2016

Application-Aware Resource Management for Cloud Platforms

Xin He
University of Massachusetts Amherst

APPLICATION-AWARE RESOURCE MANAGEMENT FOR CLOUD PLATFORMS

A Dissertation Presented

by

XIN HE

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2016

College of Information & Computer Sciences

APPLICATION-AWARE RESOURCE MANAGEMENT FOR CLOUD PLATFORMS

A Dissertation Presented

by

XIN HE

Approved as to style and content by:

Prashant Shenoy, Chair

Don Towsley, Member

Ramesh Sitaraman, Member

David Irwin, Member

James Allan, Chair
College of Information & Computer Sciences

To my wife Lele Chen.

ACKNOWLEDGMENTS

Doing this Ph.D. in UMass Amherst has been a wonderful experience for me. I really appreciate a number of people who have helped me during this five years here.

First and foremost I would like to thank my advisor Professor Prashant Shenoy for his guidance and patience during our five-year long collaboration. He taught me how to think of a problem, solve it using my skills and present my work to people. Whenever I met a barrier during my research, he was always patient to help me to pass it. This thesis would not have been completed without him. The way he faces difficulties and treats others will benefit me all through my life.

I feel lucky to work in such a good department and meet so many good professors. I am particularly grateful to the members of my thesis committee Professors Don Towsley, Ramesh Sitaraman and David Irwin. I learned a lot from them by taking their courses and collaborating with them. I would like to appreciate them for taking interest in my work and providing me with valuable comments and new perspectives on my research.

I have collaborated with researchers outside of UMass during my internships at NEC Labs and IBM. I would like to thank Pengcheng Xiong and Hakan Hacigumus for mentoring me during my two internships at NEC Labs. We worked closely on SDNs and big data platforms which then became an important part of my thesis. I have had extensive collaborations with Erich Nahum from IBM Research, and I am grateful for his support and for sharing his time and ideas with me.

I would like to thank other members of the LASS group, particularly Emmanuel Cecchet, Tian Guo, Prateek Sharma, Stephen Lee and Srinivasan Iyengar, who collaborated with me on research and provided suggestions when I met problems both in my research and in my life.

I am grateful to Karren Sacco for helping me to register conferences and arrange trips and to Leeanne Leclerc for managing my paperworks and answering my questions.

I would like to thank my three undergraduate friends: Gang Hu, Xinhao Yuan and Wenyi Huang who are also in the United States and pursuing Ph.D. in Computer Science. I enjoyed sharing my experience and thoughts with them and listening their opinions and comments.

I made many friends during my stay at Amherst and they have all enriched my life in one way or another. I would like to thank Haitian Yue, Qianwei Tang, Yang Meng, Linxiao Jin and Lichao Zhang who were my roommates and made me have a comfortable and warm home in Amherst. And I am grateful to Xiaojian Wu, Pengyu Zhang and Haopeng Zhang who invited me to their basketball group and organized basketball games on weekends.

Lastly, I would like to thank my family who have influenced me the most and were always there to support me. My parents (Xiaowen He and Shaoping Zou) encouraged me to come abroad to pursue knowledge and taught me how to be a good person. My wife Lele Chen has been accompanying with me since my high school. We shared opinions with each other, listened each other's troubles and went through all kinds of problems together. Whenever I made a decision, she always supported me and encouraged me to pursue my dream. My family has always been the source of my power and strength. I dedicate this thesis to them.

ABSTRACT

APPLICATION-AWARE RESOURCE MANAGEMENT FOR CLOUD PLATFORMS

SEPTEMBER 2016

XIN HE

B.E., TSINGHUA UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant Shenoy

Cloud computing has become increasingly popular in recent years. The benefits of cloud platforms include ease of application deployment, a pay-as-you-go model, and the ability to scale resources up or down based on an application's workload. Today's cloud platforms are being used to host increasingly complex distributed and parallel applications. The main premise of this thesis is that application-aware resource management techniques are better suited for distributed cloud applications over a systems-level one-size-fits-all approach. In this thesis, I study the cloud-based resource management techniques with a particular emphasis on how application-aware approaches can be used to improve system resource utilization and enhance applications' performance and cost.

I first study always-on interactive applications that run on transient cloud servers such as Amazon spot instances. I show that by combining techniques like nested virtualization, live migration and lazy restoration together with intelligent bidding strategies, it is feasible

to provide high availability to such applications while significantly reducing cost. I next study how to improve performance of parallel data processing applications like Hadoop and Spark that run in the cloud. I argue that network I/O contention in Hadoop can impact application throughput and implement a collaborative application-aware network and task scheduler using software-defined networking. By combining flow scheduling with task scheduling, our system can effectively avoid network contention and improve Hadoop's performance. I then investigate similar issues in Spark and find that task scheduling is more important for Spark jobs. I propose a network-aware task scheduling method that can adaptively schedule tasks for different types of jobs without system tuning and improve Spark's performance significantly. Finally, I study how to deploy network functions in the cloud. Specifically, I focus on comparing different methods of chaining network functions. By carrying out empirical evaluation of two different deployment methods, we figure out the advantages and disadvantages of each method. Our results suggest that the tenant-centric placement provides lower latencies while service-centric approach is more flexible for reconfiguration and capacity scaling.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Background and Motivations

Cloud platforms have been increasingly popular for hosting applications in recent years. The pay-as-you-go model and on-demand resource allocation abilities save customers' time and money to build IT infrastructure themselves and provide a more cost-effective way for them to run applications and deploy services. Currently, there are multiple categories in cloud computing: Infrastructure-as-a-Service (IaaS), Platform-as-a-service (PaaS) and Software-as-a-Service (SaaS). In an IaaS cloud, customers can share computing, storage and network resources in large data centers provided by cloud providers, which increases the effectiveness of the shared resources and reduces the management effort of customers. Users can deploy all kinds of applications and enjoy the low cost and high variety of resource choices in the cloud. PaaS provides a platform for users to develop and deploy their applications where users do not need to configure the development environment. SaaS uses the web to deliver applications that are managed by a third-party vendor and users do not need to install or maintain the software locally. Given the benefits provided by clouds, more and more individuals and enterprises are moving their applications from traditional individual PCs and in-house data centers to the cloud.

The first challenge is that there is no one-for-all solution for all cloud applications because they have different characteristics and resource demand varies even for the same application. For example, online web services are generally CPU intensive when they process queries from users, while most MapReduce jobs running in Hadoop and Spark are usually data intensive. But online video web services are data intensive and requires

high I/O throughput, while some MapReduce jobs like K-Means and Pagerank are CPU intensive and usually limited by CPU capacity. Therefore, mechanisms that can benefit one type of applications may not be helpful for others. Thus we need to study resource management in an application-aware fashion.

The second challenge is how to choose the best deployment method for a specific application. As we know, customers pay for specific CPU, memory, storage and network resources that they use in the cloud. Cloud platforms provide servers of differetn types to satisfy different applications' demand. For example, in Amazon EC2, there are general purpose instances as well as compute optimized, memory optimized and storage optimized instances. There are also low-cost transient servers that enable cloud providers to sell the idle servers during the off-peak periods or periods of low demand. It's important to know the tradeoffs among these choices and which type or what combinations of multiple types should be chosen.

The last challenge is, given specific resources, how can we make the resource manager understand application demands and use this information to achieve better performance. For example, in MapReduce clusters, network contention may happen because MapReduce doesn't know the underneath network conditions while network is unaware of MapReduce's demand. Hence we want to investigate these problems in deploying applications in the cloud and enable coordination between resource manager and applications.

These problems are important for both cloud providers and customers. From cloud providers' perspective, by tackling these problems, they can provide more efficient and reliable services for users. From customers' perspective, by understanding their applications' demands and cloud platforms' resource management mechanisms, they can choose cheaper and more suitable cloud services for their own scenario.

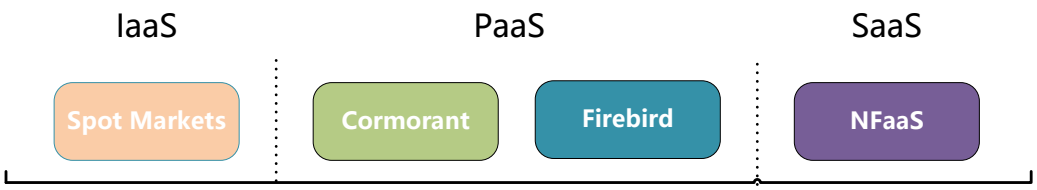


Figure 1.1. The topics and systems described in this thesis spans IaaS, PaaS and SaaS clouds.

1.2 Thesis Contributions

This thesis focuses on techniques to deployment and resource management problems in current cloud environment. I propose novel techniques that combine system-level mechanisms and application-level methods with intelligent algorithms and modeling techniques.

1.2.1 Contribution Summary

This thesis proposes application-aware techniques to improve resource management for cloud platforms. The fundamental theme of this thesis is that by making the cloud be aware of application demands and using both system-level and application-level mechanisms, we can effectively enhance the performance and reduce the cost of cloud applications. As illustrated in Figure 1.1, this thesis covers IaaS, PaaS and SaaS clouds:

- *Cost optimization in an IaaS cloud*, which focuses on using spot markets to reduce the cost of hosting always-on online services [42].
- *Performance optimization in PaaS platforms*, which focuses on improving the performance of two big data platforms – Hadoop and Spark by making network and applications collaborate with each other [83, 43].
- *Network Function Virtualization in a SaaS cloud*, where I study the advantages and disadvantages of placement strategies in a NFaaS cloud.

1.2.2 Optimizing cost of online services using spot markets

The use of cloud servers to host modern Internet-based services is becoming increasingly common. Today's cloud platforms offer a choice of server types, including non-revocable on-demand servers and cheaper but revocable spot servers. A service provider requiring servers can bid in the spot market where the price of a spot server changes dynamically according to current supply and demand for cloud resources. Spot servers are usually cheap, but can be revoked by the cloud provider when cloud resources are scarce. While it is well-known that spot servers can reduce the cost of performing time-flexible interruption-tolerant tasks, we explore the novel possibility of using spot servers for reducing the cost of hosting an Internet-based service such as an e-commerce site that must *always* be on and the penalty for service unavailability is high.

In Chapter 3, I show that by using the spot markets, it is feasible to host an always-on Internet-based service using dedicated revocable servers and achieve significant savings. I propose a cloud scheduler that reduces the cost by intelligently bidding for spot servers. Further, the scheduler uses novel VM migration mechanisms to quickly migrate the service between spot servers and on-demand servers to avoid potential service disruptions due to spot server revocations by the cloud provider.

1.2.3 Optimizing performance of big data platforms

Hadoop and Spark are popular choices for executing big data workloads over large datasets on clusters of commodity machines. Due to the distributed nature of such applications, network resource bottlenecks can adversely affect performance, especially when multiple applications share the network. Fortunately, the emergence of software-defined networking (SDN) is removing the barriers to cooperation between cluster computing platforms and the network. To explore this opportunity, I focus on how we can use the capabilities of a SDN to create a more collaborative relationship between Hadoop/Spark and the network underneath.

In Chapter 4, I present techniques to avoid network contention and improve Hadoop's performance when there is neighborhood traffic in data center. I use a SDN controller to gather global network information and control the path of network flows. Then I modify the task scheduler of Hadoop to enable it to communicate with SDN controller. By combining task scheduling and flow scheduling, I implement Cormorant – a system that can efficiently optimize network bandwidth utilization in a Hadoop cluster.

Since disk I/O is generally the bottleneck of data intensive jobs, Resilient Distributed Datasets (RDDs) was proposed to provide stable in-memory data storage and implemented in Apache Spark [88]. Since local I/O is improved in Spark, network I/O may be the bottleneck even in dedicated networks.

In Chapter 5, I first discuss existing task schedulers and why network congestion can occur in the presence of background applications. Then I propose a network-aware scheduling method that adaptively schedules tasks according to network conditions and task demands. I implement this method in Firebird – a Spark-based system running on SDNs. Compared with Spark, this system can achieve good performance in different network conditions for both data intensive and CPU intensive jobs.

1.2.4 Placement of virtualized network functions in the cloud

Network functions virtualization (NFV) offers a new way to design, deploy and manage networking services. NFV decouples the network functions, such as network address translation (NAT), firewall, intrusion detection and web caching from proprietary hardware appliances so they can run in software. A virtualized network function (VNF) may consist of one or more virtual/physical machines running different software and processes, on top of standard high-volume servers or cloud computing infrastructure.

In Chapter 6, I discuss how multiple virtualized network functions should be deployed in cloud environments, which is also called Network-Functions-as-a-Service (NFaaS). In this component of my thesis, I compare two methods of chaining network functions by

performing a series of experiments: tenant-centric and service-centric. Our results enable a cloud provider to understand the tradeoffs in deploying network functions and make judicious deployment decisions.

1.3 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 provides background on cloud computing, cloud-based applications and related resource management techniques. Chapter 3 describes how to host always-on interactive applications using cloud spot markets. Chapter 4 describes the challenges of deploying Hadoop cluster in networks shared with other applications and propose Cormorant that makes network and Hadoop collaborate with each other to achieve better performance. In Chapter 5, I discuss the resource management problems in Spark and propose Firebird that implements a novel network-aware task scheduling method which fits different types of jobs in different network conditions. In Chapter 6, I discuss how to deploy network functions in cloud environment and present two different placement strategies for deploying them. Finally, Chapter 7 summarizes my thesis contributions and discusses future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter presents background material on cloud platforms, virtualization techniques and data center networks to set the context for the rest of this thesis. More detailed related work is also discussed in each chapter.

2.1 Cloud Platforms

Cloud platforms are platforms that manage shared resources in a centralized manner and provides them to different users on demand. There are different types of clouds depending on what kind of resources are offered, for example, infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). Figure 2.1 shows the difference between these three clouds.

An IaaS cloud is a virtualized data center where the cloud provider allocates virtual machines (also referred to as virtual servers) to customers using the underlying physical servers[1]. Users can run their own unmodified applications and processes on these servers. Currently there are many infrastructure cloud platforms such as Amazon's EC2 [2], Microsoft Azure [3] and Google Compute Engine [4]. An infrastructure cloud typically supports different types of virtual machines that vary in their hardware configurations—for instance, Amazon's EC2 cloud supports over a dozen different virtual server configurations that differ in the amount of CPU, memory, disk and network allocations [2]. When using an infrastructure cloud, customers can request cloud servers on demand and the cost

[1]Some cloud platforms like IBM's Softlayer [1] also directly provide physical servers to users

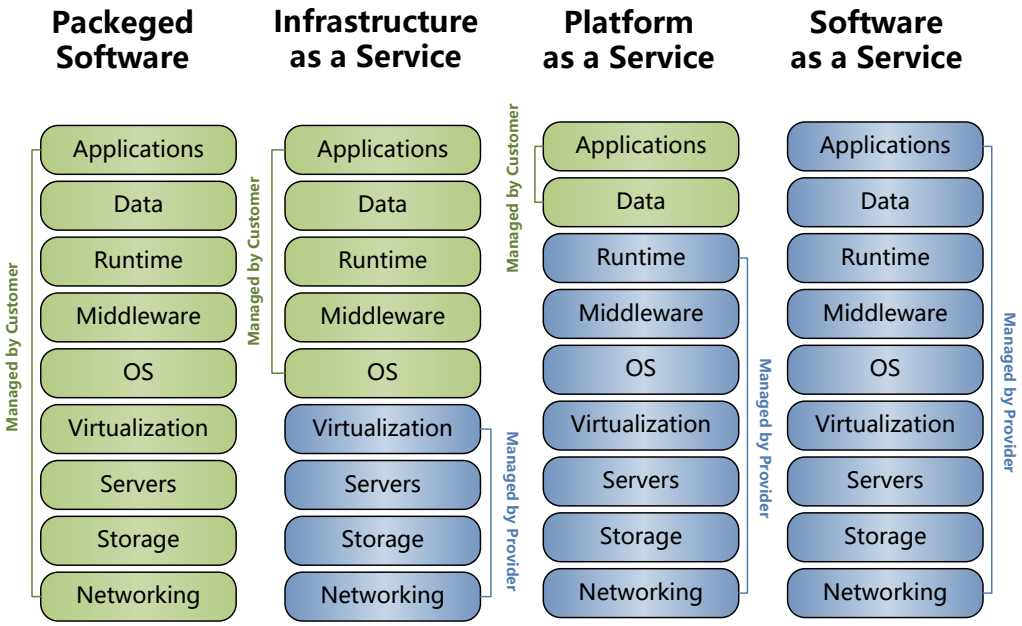


Figure 2.1. Responsibilities of different types of cloud

of a cloud server depends on the chosen configuration and is billed based on usage time. Besides providing on-demand servers, cloud providers also provide transient servers, which are significantly cheaper than on-demand servers but may be revoked at any time. These transient servers exist because they enable cloud providers to sell unused server capacity during off-peak periods. Amazon has been offering transient servers in the form of spot servers since 2009 [5]. Spot servers have a variable price determined by market conditions of the cloud such as supply and demand. By setting up a bid price that they can afford, customers can request a spot server and use it when the spot price is lower than his bid price. The spot server is revoked when the spot price goes above his bidding. Google introduced another form of transient server called preemptible VM in 2015 [6]. In contrast to an Amazon's spot server, a preemptible VM has a fixed price. A preemptible VM has a maximum life time for at most 24 hours and may also be revoked at any time.

In contrast to an IaaS cloud, a PaaS cloud doesn't provide the entire infrastructure to users. Instead, PaaS cloud only provides a platform and key services for their customers to build, run and manage applications. PaaS saves the time of managing and configuring development environment such as runtime, middleware, operating system, virtualization, servers, storage and networking. Most PaaS platforms are geared toward software development. For example, Google App Engine [7] and IBM Bluemix [8] support multiple languages and services to build and host web applications on the cloud. Altiscale [9] and Amazon EWR [10] provide MapReduce platforms such as Hadoop and Spark for developing and running big data applications.

Finally, SaaS applications are hosted in the cloud and can be accessed by users remotely. SaaS removes the need for users to install and run applications on their own computers. This eliminates the expense of hardware purchasing and maintenance, as well as software licensing and support. Generally, customers pay for this service on a monthly basis using a pay-as-you-go model. There are many common SaaS applications such as GMail which is a web based email service and Google Drive which allows users to edit their documents online and store them in the cloud.

This thesis spans all three types of cloud. Chapter 3 discusses how to deploy always-on services on spot instances in IaaS cloud. Chapter 4 and Chapter 5 optimize big data platforms which forms PaaS clouds. Chapter 6 discusses placement problems of network functions as a service (NFaaS) which are SaaS applications.

2.2 Cloud Data Center

Cloud providers supply physical or virtual machines, as well as storage and network resources for users to run their applications. These resources are provided on-demand from their large pools of servers installed in data centers. Figure 2.2 shows the architecture of a cloud data center. This section provides a brief introduction on network and virtualization techniques being used in cloud data centers.

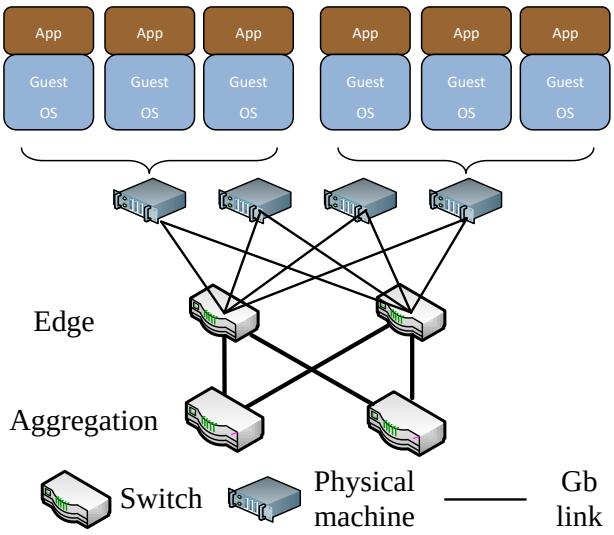


Figure 2.2. A general cloud data center topology

2.2.1 Multi-rooted tree topologies

Today's data centers contain thousands of connected servers. Recent research advocates multi-rooted tree topologies [16], where there is a large number of parallel paths between any given source and destination edge switches. One rationale for the existence of multiple paths is to achieve fault tolerance.

For example, Figure 4.1 shows two layers of switches, i.e., edge layer and aggregation layer. The edge layer switches directly connect to the servers. We can see that there are a larger number of parallel paths between any given source and destination edge switches. Note that, although high speed links are used in the network, it is currently very hard to achieve full bisection bandwidth due to the high oversubscription factor [16].

2.2.2 Software-defined network

Software-defined network (SDN) is an approach to networking that decouples the control plane from the data plane. The control plane is responsible for making decisions about where traffic is sent and the data plane forwards traffic to the selected destination. This

separation allows network administrators and application programs to manage network services through abstraction of lower level functionality by using software APIs [60].

OpenFlow is a standard communication interface among the layers of an SDN architecture, which can be thought as an enabler for SDN [56]. An OpenFlow controller communicates with an OpenFlow switch. An OpenFlow switch maintains a flow table, with each entry defining a flow as a certain set of packets by matching on 10 tuple packet information.

In this thesis, I use SDN techniques to help build a collaborative relationship between network and big data platforms, which improves network bandwidth utilization of big data clusters. From a big data platform point of view, the abstraction and SDN's control APIs allow it to (1) monitor the current status and performance of the network, and (2) modify the network with directives, for example, setting the forwarding path for non-local tasks.

2.2.3 Virtualization technologies

Server virtualization techniques are widely used in cloud computing since it provides a mechanism to partition physical resources, allowing multiple applications run on one single server with isolated computing, storage and network resources. Virtualzation provides flexible resource management mechanisms for cloud but also introduces some new challenges.

There are multiple popular virtual machine hypervisors such as Xen, KVM, VMWare annd Hyper-V [18, 50, 76, 78]. For example, Amazon EC2 uses Xen-based VMs while Microsoft Azure uses Hyper-V as its hypervisor. These hypervisors support para-virtualization or full virtualization that allows multiple operating systems to run in separate virtual machines. Since the emergence of virtual machines, there has been a lot of research on enhancing flexibility, stability and efficiency of VMs. Migration and backup are important features of VMs which increase VMs' flexibility and fault tolerance and there have been a lot of related techniques. Among these techniques, live migration [31], bounded time migration [74] and lazy restoration [44, 89, 52] are used in this thesis. For security consid-

eration, cloud providers don't provide fully access of VM hypervisor to users. Thus, users can not migrate their VMs provided by IaaS providers arbitrarily. Nested virtualization techniques address this problem by adding a nested hypervisor inside the VM provided by cloud providers and hence users can have full control of the nested VM [80].

VM placement in IaaS clouds havs been well investigated to improve data center resource utilization and reduce allocating time [25, 22, 47, 57, 37, 59]. Network-aware VM placement and application-aware VM placement methods are also proposed to provide solutions for specific demand [49, 20, 40].

In this thesis, we use advanced VM techniques to reduce downtime of running always-on services on transient servers. We also discuss how VM placement strategies impact the deployment of virtualized network functions in the cloud.

2.3 Application-aware Resource Management

Resource management is a hot research topic in the cloud computing. Application-aware resource management means the resource manager should be aware of applications' demand and make decisions accordingly. Optimizing the efficiency of IaaS cloud for specific applications has been studied by multiple effects. An energy-aware online provisioning approach for HPC applications on consolidated and virtualized computing platforms was proposed in [67]. Energy efficiency is achieved using a workload-aware dynamic provisioning mechanism. The usability of compute clouds to speed up executions of scientific workflows was investigated in [61]. Optimal resource allocation for multimedia services and applications running in the cloud based on queuing model was studied in [58]. These papers all focus on optimizing performance or resource management for specific applications in cloud environment. Their results show that application-aware mechanisms can efficiently benefit specific applications running in the cloud.

Application-aware networking in data centers has been well investigated. Wang *et al.* [79] propose application-aware networking and argue that distributed applications can

benefit from communicating their preferences to the network control-plane. Yap *et al.* have also advocated for an explicit communication channel between applications and software-defined networks, in what they called software-friendly networks [84]. Coflow [28] is a networking abstraction that expresses the communication requirements of prevalent data parallel programming paradigms. Coflows make it easier for the applications to convey their communication semantics to the network, which in turn enables the network to better optimize common communication patterns. PANE [36] proposes design, implementation, and evaluation of an API for applications to control a software-defined network. These papers focus on creating an application-aware network in system level, while in this thesis, we use both system-level and application-level techniques to create a tight relationship between MapReduce platforms and underlying networking. Our work is inspired by Xiong *et al.* who use network-aware planning to improve query processing in traditional databases [82].

CHAPTER 3

CUTTING THE COST OF HOSTING ONLINE SERVICES USING CLOUD SPOT MARKETS

Today's cloud platforms offer a variety of server types to meet the diverse needs of their hosted services. In the first part of the thesis, we consider how to use revocable spot servers to host always-on online services and reduce the cost. Since spot servers can be revoked at any time, such a revocation can potentially cause unavailability of the service for which the penalty is high. We propose to use OS and virtualization techniques to quickly move a service between revocable spot and non-revocable on-demand servers to reduce the revocation penalty. We also propose to use clever bidding strategies to reduce risk of revocations.

3.1 Background and Motivation

Cloud computing has become the paradigm of choice for building low-cost, scalable Internet-based services. Cloud providers such as Amazon AWS, Microsoft Azure [3], and Google Compute Engine [4] operate large, distributed computing infrastructures that provide computing and storage resources that can be leased by service providers. Cloud providers offer a number of benefits to service providers such as a pay-as-you-go model and flexible, on-demand allocation of resources to hosted services. A key business driver for the rapid adoption of cloud computing by service providers is the reduction in infrastructure costs. Unlike the traditional method of buying dedicated infrastructure, which must

be provisioned in advance for the peak demand, leasing cloud servers enables the service provider to scale the service as it grows over time and also exploit just-in-time allocation of capacity to handle peak workloads. Consequently, leasing cloud servers is often more economical than building dedicated infrastructure, especially for services with dynamic or growing workloads.

