronic Commerce (EC 07)*, 2007, pp. 272–278.

[81] A. Biem *et al.*, "Real-time traffic information management using stream computing," *IEEE Data Eng. Bull.*, vol. 33, no. 2, pp. 64–68, 2010.

[82] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic stream processing for the internet of things," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD 16)*, 2016, pp. 100–107.

[83] C. Hochreiner, S. Schulte, S. Dustdar, and F. Lécué, "Elastic stream processing for distributed environments," *IEEE Internet Comput.*, vol. 19, pp. 54–59, 2015.

[84] A. Shukla and Y. Simmhan, "Benchmarking distributed stream processing platforms for IoT applications," in *Technology Conf. on Performance Evaluation and Benchmarking (TPCTC 16)*, 2016, pp. 90–106.

[85] Apache Software Foundation. *Apache Storm*. (2015) [Online]. Available: http://storm.apache.org/, Accessed on: March 8, 2018.

[86] ——. *Apache Flink*. (2017) [Online]. Available: http://flink.apache.org/, Accessed on: March 8, 2018.

[87] M. D. de Assuncao, A. D. S. Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *J. Netw. and Computer Appl.*, vol. 103, pp. 1–17, 2018.

[88] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud," in *Proc. IEEE Int. Conf. on Cloud Computing (CLOUD 11)*, 2011, pp. 348–355.

[89] T. Li, J. Tang, and J. Xu, "A predictive scheduling framework for fast and distributed stream data processing," in *Proc. IEEE Int. Conf. on Big Data*, 2015, pp. 333–338.

[90] S. A. Noghabi *et al.*, "Samza: stateful scalable stream processing at LinkedIn," *Proc. of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[91] Apache Software Foundation. *Apache Spark Streaming*. (2018) [Online]. Available: http://spark.apache.org/streaming/, Accessed on: March 8, 2018.

[92] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. on Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[93] J. Kreps, N. Narkhede, and J. Rao, "Kafka : A distributed messaging system for log processing," in *Proc. ACM SIGMOD Workshop on Networking Meets Databases (NETDB 11)*, 2011, pp. 1–7.

[94] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. ACM Annu. Symp. on Cloud Computing (SOCC 13)*, 2013, pp. 1–16.

[95] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX Symp. on Networked Syst. Design and Implementation (NSDI 11)*, 2011, pp. 295–308.

[96] Apache Samza, "SAMZA-348: Configuring Samza jobs through a stream," [Online]. Available: https://issues.apache.org/jira/browse/SAMZA-348, Accessed on: March 8, 2018.

[97] Apache Spark, "SPARK-12133: Support dynamic allocation in Spark Streaming," [Online]. Available: https://issues.apache.org/jira/browse/SPARK-12133., Accessed on: March 8, 2018.

[98] R. B. Cooper, *Introduction to Queueing Theory, 2nd ed.* New York, NY: Elsevier North Holland, 1981.

[99] K. Teknomo, "Queuing theory tutorial," 2017, [Online]. Available: http://people.revoledu.com/kardi/tutorial/Queuing/, Accessed on: March 8, 2018.

[100] J. F. C. Kingman, "The single server queue in heavy traffic," *Mathematical Proc. Cambridge Philosophical Society*, vol. 57, no. 4, pp. 902–904, 1961.

[101] B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, 2014.

[102] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Proc. Int. Conf. on Distrib. Comput. Syst. (ICDCS 14)*, 2014, pp. 535–544.

[103] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proc. IEEE Int. Conf. on Cloud Eng. (IC2E 16)*, 2016, pp. 22–31.

[104] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. USENIX Symp. on Networked Syst. Design and Implementation (NSDI 16)*, 2016, pp. 363–378.

[105] G. Mariani, A. Anghel, R. Jongerius, and G. Dittmann, "Predicting cloud performance for HPC applications: A user-oriented approach," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 17)*, 2017, pp. 524–533.

[106] Intel Corporation. *Storm Benchmark.* (2018) [Online]. Available: https://github.com/intel-hadoop/storm-benchmark/, Accessed on: March 8, 2018.