Today's cloud platforms offer a variety of server types to meet the diverse needs of their hosted services. Cloud servers vary in offered resource configurations, leasing cost and service model offered to customers. For instance, *on-demand* servers offer a fixed rental cost and a *non-revocable* model, where the customers pay a fixed cost and can voluntarily relinquish the server when they no longer need it. In contrast, *spot* servers offer a variable rental cost and a *revocable* model, where the customer bids an upper limit on the price they are willing to pay for a server. The cost of these spot servers fluctuates over time and an allocated spot server may be revoked by the cloud provider when its price rises above the bid price the customer is willing to pay for the server. Spot servers allow a cloud provider to offer unused server capacity at a lower price to customers, while allowing the cloud provider to revoke these servers at any time in order to fulfill requests for higher-priced on-demand servers.

Internet-based services that use the cloud vary significantly in their service requirements. At one end of the spectrum lie data-intensive cloud applications that use cloud servers to run large data analytics tasks (e.g., using MapReduce); such "big data" applications often run in batch mode with the results made available within a specified time period. As noted in Amazon's description of their cloud service [2], spot servers are a popular choice for reducing the cost of running "interruption-tolerant" and "time-flexible" tasks, such as data-intensive batch analytics and scientific computing. Indeed, there has been recent research [26] [75] [86] [54] on using spot markets to provide non-realtime services that can be performed in batch mode at a reduced cost.

At the other end of the spectrum are *always-on* Internet-based services that serve user requests in real-time. Providers of web content such as CNN, video content such as NetFlix, application portals such as Salesforce, e-commerce portals such as Walmart.com, and social networking sites such as Facebook all belong in this category. Traditionally always-on Internet-based services have relied on dedicated deployed servers owned by the service provider or a third-party content delivery network. Recently, in part to reduce costs, there has been a trend for always-on services to use non-revocable on-demand servers from the cloud markets to meet their infrastructure needs. For instance, Netflix uses Amazon's on-demand cloud services to operate their backend origin infrastructure that stores and serves out videos [11].

Conventional wisdom has held that always-on services should be hosted using either dedicated hardware or non-revocable on-demand servers and that spot servers may not be suitable for this purpose due to potential service interruptions caused by server revocations. In contrast, batch jobs such as MapReduce-style data analytics tasks that have highly elastic deadlines can exploit spot servers to lower their costs while potentially increasing completion time; such tasks can employ checkpointing methods to periodically save their state to disk and resume from the most recent checkpoint if the computation was interrupted by the revocation of spot servers. Thus, as noted by Amazon, spot servers were designed for performing time-flexible and interruption-tolerant tasks.

In this chapter, we study the feasibility and benefits of using spot servers for running always-on Internet services, which are *neither* time-flexible nor interruption-tolerant. We study how a service provider can exploit recent advances in OS and virtualization techniques such as nested virtualization and fast migration of virtual machine state to quickly move a service from spot servers to on-demand servers upon revocation and back to spot servers when they are available again. We seek to design clever bidding algorithms that exploit the low costs of spot servers and yet proactively migrate the service to on-demand instances when faced with risk of revocation. We also seek to quantify the service un-

availability due to downtimes when the cloud platform revokes spot servers. Our overall objectives are to quantify the cost savings and service unavailability and determine whether combining clever bidding and migration technique enable spot servers to be used in a novel fashion for always-on services.

3.2 Cloud Platforms and Markets

Our work targets infrastructure clouds that lease server resources to service providers. An infrastructure cloud is a virtualized data center where the cloud provider allocates virtual machines (also referred to as virtual servers) to customers using the underlying physical servers. An infrastructure cloud typically supports different types of virtual servers that vary in their hardware configurations—for instance, Amazon's EC2 cloud supports over a dozen different virtual server configurations that differ in the amount of CPU, memory, disk and network allocations. The cost of a cloud server depends on the chosen configurations and is billed based on time of usage (e.g., hourly).

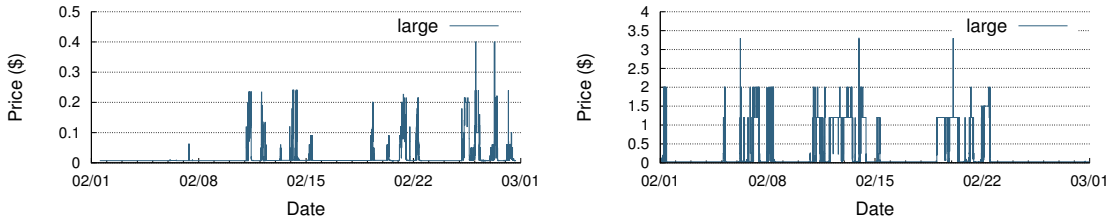
A cloud service provider can request any server type in one of two modes: on-demand and spot. On-demand cloud servers incur a fixed cost and are non-revocable. For instance, the fixed hourly price of on-demand server varies from 6 cents per hour for the small configuration to as much as $2.19 per hour for the double-extra large configuration. Importantly, once allocated, on-demand servers are non-revocable and the service provider is guaranteed availability to a server until it is no longer needed and voluntarily terminated. Since cloud platforms are provisioned with sufficient capacity to handle peak seasonal demands, they often have many unallocated and unutilized servers, which results in lost revenues due to lack of usage. Cloud providers such as Amazon have begun to offer this unused server capacity at significantly lower prices in the form of spot servers. Spot markets were first introduced by Amazon's EC2 cloud in 2009. Unlike on-demand servers, a spot server incurs a variable price and is revocable. A cloud-based service provider may request a spot server of any configuration by specifying the maximum hourly price they are willing to

pay for such a server (also known as the bid price). Since the price of spot servers varies continuously, the request is granted only if the current price is below the customer's bid price. Furthermore, if the spot price rises above the bid price at any point in the future, the server is revoked. As shown in Figure 3.1, the price of a spot server fluctuates over time based on supply-demand considerations. Prices are low when there is plenty of unused capacity in relation to demand and the price rises when there is more demand for spot servers or increased demand for on-demand servers, both of which causes the customers with the low bid prices to lose these allocated servers (which are then re-allocated to higher paying on-demand customers).

Researchers have studied the dynamics of spot markets. Each server configuration has its own spot market with fluctuating prices. The different spot markets exhibit different types of dynamics and the price can also spike up during periods of extreme scarcity. As shown in Figure 3.1, the price of a large server can be as low as few cents per hour for long periods and can spike to as much as $3/hr during high-demand periods. Other than the variable price and revocable nature, spot servers are identical to on-demand servers in all other respects such as their resource configurations. They are also billed on an hourly basis, based on the spot price (*not the bid price*) at the beginning of each hour. Partial hours are not billed if a spot server is revoked before the end of an hourly billing period. Researchers have also observed that upon being revoked, a spot server is given upto a 2 minute grace period to save all unsaved memory state to disk and execute a graceful shutdown (failing which it is forcibly terminated) [53]—while this was an "undocumented" feature of spot servers, Amazon has recently made this grace period official policy by providing an explicit two minute warning prior to revoking a spot server.

3.3 A Cloud Scheduler for Always-On Services

We design a cloud scheduler that procures servers in the cloud markets to host an always-on Internet-based service while minimizing both the cost and the unavailability of



(a) Varying spot price of a small server

(b) Varying spot price of a large server

Figure 3.1. Spot prices over a month long period in Amazon's US East-1 region. The prices across markets even within the same region are not strongly correlated, a fact we use in our multi-market bidding algorithms.

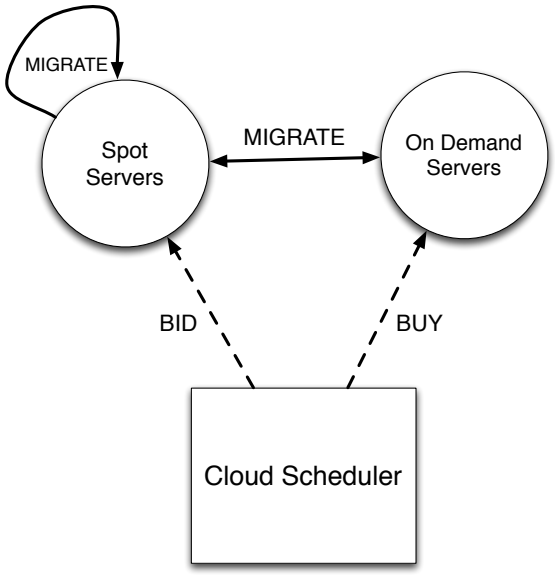


Figure 3.2. Interactions between the cloud scheduler and the cloud markets.

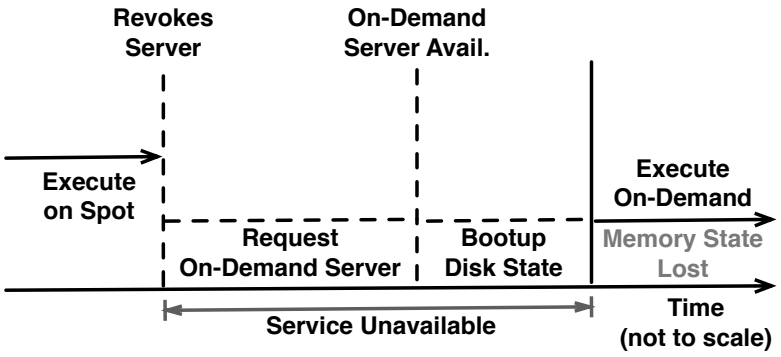


Figure 3.3. A naive approach to migrating from spot to on-demand server that involves substantial service unavailability and loss of memory state.

the service. A naive approach for using spot servers to host an always-on service is depicted in Figure 3.3. In this case, the service runs on a spot server for a period of time and the spot server is then revoked by the cloud provider, resulting in the service to be unavailable. Upon revocation, the cloud scheduler immediately requests an on-demand server to replace the revoked spot server. When the on-demand server is (eventually) allocated by the cloud provider, the Internet service is restarted on the new server. This naive baseline approach has two limitations: (i) any memory state of the spot server is lost upon revocation, and (ii) the service is unavailable from the time of revocation to the instant where the service is restarted on a new on-demand server. Note that even in this naive approach, the *disk state of the service is preserved*, since we assume that networked storage volumes are used by the service, so all data on the storage volume is preserved when the server is revoked and the volume can simply be re-attached to the new on-demand server (such networked storage volumes are referred to as EBS volumes in Amazon's EC2 cloud).

Our cloud scheduler uses a combination of intelligent bidding strategies and OS and virtualization-based techniques to address the two drawbacks of the naive approach. Our scheduler seeks to (i) eliminate any loss of memory state by migrating any such state to the new server, and (ii) reduce service unavailability or eliminate it completely in some scenarios. Figure 3.2 provides an overview of the server transitions implemented by the cloud scheduler. We next describe the two key components of the scheduler.

3.3.1 Bidding Algorithms

The cloud scheduler's bidding algorithm seeks to achieve two goals: (i) determine what prices to bid when acquiring spot servers so as to reduce the frequency of revocations and achieve cost savings over solely using on-demand servers, and (ii) determine when to transition from spot servers to on-demand servers and vice versa. As noted earlier, when requesting a spot server, the cloud platform requires a maximum price p_b to be specified by the service provider. Since the cost of a spot server fluctuates over time based on supply

and demand considerations, this maximum price p_b, also known as the bid price, is the upper limit that the service provider is willing to pay for the spot server. Hence, when the instantaneous spot price $p_{sp}(t)$ rises above the bid price p_b, the spot server is revoked by the cloud provider.

The bidding algorithm must intelligently choose the bid price p_b to achieve its goals. In general, a higher bid price reduces the chances that the spot price will rise above the bid and reduces the chances (and frequency) of server revocation. However, there is a risk that the spot price could increase but still stay below in the bid price, resulting in more cost and lower savings when compared to a pure on-demand model. In contrast, a lower bid price increases the chances of a revocation but can also lower costs.

The cloud scheduler implements two variants of the bidding algorithm. Note that whenever the spot price rises above the price of an on-demand servers, the cost savings vanish and it is more cost-effective to transition to an on-demand server and pay the fixed on-demand price over paying a even higher spot price. In the *reactive* version, the bid price is set to the price of an on-demand server i.e., $p_b = p_{on}$, where p_{on} denotes the cost of an on-demand server. Hence, setting $p_b = p_{on}$ ensures that the cloud platform will revoke the spot server whenever the spot price increases above the on demand price–forcing a migration (transition) to an on-demand server.

An alternative approach, which we refer to the *proactive* version, the bid price is set to a value higher than the on-demand price: $p_b = k \cdot p_{on}, \quad k > 1$. In this case, the bidding algorithm continuously tracks the fluctuating spot price $p_{sp}(t)$ and whenever the spot price rises above the on-demand price, the algorithm *voluntarily and proactively* transitions to an on-demand server to pay the fixed on-demand price over paying the higher spot price. Since the migration to an on-demand server is voluntary, the cloud scheduler has more time and flexibility to make the transition, which in turn allows *service unavailability to be virtually eliminated*. Note that in the reactive approach, the transition must be made within a limited time duration before the server is revoked, while in the proactive case, the cloud

scheduler can wait until the migration has completed before relinquishing the spot server. In the extreme case of the proactive version, the bidding algorithm can bid the highest bid that is allowed by the cloud platform (e.g., a large multiple k of the on-demand price) which gives the greatest flexibility[1] Regardless of the actual bid, a large sharp spike of the spot price above the bid price will cause the spot server to be revoked by the cloud platform before the proactive algorithm can begin (or finish) its voluntary migration.

After transitioning to an on-demand server, the bidding algorithm continues to monitor the spot price $p_{sp}(t)$ and can again request a spot server when $p_{sp}(t)$ falls below the on-demand price p_{on} and initiate a reverse migration from an on-demand to the spot server; such migrations are also voluntary and can take as long as needed to migrate the service.

Thus, both the reactive and proactive version of the bidding algorithms involve the following steps:

1. *Forced Migration.* If the $p_{sp}(t) > p_b$ and the algorithm holds a spot server, then the spot instance is terminated by the cloud provider. The algorithm is forced to migrate the spot server to an on-demand server.

2. *Planned Migration.* If $p_b \geq p_{sp}(t) \geq p_{on}$ near the end of a billing period (i.e., billing hour) and the algorithm holds a spot server, it reduces cost by voluntarily migrating to an on-demand server.

3. *Reverse Migration.* If $p_{on} > p_{sp}(t)$ near the end of a billing hour and the algorithm currently holds an on-demand server, it reduces cost by re-procuring and migrating back to a spot server.

Note that planned migrations are more desirable than forced migrations, since there is more time to migrate the service in the former, resulting in less disruption to the service.

[1]Note that cloud providers do not allow an infinite bid price. The largest bid price currently allowed by Amazon is four times the on-demand price which we use in our proactive algorithm.

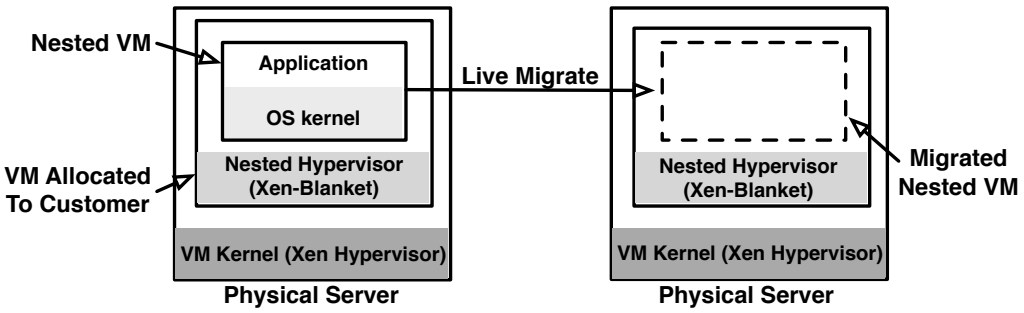


Figure 3.4. Nested virtualization and live migration of a nested virtual machine.

Whereas with a forced migration there is only a short time window before the spot server is terminated.

3.3.2 OS Mechanisms

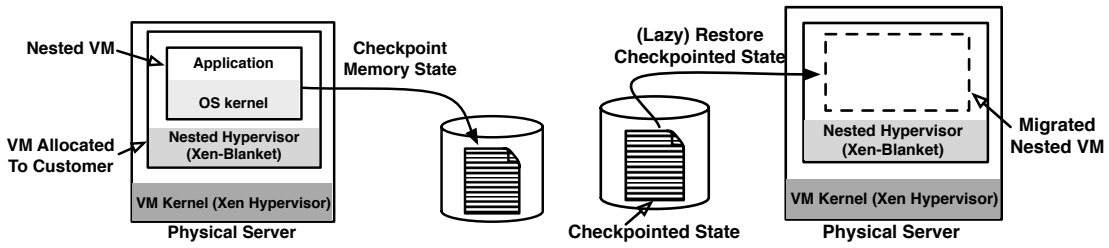
The cloud scheduler uses four well-known OS-level mechanisms to implement migrations from spot servers to on-demand servers and vice versa. While these OS-level mechanisms were proposed elsewhere, they have not been used in cloud platforms previously, nor has this novel combination been studied previously in the cloud context.

We assume that migrations from spot to on-demand servers and back is implemented at the virtual machine level. Virtual machine migration is transparent to the OS and the applications and does not require *any* modifications to either, allowing the technique to apply to all applications (here, cloud services) and operating systems unmodified. Our cloud scheduler employs three variants of virtual machine migration, as described below, to achieve different goals.

Nested virtualization. All common cloud platforms are virtualized and allocate virtual servers in the form of virtual machines. As a result, migration of virtual machines (VMs) is feasible in cloud platforms. Unfortunately, however, today's cloud platforms do not expose migration capabilities of virtual machines to customers and retain this control for themselves. Since the ability to migrate virtual machines from spot to on-demand servers

and back is central to our approach, the cloud scheduler uses a mechanism called *nested virtualization* to achieve this goal in today's cloud platforms. Nested virtualization involves running a virtual machine *inside* another virtual machine and the application runs inside the nested virtual machine (see Figure 3.4). The advantage of nested virtualization is that it allows complete control of the nested virtual machine to the user without requiring any privileged access to the native virtual machine. Since cloud platforms allow a customer to run any OS kernel inside their virtual servers, a customer can easily run a nested virtual machine kernel, instead of a regular OS kernel, and run the second, nested VM inside the virtual server. In such a scenario, we only need to migrate the nested virtual machine from one virtual server to another (e.g., spot to on-demand) without migrating the outside virtual machines. Nested virtualization was proposed in [80] and has been implemented in Xen, a widely used open-source virtualization platform, in the form of Xen-Blanket, which is compatible with Amazon's cloud servers that also use Xen. Experiments reported in [80] show only a modest overhead due to the second nested virtualization layer.

Live migration. Live virtual machine migration is a technique where an entire virtual machine is migrated from one physical server to another while the OS and resident applications continue to execute without requiring any downtime. Live VM migration techniques were proposed over a decade ago and are now supported by most common commercial and open-source virtual machine products (e.g., VMWare, Xen) [30]. Live migration is implemented by interactively copying the memory pages of the virtual machine from the source server to a destination server while the OS and applications continue to run. Since the VM is running during this migration process, memory pages will continue to be modified. Hence, live migration operates in rounds, where each round involves sending memory pages modified since the previous round. After several round of incremental transfers, the difference between the source and destination servers shrinks, and the virtual machine is momentarily paused to send the final set of changed memory pages. The VM at the destination is resumed and the source VM is terminated. This allows the application and its



(a) Checkpoint (b) Restore

Figure 3.5. VM Memory checkpointing and restoration.

network connections to smoothly transition to the new server and no network reconfiguration is needed (the IP address remains unmodified when transferring the VM within a LAN). We note that VM migration techniques typically only transfer memory state of the virtual machine and do not transfer disk state since disk state is assumed to be stored on a network disk that can simply be re-attached to the destination server. By using nested virtualization, our cloud scheduler can live migrate nested virtual machines as shown in Figure 3.4. Further, the cloud scheduler uses techniques such as virtual private cloud [2] that allow customer control over the assignment of IP addresses to one's virtual machines to ensure that the address assigned to the nested VM on a spot server can be transparently reassigned to an on-demand server upon migration and vice versa.

Bounded memory checkpointing. Live migration is an attractive and straightforward method for transparently migrating a virtual machine from one server to another. The main limitation of this approach, however, is the potential large latency involved in memory copying, especially for larger server configurations that have substantial amount of memory (e.g., tens of GB of RAM). While these larger latencies can be easily accommodated during planned or reverse migrations initiated by the bidding algorithm, where there is flexibility in determining how much in advance to start the migration process, they may not be feasible for forced migrations. When a cloud platform revokes a spot server, there is a limited window of time to execute a graceful shutdown and this period may not suffice to live

migrate a nested VM with large amounts of memory. Consequently a different approach is needed to quickly save memory state during forced migrations.

Memory checkpointing of a virtual machine in the form of *suspend-resume* involves writing out the entire memory contents of a VM to disk prior to suspending the virtual machine, and then resuming it at a later time by loading the checkpointed memory state (see Figure 3.5). Such suspend-resume support is already built into all virtual machine products and is an alternate approach to live migration for capturing memory state and resuming the VM on a different machine. However, writing of the memory contents of a virtual machine to a network disk can also involve a substantial latency for servers with substantial amounts of RAM. Fortunately, we need not wait to initiate memory checkpointing until a revocation is in progress and can instead checkpoint memory periodically in the background. In this case, memory contents are asynchronously written to disk in the background periodically and upon a renovation, only the incremental modified memory state since the most recent checkpoint needs to be written out, making the capturing of memory state very quick (and well suited to the limited time window available during a forced migration). Our cloud scheduler uses a recently proposed checkpointing technique called Yank [74] that provides an *upper bound* on the time needed to complement a checkpoint – given a bound τ, it dynamically adjusts the periodicity of the background checkpointing process to ensure that the incremental state does not exceed a threshold and can always be written out within the bound of τ seconds. By setting the bound to a small time window allowed by the cloud platform during a revocation, our cloud scheduler can ensure that all of the memory contents are safely captured to disk and the nested VM can be resumed on a different cloud server; we assume that network disks are used for the purpose of capturing memory state so that the disk is still available after a spot server has terminated.

Lazy VM restore. While bounded memory checkpointing allows suspension of the VM's memory state to complete within a small, bounded time period, the resume part of the suspend-resume process can still incur a latency of tens of seconds—since it requires

reading the saved memory state from disk into RAM prior to the resumption of the nested VM. For larger cloud server configurations, this may involve reading tens of gigabytes of RAM state from disk. Hence, we employ an OS mechanism called lazy restore that substantially speeds up the resumption of a virtual machine from its saved memory state. Lazy restore [44, 89, 52] involves reading in only a small subset of the memory pages and resuming execution. The remaining memory state is read concurrently in the background as the VM executes. In the event the executing VM accesses a memory page that has not yet been read from disk, the corresponding memory page is fetched on-demand from disk (akin to how a page fault is handled in traditional operating systems). Lazy restore only requires a small fraction of the memory state to be read from disk before execution can be resumed, allowing for fast resumes and very small downtimes. Of course a downside is that the VM execution may be slower for a period of time due to the page faults that are seen while the remaining memory state is being loaded from disk in the background.

This novel combination of the four OS mechanisms makes it feasible to implement forced, planned and reverse migrations of our bidding algorithm in today's cloud platforms.

3.4 Evaluation of the Cloud Scheduler

We use empirical micro-benchmark measurements on Amazon's EC2 cloud as well as simulations seeded by Amazon's spot price traces to drive our evaluation. We evaluate the bidding algorithms and migration mechanisms employed by our cloud scheduler in three different scenarios. The simplest is the *single-region single-market* scenario where the cloud scheduler procures servers of a single size from a single spot market at a single geographical region, migrating to on-demand servers of the same size when necessary. More complex is the *single-region multi-market* scenario where the cloud scheduler has the option to buy servers of different sizes from different spot markets, though all of the servers are hosted at a single region. The most complex situation is the *multi-region multi-market* scenario where the cloud scheduler can procure servers of different sizes from different

Instance type	US east (s)	US west (s)	EU west (s)
On-demand	94.85	93.63	98.08
Spot	281.47	219.77	233.37

Table 3.1. Average Start-up Time of On-demand and Spot instances

	Live migrate (s)	Memory checkpointing (s)	Disk copy (s)
Inside US East	58.5	28.9	–
Inside US West	57.1	28.8	–
Inside EU West	58.2	28.05	–
US East to US West	73.7	–	122.4
US East to EU West	74.6	–	140.5
US West to EU West	140.2	–	171.6

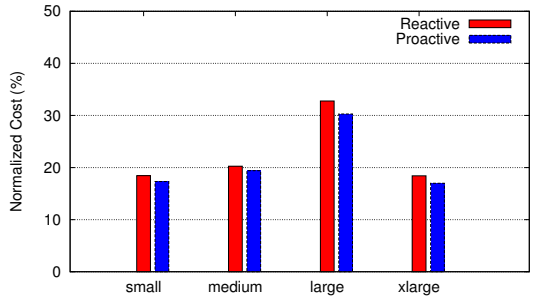
Table 3.2. Overhead of migration mechanisms.

markets across any of the regions offered by the cloud provider. Intuitively, the attainable cost reduction should increase with each scenario since the cloud scheduler has more options for lowering the cost. However, the migration becomes more complex—a multi-market strategy involves packing multiple nested VMs onto a larger spot or on-demand server, while multi-region involves migration across regions that could be more complex and expensive.[2]

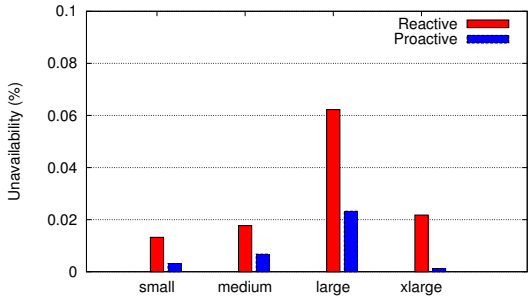
3.4.1 Microbenchmarks

We ran the XenBlanket nested hypervisor on Amazon's cloud servers and conduct a series of micro benchmark measurements to capture the overheads of various migration mechanisms; these measured values are then used to parameterize subsequent simulation experiments. We first measure the latency to allocate an on-demand and spot server of different sizes in different regions. Table 3.1 shows the mean measured values across multiple runs and shows that typical allocation times are around 1.5 minutes for an on demand

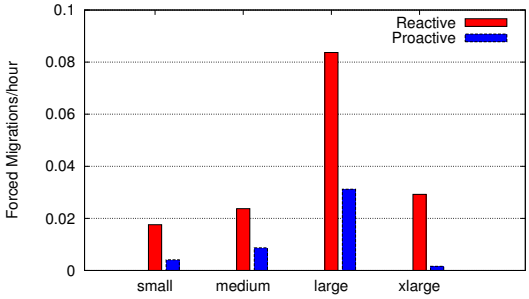
[2]WAN VM migration across regions involves additional network reconfigurations [81] that also add to the overheads.



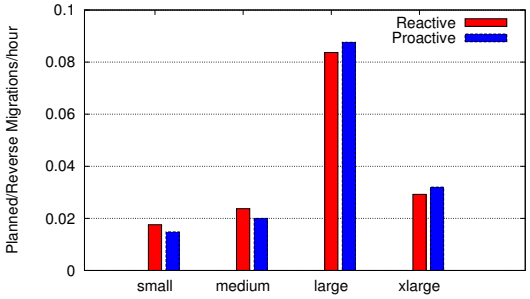
(a) Both proactive and reactive provide significantly smaller cost than the baseline.



(b) Proactive mitigates service unavailability better than reactive.



(c) Proactive has a smaller number of forced migrations per server per hour.



(d) Proactive and reactive have similar number of planned/reverse migrations per server per hour.

Figure 3.6. A comparison of proactive versus reactive bidding algorithms.

server and between 3.5 to 4.5 minutes for spot servers. Next, we measure the latency to live migrate a nested VM with 2GB of RAM within and across regions. Table 3.2 shows that live migration latency is around 1 minute for intra-data center migration and varies from 73 to 140 seconds for cross region migrations. While LAN migration can use networked storage and does not require disk state transfers, cross-region WAN migration does and the table shows that cross-datacenter copying of disk state take between 2 to 3 minutes per GB of disk state. We also benchmark the latency of memory checkpointing, which involve writing memory pages sequentially to a network attached disk and observe a latency of 28s per GB of memory state (VM restoration latencies which read this data back from disk are similar). In contrast, we assume a lazy restoration latency of 20s, which is independent of memory size, based on measurements reported in [44].

In our microbenchmarks, we conducted multiple runs to obtain estimated startup times and migration times. In addition to these measured parameters, we also gathered published

spot price history for Amazon's spot servers. In our simulations, we sampled the empirically observed distributions and used a different sample for each simulation run. We report results for small, medium, large and xlarge spot servers at four Amazon regions: US East 1A, US East 1B, US West 1A and Europe West 1A.

3.4.2 Proactive versus Reactive Bidding

We start with the simplest scenario where our bidding algorithm described in Section 3.3.1 uses a single market (either small, medium, or large) in a single region (us-east). The bidding algorithm alternately uses servers procured in the chosen spot market in the us-east region or an on-demand server obtainable at the same region. We study the two variants of the bidding algorithm described earlier, proactive and reactive, both using bounded checkpointing with lazy restore for migration.[3] To estimate cost savings from using the spot market, we use the cost of using only on-demand servers to host the service as the baseline. As shown in Figure 3.6(a), both proactive and reactive approaches show a significant reduction in cost achieving 17% to 33% of the baseline cost of not using the spot market at all. However, proactive does achieve a slightly smaller cost than reactive in all three markets. More importantly, the proactive algorithm achieves significantly less service unavailability than the reactive algorithm in *all* markets (cf. Figure 3.6(b)). Specifically, the unavailability of the proactive algorithm is smaller by a factor that ranges from 2.5 to 18 when compared to the reactive algorithm. The reason is that the proactive algorithm significantly reduces the number of forced migrations in comparison with the reactive algorithm as shown in Figure 3.6(c). Specifically, the proactive algorithm migrates its servers from the spot market to the on-demand market before it is forced to do so, giving it more time to perform the migration, in turn reducing the possibility of the service being unavailable during the migration process. Figure 3.6(d) shows that proactive and reactive algorithms have similar number of planned/reverse migrations.

[3]Results for planned live migrations are similar and omitted here.

The results for other regions are also similar to what we presented above for us-east. Thus, we conclude that it is better to be proactive rather be reactive, both from the perspectives of cost and unavailability. Henceforth, we will use the proactive bidding algorithm and its variants in all our subsequent evaluations.

3.4.3 Evaluating the Migration Mechanisms

We next evaluate the efficacy of four different combinations of migration mechanisms for the proactive bidding algorithm: memory checkpointing (with standard restore), memory checkpointing with lazy restore, live migration with checkpointing and live migration with checkpointing and lazy restore. The service unavailability of each combination is shown in Figure 3.7 for small servers in the US East 1a region; we report results for normal case as well as a pessimistic case where all migration mechanisms exhibit worst case behavior. Pure checkpointing alone has the worst unavailability of 0.018% due to the long latency needed to read the save memory state from disk prior to resuming the virtual machine. The unavailability improves significantly to 0.004% when lazy restore is used to speed up the resumption of a checkpointed VM. Similarly live migration with checkpointing has higher unavailability of 0.0095% since any forced migrations employ checkpointing with its longer downtimes. The final combination of using live migration when possible, and checkpointing with lazy restore for any forced migration has the smallest unavailability of 0.002% (roughly factor of two better than checkpointing with lazy restoration alone). According to [31] and [68], in the worst case, the downtime during migration of a 4GB virtual machine can be 10s, and migration of a 2GB VM causes down time of as much as 4s. The worst case of memory restore is copying the whole memory to the new VM while restoring. In our measurement, the time to copy a 2GB disk file which is less than 120s inside a region. The pessimistic scenarios, which assume pessimistic values of a 10s outage for live migration, and 120s latency for lazy restoration, see uniformly higher unavailability for all mechanisms, with the best unavailability of 0.017% for live migration with check-

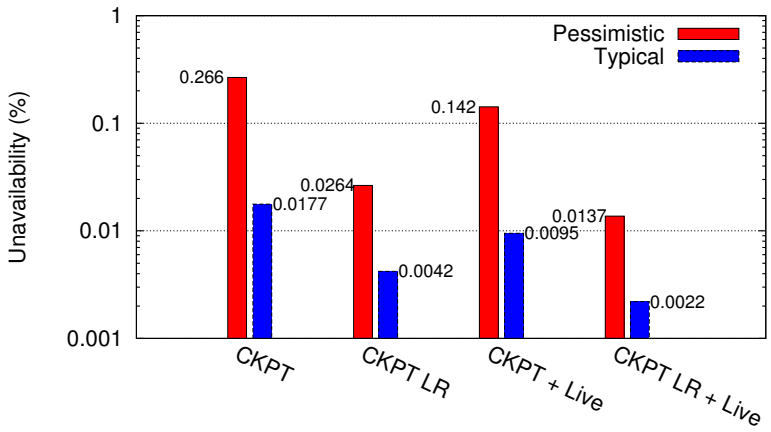


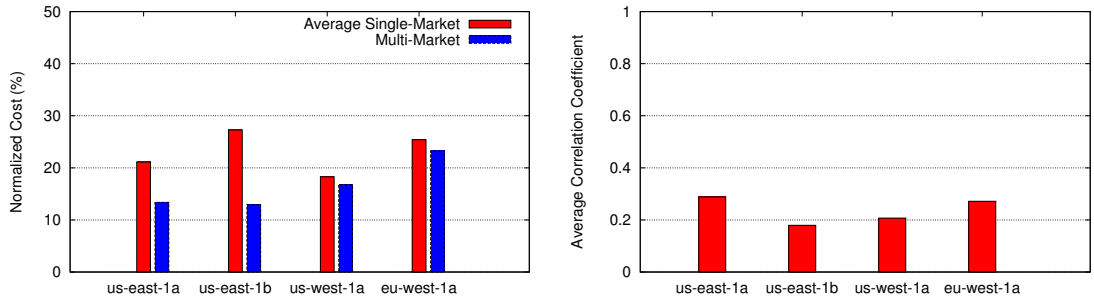
Figure 3.7. Comparison of different migration mechanisms using proactive bidding. (Unavailability percent is plotted in log-scale)

pointing and lazy restore. Thus we conclude that pure checkpointing is not desirable due to its higher unavailability, when used alone or in combination with live migration. However, when used with lazy restoration, the technique provides unavailability values that make it feasible for always-on services, with live migration further halving the unavailability of the service.

3.4.4 Multi-Market Bidding Strategies

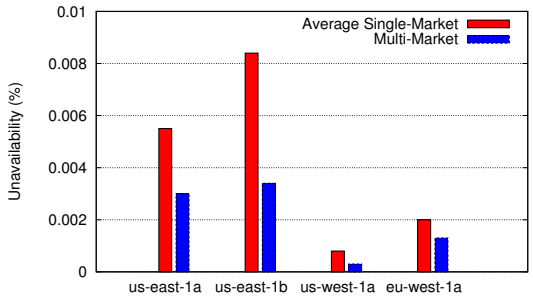
We study the benefit of bidding in multiple spot markets in comparison with bidding in a single spot market within a given region. The intuitive reason why multiple markets can decrease the cost is that when one spot market has a price rise the other markets in the same region *may* not experience a similar rise. So, our cloud scheduler can move its servers from the pricier spot market to one of the cheaper ones.

We modified our proactive bidding algorithm of Section 3.3.1 to use multiple markets within the same region as follows. In the planned migration step, we look to see if there is *any* spot market in the same region that has a cheaper price than the on-demand price. If so, the algorithm bids in the cheapest available spot market in that region and migrates the spot server to that market. If not, the algorithm migrates the spot server to the on-demand server



(a) Bidding in multiple markets decreases the cost in comparison with single-market schemes.

(b) The price correlation between the different spot markets within a region is low.



(c) Bidding in multiple markets decreases unavailability in comparison to single-market schemes.

Figure 3.8. The benefits of bidding in multiple markets within the same region.

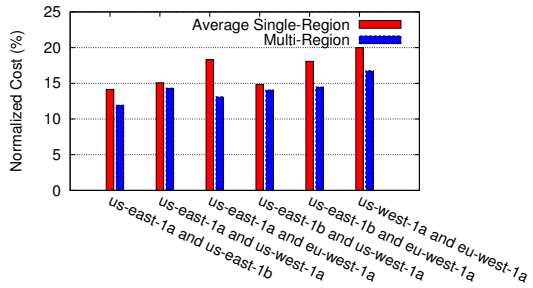
as it is currently cheaper than any of the spot servers. The forced and reverse migration steps work the same as before.

We evaluated our multi-market bidding algorithm in all regions and show the results in Figure 3.8. As shown in Figure 3.8(a), a multi-market scheme was able to reduce the cost by 8% for us-west-1a to 52% for us-east-1b in comparison with the average cost of the single-market schemes in those regions. The reason for the reduction is that price correlation between the different markets is low as shown in Figure 3.8(b), i.e., when the price spikes up in one of the spot markets, another of the spot markets in the same region may not have an equivalent increase. Our multi-market bidding algorithm exploits the lack of correlation to move servers from the costlier to the cheaper spot market.

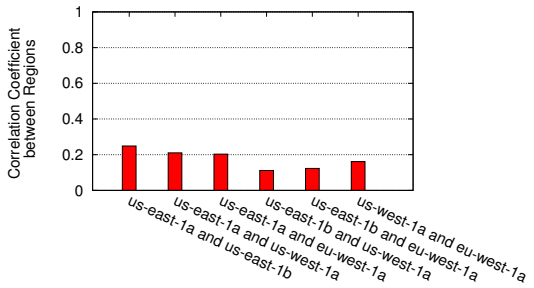
3.4.5 Multi-Region Bidding Strategies

We study multi-region bidding algorithms that can move servers between spot markets both within a given region as well as *across* different regions. Our multi-region algorithm is identical to the multi-market algorithm described in Section 3.4.4 except that the algorithm looks for the cheapest market both within and across regions for migration. We evaluate our multi-region bidding algorithm on pairs of regions and we show the results in Figure 3.9. To normalize the cost achieved by our multi-region algorithm, we use the lowest on-demand cost available in the two allowable regions as the baseline. As we show in Figure 3.9(a), our multi-region strategy achieves 12-17% of the baseline cost, resulting in a significant cost reduction in comparison to the baseline of not using the spot markets at all. Further, our multi-region algorithm results in a normalized cost that is 5-28% smaller than the average cost achieved by the single-region bidding algorithm operating in each of the two regions. The reason for the additional cost savings is that the prices across two regions have a low correlation as shown in Figure 3.9(b). Therefore, when the spot price increases in one region, our multi-region algorithm is able find cheaper prices in the other region.

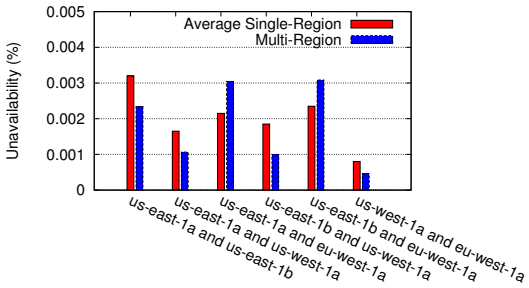
However, service unavailability can actually *increase* in some cases with multi-region bidding as can be seen in Figure 3.9(c). The reason is that regions such as us-east-1a and us-east-1b that tend to have cheaper prices, also have greater variability in those prices (cf. Figure 3.10). Whereas the eu-west region tends to be more expensive but the prices are more stable. Since our multi-region bidding algorithm migrates its servers to spot markets primarily based on a lower price, it can sometimes migrate to lower cost regions (such as us-east) with more volatile prices. Markets with larger price volatility can cause more migrations as the prices fluctuate making these markets more expensive at times than the other markets. The increased migration causes more unavailability. Bidding algorithms that also consider price stability instead of greedily opting for the cheapest price is a topic for future research.



(a) Bidding in multiple regions decreases the cost in comparison with single-region schemes.



(b) The price correlation across regions is low, enabling the multi-region algorithm to avoid price hikes by switching to a cheaper alternate region.



(c) Multi-region unavailability can increase in cases when the lower-priced markets such as us-east happen to also be less stable.

Figure 3.9. A comparison of multi-region versus single-region bidding algorithms. Both algorithms bid in multiple markets within their allowable regions.

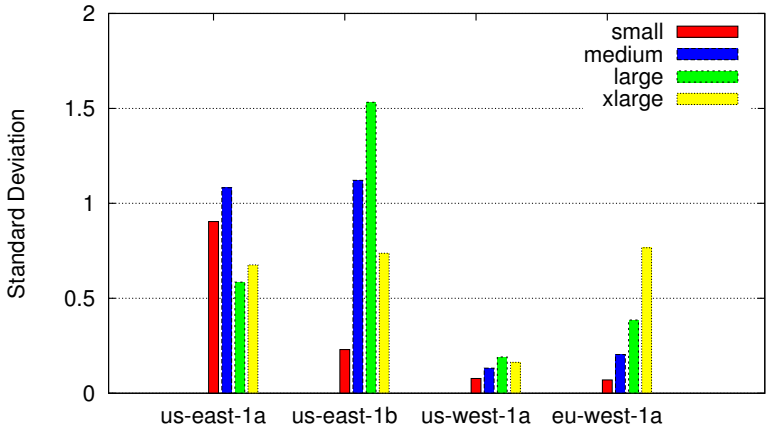
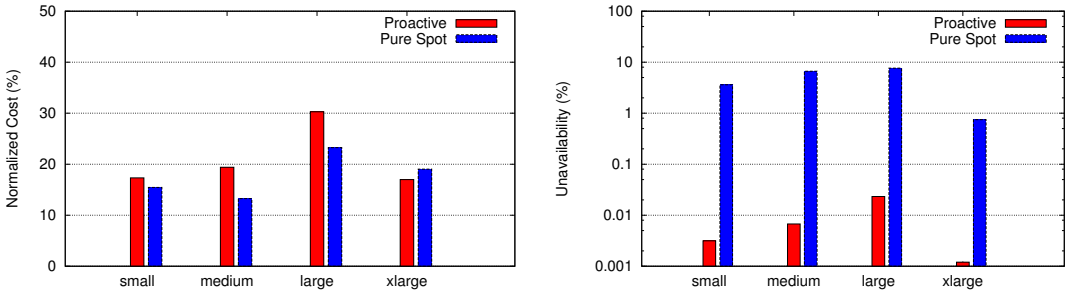


Figure 3.10. The prices of us-east are more variable than us-west or eu-west.



(a) Using pure spot instances can slightly reduce the total cost

(b) Using pure spot instances largely increases unavailability

Figure 3.11. A comparison of proactive method versus using only spot servers.

3.5 Cost and Availability Analysis

In the previous section, we show that by using nested VMs and migrating between on-demand instances and spot instances, we can achieve a significant reduction in cost over using on-demand instances alone. In this section, we go a step further to show the advantage of our method over current spot market.

Figure 3.11 compares our proactive method with using spot instances alone. We find that although using spot instances reduce cost in some markets, its availability is quite bad. In small, medium and large markets, unavailability can exceed 1% which is not acceptable for always-on internet services. Further, since the price may be over bid limit for a long period, services can be unavailable for hours or even days. Hence, using spot instances alone are not a good choice for hosting always-on internet services, as conventional wisdom has held.

As table 3.3 shows, our method combines the advantage of on-demand and spot market and provides a solution with low cost and high availability to host always-on internet services in current cloud platforms.

	Cost	Availability
Only On-demand	High	High
Only Spot	Low	Low
Using migration mechanisms	Low	High

Table 3.3. By using a combination of on-demand and spot servers, we can achieve low cost and high availability for online services

	Amazon VM (Mbps)	Nested VM (Mbps)
Network TX	304	304
Network RX	316	314
Disk Read	304.6	297.6
Disk Write	280.4	274.2

Table 3.4. Network and Disk I/O performance of nested VMs is comparable to Amazon's native VMs

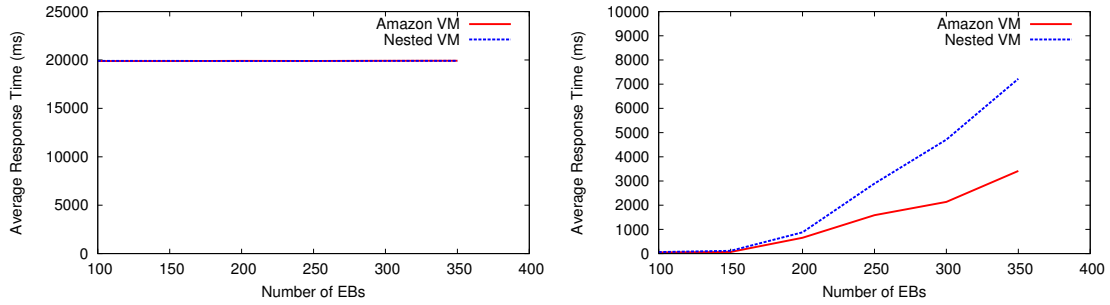
3.6 Impact of System Performance on Cost

Although nested VMs on spot instances provide good savings, nested virtualization can also impose system performance overheads. In this section, we quantify these system performance overheads and study their impact on the eventual cost savings.

3.6.1 Disk and Network I/O Overheads

Since we use a second hypervisor to host our nested VMs, our system will incur performance overheads. We compare the system performance of Amazon VMs and nested VMs (using the xen-blanket nested hypervisor). In our experiments, we use Amazon EC2 m3.medium VMs which has 1 virtual CPU and 3.75 GB memory, using HVM virtualization instances and Elastic Block Store (EBS). When creating a nested VM, we only distribute 3 GB memory to it because dom0 needs some memory to hold its service. Network address translation (NAT) is used to provide transparent network access to the nested VMs.

We first measure the network I/O and disk I/O overhead. We use iperf to get a measurement of network throughput. From Table 3.4 we can see that both the transmitting rate and



(a) If browsers fetch images while browsing, nested VM's performance is no worse than Amazon VM

(b) If browsers don't fetch images while browsing, nested VM's performance is upto 50% worse than Amazon VM

Figure 3.12. The overhead of nested VM depends on the type of service it provides

receiving rate of nested VM matches the throughput of Amazon VM. Then we ran *dd* to measure disk I/O. System caches at all layers were flushed before reading and writing 2GB of data from the root file system. Table 3.4 shows that disk I/O performance is only degraded by 2%. These results show that disk and network I/O performance of nested virtual machine instances is close to Amazon's native VMs.

3.6.2 CPU Overhead Benchmarking

The original Xen-blanket paper [80] provided detailed results on the CPU overheads imposed by the Xen-blanket nested hypervisor. We use TPC-W as an example benchmark application to verify their results in Amazon's EC2 cloud. TPC-W is a web benchmark that emulates an online e-commerce store. We use a Java servlets-based multi-tiered configuration of the TPC-W shopping website. Our experiment injects an "ordering workload" where 50% of the clients only browse the website and the remaining 50% execute order transactions. TPC-W allows us to measure the influence of the extra xen-blanket hypervisor on the response time perceived by the clients of an interactive web application.

We perform the above experiment with two common configurations: 1) browsers fetch images from the server while browsing 2) browsers don't fetch images from the server while browsing. The first configuration emulates a case where the entire website, including

the images, is served by our server VMs. The second configuration emulates a case where only the base web page is served by our server VMs and the embedded images are cached and served by a third-party content delivery service. Figure 3.12 shows the response time under a varying load imposed by different number of emulated browsers. Figure 3.12(a) shows the result under the first configuration. We can see that nested VMs can achieve similar performance as Amazon VMs. This is because when browsers get images from the server, the benchmark is I/O bound and xen-blanket can provide efficient I/O. Figure 3.12(b) gives the result under the second CPU-intensive configuration; in this case, the CPU overhead depends on the load and in the worst case, we see that nested VMs incur up to a 50% overhead over Amazon VMs.

From our system measurements, we observe that disk and network intensive services will see close to native performance and achieve most of the cost savings. For CPU-intensive workloads, the overheads depends on the actual load and can reduce the cost savings (since additional capacity is needed to service a particular load). In the worst case, performance may be halved, yielding actual savings of 12%-34% of the baseline cost. Of course, Xen-Blanket is a research prototype of a nested hypervisor and a commercial ested hypervisor implementation may be able to optimize the performance overhead and yield better savings.

3.7 Related Work

There has been recent research on cloud spot servers, but much of prior work has focused on interrruption-tolerant batch jobs. The use of spot servers to reduce the cost of data-intensive MapReduce batch jobs has been studied in [53] and [26]. Optimal bidding strategies that minimize completion times of short batch jobs have also been studied in [86], [72], and [75]. Checkpointing techniques for batch jobs running on spot servers were studied in [85]. In contrast, our work focuses on using spot instances of always-on services that interact with users in real-time.

Our work builds on a large body of work in virtualization techniques [18]. Live migration of virtual machines was studied in [30], while checkpointing techniques for virtual machines have been studied in [32, 74]. Nested virtualization in the context of the Xen virtual machine platform was proposed in [80]. Lazy restoration methods have been studied in [89, 44, 52]. SpotCheck [70] is a system that uses nested virtualization and migration mechanisms to manage server pools based on spot and on-demand servers. Our work assumes the presence of such system level mechanisms and examines a range of bidding and migration policies that use these mechanisms in the cloud context.

3.8 Conclusions

In this chapter we studied the efficacy of using spot servers to lower the cost of hosting always-on Internet services. We proposed a cloud scheduler that combines bidding algorithms and migration techniques to reduce, or nearly eliminate, unavailability by migrating a spot server to an on-demand server when needed. Our results demonstrated the feasibility of using our proactive approach to provide availability levels that are close to levels desirable for always-on services, at nearly one-third to one-fifth of the cost of the traditional approach of using on-demand servers. As part of future work, we plan to design more sophisticated bidding strategies that take spot price stability into account to further reduce server revocation frequency, and hence, service unavailability.

CHAPTER 4

OPTIMIZING MAPREDUCE WITH COLLABORATIVE SOFTWARE-DEFINED NETWORKING

MapReduce is a popular choice for executing analytic workloads over large datasets on clusters of commodity machines. Due to the distributed nature of such systems, network resource bottlenecks can adversely affect performance, especially when multiple applications share the network. In this chapter, we propose to improve network utilization by using the capabilities of SDN to create a collaborative relationship between MapReduce and the network underneath.