[107] A. Verma, L. Cherkasova, and R. Campbell, "ARIA: Automatic resource inference and allocation for mapreduce environments," in *Proc. ACM Int. Conf. on Autonomic Computing (ICAC 11)*, 2011, pp. 235–244.

[108] Microsoft, "Measuring maximum sustainable engine throughput," [Online]. Available: https://msdn.microsoft.com/en-us/library/cc296884(v=bts.10).aspx, Accessed on: March 8, 2018.

[109] C. Davatz, C. Inzinger, J. Scheuner, and P. Leitner, "An approach and case study of cloud instance type selection for multi-tier web applications," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 17)*, 2017, pp. 534–543.

[110] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from Data*. New York, NY: AMLBook, 2012.

[111] Luis Martin Garcia. *TCPDUMP/LIBPCAP Public Repository*. (2017) [Online]. Available: http://www.tcpdump.org/, Accessed on: March 8, 2018.

[112] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[113] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating skew in mapreduce applications," in *Proc. ACM SIGMOD Conf. on Manage. of Data*, 2012, pp. 25–36.

[114] C. Delimitrou and C. Kozyrakis, "HCloud: Resource-efficient provisioning in shared cloud systems," in *ACM SIGOPS Operating Syst. Rev.*, 2016, vol. 50, no. 2, pp. 473–488.

[115] N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov, "Augmenting elasticity controllers for improved accuracy," in *Proc. IEEE Conf. on Autonomic Computing (ICAC 16)*, 2016, pp. 117–126.

[116] R. Hu, J. Jiang, G. Liu, and L. Wang, "KSwSVR: A new load forecasting method for efficient resources provisioning in cloud," in *Proc. IEEE Conf. on Services Computing (SCC 13)*, 2013, pp. 120–127.

[117] J. Jiang, J. Lu, G. Zhang, and G. Long, "Optimal cloud resource auto-scaling for web applications," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 13)*, 2013, pp. 58–65.

[118] S. Imai, S. Patterson, and C. A. Varela, "Maximum sustainable throughput prediction for data stream processing over public clouds," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 17)*, 2017, pp. 504–513.

[119] Y. Liu, N. Rameshan, E. Monte, V. Vlassov, and L. Navarro, "ProRenaTa: Proactive and reactive tuning to scale a distributed storage system," in *Proc. IEEE Symp. on Cluster, Cloud and Grid Computing (CCGrid 15)*, 2015, pp. 453–464.

[120] G. D. F. Morales and A. Bifet, "SAMOA: Scalable advanced massive online analysis," *J. Mach. Learning Res.*, vol. 16, pp. 149–153, 2015.

[121] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE Netw.*, vol. 14, no. 3, pp. 30–37, 2000.

[122] Twitter. *Twitter API Reference*. (2018) [Online]. Available: https://dev.twitter.com/docs, Accessed on: March 8, 2018.

[123] ADS-B Exchange, "ADS-B Exchange - world's largest co-op of unfiltered filght data," 2018, [Online]. Available: https://www.adsbexchange.com/, Accessed on: March 8, 2018.

[124] R. Buyya *et al.*, "A manifesto for future generation cloud computing: Research directions for the next decade," *arXiv:1711.09123 [cs.DC]*, 2017, [Online]. Available: https://arxiv.org/abs/1711.09123, Accessed on: April 19, 2018.

[125] D. P. Anderson, E. Korpela, and R. Walton, "High-performance task distribution for volunteer computing," in *Int. Conf. on e-Science and Grid Computing (e-Science 05)*, 2005, pp. 196–203.

[126] Gridcoin community, "Gridcoin," [Online]. Available: https://gridcoin.us/, Accessed on: April 10, 2018.

[127] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi, "Fog computing conceptual model," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. SP-500-325, 2018.

[128] Q. Pu *et al.*, "Low latency geo-distributed data analytics," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 421–434, 2015.

[129] A. Jonathan, M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data intensive computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3229–3242, 2017.