4.1 Background and Motivation

Running analytic queries on large, diverse, and ever-growing datasets, so-called big data processing, has become an essential part of business processes for enterprises. MapReduce [33] (and Hadoop as the open source version of MapReduce) has emerged as a framework for processing large amounts of structured and unstructured data in parallel across a large number of machines, in a reliable and fault-tolerant manner. However, due to the distributed nature of the framework, network bandwidth has always been a scarce resource that limits the MapReduce's performance [15]. Moreover, this problem becomes even more challenging if the network is shared with other applications as well [73].

One cause of the problem is the current separation between the decisions MapReduce and networking make with respect to resource allocation. MapReduce does not explicitly monitor underlying network status, nor does it try to modify its activities due to this status. Similarly, the networking layer does not base its resource allocation on any insight into the

specific expected behavior of a MapReduce task. As a result, when the network is shared with other applications, it simply tries to deliver equal network service to all applications. In this chapter we explore the issue of whether or not better performance can be obtained by changing the fundamental relationship between MapReduce and network routing by exploiting the cooperative capabilities offered by software-defined networking (SDN) [56, 60]. We focus on MapReduce workloads generated by Hive as representative of a widely used approach to executing decision support queries over large data sets.

Data center applications initiate connections between a diverse range of hosts and can require significant aggregate bandwidth. Data center topologies often implement a multirooted tree with increasing speed links but decreasing aggregate bandwidth moving up the hierarchy[1]. These multi-rooted trees have many paths between all pairs of hosts. A key challenge is to simultaneously and dynamically forward flows along these paths to minimize or reduce link oversubscription and to deliver acceptable aggregate bandwidth. Unfortunately, existing network forwarding protocols are optimized to select a single path for each source/destination pair in the absence of failures. Such static single-path forwarding can significantly underutilize multi-rooted trees. The state of the art forwarding in data center environments uses ECMP [45] (Equal Cost Multipath) to statically stripe flows across available paths using flow hashing. This static mapping of flows to paths does not account for either current network utilization or individual flow size.

Recently, Hedera [15] has been proposed as a dynamic flow scheduling system for generic workloads in data centers with multi-rooted tree topologies. Hedera is a substantial improvement over the network status and flow size oblivious ECMP algorithm. In view of this, we have chosen Hedera as flow scheduler for our experiments.

However, Hedera only goes part of the way to collaborative, network-aware scheduling between task managers and flow schedulers. That is, Hedera is invoked after the tasks

[1]Cisco Data Center Infrastructure 2.5 Design Guide. www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf.

have already been selected, and seeks to schedule and route the resulting flows given that the tasks selected are fixed. In this work we aim to discover if completing the transition is effective – that is, if an even tighter integration between the task manager and the flow scheduler can yield better performance.

Another important general idea in reducing the impact of bandwidth limitations in Map Reduce computations is to place jobs "close to" their data, thus reducing the amount of data that must be transferred. A relevant piece of work along these lines is Mantri [17]. Mantri can yield much better performance than location-oblivious placement of tasks; in view of this, we have implemented Mantri as our task scheduler. However, while Mantri impacts the flows that a particular application will generate, it does not manage those flows, nor does it monitor or react to the status of the network. As such, Mantri is also complementary to our work.

Thus, the main goal of this section is to explore the following: given that we are using a state-of-the-art flow scheduling algorithm (Hedera) and a state-of-the-art task-placement algorithm (Mantri), is there still room for further improvement by exploiting the capabilities of software-defined networking (SDN) to establish a collaborative relationship between a system executing decision support queries over Hadoop and the network providing the communication below? We provide an initial answer of "yes" and also lay the groundwork for future follow-on work exploring this question.

Leveraging SDN for better performance of analytical queries was also considered in [82]. However, the scenarios considered in [82] are limited to traditional relational query processing, while our work focuses on MapReduce systems. One important difference is that in relational systems, query processing and the storage management are tightly coupled, whereas in Hadoop-based systems they are managed separately (MapReduce processing and HDFS file system). This separation calls for different management and optimization techniques for task and flow scheduling, which we study in this work.

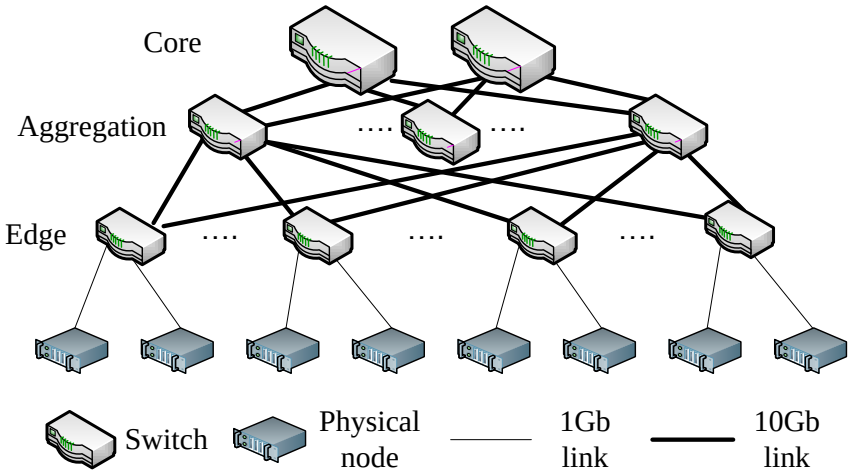


Figure 4.1. A common multi-rooted hierarchical tree

4.2 Data Center Networks and SDN

In this section, we give a brief background on some components to help presentation of the methods in the sequel.

4.2.1 Multi-rooted tree network topologies

Today's data centers could consist of thousands of connected servers. The recent research advocates multi-rooted tree topologies [16], where there are a larger number of parallel paths between any given source and destination edge switches. One rational for the existence of multiple paths is to achieve fault tolerance.

For example, Figure 4.1 shows three layers of switches, i.e., edge layer, aggregation layer, and the core layer. The edge layer switches directly connect to the servers. We can see that there are a larger number of parallel paths between any given source and destination edge switches. Note that, although 10Gb links are used in the aggregation layer and the core layer, which are more powerful than 1Gb links in the edge layer, it is currently very hard to achieve full bisection bandwidth due to the high oversubscription factor [16].

4.2.2 Software-defined networking and OpenFlow

SDN is an approach to networking that decouples the control plane from the data plane. The control plane is responsible for making decisions about where traffic is sent and the data plane forwards traffic to the selected destination. This separation allows network administrators and application programs to manage network services through abstraction of lower level functionality by using software APIs [60]. From the Hadoop point of view, the abstraction and the control APIs allow it to (1) monitor the current status and performance of the network, and (2) modify the network with directives, for example, setting the forwarding path for non-local tasks.

OpenFlow is a standard communication interface among the layers of an SDN architecture, which can be thought of an enabler for SDN [56]. An OpenFlow controller communicates with an OpenFlow switch. An OpenFlow switch maintains a flow table, with each entry defining a flow as a certain set of packets by matching on 10 tuple packet information.

When a new flow arrives, according to OpenFlow protocol, a "PacketIn" message is sent from the switch to the controller. The first packet of the flow is delivered to the controller. The controller looks into the 10 tuple packet information, determines the *egress port* (the exiting port) and sends "FlowMod" message to the switch to modify a switch flow table. When an existing flow times out, according to OpenFlow protocol, a "FlowRemoved" message is delivered from the switch to the controller to indicate that a flow has been removed.

For example, we show a 4-port OpenFlow switch S_{E0} serving as an edge switch in Figure 4.2. Two nodes $N_{0,1}$ are connected to S_{E0} at ports 0,1 and two aggregation switches S_{A0} and S_{A1} are connected to the switch at ports 2,3, respectively. There is a receiver and a transmitter behind each port of the switch. When a new flow $Flow_0$(from N_0 to N_2) arrives, a "PacketIn" message is sent from the switch S_{E0} to the controller. The controller looks into the 10 tuple packet information, determines the egress port and sends a "FlowMod" message to the switch to modify a switch flow table. The following packets in the same

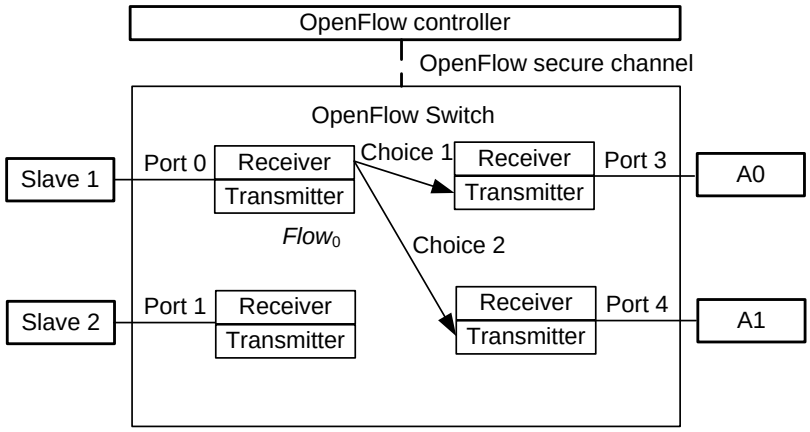


Figure 4.2. Inside an OpenFlow switch

flow will be sent through the same egress port. Because there are two aggregation switches, i.e., two paths from N_0 to N_2, the OpenFlow controller can have two options to determine the egress port. That is, the egress port can be 1 or 2, which means the flow can go through the aggregation switch S_{A0} or the aggregation switch S_{A1}.

4.3 Cormorant Design

In this section, we describe the system that we designed and implemented to evaluate the promise of SDN for improved Hadoop MapReduce query processing.

4.3.1 System architecture

Figure 4.3 shows the overall system architecture. The system is mainly comprised of Hadoop (with Master/NameNode and Slave/DataNode servers deployed in separate nodes), a network information manager, and an OpenFlow controller.

The basic operation of the system is as follows: The OpenFlow controller collects all the flow information from all the OpenFlow switches and periodically generates a snapshot of current network status. This information is stored at the Network Information Manager (NIM) and can be shared by the task scheduler, the replica scheduler and the flow scheduler. When Hive receives a query, it translates the query into several map reduce jobs and submits the jobs to the Hadoop master. Based on the network status snapshot, the task scheduler at

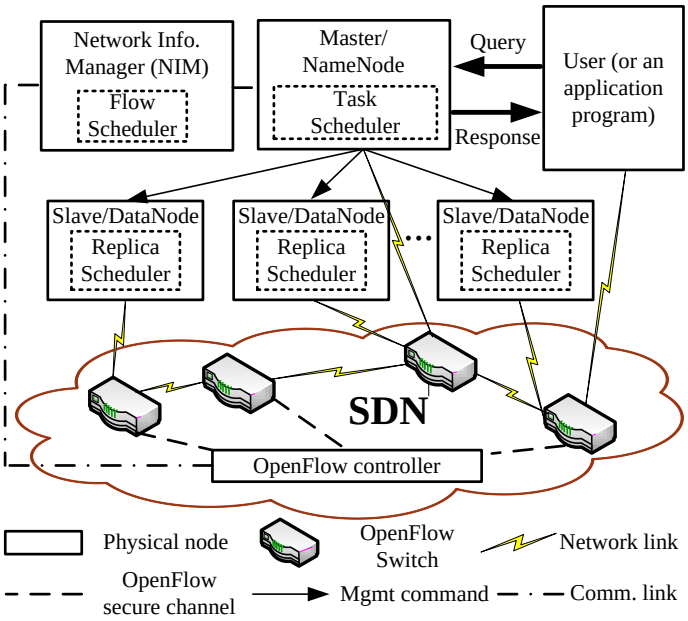


Figure 4.3. Cormorant system architecture

Table 4.1. Notations

Cap	port capacity (1Gbps in our setting)
N	a physical node
$Flow$	a network flow defined by 10 tuples
Flow	a set of all the flows
P_{Flow}	a random variable that denotes a path for $Flow$
$\mathbf{P}_{Flow}$	a sample space of all candidate paths for $Flow$
p_{Flow}	a physical path in a sample space $\mathbf{P}_{Flow}$
$A(p_{Flow})$	available bandwidth of path p_{Flow}
$t \in \mathbf{Task}$	a task in a task set

the master node assigns tasks to different slaves; the replica scheduler at each slave node selects replicas; and the flow scheduler schedules the flows. After all the jobs finish, the query results are returned to the user.

Table 4.1 lists the notations for the rest of the chapter.

4.3.2 Network Information Manager (NIM)

The NIM updates and inquires about the information on the current network state by communicating with the OpenFlow controller. Network information includes the network topology, queues, links, and their capabilities. It is important to keep this information up-to-date as inconsistencies can lead to under-utilization of network resources as well as bad

query performance. The NIM maintains a network status snapshot by collecting traffic information from OpenFlow switches. When a scheduler sends an inquiry to the NIM to inquire $A(p_{Flow})$, it will return the current available bandwidth of the flow by finding out the hop along the whole path that has the minimum available bandwidth (bottleneck).

Besides inquiring $A(p_{Flow})$, i.e., the available bandwidth for $Flow$ with a specific path p_{Flow}, the schedulers can also ask for a list of candidate paths. In this case, the NIM can select the best path that has the maximum $A(p_{Flow})$. Based on the best path information, the OpenFlow controller can send a "FlowMod" message to the switch to modify the switch flow table to add the best path.

4.3.3 Task/Replica scheduler

We follow the basic idea for task scheduler proposed in Mantri [17], i.e., placing a task close to its data. Compared with the default task scheduler which uses the static node-local, rack-local, and non-local tags, the improved tasks scheduler uses real-time global network status information for all the tasks. It greedily selects a task with *the most available bandwidth* from the data node to the taskTracker. Note that we assume that more available bandwidth may make the task finish faster and this approach is only "local" optimal for this task but may not be "global" optimal for all the tasks.

We apply Algorithm 1 for task sets. It picks the one that has the maximum available bandwidth (line 9). Finally, it compares the maximum available bandwidth with a threshold (a configurable system parameter). It returns the task if the maximum available bandwidth is more than the threshold and return no task if the maximum available bandwidth is less than the threshold (which means there is serious network congestion and/or there is no available slots and it may be better to postpone executing this task until the situation improves).

There is one key parameter P in Algorithm 1, which denotes an "abstract" path from N_d to N_c. It is called an "abstract" path because it is defined from a "MapReduce" point

Algorithm 1: Select task from a task set

```
Input: taskTracker at node N_c which is asking the master for a task to execute. task set Task;
Output: the task t_exec ∈ Task to be executed;
Max = −infinity; t_exec = null;
for t ∈ Task do
    NodeSet Datanode=t.getSplitLocations();
    for N_d ∈ Datanode do
        Path P = new Path(N_d, N_c);
        if A(P) > Max then
            Max = A(P); t_exec = t;
        end
    end
end
if Max > Threshold then
    Return t_exec;
end
else
    Return null;
end
```

of view, which is different from a physical path that is defined from a "Network" point of view. We use a discrete random variable P to denote an abstract path and use p to denote a physical path. We use $\mathbb{P} = \{p_1, p_2, ..., p_n\}$ to denote the sample space of all the n candidate physical paths. $A(P)$ is calculated as the average of the available bandwidth of all the n candidate physical paths.

When a task is executed, which of its replicas to choose is determined by the slave. This means the replica scheduler work independently with task scheduler which could cause inconsistency. So we also modified the source code of HDFS to make them work collaboratively. When a taskTracker needs to read a chuck, it also selects the replica that has *the most available bandwidth* to the taskTracker.

4.3.4 Flow scheduler

We adopt the flow scheduler design in Hedera [15], i.e., the scheduler aims to assign flows to *nonconflicting* paths.When a new flow arrives at one of the edge switches, according to OpenFlow protocol, a "PacketIn" message is sent from the switch to the controller. The first packet of the flow is delivered to the controller. The controller then chooses a

path whose available bandwidth can best accommodate this flow and schedule the flow to that path. Note that, we again assume that more available bandwidth will make the flow run faster and this approach is only "local" optimal for this flow but may not be "global" optimal for all the flows.

4.3.5 Collaborative schedulers

We described three improved schedulers in the previous three subsections. However, they may not be able to deliver the best optimized performance if they work separately rather than working collaboratively as shown below. (1) Task/replica scheduler only. Although we select a task with *the most available bandwidth* from the data node to the taskTracker, the most available bandwidth is not guaranteed at run-time if the task scheduler does not work collaboratively with a flow scheduler. (2) Flow scheduler only. If none of the candidate paths has enough available bandwidth due to neighbor traffic, the flow scheduler will not have good choice and the scheduled task may be take long to be executed.

In order to improve this, we build the collaborative schedulers as shown in Figure 4.4. (1) Network status snapshot is built by leveraging SDN. (2) The collaborative task scheduler chooses the best task with *the most available bandwidth* based on network status and Algorithm 1. $A(P)$ is calculated as the maximum available bandwidth of all the n candidate physical paths. (3) The replica scheduler chooses the replica accordingly. (4) The flow scheduler leverages the path configuration handler and schedules the physical path that has *the maximum available bandwidth* which corresponds to the task scheduler's choice.

Note that, collaborative schedulers are not simply putting the improved task/replica/flow schedulers all together. For example, in the improved task scheduler, $A(P)$ is calculated as the average because the scheduler is uncertain about the physical path. However $A(P)$ is calculated as the maximum in the collaborative one because the collaborative task scheduler is sure about the physical path.

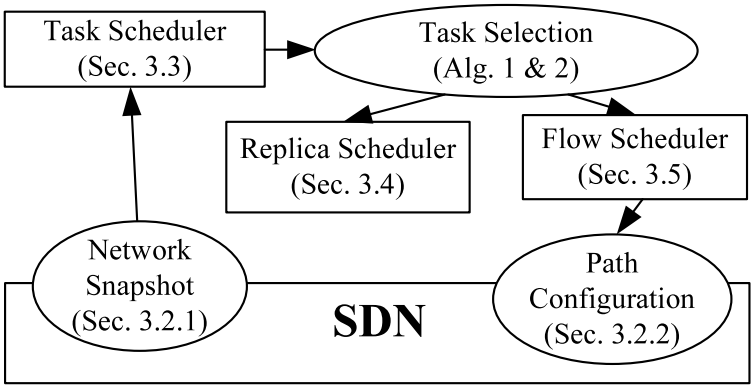


Figure 4.4. Collaborative relationship among schedulers

4.4 Experimental Setup

In this section, we describe our experimental setup and system implementation details.

4.4.1 Hardware and Topology

Our test bed as shown in Figure 4.5 consists of 17 physical nodes N_{0-16}. Each of the machines has an Intel Xeon E5-2440 2.4GHz Hexa-Core CPU, 32GB of RAM, 1TB 7200rpm disk running Linux with kernel 2.6.32. Six of the machines N_{10-15} are installed with a 4-port Gigabit NetFPGA card and perform as OpenFlow switches. Seven of the machines $N_{0-3,5-7}$ are used for Hadoop MapReduce deployment with one master (at N_0) and six slaves (at $N_{1-3,5-7}$). N_0 and N_4 are also used for generating neighbor network traffic. N_8, N_9 and N_{16} are used to run network information manager(NIM), Openflow Controller, and client emulator, respectively.

4.4.2 Benchmark and Traffic

We run Hadoop MapReduce 1.2.1 and apply most of the default settings. We use Hive 0.11 and run an OLAP benchmark TPC-H with a scaling factor of 100. We report the query execution time and we do not consider loading time as part of the benchmark results. All query results are saved into Hive tables, which are also stored in HDFS.

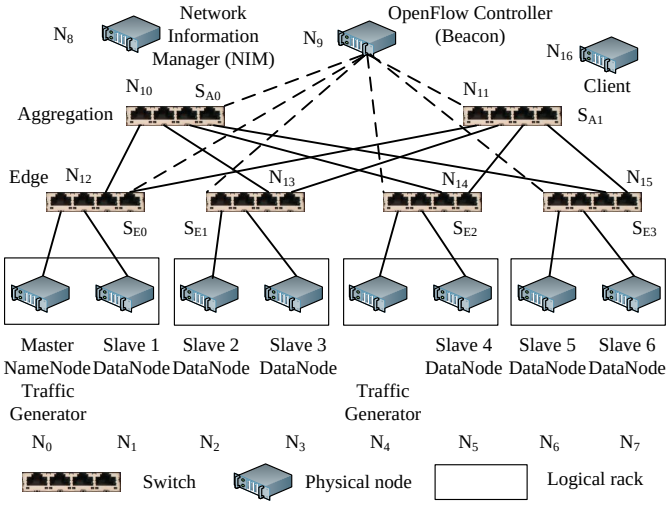


Figure 4.5. Hadoop MapReduce Setup

We use nodes N_0 and N_4 to create the *neighbor contention traffic* emulated by `iperf` [2]. Neighbor contention traffic emulates the shared network environment where Hadoop (the main application) is running along with other applications whose network traffic is captured in the neighbor contention traffic. More specifically, we create 12 flows in total. $N_{i\to j}$ denotes a flow from node N_i to node N_j. The 12 flows are $N_{0\to1}$, $N_{0\to2}$, $N_{0\to3}$, $N_{1\to0}$, $N_{2\to0}$, $N_{3\to0}$, $N_{4\to5}$, $N_{4\to6}$, $N_{4\to7}$, $N_{5\to4}$, $N_{6\to4}$, $N_{7\to4}$. We define several levels of contention as show in Table 4.2. For example, when the contention traffic is "Low", we have 3 large flows $N_{3\to0}$, $N_{7\to4}$ and $N_{0\to2}$ with 800Mbps each, and 9 small other flows with 50Mbps each. We have 4 large flows with 800Mbps and 850Mbps each in "Medium" and "High" levels, respectively.

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

Table 4.2. Contention traffic levels

Levels	Large flow (800Mbps)	Small flow (50Mbps)	Util.
None	-	-	0%
Low	$N_{3\to0}$, $N_{7\to4}$, $N_{0\to2}$	Others	36%
Medium	$N_{3\to0}$, $N_{7\to4}$, $N_{0\to2}$, $N_{4\to6}$	Others	45%
High	Medium with 850Mbps	Others	48%

The NIM collects network status information from the OpenFlow controller at a 5 second polling rate. We set the threshold for Algorithm 1 to 100Mbps. Note that, we adopt these settings in order to run our experiments. We do not claim that they are the best settings.

4.5 Evaluation

In this section, we report our experimental results with TPC-H benchmark with a scaling factor of 100. Each experiment is run three times and the average (with the standard deviation if applicable) is reported. In Section 4.5.1, we first summarize all TPC-H queries' performance under 5 different scenarios as shown in Table 4.3, i.e., **default** (default Hadoop), **task/replica scheduler only**, **flow scheduler only**, **collaborative** and **no traffic** (default Hadoop without any neighbor traffic).

Table 4.3. Schedulers used in different scenarios

Scenarios	Task Scheduler	Flow Scheduler	Traffic
default	default	ECMP	yes
task/rep. sched.	improved	ECMP	yes
flow sched.	default	improved	yes
collaborative	collaborative	collaborative	yes
no traffic	default	ECMP	no

4.5.1 Summary of TPC-H queries' performance

In this section, we discuss the experimental results for all TPC-H queries (Q1-Q22) as shown in Figure 4.7. The y-axis is the query execution time and x-axis shows 5 different

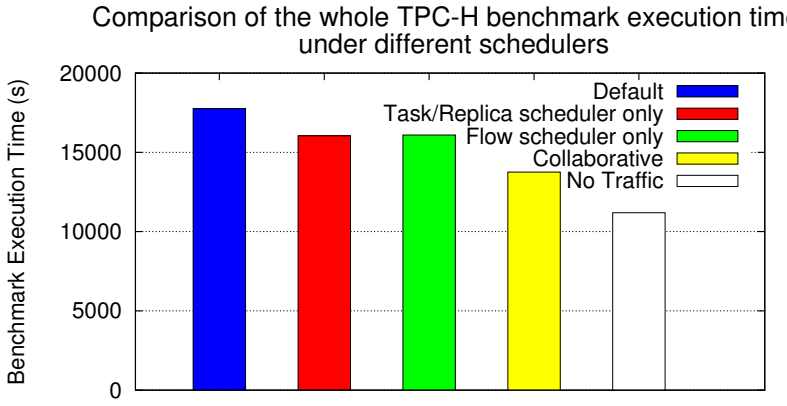


Figure 4.6. Comparison of the whole TPC-H benchmark execution time

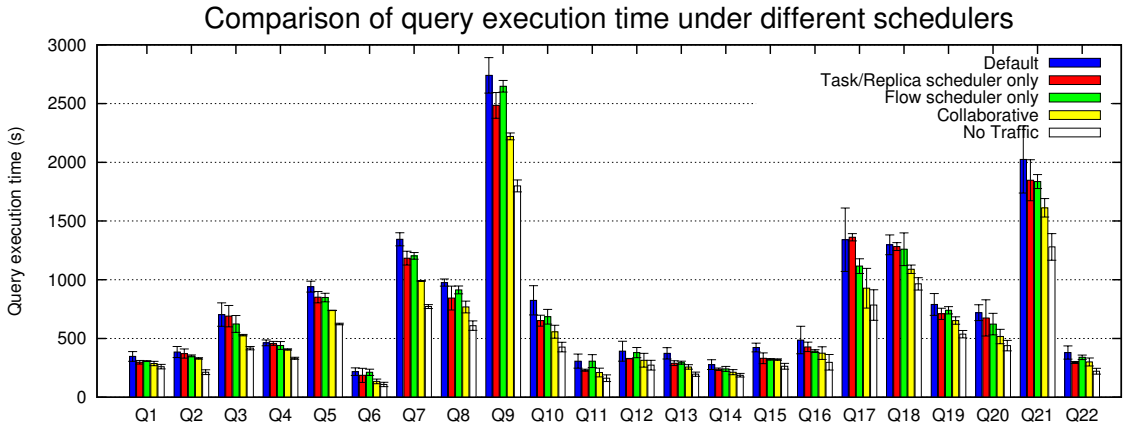


Figure 4.7. Details of TPC-H benchmark query execution time

scenarios for each query. Besides the other default settings of Hadoop, we set the number of replica to one, the chunk size to 512MB, the number of map slots to three and the number of reduce slots to three. Neighbor network traffic is medium, i.e., 45% network bandwidth utilization. More case studies under different settings can be found in Section 4.5.3.

We summarize and compare the whole TPC-H benchmark execution time in Figure 4.6 and we enumerate the details of each query execution time in Figure 4.7. We have the following observations: (1) Figure 4.6 shows that it takes on average 17757s, 16043s, 16090s, 13756s and 11191s for default, task/replica scheduler only, flow scheduler only, collaborative and no traffic scenarios. That is, the overhead brought by network traffic is 58.677%, 43.363%, 43.777% and 22.925% for default, task/replica scheduler only, flow scheduler

only and collaborative. Although the overhead is reduced when task/replica scheduler or flow scheduler is used, the reduction is quite limited (only about 15%). When all the schedulers are working collaboratively, we can achieve benefits even more than simply adding up their own benefits (35%). (2) From Figure 4.7, we observe that all the query execution times are reduced when we use task/replica scheduler or flow scheduler alone. Across all of the benchmark queries, the collaborative case always delivers the best performance. For different queries, different improvements are achieved. For some of the queries, e.g., Q3, the flow scheduler achieves greater improvement than the task/replica scheduler. However, for some other queries, e.g., Q6, the task/replica scheduler achieves greater improvement than the flow scheduler. We study and compare task/replica vs. flow scheduler contributions with respect to Q3 and Q6 in the next subsection.

4.5.2 Task/replica vs. flow scheduler contributions

First, we study the task/replica scheduler and flow scheduler contributions for Q3. Q3 involves "join" operations among `customer`, `orders`, and `lineitem` tables followed by "group by" and "order by" operations. Accordingly, when the query is submitted, it is transformed into four MapReduce jobs: (job 1) Join `customer` and `orders` tables on c.c_custkey = o.o_custkey with predicate c.c_mktsegment = 'BUILDING'. (job 2) Join the result of job 1 with `lineitem` table with predicates and calculate the value inside "sum". (job 3) Calculate the sum value according to the "group by" operator. (job 4) Order the final results and return the top 10 rows. We use default task/replica/flow schedulers, turn off the contention traffic and run the query. It is observed that, 89% of the total execution time is spent on join with predicates (jobs 1 and 2, especially job 2) while 11% of the time is spent on aggregation (jobs 3 and 4).

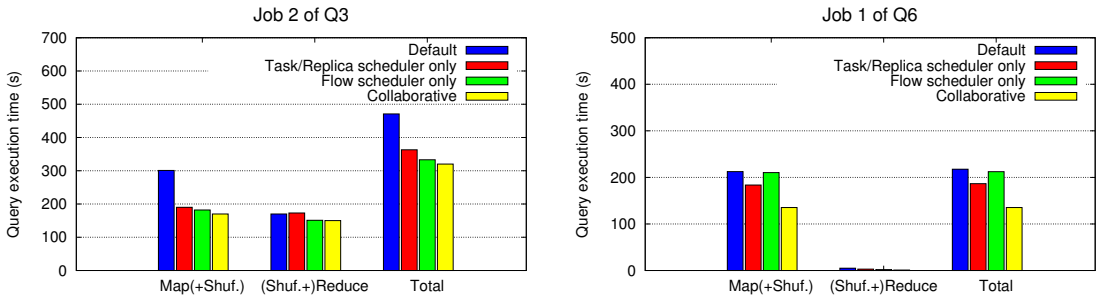
Because most of the execution time is spent on join, especially job 2, we record the total number of map and reduce tasks in a single run for job 2. There are 170 map tasks and 81 reduce tasks. Of all the map tasks, 17 tasks are non-local ones in which a taskTracker

needs to load data remotely from a separate data node. We denote the time interval between the first map task begins and the last map task ends as "Map" and the time interval between the last map task ends and the last reduce task ends as "Reduce" for job 2 of Q3 in the left part of Figure 4.8. The "Total" is the sum of the two, which is the execution time of the job. We also compare the four scenarios in "Map", "Reduce" and "Total" intervals. Note that, because we have one master(jobTracker) and six slaves(taskTrackers), the maximum number of usable reduce slots is 18. Since there are 81 reduce tasks, at most 18 reduce tasks are running during the "Map" time interval and at least 63 reduce tasks are running during the "Reduce" time interval. This means the shuffling phase (Shuf.) spans across both the "Map" and the "Reduce" time intervals. We can see that, due to the different design of task/replica scheduler and flow scheduler, (1) Task/replica scheduler contributes almost the same as flow scheduler for the reduction of the execution time for the "Map" interval. (2) Flow scheduler contributes significant more than task/replica scheduler for the "Reduce" interval. (3) As a result, the flow scheduler contributes more than the task/replica scheduler for the "Total" job execution time.

Second, we study the task/replica scheduler and flow scheduler contributions for Q6. When the query is submitted, it is transformed into a single MapReduce job: (job 1) scans `lineitem` table with predicates and then calculates the sum of revenues. We also compare the four scenarios in "Map", "Reduce" and "Total" intervals for job 1 of Q6 in the right part of Figure 4.8. This job is comprised of 148 map tasks and 1 reduce task. We have similar observations (1) and (2) for this job as for the job 2 of Q3. However, the length of "Map" interval (around 170s to 200s) is much longer than that of the "Reduce" interval (around 5 seconds) in this job. As a result, task/replica scheduler contributes more than flow scheduler for the "Total" job execution time.

Finally, we can summarize the different contributions of task/replica scheduler and flow scheduler for different TPC-H queries. Assume that we can decompose the query execution time into "Map" and "Reduce" parts. If the "Map" part is smaller than or comparable to

the "Reduce" part, then the flow scheduler generally contributes more than the task/replica scheduler. Otherwise, the task/replica scheduler contributes more than the flow scheduler. It is important to note that our collaborative schedulers cover the benefits of both.



(a) Flow scheduler contributes more in Job2 of Q3 (b) Task/replica scheduler contributes more in Job1 of Q6

Figure 4.8. Comparison of schedulers' contribution to reduce Q3 and Q6's job execution time

4.5.3 Case studies of Q3 in TPC-H

In this section we present the experimental results for Q3 for a set of case studies. The main motivation is to study how much improvement our collaborative method provides when the system environment or settings changes. More specifically, we systematically change the system settings to generate the cases as shown in Table 4.4. We compare the query performance under (1) different levels of traffic in Section 4.5.3.1. (2) different number of replicas in Section 4.5.3.2. (3) different chunk sizes in Section 4.5.3.3. For each case, we report on 4 scenarios (default, task scheduler only, flow scheduler only, collaborative). We choose Q3 for case studies due to two reasons: (1) it is complicated, which involves selection, projection, join, and aggregation operations; (2) its performance improvement based on the schedulers is in the middle of all the queries that we consider in Figure 4.7.

Table 4.4. Summary of case studies

#	Contention traffic	# of replicas	Chunk size(MB)
1	**Change**	1	512
2	medium	**Change**	512
3	medium	1	**Change**

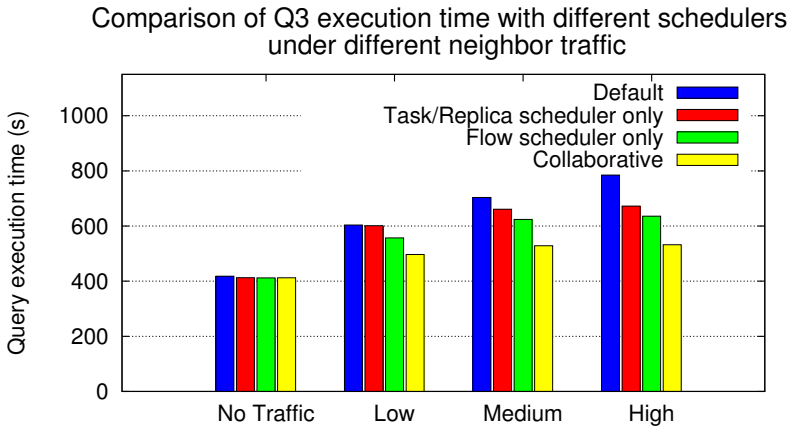


Figure 4.9. Comparison of Q3's execution time with different levels of neighbor traffic

4.5.3.1 Case 1: different levels of neighbor traffic

In this case we compare the query execution time of Q3 with different levels of neighbor traffic emulated by iperf as shown in Figure 4.9. We can see that, (1) When the levels of neighbor traffic increases from none to low until medium and high, the query execution time increases accordingly. (2) The improvement based on task/replica scheduler and flow scheduler over the default becomes more obvious when the levels of neighbor traffic increases. (3) When all the schedulers work collaboratively, we can achieve the most improvement. (4) The previous observation also applies to the improvement of collaborative scheduler under dynamic traffic (not shown in the figure), which is in-between of low and high levels of neighbor traffic.

4.5.3.2 Case 2: different number of replicas

In this case we compare the query execution time of Q3 with different numbers of replicas as shown in Figure 4.10. We can see that, (1) When the number of replicas increases from 1 to 3, the query execution time with default schedulers remains nearly constant. On

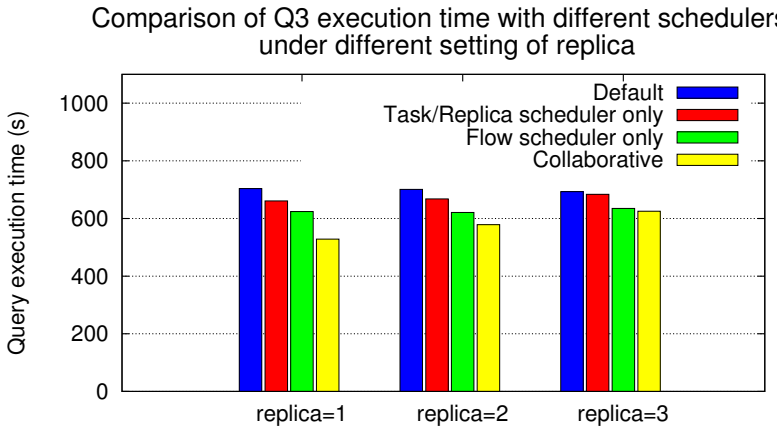


Figure 4.10. Comparison of Q3's execution time with different number of replicas

one hand, more replicas provides the default schedulers with more replica candidates to choose from and avoids network congestion. On the other hand, more replicas implies more copies of intermediate results to be written (because HDFS enforces the same number of copies of intermediate results), which increases the chance of encountering network contention. (2) When the number of replicas increases from 1 to 3, the contribution of task/replica schedulers becomes less significant. This is because more replicas means more replica candidates to choose from. Therefore the impact of task/replica schedulers becomes less significant. (3) Because of (2), although there is still improvement based on task/replica scheduler and flow scheduler over the default, the improvement is offset when the number of replicas increases. And we observe less improvement of collaborative schedulers when the number of replicas increases.

4.5.3.3 Case 3: different chunk sizes

In this case we compare the query execution time of Q3 with different chunk sizes as shown in Figure 4.11. We make the following observations. (1) When chunk size increases from 64MB to 512MB, query execution time of Q3 with default schedulers, task/replica schedulers only and flow scheduler only cases decreases. The decrease is mainly due to the decrease in the scheduling overhead, which is reported in related work [48]. For ex-

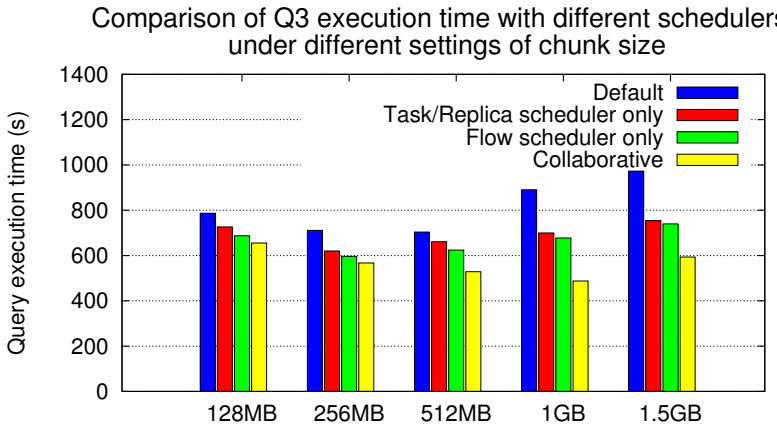


Figure 4.11. Comparison of Q3's execution time with different chunk sizes

ample, as reported in [48], a micro-benchmark shows that the processing time for scanning a 10GB file with 5GB chunk size is more than thrice faster than scanning the same file with 64MB chunk size. (2) However, when chunk size increases from 512MB to 1.5GB, query execution time of Q3 with default schedulers, task/replica schedulers only and flow scheduler only cases increases. The main reason for this is that, the number of total map tasks decreases due to the increase in the chunk size. This results in a decreased number of non-local tasks and decreased chance of network traffic contention. (3) We still observe the improvement based on task/replica scheduler, flow scheduler and collaborative schedulers over the default schedulers.

4.6 Related work

We discuss related work from two areas – the databases and the networking.

From the database perspective: there are a plethora of work on optimizing Hadoop/MapReduce performance. CoHadoop [34] extends HDFS and adds metadata to NameNode so that it allows applications to control where data are stored. HaLoop [24] caches the reused data on local disks and then improves performance by leveraging this data locality in the scheduler. Hyracks [21] is a data-parallel runtime platform designed to perform data-processing tasks on large amounts of data using large clusters. Because data is processed in a pipelined

manner, Hyracks can push a very large amount of data to the network thereby potentially creating extreme network contention during the run-time. (A similar observation has been made in other recent large scale data processing systems, such as Apache Spark.) All of the above work treats the network as a black box. Our contribution is orthogonal and is complementary to this, making network work collaboratively with Hadoop/MapReduce. The work that is most related to ours is [17], [15] and [82]. We have discussed our differentiation from these work in the introduction.

From the networking perspective: Wang *et al.* [79] propose application-aware networking, and argue that distributed applications can benefit from communicating their preferences to the network control-plane. Yap *et al.* have also advocated for an explicit communication channel between applications and software-defined networks, in what they called software-friendly networks [84]. PANE [36] proposes design, implementation, and evaluation of an API for applications to control a software-defined network. Sinbad [27] proposes a network balanced data placement method. Varys [29] uses coflows to optimize network scheduling. Both [27] and [29] assume data center is dedicated and all network flows can be controlled by their system. While our work focuses on how to improve Hadoop performance in a shared network environment with dynamic uncontrolled flows. The work that is most related to ours is [15]. [15] focuses on the flow scheduler work-alone case where the flow scheduler has to "estimate" the demand of a flow. However, in our work, the flow scheduler consistently follows the scheduling decisions made by the task scheduler. As a result, our collaborative schedulers achieve much more improvement over their work-alone flow scheduler.

4.7 Conclusions

In this chapter, we propose Cormorant, a Hadoop-based system with collaborative software-defined networking for executing analytic queries. Unlike previous work, in which Hadoop works independently from the network underneath, our system enables

Hadoop and the networking layer to work together to improve network utilization and reduce query execution times. As our experiments with an implementation show, this improvement goes beyond that achievable by the state of the art approach of combining optimizing task schedulers and flow schedulers without collaboration. We believe that our work shows early promise for achieving one of the often-cited goals of SDN, i.e., tightly integrating applications with the network to improve performance

CHAPTER 5

NETWORK-AWARE TASK SCHEDULING FOR SPARK

Recently Spark has become a popular data processing platform that runs on commodity cluster because of its fast in-memory computing which allows users to cache data in servers' memory and query it repeatedly. However, since network I/O is much slower than local memory I/O in Spark, network can be a bottleneck for data intensive jobs. Current delay scheduling method can't solve this problem because it's agnostic to network conditions of the cluster. In this section, I propose a network-aware scheduling method to solve current scheduling issues and improve the performance of Spark. We implement our method in a system called Firebird running on top of SDNs.

5.1 Background and Motivation

Running analytic queries on large, diverse, and ever-growing datasets, also referred to as big data processing, has become an essential part of business processes for enterprises. MapReduce [33] has emerged as a framework for processing large amounts of structured and unstructured data in parallel across a large number of machines, in a reliable and fault-tolerant manner. Hadoop and Spark are two popular platforms that implement MapReduce framework.

MapReduce platforms provide computation, storage and network resources to users and efficient management of these resources is essential for platform performance. Current resource managers like Yarn and Mesos mainly focus on managing computing and memory resources in a cluster. Recent research has found that for data intensive jobs, network and

disk I/O are major performance bottlenecks in MapRedce platforms like Hadoop [41]. But these aspects are not handled by current resource managers.

As server memory grows in size, it has been feasible to use persistent memory structure to address the issue of slow disk I/O. Resilient Distributed Datasets (RDDs) have been proposed to provide high efficiency data reuse which reduces disk I/O by using in-memory data storage. RDD is implemented in Apache Spark which performs up to 100 times faster than Hadoop [88].

However, due to the distributed nature of the MapReduce framework, the network I/O has always been a scarce resource that limits MapReduce's performance [15]. Neither Spark nor Hadoop explicitly manage network resources. Unbalanced data locality can cause biased network utilization, which means data contention can happen and harm a cluster's performance during job execution [17]. Moreover, this problem becomes even more challenging when the network is shared with other applications [73].

The main cause of the problem is the separation between the current resource manager and the network i.e., the resource manager is agnostic to the underlying network while the network neglects any MapReduce-specific customization requests. As a result, if there is a node with adequate computing resources, but limited network bandwidth, the network becomes the bottleneck when many non-local tasks are scheduled to and executed on this node. In traditional MapReduce frameworks like Hadoop, this problem is not severe because local disk I/O is slow. But in Spark, in-memory data storage accelerates local data I/O so that network I/O is more likely to be the bottleneck and network congestion will largely impact the performance of Spark jobs.

In this context, we focus on the problem of how to create a more collaborative relationship between Spark and networking in order to improve the performance by exploiting the capabilities offered by software-defined networking (SDN) [56, 60].

It is desirable to create a more collaborative relationship between Spark query processing and the underlying networking where there is a more direct and continuous communi-

cation between those two components for improved data processing. It is intuitive that if the state of the network is more visible to Spark, it can make better decisions on scheduling tasks across the distributed nodes. As pointed out earlier, the problem is that, data processing and network are separate entities that do not directly communicate with each other and network is generally managed distributedly which makes it hard to create the relationship between network and Spark. However, the emerging software-defined networking (SDN) can remove those communication barriers by providing direct APIs for applications to monitor and control the state of the network. Our goal in this chapter is to explore SDN to create an efficient collaborative distributed query processing environment with Spark.

In this chapter, we analyze scheduling problems of existing task schedulers in MapReduce frameworks and then propose a network-aware scheduling method that optimizes task scheduling based on network status of a cluster. We model different scheduling methods and analyze their performance under unbalanced data locality. Then we implement our method in Spark and build an actual system running on top of software-defined networking. In contrast to previous work, our work aims at building a collaborative relationship between Spark and a SDN-based network.

5.2 Spark and SDN

In this section, we describe Spark and SDNs by way of background for subsequent sections.

5.2.1 Spark Background

Apache Spark is a MapReduce-like cluster computing framework that can execute data-parallel tasks. When a Spark job is submitted, the Spark master divides it into stages according to shuffle operations (one stage per shuffle operation). Figure 5.1 shows how one Spark stage is executed. The input data is RDD partitions stored on disk or in memory. Each RDD partition corresponds to one map task in Spark. Map tasks are assigned to

executors by a Spark master and processed into key-value pairs. Then the shuffle operation deals with values for each unique key and generates output.

In contrast to traditional MapReduce engines, Spark introduces a distributed memory abstraction called Resilient Distributed Dataset (RDD). Each RDD is a read-only data structure created from data in stable storage or through a transformation on existing RDDs. Each RDD is divided into partitions that can be stored in memory or disk on individual servers. Users can explicitly cache RDDs in memory by using persist() or cache() function in Spark API. When servers have enough memory, a lot of data can be kept in memory which makes Spark much faster for jobs that need to iteratively read the same dataset, e.g. machine learning jobs and SQL queries. Experimental results show that Spark can be up to 100 times faster than Hadoop for specific jobs [88]. To deal with data lost during failures, Spark uses a directed acyclic graph (DAG) to record transformations that created the RDD. Importantly, transformations are coarse-grained in that they apply the same operation to each of an RDD's partition in parallel. Thus, if a node fails, the RDD partitions on this node can be re-generated on other nodes.

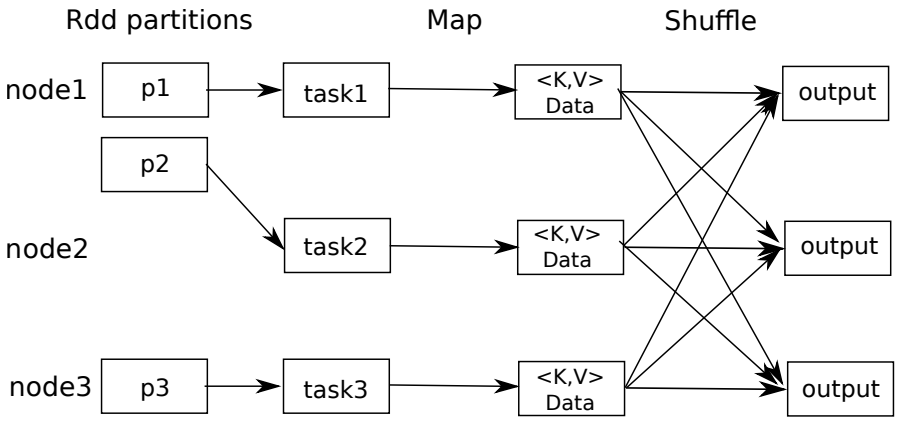


Figure 5.1. Execution of a Spark stage

If the distribution of RDD partitions is not well balanced among nodes, e.g. one node has all RDD partitions or one node has no RDD partitions, there can be data shipping between nodes during execution (in Fig 5.1 p2 is shipped from node1 to node2). Data distribution on a Spark cluster may be unbalanced for two reasons: 1) the data distribution in stable storage such as HDFS is not balanced 2) some nodes lose data because of node failures. To deal with this unbalanced data distribution, the task scheduler should judiciously schedule tasks to make full use of computation resource of the cluster as well as reduce latency caused by data shipping. Similar to Hadoop, Spark uses a locality-aware scheduler which divides tasks into 5 different locality levels and schedule tasks with high locality first. The five locality groups are:

PROCESS_LOCAL: the data is in the same JVM as the current executor. Data in this level is already deserialized and cached in memory, so it's very fast.

NODE_LOCAL: the data is on the same server as the current executor. Data can either be in local disk or HDFS directory on the node. If data has been used recently, it may also be cached in memory by OS file caching.

NO_PREF: the data has no locality preference which means data is not on the same node as the current executor and it's also not on other nodes that have executors. One example is that data is stored in Amazon S3.

RACK_LOCAL: the data is not stored on the same node as the current executor but on the same rack as the current executor. This level assumes the data shipping within a rack is faster than that between different racks.

ANY: the data is neither on the same node nor on the same rack as the current executor. Different from NO_PREF, data has its locality preference which generally means the node who owns the data can also execute this task. Therefore, this task may potentially be executed as PROCESS_LOCAL or NODE_LOCAL if it's not scheduled to the current executor.

For simplicity, in this chapter, we treat PROCESS_LOCAL and NODE_LOCAL as "local" and the other 3 levels as "non-local". To monitor the network status of the whole cluster, we assume input data is stored in the cluster nodes rather than file systems like Amazon S3.

5.2.2 Software-defined networking

SDN is a new approach to networking that decouples the control plane from the data plane. The control plane is responsible for making decisions about where traffic is sent and the data plane forwards traffic to the selected destination. This separation allows network administrators and application programs to manage network services through abstraction of lower level functionality by using software APIs [60]. From Spark's point of view, the abstraction and the control APIs allow it to (1) monitor the current status and performance of the network, and (2) modify the network with directives, for example, setting the forwarding path for non-local tasks.

OpenFlow is a standard interface among the layers of an SDN architecture, which can be thought of an enabler for SDN [56]. An OpenFlow controller communicates with an OpenFlow switch. An OpenFlow switch maintains a flow table, with each entry defining a flow as a certain set of packets by matching on 10 tuple packet information as shown below.

$$Flow ::= [InPort, VLANID, MACSrc, MACDst,$$
$$MACType, IPSrc, IPDst, IPProto, PortSrc, PortDst]$$

When a new flow arrives, according to OpenFlow protocol, a "PacketIn" message is sent from the switch to the controller. The first packet of the flow is delivered to the controller. The controller looks into the 10 tuple packet information, determines the *egress port* (the exiting port) and sends "FlowMod" message to the switch to modify a switch flow table. When an existing flow times out, according to the OpenFlow protocol, a "FlowRemoved" message is delivered from the switch to the controller to indicate that a flow has been

removed. The controller can also send a "PortStatus" message to switches to get statistics like TX data and RX data. By periodically enquiring port status of all switches, controller can compute current network utilization and available bandwidth of all links in the cluster.

5.3 Task Scheduling

In MapReduce platforms like Hadoop and Spark, data may not be on the same node as the executor, resulting in network traffic between nodes. For reduce tasks, since each task reads roughly equal amounts of data from all nodes [87], data locality based scheduling doesn't provide much benefit. However, since each map task is executing data on a specific node, if data locality of a cluster is not well balanced, map tasks can cause network congestion which may harm the performance of whole system. So in this chapter, we mainly focus on scheduling of map tasks. In this section, we'll first introduce current scheduling methods and describe the problems of these schedulers. Then we propose our network-aware scheduling method based on knowledge of network status of a cluster. Finally, we create models of these scheduling methods to analyse their performance under unbalanced data locality.