[130] B. Heintz, A. Chandra, and R. K. Sitaraman, "Trading timeliness and accuracy in geo-distributed streaming analytics," in *Proc. ACM Symp. on Cloud Computing (CLOUD 16)*, 2016, pp. 361–373.

[131] K. E. Maghraoui, T. Desell, B. K. Szymanski, and C. A. Varela, "Malleable iterative mpi applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 3, pp. 393–413, 2009.

[132] Amazon Web Services. *AWS Lambda*. (2018) [Online]. Available: https://aws.amazon.com/lambda/, Accessed on: April 10, 2018.

[133] Microsoft. *Microsoft Azure Functions*. (2018) [Online]. Available: https://azure.microsoft.com/en-us/services/functions/, Accessed on: April 11, 2018.

[134] Google. *Google Cloud Functions* [Online]. Available: https://cloud.google.com/functions/, Accessed on: April 10, 2018.

APPENDIX A

Peak Calculations for Model 1

We take the derivative of f_1 in Eq. (5.2) to find when f_1 gets to its peak.

$$\frac{\partial f_1}{\partial m} = \frac{\partial}{\partial m}\left(\frac{1}{w_0 + w_1 \cdot \frac{1}{m} + w_2 \cdot m + w_3 \cdot m^2}\right). \tag{A.1}$$

Using the chain rule with $u = w_0 + w_1 \cdot \frac{1}{m} + w_2 \cdot m + w_3 \cdot m^2$, Eq. (A.1) can be further calculated as follows:

$$\begin{aligned}
\frac{\partial f_1}{\partial m} &= \frac{\partial}{\partial u}\left(\frac{1}{u}\right) \cdot \frac{\partial u}{\partial m} \\
&= -\frac{1}{u^2} \cdot \left(-\frac{w_1}{m^2} + w_2 + 2w_3 m\right) \\
&= \frac{1}{u^2} \cdot \frac{w_1 - w_2 m^2 - 2w_3 m^3}{m^2} \\
&= \frac{w_1 - m^2(w_2 + 2w_3 m)}{(um)^2} \\
&= \frac{w_1 - m^2(w_2 + 2w_3 m)}{\left((w_0 + w_1 \cdot \frac{1}{m} + w_2 \cdot m + w_3 \cdot m^2) \cdot m\right)^2} \\
&= \frac{w_1 - m^2(w_2 + 2w_3 m)}{(w_0 \cdot m + w_1 + w_2 \cdot m^2 + w_3 \cdot m^3)^2}.
\end{aligned} \tag{A.2}$$

From the trained results for Model 1 in Table 5.2, we have only two cases where f_1 can have a peak: 1) $w_2 = 0$ and $w_3 \neq 0$, and 2) $w_3 \neq 0$ and $w_2 = 0$. For these two cases, we calculate the peaks of f_1 as follows:

1. For $w_2 = 0$ and $w_3 \neq 0$, we find when the slope (A.2) is zero:

$$\frac{\partial f_1}{\partial m} = \frac{w_1 - 2w_3 m^3}{(w_0 \cdot m + w_1 + w_3 \cdot m^3)^2} = 0 \tag{A.3}$$

$$m^* = \left(\frac{w_1}{2w_3}\right)^{\frac{1}{3}} \quad \text{if} \quad w_0\left(\frac{w_1}{2w_3}\right)^{\frac{1}{3}} + \frac{3}{2}w_1 \neq 0. \tag{A.4}$$

2. For the case $w_2 \neq 0$ and $w_3 = 0$, we find when the slope (A.2) is zero:

$$\frac{\partial f_1}{\partial m} = \frac{w_1 - w_2 m^2}{w_0 \cdot m + w_1 + w_2 \cdot m^2} = 0 \tag{A.5}$$

$$m^* = \frac{\sqrt{w_1}}{\sqrt{w_2}} \text{ if } \sqrt{w_2} \neq 0 \text{ and } w_0 \frac{\sqrt{w_1}}{\sqrt{w_2}} + 2w_1 \neq 0. \tag{A.6}$$