5.3.1 Naive Scheduling

Naive scheduling was implemented in early versions of Hadoop. In naive scheduling, tasks are divided into different locality sets based on distance to the executor. When the scheduler receives a heartbeat from an idle node, it searches for available tasks from node-local set first, then rack-local set and at last non-local set. It does so because it believes that closer tasks can have better data transfer rate. Algorithm 2 shows the pseudo code of naive task scheduling.

There are several drawbacks to naive scheduling. The first one is that whenever the task scheduler receives a heartbeat from a node that has an idle slot, a task is scheduled to it instantly. If a job's data is only stored on a small fraction of nodes, during job execution,

Algorithm 2: Naive task Scheduling

Input: Node n that has an idle slot
Output: Task t to be assigned to n

```
for j in jobs do
    if j has unassigned task t with data on n then
        return t;
    end
    else if j has unassigned task t with data on nodes in the same rack with n then
        return t;
    end
    else if j has unassigned task t then
        return t;
    end
end
return null;
```

a task is scheduled to the node that sends the first heartbeat rather than the node with data. Therefore task locality can be very poor. For example, if a job has data on 20% of nodes, only 20% tasks are local tasks on average.

The second problem is that in rack-local and non-local task sets, naive scheduling simply picks the first available task without further considering current cluster status, especially network conditions. The execution time of some tasks may be longer than other similar tasks because of network congestion and these outlier tasks will influence the performance of the whole system.

5.3.2 Delay Scheduling

Delay scheduling [87] was proposed to solve the first problem in naive scheduling. The main idea of delay scheduling is to delay launching a non-local task. During the delay, the task scheduler may find a data-local node to execute the current task. Algorithm 3 shows the pseudo code of delay task scheduling being used in Apache Spark.

Using a simple technique, delay scheduling solves the locality problem of naive scheduling in small jobs. But in large jobs, if the data distribution is unbalanced, the second problem of naive scheduling still exists. For example, in Figure 5.2 if a job has no data partition on node A and twenty partitions in other nodes, and each node has ten slots to execute

Algorithm 3: Delay task Scheduling

Input: Node n that has an idle slot, rack-local waiting time T_r, non-local waiting time T_{na}
Output: Task t to be assigned to n

```
Initialize j.lastlaunchtime to current time for all jobs j.
for j in jobs do
    if j has unassigned task t with data on n then
        set j.lastlaunchtime to current time ;
        return t;
    end
    else if j has unassigned task t with data on nodes in the same rack with n then
        if currenttime − j.lastlaunchtime > T_r then
            return t;
        end
    end
    else if j has unassigned task t then
        if currenttime − j.lastlaunchtime > T_na then
            return t;
        end
    end
end
return null;
```

tasks. We assume the size of each partition is 1Gb. Here we use a concept called data processing rate (hereinafter referred to as DPR) to measure the computation intensity of a task. DPR means the amount of data that can be processed per second by the current CPU if the data I/O is unlimited. This metric is influenced not only by task type but also by CPU capability. So we usually say DPR under current environment. We assume the DPR of the tasks in the current environment is 1Gb/s. If node A is not in the cluster, all tasks are executed locally and it only takes 4 seconds to complete them. But when we add node A to the cluster, tasks will be scheduled to node A after waiting a small amount of time. Node A will have 10 non-local tasks to execute. If the incoming network bandwidth of node A is only 1Gb/s, the tasks will congest on node A which means their actual executing rate is only 100Mb/s. It will take 10 seconds to finish them in this case. From this scenario, we see that unbalanced data distribution hurts the performance of the cluster. It takes longer to finish the tasks when we add a new node to the cluster which makes data unbalanced. This task scheduling is worse than ignoring node A and executing all tasks locally. And it can be even worse if there is neighborhood traffic from other nodes to node A.

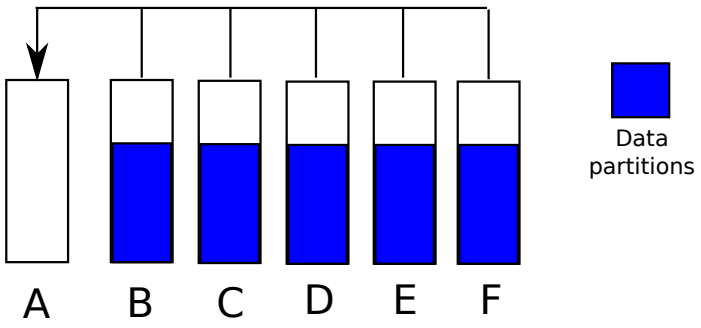


Figure 5.2. Examples of unbalanced data distribution

In delay-based scheduling, one method to solve this problem is to increase the waiting time to achieve better locality. Figure 5.3 shows an example of the relationship between waiting time and locality. According to Algorithm 3, if the waiting time is larger than task execution time, no non-local task will be scheduled and all tasks will be local tasks. However, although longer waiting time increases data locality, it also means wasting of computing resource. So in practice, we need to balance these two factors and find the optimal waiting time to obtain the best performance. Since for different jobs (data intensive, CPU intensive) and different data locality (well balanced, unbalanced) the optimal waiting time is different, it's impossible to find an best value that fits all scenarios. It's also hard to find the best waiting time each time before a job starts because this requires knowledge of the CPU DPR before a job starts but CPU DPR varies even in different stages of the same job.

5.3.3 Network-aware Scheduling

The problems in existing schedulers occur because they don't consider network status in a cluster. We observed that network congestion mainly occurs at nodes with few data blocks or with large incoming neighborhood traffic, which results in network congestion at the node. To solve this problem, we propose a new network-aware scheduling method

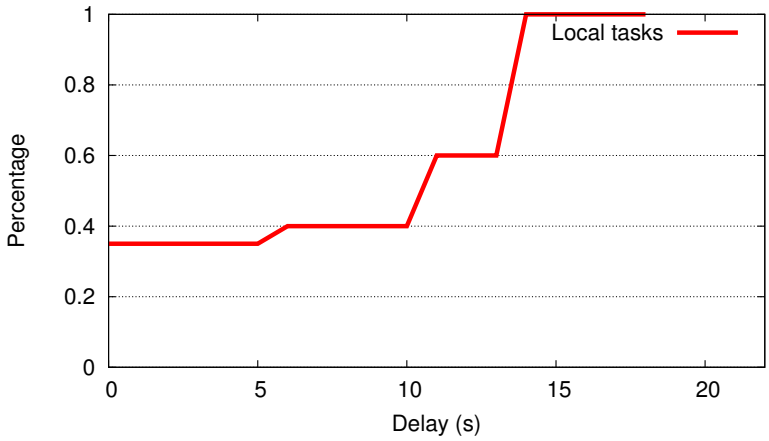


Figure 5.3. Relationship between waiting time and locality

which aims to avoid network congestions and achieve efficient scheduling to increase the throughput of the whole system. Our network-aware task scheduler is implemented based on a delay scheduler and uses the global network status to help make scheduling decisions. When a node requires a task, if there is no local task to schedule, the scheduler will first check if the node has enough available bandwidth to accommodate a new non-local task. If so, the scheduler finds a task with enough available bandwidth from data node to the executor node, otherwise, no task is scheduled at this time. The pseudo code of our method is shown in Algorithm 4.

Compared with current delay scheduling method, our method has three advantages. The first is that while scheduling non-local tasks, it considers the network overhead of the node that has idle slots, which can prevent scheduling too many non-local tasks to the same node. The second advantage is that since the scheduler can get global network status, it can pick a non-local task that will not experience congestion during data shipping, which accelerates the execution process. Finally, our method can achieve better performance than existing scheduling methods without any system tuning or parameter adjustment when executing different kind of jobs.

Algorithm 4: Network-aware task Scheduling

Input: Node n that has an idle slot, rack-local waiting time T_r, non-local waiting time T_{na}
Output: Task t to be assigned to n

```
Initialize j.lastlaunchtime to current time for all jobs j.
for j in jobs do
    if j has unassigned task t with data on n then
        set j.lastlaunchtime to current time ;
        return t;
    end
    else if currenttime − j.lastlaunchtime > T_r then
        for task t in j.rack-local do
            compute t.DPR
            compute availableCapacity which equals bandwidth of the best path from t.split to n
            if availableCapacity¿t.DPR then
                return t;
            end
        end
    end
    else if currenttime − j.lastlaunchtime > T_na then
        for task t in j do
            compute t.DPR
            compute availableCapacity which equals bandwidth of the best path from t.split to n
            if availableCapacity¿t.DPR then
                return t;
            end
        end
    end
end
return null;
```

5.3.4 Analysis of scheduling methods

The previous sections provide an overview of different scheduling method. In this section, we analyse and compare these methods in detail. We first summarize these schedulers as three classes of scheduling models and then analyse their performance in theory.

5.3.4.1 Scheduling Models

In Spark, scheduling methods can be separated into three classes based on how they deal with unbalanced data locality.

Non-local preferred scheduling. The key idea of this model is that when there is no local task on the idle executor, non-local preferred scheduling prefers to schedule a non-local task. Both naive scheduling and delay scheduling with small waiting time can be seen as non-local preferred scheduling. If network capacity is adequate and there is no data contention, then there is no overhead caused by unbalanced locality because non-local tasks can be executed at the same processing rate as local tasks. If the data processing rate is too large or network capacity is inadequate, there will be overhead because network capacity is not sufficient to accommodate non-local tasks and data contention will cause delay of non-local tasks.

Local preferred scheduling. In this model, when there is no local task on the idle executor, it prefers waiting to schedule a non-local task. Thus, most tasks are executed locally and non-local tasks are avoided during processing. Delay scheduling with long waiting times can be seen as local preferred scheduling. Since extra data on some nodes can not be digested by other nodes and nodes with fewer data chunks are not well utilized, the overhead will be high if data locality is highly unbalanced.

Adaptive scheduling. The key idea of adaptive scheduling is that while scheduling non-local tasks, the scheduler considers network capacity and DPR of current tasks to avoid data contention caused by either unbalanced locality or limited network capacity. Our network-aware scheduling can be seen as adaptive scheduling.

5.3.4.2 Scheduling Analysis

In this section, we analyze different scheduling models for unbalanced-locality datasets. Table 5.1 lists the notations for this subsection.

M	Number of servers
D	Data size (bytes)
R	DPR of all cores of a server (bytes/s)
C	Network bandwidth of a server (bytes/s)
α	Percentage of local tasks

Table 5.1. Notations

In Spark, the execution time of a job is determined by the last finished task. Unbalanced locality means that some nodes have more data chunks while others have less. If one node has a lot more data chunks than the other nodes, then it has to ship its data to other nodes so that its outgoing network can be a bottleneck. If one node has fewer data chunks than the other nodes, then it has to get data from other nodes so that its incoming network can be a bottleneck. Thus we analyze how scheduling methods perform and how a network-aware scheduler achieves optimal scheduling in these two scenarios.

We first analyze an extreme version of the first scenario: one node has all of the data and there is no data on other nodes. These tasks can either be scheduled as local tasks or non-local tasks. We assume α is the percentage of local tasks and $(1-\alpha)$ is the percentage of non-local tasks. The execution time of tasks on Node 1 is $\frac{\alpha D}{R}$ and the execution time of tasks on other nodes is $\frac{(1-\alpha)D}{min(C,(M-1)R)}$. So the execution time of all tasks should be

$$T = max(\frac{\alpha D}{R}, \frac{(1-\alpha)D}{min(C,(M-1)R)}) \tag{5.1}$$

According to this equation, we find that the only difference among different scheduling methods is α and we'll see how α is determined in different scheduling models.

Under non-local preferred scheduling, α is generally $\frac{1}{M}$ which means tasks are evenly distributed to all nodes. And the execution time is:

$$T_{np} = \begin{cases} \frac{D}{MR} & \text{if } (M-1)R < C, \\ \frac{(M-1)D}{MC} & \text{if } (M-1)R \geq C \end{cases}$$

Under local preferred scheduling, α is generally close to 1, so we just use 1 here. This means that no non-local task is assigned and all tasks are executed locally. The execution time is:

$$T_{lp} = \frac{D}{R}$$

Under network-aware scheduling, α is determined by data processing rate and network capacity. If $(M-1)R < C$ which means network capacity is sufficient for all non-local tasks, $\alpha = \frac{1}{M}$. If $(M-1)R \geq C$ which means network capacity is not enough, we only assign non-local tasks that can be digested by network. Since local task DPR is R and non-local task DPR is C, the fraction of local tasks is $\alpha = \frac{R}{R+C}$. So the execution time is:

$$T_{na} = \begin{cases} \frac{D}{MR} & \text{if } (M-1)R < C, \\ \frac{D}{R+C} & \text{if } (M-1)R \geq C \end{cases}$$

Table 5.2 gives a comparison of the three models. From this table, we can find that while $(M-1)R < C$, non-local preferred and network-aware methods performs M times better than local preferred method because they make use of those nodes without data to help accelerate execution. But while $(M-1)R \geq C$, there is difference between non-local preferred method and network-aware method. If we assume M is very large, the execution time of non-local preferred method is approximately $\frac{D}{C}$. For some data-intensive jobs like TPC-H queries, $C << R$, so non-local preferred method can be much worse than local preferred method because of data contention. Network-aware method can double outperform both non-local preferred method and local preferred method under the condition $C \approx R$.

	$(M-1)R < C$	$(M-1)R \geq C$
Non-local Preferred	$\frac{D}{MR}$	$\frac{(M-1)D}{MC}$
Local Preferred	$\frac{D}{R}$	$\frac{D}{R}$
Network-aware	$\frac{D}{MR}$	$\frac{D}{R+C}$

Table 5.2. Execution time of three scheduling models in scenario 1

Next in the second scenario, most nodes have the same amount of data and there is no data on one node (Fig 5.2). We assume that α is the fraction of local tasks and all non-local tasks are executed by Node 1. The execution time of tasks on Node 1 is $\frac{(1-\alpha)D}{min(R,C)}$ and the execution time of tasks on other nodes is $\frac{\alpha D}{(M-1)R}$. So the execution time of all tasks should be

$$T = max(\frac{(1-\alpha)D}{min(R,C)}, \frac{\alpha D}{(M-1)R}) \tag{5.2}$$

We will also analyze different methods by finding how α is determined.

Under non-local preferred scheduling, α is generally $\frac{M-1}{M}$ which means tasks are evenly distributed to all nodes. And the execution time is:

$$T_{np} = \begin{cases} \frac{D}{MR} & \text{if } R < C, \\ \frac{D}{MC} & \text{if } R \geq C \end{cases}$$

Under local preferred scheduling, α is generally larger, so we just use 1 here. This means that no non-local task is assigned and all tasks are executed locally. The execution time is:

$$T_{lp} = \frac{D}{(M-1)R}$$

Under network-aware scheduling, if $R < C$ which means network capacity is sufficient for all non-local tasks, $\alpha = \frac{M-1}{M}$. If $R \geq C$ which means network capacity is not enough, we only assign non-local tasks that can be digested by network. The fraction of local tasks is $\alpha = \frac{(M-1)P}{(M-1)P+C}$. So the execution time is:

	$(M-1)R < C$	$(M-1)R \geq C$
Non-local Preferred	$\frac{D}{MR}$	$\frac{D}{MC}$
Local Preferred	$\frac{D}{(M-1)R}$	$\frac{D}{(M-1)R}$
Network-aware	$\frac{D}{MR}$	$\frac{D}{(M-1)R+C}$

Table 5.3. Execution time of three scheduling models in scenario 2

$$T_{na} = \begin{cases} \frac{D}{MR} & \text{if } R < C, \\ \frac{D}{R+C} & \text{if } R \geq C \end{cases}$$

Table 5.3 gives a comparison of the three models in this scenario. From this table, we find that while $(M-1)R < C$, the 3 methods performs almost the same when M is large. But while $R \geq C$, especially if $R >> C$, the execution time of non-local preferred method is much worse than the other 2 methods.

From our above analysis, we see that non-local preferred method is weak in data-intensive jobs and local preferred method is weak in CPU-intensive jobs. But our network-aware method performs well in both cases and achieves optimal scheduling. Here we only picks 2 simple scenarios to show a theoretical comparison. In practice, there can be more complicated circumstances and the randomness of task assignment may also affect the performance. We do not claim network-aware scheduling can always achieve optimal scheduling. And usually in real data centers, there is data on every node, so all nodes would execute local tasks at the beginning, and non-local tasks emerge when there is no more local task on an idle node. So the difference between the three scheduling methods may be observed in the latter portion of a stage's execution.

5.4 Implementation

In this section, we first introduce the overall architecture of our implementation and then describe the functionality of each component in detail. Finally we show how they work together to achieve network-aware scheduling in a real Spark cluster.

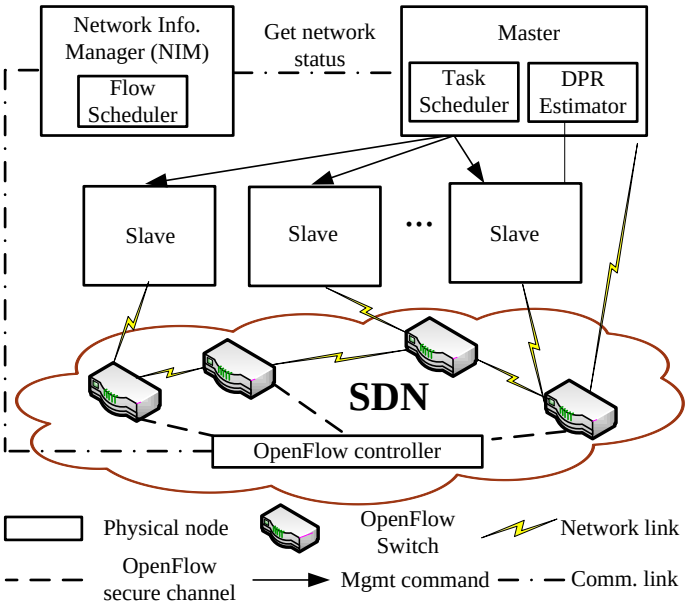


Figure 5.4. System Architecture of Firebird

5.4.1 Architecture

We implemented a system called Firebird using Spark 1.3.1 in conjunction with software-defined network. Our system consists of two modules: network module and scheduling module. The network module is a network information manager (NIM) and flow scheduler. The scheduling module consists of two components: DPR estimator and task scheduler. Figure 5.4 shows the architecture of our system.

5.4.2 Network Information Manager

The NIM obtains and updates information and state of the current network by communicating with the OpenFlow controller. The network information includes the network topology (hosts, switches, ports), queues, links, and their capabilities. The NIM also hosts the switch information such as its ports' speeds, configurations, and statistics. It is important to keep this information up-to-date as inconsistencies can lead to under-utilization of network resources as well as poor task scheduling. The NIM maintains a network status snapshot by collecting traffic information from OpenFlow switches periodically.

When a scheduler sends an inquiry to the NIM to inquire the available bandwidth between M and N, the NIM returns the maximum bandwidth of all paths from M to N.

5.4.3 Flow scheduler

In NIM, we follow the flow scheduler design of Hedera [15], i.e., the scheduler aims to assign flows to *nonconflicting* paths.When a new flow arrives at one of the edge switches, according to OpenFlow protocol, a "PacketIn" message is sent from the switch to the controller. The first packet of the flow is delivered to the controller. The controller examines the 10 tuple packet information and forwards the information to the flow scheduler. Based on the packet information, the flow scheduler identifies the source and the destination of the flow. The flow scheduler compares the available bandwidth for all the candidate physical paths for this flow. It then chooses the path that has the maximum available bandwidth. The best path is decomposed into multiple hops. For each hop, the flow scheduler uses the path change handler to configure the path, i.e., asks the OpenFlow controller to send a "FlowMod" message to the switch to modify a switch flow table. Note that, we again assume that more available bandwidth will make the flow run faster and this approach is only "locally" optimal for this flow but may not be "globally" optimal for all the flows.

In our initial experiments, we find that in a dedicated network, the network bottleneck is usually the node network capacity rather than intermediate path capacity. So flow scheduler shows little improvement. But when network is shared with other applications, the flow scheduler can significantly reduce the risk of network contention. So we implement flow scheduler to enable our system to fully utilize network resource in both cases.

5.4.4 DPR Estimator

In traditional cluster computing platforms like Hadoop, since the bottleneck for data intensive jobs is disk I/O, it's hard to estimate CPU data processing rate. Since network capacity is not a problem, it's also unnecessary to schedule non-local tasks according to DPR. But in in-memory cluster computing platforms such as Spark, things are different.

Memory I/O replaces disk I/O which means local I/O is much faster and is no longer the bottleneck. Hence we can compute CPU DPR of a finished task by simply dividing data size by execution time. For simplicity, we assume that all nodes in the cluster are homogeneous. We observe that the DPRs of tasks in the same stage of a job are similar. So the DPR of a new task can be estimated as the average DPR of finished tasks in current stage.

5.4.5 Task Scheduler

For process-local and node-local tasks, since they don't generate network traffic, our task scheduler follows the design of default task scheduler. According to Algorithm 4, when a node is idle, the scheduler first searches for tasks in node-local sets. If there is no task returned, it continues to search in rack-local and non-local task set. For each task in rack-local and non-local task sets, DPR estimator first estimates the bandwidth that the task requires. Then task scheduler inquires NIM to get the available bandwidth between data node and execution node. Since the real flows are generated several seconds after tasks are scheduled, to avoid contention of simultaneously scheduled tasks, task scheduler also records the reserved bandwidth for scheduled tasks before the real flows are generated. By combining NIM's bandwidth information and reserved bandwidth, task scheduler computes the real available bandwidth to see if it is enough to accommodate this task. If so, the task is scheduled to the idle executor, and if not, it tries the next task. If there are no tasks that can be scheduled, the executor remains idle until next heartbeat.

5.5 Experimental Evaluation

In this section, we describe our experimental settings and the experimental results.

5.5.1 Experimental Settings

5.5.1.1 Hardware settings

Our experimental testbed as shown in Figure 5.5 consists of 17 physical nodes N_{0-16}. Each of the machines has an Intel Xeon E5-2440 2.4GHz six core CPU, 32GB of RAM,

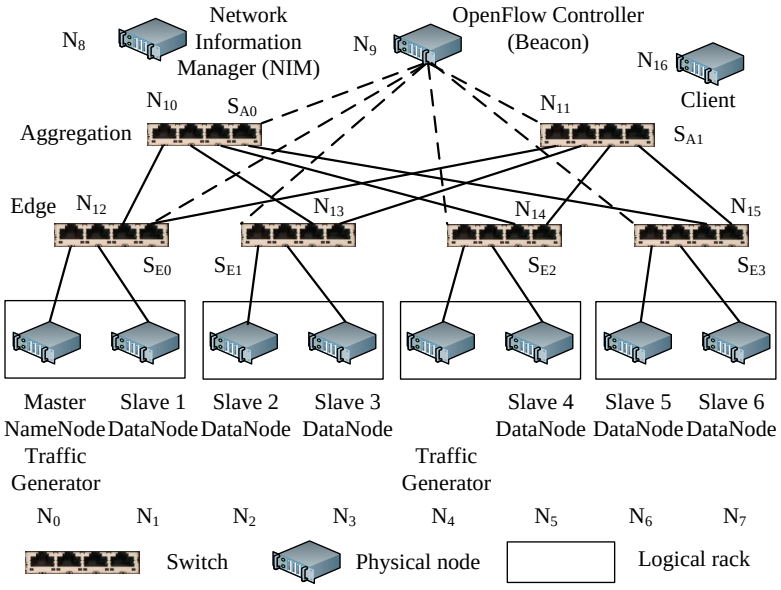


Figure 5.5. Spark Cluster Setup

1TB 7200rpm disk running Linux with kernel 2.6.32. Each of the machines has two Gigabit ethernet NICs. Six of the machines N_{10-15} are installed with a 4-port Gigabit NetFPGA card serving as a 4-port OpenFlow switch. Hence, N_{10-15} operate as OpenFlow switches. Seven of the machines N_{1-3}, N_{5-7} are used for both Spark and HDFS deployment with one master (at N_0) and six slaves (at N_{1-3}, N_{5-7}). N_4 is used for generating neighborhood network traffic. N_8, N_9 and N_{16} are used to run network information manager(NIM), Openflow Controller, and client emulator, respectively.

There are two networks in our testbed, a management network and an OpenFlow network. The first NIC of each machine is connected by a Gigabit Cisco switch, which forms the management network. The second NIC of N_{0-7} is connected by a 4-port Gigabit NetFPGA OpenFlow switch, which forms the OpenFlow network. All Spark traffic goes through the OpenFlow network. We use an open source OpenFlow controller Beacon [1] as our OpenFlow controller. The OpenFlow network is built following a similar network topology in prior work [15] to enable multiple paths from a source host to a destination host.

[1] https://openflow.stanford.edu/display/Beacon/Home

5.5.1.2 Benchmark

We use three different types of jobs: TPC-H Q6, Kmeans and Word Count as examples to evaluate the performance of our method.

TPC-H Query 6. TPC-H queries are known as data intensive jobs and the DPR of core is 500Mb/s-800Mb/s in our experimental environment. We pick Query #6 (Q6) because it's easy to analyze: its reduce phase is very short and most of execution time is spent on map tasks. We show details of Q6 below.

```
select
  sum(l_extendedprice*l_discount) as revenue
from
  lineitem
where
  l_shipdate >= '1994-01-01'
  and l_shipdate < '1995-01-01'
  and l_discount >= 0.05 and l_quantity < 24
  and l_discount <= 0.07;
```

We use TPC-H dataset with scaling factor of 100GB generated by dbgen and a chunk size of 512MB. This query is a single-stage job with a lot of map tasks but only 1 reduce task.

Word Count. Word count is a less data intensive job compared with TPC-H queries. The DPR of one core is 250Mb/s-350Mb/s in our experimental environment. In our experiment, the dataset of word count is 50G and the chunk size is 512MB. It's also a single-stage job.

Kmeans. In contrast to TPC-H and Word Count, Kmeans is a multi-stage CPU intensive job. Since it's multi-stage, the DPR varies in different stages, but in the range of 10Mb/s-200Mb/s in our experimental environment. In our experiment, the dataset size is 5GB and consist of 30 clusters of 3D points. The chunk size of the dataset is 64MB.

We run each job multiple times and discard the result of the first run because in the first run data is not cached in memory.

5.5.2 Dedicated Network

In this section, we first consider an idle (dedicated) network that only sees Spark traffic. We compare the performance of three types of schedulers mentioned in section 5.3.4. Default delay scheduler with a delay of three seconds performs as non-local preferred scheduler. Local preferred scheduler is implemented by setting the waiting time longer than task execution time so that non-local tasks are not scheduled. In order to test Firebird's performance under different environment, we vary the number of cores that can be used on each server by varying the configuration file of Spark.

Table 5.4 shows the number of data chunks on each node for each workflow. We can see from the table that naturally distributed data is not well balanced.

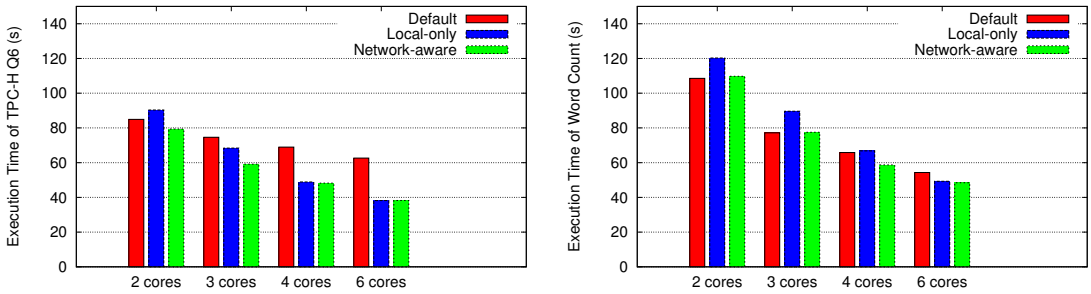
	S1	S2	S3	S4	S5	S6
TPC-H Q6	28	21	17	24	31	28
Word Count	22	19	12	16	19	18
K-Means	15	11	10	19	12	16

Table 5.4. Number of chunks on each node.

Figure 5.6 shows the performance of the three schedulers with different number of cores. From figure 5.6(a), we can see that network-aware scheduler performs almost the same as local preferred scheduler and up to 39% better than non-local preferred scheduler in 4 core and 6 core scenarios. Network-aware scheduler performs 5-20% better than the other two schedulers in 2 core and 3 core scenarios.

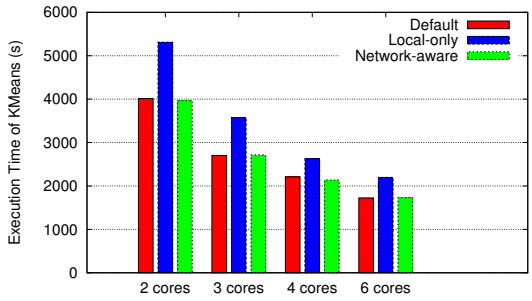
From Figure 5.6(b), we observe that our network-aware scheduler is 10% better than local preferred scheduler in 4-core and 6-core scenarios and is 10% better than non-local preferred scheduler in 2-core, 3-core and 4 core scenarios.

From Figure 5.6(c), we can see that for CPU intensive jobs like KMeans, local preferred scheduler performs worse than the other 2 schedulers. In 2-core scenario, local preferred scheduler is 24% worse than the other 2 schedulers. This result justifies our conclusion in Section 5.3.4.2.



(a) Execution time of TPC-H q6

(b) Exectution time of word count



(c) Execution time of Kmeans

Figure 5.6. The execution time of different jobs while using 3 different schedulers

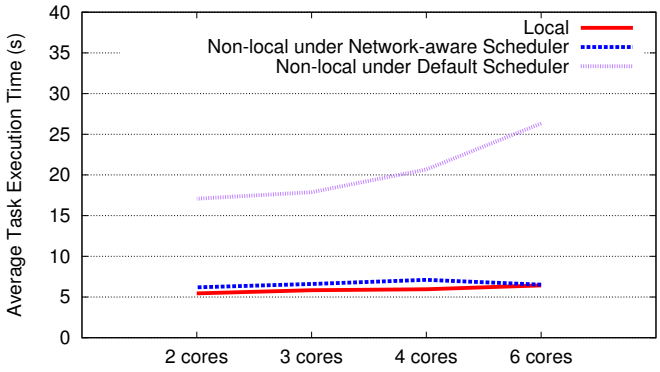


Figure 5.7. Average execution time of local and non-local tasks

The experimental results are in line with our analysis in Section 5.3.4.2 that non-local preferred method is weak in data-intensive jobs and local preferred method is weak in CPU-intensive jobs. From these three graphs, we can also see that our network-aware scheduler is always the best of the three schedulers no matter the job is data intensive or CPU intensive. When the job is CPU intensive and the number of cores is small, network-aware scheduler performs similar to non-local preferred scheduler. When the job is data intensive and the number of cores is larger, network-aware scheduler performs similar to local preferred scheduler. Also network scheduler performs better than both for scenarios in between.

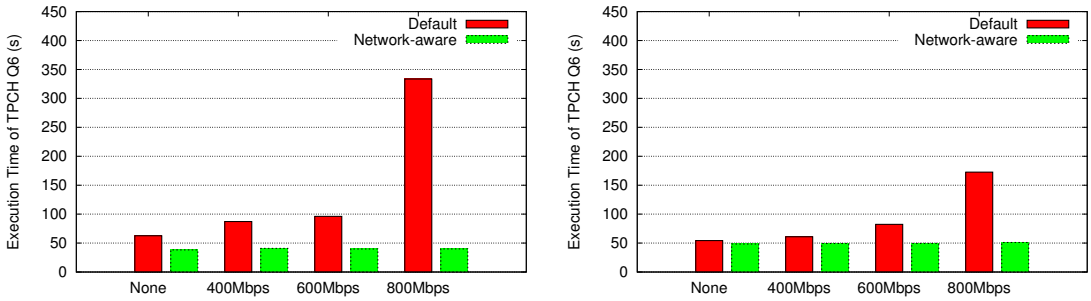
Figure 5.7 shows the average execution time of tasks in TPC-H Q6. We observe that as the number of cores increases, when non-local preferred scheduler is used, the execution time of non-local tasks increases because of network contention. While using network-aware scheduler, the execution time of non-local tasks doesn't increase and is still close to local tasks. This means that scheduling tasks according to available bandwidth is reasonable and it can avoid outlier tasks efficiently.

These experiments show that our network-aware scheduler is efficient for both CPU intensive and data intensive jobs. Hence it doesn't need any system tuning while executing different jobs which is very convenient to users in practice.

5.5.3 Shared Network

In real data centers, network is often shared by different types of applications. The performance of Spark can be influenced by network traffic generated by other applications. We expect that by being aware of current network status, our network-aware scheduler can improve Spark's performance under shared network.

To create background traffic, we add a flow generated by iperf from an idle node (traffic generator) to N3. From figure 5.8, we can see that as the iperf flow becomes larger, the execution time of using default scheduler is highly influenced. Since N3 has less data than



(a) Execution time of TPCH Q6 under outside network traffic

(b) Execution time of WordCount under outside network traffic

Figure 5.8. Network traffic generated by other applications can largely influence the performance of Apache Spark

other nodes, after local tasks are finished, it would be assigned non-local tasks and read data from other nodes. Since the background traffic limits the available bandwidth that N3 can use, the data transfer speed of these non-local tasks is slow and the execution time of the whole job is influenced. While the iperf flow is 800Mb/s, the execution time becomes five times longer in TPCH Q6 and 3 time longer in Word Count than that with no background traffic. But network-aware scheduler is not influenced much by this background traffic because it can acquire the network status and avoid assigning non-local tasks to N3. The experiment shows that network-aware scheduler performs nine times better in TPCH Q6 and three times better in Word Count than default scheduler under 800Mb/s background traffic.

5.6 Related Work

In this section, we discuss related work from two areas: scheduling and networking.

In the scheduling context, there has been a plethora of work on optimizing MapReduce performance. CoHadoop [34] extends HDFS and adds metadata to NameNode so that it allows applications to control where data are stored. HaLoop [24] caches the reused data on local disks and then improves performance by leveraging this data locality in the scheduler. Hyracks [21] is a data-parallel runtime platform designed to perform data-processing tasks

on large amounts of data using large clusters. Because data is processed in a pipelined manner, Hyracks can push a very large amount of data to the network thereby potentially creating extreme network contention during the run-time.

All of the above work treats the network as a black box. Our contribution is complementary by considering network status when scheduling task.

From the networking perspective, Wang *et al.* [79] propose application-aware networking, and argue that distributed applications can benefit from communicating their preferences to the network control-plane. Yap *et al.* have also advocated for an explicit communication channel between applications and software-defined networks, in what they called software-friendly networks [84]. PANE [36] proposes design, implementation, and evaluation of an API for applications to control a software-defined network. Sinbad [27] is a system that identifies imbalance and adapts replica destinations to navigate around congested links. It can be seen as a network-aware data placement method. Coflows [28] is a a networking abstraction that allows cluster applications to convey their communication semantics to the network. Based on [28], Varys [29] enables data-intensive frameworks to use coflows to optimize network scheduling. Both [27] and [29] assume data center is dedicated and all network flows can be controlled by their system. While our system can work in both dedicated and shared network environments with dynamic uncontrolled flows. Ousterhout *et al.* [62] argue that network bandwidth doesn't have significant impact on the performance of big data platforms because disk is generally the bottleneck of data I/O. However, Trivedi *et al.* [77] show that network bandwidth matters for Spark jobs. We find that Ousterhout's opinion is true when all data is stored on disk and network bandwidth is adequate. But if input data is stored in memory, their opinion doesn't hold and network bandwidth becomes important for improving Spark's performance.

5.7 Conclusions

In this chapter, we propose a network-aware scheduling method that exploits software-defined networking and implement it in Spark. Unlike previous work, in which Spark works independently from the network underneath, our system enables Spark and the networking layer to work together to improve network utilization and reduce job execution times. As our experiments show, this improvement can be huge when network contention is heavy. We show that our network-aware scheduling method doesn't require any system tuning when facing different kind of jobs.

CHAPTER 6

PLACEMENT STRATEGIES FOR A NFAAS CLOUD

Enterprises that host services in the cloud need to protect their cloud resources using network services such as firewalls and deep packet inspection systems. While middleboxes have typically been used to implement such network functions in traditional enterprise networks, their use in cloud environments by cloud tenants is problematic due to the boundary between cloud providers and cloud tenants. Instead we argue that network function virtualization is a natural fit in cloud environments, where the cloud provider can implement Network Functions as a Service using virtualized network functions running on cloud servers, and enterprise cloud tenants can employ these services to implement security and performance optimizations for their cloud resources. In this chapter, we focus on placement issues in the design of a NFaaS cloud and present two placement strategies—tenant-centric and service-centric—for deploying virtualized network services in multi-tenant settings. We discuss several tradeoffs of these two strategies. We implement a prototype NFaaS testbed and conduct a series of experiments to show to quantify the benefits and drawbacks of our two strategies.

6.1 Introduction

Traditionally enterprises have used middleboxes to implement various security and performance functions in their enterprise networks. These network functions include firewalls, deep packet inspection systems, and proxy caches among others.

As enterprise networks have become more dynamic in their needs, the use of specialized hardware middleboxes to implement network functions has become a drawback rather

than a benefit. Network function virtualization (NFV) has emerged as a potential solution to enable enterprises to flexibly deploy and reconfigure network functions on-demand to handle network dynamic, scalability and security needs.

At the same time, enterprises have begun to move backend applications from in-house data centers to the cloud. The cloud's pay-as-you-go model and on-demand resource allocation abilities are attractive to enterprises for hosting their application in a more cost-effective fashion while also handling workload dynamics. Indeed many new enterprises are entirely cloud-based where their entire IT infrastructure—both internal and external facing applications—are cloud based.

In such scenarios, an enterprise needs to implement network security and performance functions in the cloud to guard their cloud-based servers—in order to implement the same network security and performance policies they would have implemented in their enterprise network. Since deploying or leasing middleboxes in a public cloud is not always possible, the use of NFV to implement these functions using commodity cloud servers is an attractive option.

In many cases, the cloud providers may themselves offer network functions as a service (NFaaS) to cloud-based enterprises so that they can lease storage and servers as well as appropriate network services to configure and guard their resources.

Motivated by such scenarios, in this chapter, we study the design of a NFaaS cloud. Specifically we assume that a NFaaS cloud provides different network functions (e.g., firewall, IDS, caching, etc) that can be leased by a cloud-based enterprise for their cloud IT infrastructure.

We specifically examine how a cloud provider should design a multi-function multi-tenant NFaaS cloud from the placement perspective. We propose two different placement strategies for a multi-tenant NFaaS cloud and discuss the advantages and disadvantages of each approach. We conduct an experimental evaluation of these approaches using a small prototype NFaaS cloud and quantify their benefits and overheads. We believe that our

insights can provide design guidelines on the placement of virtualized network functions in future NFaaS clouds.

6.2 Background and Related Work

In this section, we present background on cloud computing and network function virtualization.

6.2.1 Cloud Computing Background

We consider an enterprise that hosts its IT needs using cloud resources. The enterprise is assumed to acquire servers and storage resources from an Infrastructure-as-a-Service (IaaS) cloud to run its applications on this cloud-based infrastructure. Such a cloud-based IT infrastructure needs to incorporate network security policies and performance optimization—like an in-house enterprise network. That is, network traffic coming into the enterprise's cloud servers needs to undergo security checks (e.g., using firewalls, deep packet inspection systems, virus scanners, etc.) and may be subject to performance optimization (e.g., using in-network caches).

Traditionally such features have been implemented using middleboxes—dedicated hardware devices that are deployed in the network—to perform these functions.

In a cloud-based setting, we assume that the infrastructure cloud provider provides these functions as cloud services—allowing the enterprise to lease firewalls, caches, etc. similar to leasing servers and storage. We assume that the cloud provider supports a rich mix of network services that may be needed by an enterprise. The same benefit as infrastructure clouds hold in this case such as the pay as you go model, the ability to scale up service capacity, and on-demand resource allocation.

6.2.2 Network Function Virtualization

While the cloud provider can provide cloud-based network services by deploying middleboxes on behalf of cloud custom, it is more effective to use network function virtualization to implement these services using commodity servers.

In this case, a service such as a firewall, IDS or a cache is implemented as software that runs inside a virtual machine and the VM runs on commodity servers. Virtualizing network functions has become popular since it offers a number of benefits over the middlebox approach – such as reducing capital cost, shortening deployment cycles and the ability to handle the needs of a dynamic network.

In our scenario, NFV is a natural fit since the cloud provider is already leasing servers and can use these commodity servers to deploy virtualized functions and offer the network functions as a service (NFaaS) to customers.

A customer can lease various network functions as cloud services and chain them together to implement the desired network security policies and performance optimization. For instance, the customer (i.e., the enterprise) could lease a firewall service, an IDS service, a DNS service, a cache service and configure them so that network traffic flow through them transparently.

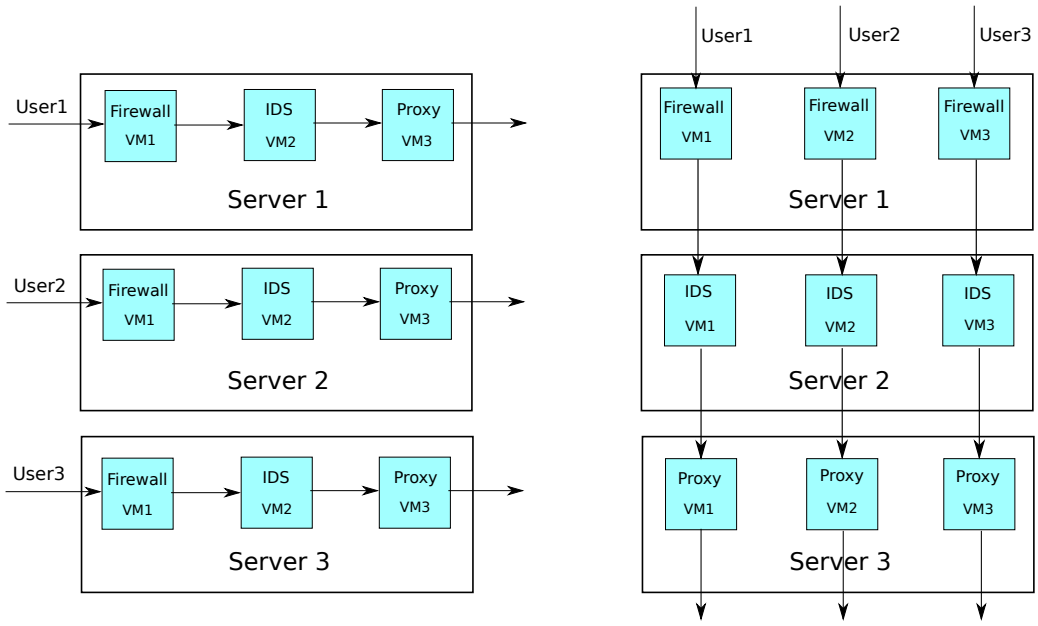
6.3 Placement Issues in a NFaaS Cloud

Many design issues arise in deploying a NFaaS cloud. In this chapter, we specifically focus on placement issues in a multi-tenant NFaaS cloud, which we discuss next.

6.3.1 Placement Strategies

A cloud provider can employ one of two placement strategies in a multi-function multi-tenant NFaaS cloud: tenant-centric and service-centric.

In a tenant-centric approach, VMs comprising all network services leased by a tenant are mapped onto a single server or a group of co-located servers (e.g., servers on the same



(a) Tenant-centric Placement Strategy.

(b) Service-centric Placement Strategy.

Figure 6.1. Two placement strategies of deploying network functions in a multi-tenant NFaaS cloud

network rack). Fig 6.1(a) shows tenant-centric placement for three different tenants, each of whom is using three different network services. While the figure shows all services resident on a single server, higher capacity services may require multiple servers that are co-located.

In contrast, a service-centric placement approach maps VMs running the same service for different tenant on the same server or on a co-located group of servers. Fig 6.1(b) which depicts this approach shows that each server (or rack(s)) hosting the same service.

6.3.2 Tradeoffs

The two placement strategies offer a number of tradeoffs – both from a cloud provider's and a cloud tenant's perspective.

Network latency: The primary advantage of customer-centric service placement is that it optimizes network latencies of packets as they traverse from one network function to

the next (e.g., from firewall to IDS). Since the services belonging to a tenant are resident on the same server or the same rack, network latency is minimized. In the service-centric placement approach, services belonging to a tenant may reside on different racks, requiring packets to traverse to multiple switches when going from one service to the next.

Flexibility and Scaling: In many scenarios, existing network functions might need to be updated. For example, a tenant may want to implement new network functions in their network, or may choose to replace one IDS with another. In scenarios where a tenant's network traffic is increasing, the resource allocated to a NF may have to be scaled up ("elastic scaling"). In a tenant-centric approach, such reconfiguration require free resources on the server or rack hosting the tenant's current services. If such idle capacity is unavailable, either other tenant have to be moved to other servers/racks to free up resources or the new service has to be placed on a more "distant" server, diminishing the latency advantage of the approach. A service-centric approach does not suffer from such a drawback, since new services (or resizing of existing ones) can be achieved by choosing any server with sufficient capacity without regard to network proximity.

Packing and Resource Utilization: A tenant-centric approach enables hosting of heterogeneous services on the same server. This enable a CPU-intensive (but less I/O intensive) service to be co-located with an I/O-intensive (but less CPU-intensive) one, yielding better utilization of various resources of potentially a denser packing—a strategy that has been successfully employed in general-purpose VM placement.

A service-centric approach hosts homogeneous services (belonging to different tenants) on a server of packing density is determined by the most bottlenecked resources of each service. However, homogeneity of services is not without advantage. Since all service on a server run the same code, it provides opportunities for better memory and cache utilization (e.g., page cache sharing across services). It may also be possible to employ containers (lightweight VMs) rather than VMs to further exploit this homogeneity.

Performance Interference: Whenever multiple services reside on a physical server, there is the potential for performance interference. In the service-centric case, such interference will be cross-tenant, while in the tenant-centric case, the interference will be cross-service interference. Fortunately, resource partitioning features of VMs (or containers) can isolate co-resident services and minimize the impact of such interference.

In summary, whether to user tenant-based or service-based placement depends on the cloud provider's objective – tenant-based deployment has better latency, potentially better packing but less flexibility for reconfigurations, capacity scaling than a service-based approach and vice versa.

6.4 Experimental Evaluation

In this section, we quantify the benefits and overheads of the two placement strategies using an experimental evaluation.

6.4.1 Prototype NFaaS Cloud

We have built a prototype NFaaS cloud using several open-source components. Out prototype provides three network services: (i) a network firewall that is implemented using Linux *IPtables*, (ii) an intrusion detection system that is implemented using *Snort* 2.9.8.2, and (iii) an in-network web cache that is implemented using *Squid* 3.3.8. All three components are popular and widely-used system that we deploy as virtualized services inside virtual machines. The use of virtualization enables benefits such as rapid deployment, flexible placement and flexible resizing when needed. Further, we implement both tenant-centric and service-centric placement in our NFaaS prototype. In both cases, we use Linux bridging to enable a tenant to "chain" network functions as per their needs (see Figure 6.2). We do so by appropriately configuring routing tables on each VM. In tenant-centric placement, services belonging to a cloud tenant are placed on the same server when possible (or

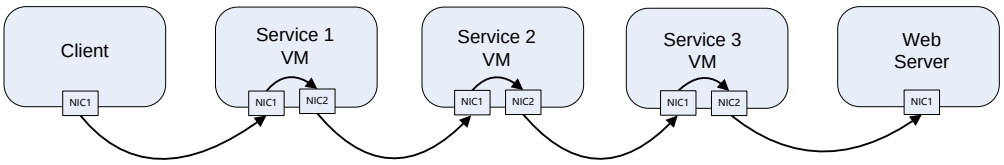


Figure 6.2. Experimental Setup

on nearby servers when VMs are too large to pack onto a single server). In service-centric placement, VMs belonging to a service are packed together for co-location.

6.4.2 Experimental Setup

We deployed our NFaaS prototype on a testbed of five physical server as shown in Figure 6.2. Each server has an Intel Xeon X3430 2.4GHz Quad-Core CPU, two gigabit physical NICs, 8 GB of RAM and 1TB 7200 RPM disk. All machines run Ubuntu 12.04 and use KVM as a virtual machine (VM) hypervisor. We use one server to house clients and one to house web servers belonging to tenants. The other three servers run virtualized network functions inside VMs. Each VM is pinned to a physical CPU core with 2048 MB RAM allocated. We also create two virtual NICs that allow network traffics to traverse from one NIC to the other. Client HTTP traffic is generated using *httperf* [12] on one server, and this traffic traverses through a tenant's three network services before reaching the web server. We monitor and measure resource utilization of various VMs using the *dstat* [13] tool.

We configure the *iptables* firewall with 1000 rules and configure *Snort 2.9.8.2* with the default rule set 2982. Finally, we configure *Squid 3.3.8* to cache frequently accessed web pages. In tenant-based deployment, each physical server contains three VMs belonging to a tenant running different network functions connected as a chain through Linux bridges as illustrated in Figure 6.1(a). In service-based deployment, each physical server contains three VMs running the same network function belonging to different tenants as illustrated in Figure 6.1(b).

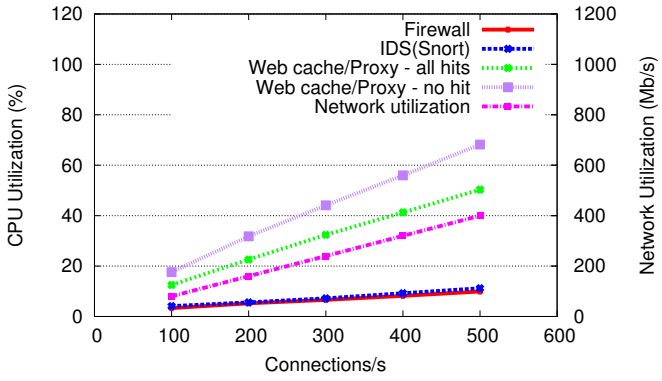


Figure 6.3. CPU and network utilization of network functions

6.4.3 Network Function Micro Benchmark

We first configure our network service based on Figure 6.2 in which each network runs inside its own VM. Client VMs are instructed to contact the web server, at a specified workload intensity, to get a 50 KB web page for 1 minute. All HTTP traffic traverses through these three NFs, and we measure the average resource utilization of NFs on their corresponding VMs. As we increase network traffic, from 100 to 500 connections per second, the CPU utilization of all network functions increase linearly as shown in Figure 6.3. Specifically, the web proxy is more sensitive to increasing network traffic than the firewall and IDS, with up to 60% more CPU consumption. The network utilization also grows directly with increasing network traffic. In addition, we find the disk and memory utilization vary only slightly with increasing network traffic. This is because the memory state of the firewall and IDs depend on factors such as the size of the rules set and not on the workload. Also, while the cache size depends directly on the number of frequently accessed pages, it is not strongly related to the request rate. We thus omit these measurement results but use them in our simulation (see Table 6.1).

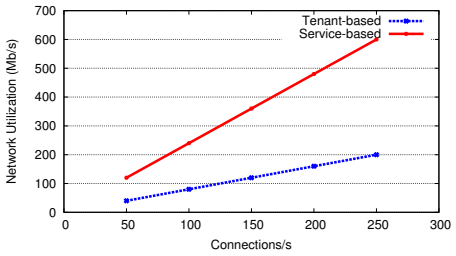
*R*esult: The CPU and network utilization of our services increase nearly linearly with the workload while memory and disk utilization are less sensitive to the request rate.

6.4.4 Virtualization Overhead

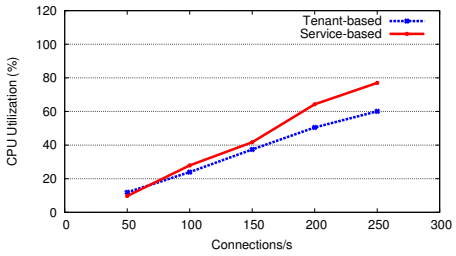
As shown in 6.4.3, NFs are heavily CPU and network consuming. In this experiment, we measure underlying hypervisor's CPU and network utilization for both placement approaches. We set up bare-bone VMs for tenant-based and service-based respectively as shown in Figure 6.1. The network traffic of each tenant is configured to be the same for both approaches. We instrument the hypervisors in both tenant-based and service-based deployments to measure the utilization.

In Figure 6.4(a), we show that the NIC bandwidth consumed by tenant-based deployment is only 1/3 of that consumed by service-based deployment. This is because in tenant-based approach, traffic between network functions goes through internal bridges within a physical server's boundary while in service-based approach, the traffic goes between physical servers through physical NICs. This makes tenant-based approach much more bandwidth efficient than service-based approach. However, this bandwidth saving is not acquired for free. From Figure 6.4(b), we can see that although NIC throughput of service-based approach is three time than that of tenant-based, the CPU utilization of service-based approach is only slightly higher. When network throughput of physical server is fixed, the CPU utilization of tenant-based approach is three times of service-based approach (Figure 6.4(c)). This is because forwarding packets through internal bridges in tenant-based approach is CPU consuming and brings overhead for the hypervisor.

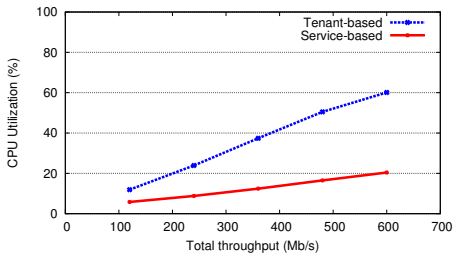
Conclusion: Tenant-based deployment is less NIC bandwidth consuming than tenant-based deployment because packets go through internal bridges between network functions. Both tenant-based and service-based deployment incur non negligible hypervisor CPU overhead that increases linearly with network traffics. Specifically, to achieve the same throughput for each tenant, service-based deployment requires much more network bandwidth and slightly more CPU resources than service-based deployment.



(a) Networkutilization of tenant-based approach and service-based approach



(b) CPU utilization of tenant-based approach and service-based approach



(c) CPU utilization of tenant-based approach and service-based approach when network throughput of each physical server is fixed

Figure 6.4. Hypervisor CPU and Network utilization of the two deployments.

6.4.5 Packing Efficiency

Next, we compare the packing efficiency of the tenant-based and service-based approaches. The packing efficiency is measured as the number of servers required to handle the same workload level—that is, we simulate different scenarios in which NFaaS customers request different resources for their network functions. Specifically, we divide NFaaS customers into three groups, i.e., small, medium and large, based on their resource requirements. In Table 6.1, we list resource requirements of different customers in the form of CPU and Memory. These requirement values are taken directly from our micro benchmarks and correspond to NFs handling 300, 600 and 1200 connections per second respectively. In addition, the NFaaS cloud is simulated with four different types of physical servers and we show the corresponding configurations in Table 6.2. Finally, we choose to use *best fit* algorithm that places all network functions of next customer to eligible servers

with smallest amount of available CPU cores, for both deployment modes. Here a server is eligible only if it has enough available CPU and memory capacity.

We first look at the total number of small physical servers required to place large-sized customers. As shown in Figure 6.5(a), deploying NFs service-basedly can save up to 30% server resources compared to tenant-based deployment. In essence, service-based deployment outperforms tenant-based deployment because it has more eligible servers to choose from during each placement decision. For example, any physical server with less than 1.4 CPU cores is not eligible for tenant-based deployment.

Next we compare the number of large servers required to host a thousand customers with different resource requirements. We observe that the benefits of service-based deployment over tenant-based deployment increase from 3% to 30% when we need to handle customers with more resource demands. Similarly, this is because a larger customer leads to more potential waste of spare server resources.

Finally, we total the number of servers required to run NFs for one thousands customers of varying resource requirements.

In Figure 6.5(c), we show that tenant-based placement performs worse when only has access to small servers with limited resources. As we increase the server size, the difference between tenant-based and service-based placements converges. This is because having access to larger servers offsets the resource constraints exerted by tenant-based placement.

After packing efficiency simulation, we perform a following simulation to compare the elasticity of the two approaches. We assume that a user may change the size of his VMs in use in two scenarios. In the first scenario, a user upgrades or degrades a single VMs because he wants to change the configuration of a specific network function (e.g., add or delete rules in firewall) . In the second scenario, a user upgrades or degrades his VM suite because of network throughput change. If a physical server does not have enough resource to host an upgraded VM, the VM is migrated to another physical server. When using tenant-based method, VMs of the same user should still be on the same physical server after migration.

Degraded VM remains on the same physical server but extra CPU, memory and bandwidth resources are released. In our simulation, each user chooses to upgrade, hold or degrade his VMs with equal probability after VMs start. Each simulation is executed 100 times and we use the average number of migrations to measure the two methods' ability to handle elasticity of demand.

In Figure 6.5(d), we show that tenant-based placement performs 58% more migrations in single VM modification scenario and 104% more migrations in VM suite modification scenario than service-based placement. Tenant-based placement performs bad because whenever resources are insufficient for upgraded VMs, tenant-based placement has to migrate all three VMs of a user while service-based placement may only need to migrate one VM. Tenant-based placement performs worse in VM suite modification scenario because it's more possible to cause resource insufficiency when three VMs on the same physical server upgrade at the same time.

Result: The service-centric approach has higher packing efficiency than a tenant-centric deployment, with up to 30% fewer physical servers. In particular, tenant-based approach has poor packing efficiency when placing customers with higher resource demands or on to smaller servers.

6.4.6 End-to-end Performance

In this experiment, we evaluate the end-to-end response time and bandwidth of deploying a network service using either tenant-centric or service-centric placement. We create a logical topology where user generated requests have to traverse firewall, IDS and then web proxy in sequence. Similar to Figure 6.1, for the tenant-centric approach, we place all three services on the same quad-core server, while for the service-centric approach, we place these services on three different servers.

For each physical topology, either tenant-based or service-based, we begin the experiment by sending HTTP requests to fetch web pages from the web server. Specifically, we

Customer Type	Throughput	Firewall	IDS	Web Proxy
Small	250Mbps	(0.1, 71)	(0.1, 462)	(0.5,147)
Medium	500Mbps	(0.2, 71)	(0.2, 470)	(1, 151)
Large	1Gbps	(0.4, 71)	(0.4, 485)	(2, 160)

Table 6.1. NFaaS customer configuration for simulations. Resource requirements are specified in the form of network throughput. We also show the corresponding number of CPU cores and memory (MB) required by each network function in the table.

Server Type	Cores	Memory (GB)	Network Bandwidth
Small	4	8	10Gbps
Medium	6	12	15Gbps
Large	8	16	20Gbps
Xlarge	12	24	30Gbps

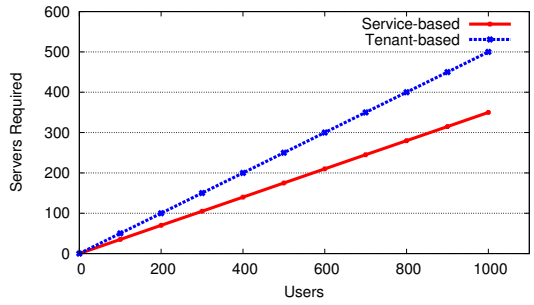
Table 6.2. Server configurations for our simulations.

limit the request rate to 100 connections per second to avoid bandwidth congestion and make sure that all user requests go through all three middleboxes. We run each experiment for one minute and measure the average response time over all requests. As shown in Figure 6.6, tenant-centric placement incurs up to 20% lower latency, for both web page sizes, under similar server load. The response time difference is because packets traversing through multiple physical servers in the service-centric placement. We expect to observe an even higher performance gap if middleboxes were running in different racks for service-based deployment.

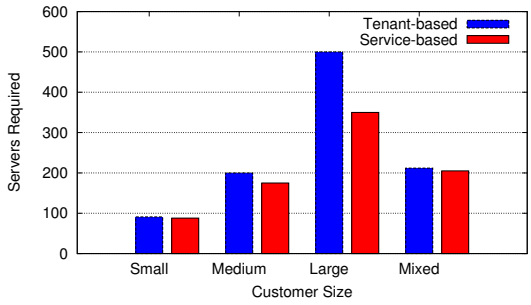
Result: Deploying network functions using a tenant-centric approach can lead to better end-to-end response time than the service-centric approach.

6.5 Related Work

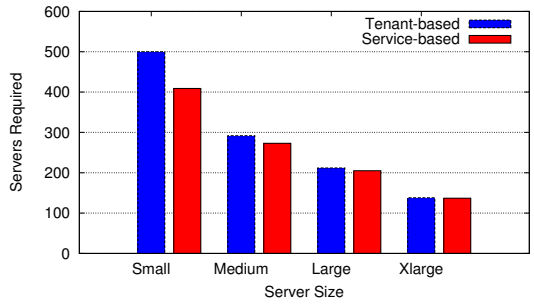
The promise of implementing software-based network functions and running them on commodity high-volume servers has attracted significant attention from the research com-



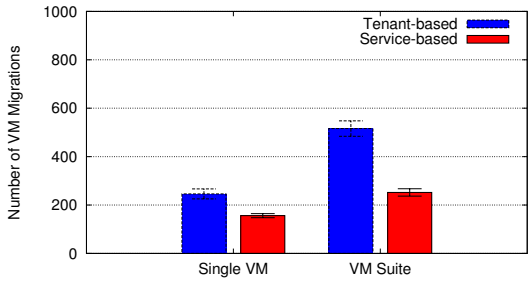
(a) Packing for Large-sized Customers.



(b) Impact of Customer Types. Mixed means equal number of each type of customers.



(c) Impact of Server Sizes.



(d) Number of migrations in two elasticity scenarios

Figure 6.5. Comparison of packing efficiency and elasticity for tenant-based and service-based deployment under different scenarios.

munity [35]. In NFV, individual network functions are implemented in software and use virtualization to replace its hardware counterpart. Efforts to improve packet processing performance using commodity NICs [14, 65] and packet transfers in the virtualized environments [66, 46] have significantly improve the performance of running virtualized NFs in the commodity settings. Almost always, NFs are used in combination to form network services and these end-to-end services introduce new problems in managing the end-to-end performance [55, 69, 63, 38], especially in the multi-tenant cloud environments [71, 23] . In addition, it is beneficial to provide dynamic scaling ability to cloud-based NFs by either migrating to a more powerful VM [19, 51] or replicating network states with and without explicitly considering state consistency [64, 39]. Unlike traditional VM placement [25, 22, 47, 57] where the VM resource needs may not be known in advance, in

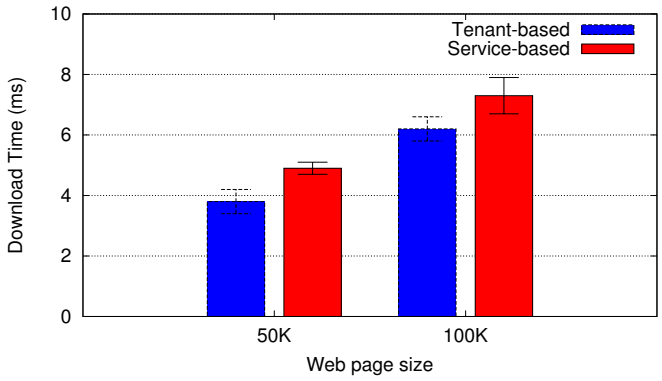


Figure 6.6. Comparison of response time. Tenant-based deployment can achieve up to 20% improvement comparing to service-based deployment when all network functions run in the same rack.

our case, the characteristics of each network function is known in advance, enabling more optimal placement.

6.6 Conclusions

In this chapter, we studied how network functions as a service can be deployed in cloud environments. We studied two different placement strategies, tenant-centric and service-centric, for deploying network functions in the cloud. Our experimental evaluation using a NFaaS prototype and simulations show that tenant-centric placement can achieve better network performance because it avoids cross-service traffic from traversing network switches, which saves physical bandwidth and reduces network delay. In contrast, the service-centric approach is easier to manage and deploy; simulations using real measurements show that this approach yields better resource utilization in the cloud. Our results demonstrate the tradeoffs of the two approaches and provide guidance on which approach to choose based on the overall design goals.

CHAPTER 7

SUMMARY AND FUTURE WORK

7.1 Thesis Summary

This thesis has explored how application-aware technologies can be used to improve resource management for the cloud. I have proposed a set of automated techniques to reduce cost or enhance performance of different applications that run in the cloud.

First we proposed to use bidding strategies together with virtualization techniques to run always-on internet services on cloud spot markets. Our evaluation demonstrated the feasibility of using our proactive approach to provide availability levels that are close to levels desirable for always-on services, at nearly 1/5 to 1/3 of the cost of the traditional approach of using on-demand servers.

Next, we proposed to use software-defined network to help create a communication channel between networks and applications. We apply this technique to big data platforms which require a lot of data shipping between nodes.

We found that for data intensive jobs in a Hadoop cluster, network contentions may happen and influence the performance of the whole cluster. We demonstrated that by enabling Hadoop and underneath network work with each other, the network contention in Hadoop cluster can be effectively avoided. By using collaborative schedulers in Hadoop, the performance of Hadoop is improved by up to 37% and the impact of neighborhood traffic can be reduced by 60%.

Then we presented the task management problems in current Spark platform. We found that previous schedulers can not achieve good performance for different tasks in different network conditions. I proposed a network-aware scheduling method that can adaptively

schedule tasks according to current network conditions and tasks' network demand. The evaluation results show that our system that implemented network-aware scheduling outperforms default Spark by up to 9 times in some cases and can always achieve good performance in different scenarios without system tunning.

Finally, we discussed how network as a service can be deployed in cloud environment. We proposed two placement methods of chaining network functions. By conducting a series of experiments and simulations, we showed the trade-offs between the two methods and provide guidance for cloud service providers to make the choice.

7.2 Future Work

In this section we discuss some future research directions that have emerged from the work in this dissertation.

Potential of transient servers: In Chapter 3, we showed how to use transient servers in spot market to deploy always-on services. Transient servers also exist in other forms like Google's preemptible VMs. Traditionally, people use these transient servers to run batch jobs. However, our study shows that with replication, migration and backup mechanisms, transient servers can be used for applications in a larger range. In this thesis, we only discuss how to use VM migration and backup to deploy web services on transient servers. There are several interesting extensions to our work: how to deploy other applications on transient servers, how these applications are impacted by server revocations and how to use other mechanisms such as application level backup and replication to make these applications cheaper to deploy and more fault tolerant in use.

Advanced scheduling in big data platforms: In Chapter 4 and Chapter 5, we used network-aware task scheduling to improve big data platforms' performance. Besides network-aware task scheduling, we can also use cluster structure and network information to further improve these platforms performance. For example, currently we only focus on task scheduling of a single job. It could be interesting to find how to optimally schedule tasks

of different jobs with different locality properties. Furthermore, if the jobs have different priority, the problem becomes more complicated and new policies need to be designed. We can also make the data storage scheduling network-aware and proactively transfer data between nodes according to current conditions and future job schedule. If these schedulers could collaborate with each other, we can maximize the resource utilization and optimize the performance of big data platforms.

Design and Implementation of a NFaaS Cloud: In Chapter 6, we discussed the placement strategies for provisioning network functions as service in the cloud. To provide a real NFaaS cloud, there is still a lot of work to do. We need to consider what virtualization techniques should be used and what scheduling algorithms can be applied. The aiming of NFaaS clouds is to provide stable and secure network functions in an on-demand manner with high flexibility and performance.

BIBLIOGRAPHY

[1] IBM's Softlayer http://www.softlayer.com/.

[2] Amazon EC2 http://aws.amazon.com/.

[3] Microsoft Azure https://azure.microsoft.com/.

[4] Google computing platform https://cloud.google.com/.

[5] Amazon EC2 Spot Instanceshttp://aws.amazon.com/ec2/purchasing-options/spot-instances/.

[6] Preemptible Virtual Machines https://cloud.google.com/preemptible-vms/.

[7] Google App Engine https://cloud.google.com/appengine/.

[8] IBM Bluemix http://www.ibm.com/cloud-computing/bluemix/.

[9] Altiscale Big Data Platform urlhttps://www.altiscale.com/big-data-platform.

[10] Amazon EWR https://aws.amazon.com/elasticmapreduce/.

[11] http://aws.amazon.com/solutions/case-studies/netflix/.

[12] https://sourceforge.net/projects/httperf/.

[13] http://dag.wiee.rs/home-made/dstat/.

[14] Intel Data Plane Development Kit. http://dpdk.org.

[15] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *Proc. of NSDI* (2010).

[16] Al-Fares, Mohammad, Loukissas, Alexander, and Vahdat, Amin. A scalable, commodity data center network architecture. In *Proc. of SIGCOMM* (2008).

[17] Ananthanarayanan, Ganesh, Kandula, Srikanth, Greenberg, Albert, Stoica, Ion, Lu, Yi, Saha, Bikas, and Harris, Edward. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of OSDI* (2010).

[18] Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauer, Rolf, Pratt, Ian, and Warfield, Andrew. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[19] Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauer, Rolf, Pratt, Ian, and Warfield, Andrew. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177.

[20] Biran, Ofer, Corradi, Antonio, Fanelli, Mario, Foschini, Luca, Nus, Alexander, Raz, Danny, and Silvera, Ezra. A stable network-aware vm placement for cloud systems. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (2012), IEEE Computer Society, pp. 498–506.

[21] Borkar, Vinayak, Carey, Michael, Grover, Raman, Onose, Nicola, and Vernica, Rares. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of ICDE* (2011).

[22] Breitgand, David, and Epstein, Amir. Sla-aware placement of multi-virtual machine elastic services in compute clouds. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops* (2011), IEEE, pp. 161–168.

[23] Bremler-Barr, Anat, Harchol, Yotam, Hay, David, and Koral, Yaron. Deep packet inspection as a service. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2014), CoNEXT '14, ACM, pp. 271–282.

[24] Bu, Yingyi, Howe, Bill, Balazinska, Magdalena, and Ernst, Michael D. Haloop: Efficient iterative data processing on large clusters. In *Proc. of VLDB* (2010).

[25] Calcavecchia, Nicolò Maria, Biran, Ofer, Hadad, Erez, and Moatti, Yosef. Vm placement strategies for cloud scenarios. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (2012), IEEE, pp. 852–859.

[26] Chohan, Navraj, Castillo, Claris, Spreitzer, Mike, Steinder, Malgorzata, Tantawi, Asser, and Krintz, Chandra. See spot run: using spot instances for mapreduce workflows. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), USENIX Association, pp. 7–7.

[27] Chowdhury, Mosharaf, Kandula, Srikanth, and Stoica, Ion. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 231–242.

[28] Chowdhury, Mosharaf, and Stoica, Ion. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks* (2012), ACM, pp. 31–36.

[29] Chowdhury, Mosharaf, Zhong, Yuan, and Stoica, Ion. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 443–454.

[30] Clark, C., Fraser, K., Hand, S., Hansen, J., Jul, E., Limpach, C., Pratt, I., and Warfiel, A. Live migration of virtual machines. In *Proceedings of Usenix NSDI Symp.* (May 2005).

[31] Clark, Christopher, Fraser, Keir, Hand, Steven, Hansen, Jacob Gorm, Jul, Eric, Limpach, Christian, Pratt, Ian, and Warfield, Andrew. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 273–286.

[32] Cully, Brendan, Lefebvre, Geoffrey, Meyer, Dutch, Feeley, Mike, Hutchinson, Norm, and Warfield, Andrew. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th NSDI Symp.* (2008), San Francisco, pp. 161–174.

[33] Dean, Jeffrey, and Ghemawat, Sanjay. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI* (2004).

[34] Eltabakh, Mohamed Y., Tian, Yuanyuan, Özcan, Fatma, Gemulla, Rainer, Krettek, Aljoscha, and McPherson, John. Cohadoop: Flexible data placement and its exploitation in hadoop. In *Proc. of VLDB* (2011).

[35] ETSI. Network functions virtualization: Architectural framework. `http://www.etsi.org/deliver/etsi\_gs/nfv/001_099/002/01.01.01\_60/gs\_nfv002v010101p.pdf`, 2013.

[36] Ferguson, Andrew D., Guha, Arjun, Liang, Chen, Fonseca, Rodrigo, and Krishnamurthi, Shriram. Participatory networking: An api for application control of sdns. In *Proc. of SIGCOMM* (2013).

[37] Garey, Michael R, Graham, Ronald L, and Ullman, Jeffrey D. Worst-case analysis of memory allocation algorithms. In *Proceedings of the fourth annual ACM symposium on Theory of computing* (1972), ACM, pp. 143–150.

[38] Gember, Aaron, Krishnamurthy, Anand, John, Saul St., Grandl, Robert, Gao, Xiaoyang, Anand, Ashok, Benson, Theophilus, Akella, Aditya, and Sekar, Vyas. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR abs/1305.0209* (2013).

[39] Gember-Jacobson, Aaron, Viswanathan, Raajay, Prakash, Chaithan, Grandl, Robert, Khalid, Junaid, Das, Sourav, and Akella, Aditya. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 163–174.

[40] Gupta, Abhishek, Kale, Laxmikant V, Milojicic, Dejan, Faraboschi, Paolo, and Balle, Susanne M. Hpc-aware vm placement in infrastructure clouds. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on* (2013), IEEE, pp. 11–20.

[41] He, Bingsheng, Yang, Mao, Guo, Zhenyu, Chen, Rishan, Su, Bing, Lin, Wei, and Zhou, Lidong. Comet: Batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 63–74.

[42] He, Xin, Shenoy, Prashant, Sitaraman, Ramesh, and Irwin, David. Cutting the cost of hosting online services using cloud spot markets. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, ACM, pp. 207–218.

[43] He, Xin Shenoy Prashant. Firebird: Network-aware task scheduling for spark using sdns. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)* (2016), IEEE.

[44] Hines, Michael, and Gopalan, Kartik. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of ACM VEE Conference* (March 2009).

[45] Hopps, C. Analysis of an equal-cost multi-path algorithm. *RFC 2992, IETF* (2000).

[46] Hwang, Jinho, Ramakrishnan, K. K., and Wood, Timothy. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 445–458.

[47] Jayasinghe, Deepal, Pu, Calton, Eilam, Tamar, Steinder, Malgorzata, Whally, Ian, and Snible, Ed. Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement. In *Services Computing (SCC), 2011 IEEE International Conference on* (2011), IEEE, pp. 72–79.

[48] Jiang, Dawei, Ooi, Beng Chin, Shi, Lei, and Wu, Sai. The performance of mapreduce: An in-depth study. In *Proc. of VLDB* (2010).

[49] Jiang, Joe Wenjie, Lan, Tian, Ha, Sangtae, Chen, Minghua, and Chiang, Mung. Joint vm placement and routing for data center traffic engineering. In *INFOCOM, 2012 Proceedings IEEE* (2012), IEEE, pp. 2876–2880.

[50] Kivity, Avi, Kamay, Yaniv, Laor, Dor, Lublin, Uri, and Liguori, Anthony. Kvm: the linux virtual machine monitor, 2007.

[51] Kivity, Avi, Kamay, Yaniv, Laor, Dor, Lublin, Uri, and Liguori, Anthony. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, pp. 225–230.

[52] Lagar-Cavilla, A., et al. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of ACM EuroSys* (2009), pp. 1–12.

[53] Liu, Huan. Cutting mapreduce cost with spot market. In *3rd USENIX Workshop on Hot Topics in Cloud Computing* (2011), pp. 1–5.

[54] Marathe, Aniruddha, Harris, Rachel, Lowenthal, David, de Supinski, Bronis R., Rountree, Barry, and Schulz, Martin. Exploiting redundancy for cost-effective, time-constrained execution of hpc applications on amazon ec2. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (New York, NY, USA, 2014), HPDC '14, ACM, pp. 279–290.

[55] Martins, Joao, Ahmed, Mohamed, Raiciu, Costin, Olteanu, Vladimir, Honda, Michio, Bifulco, Roberto, and Huici, Felipe. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 459–473.

[56] McKeown, Nick, Anderson, Tom, Balakrishnan, Hari, Parulkar, Guru, Peterson, Larry, Rexford, Jennifer, Shenker, Scott, and Turner, Jonathan. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.* (2008).

[57] Mills, Kevin, Filliben, James, and Dabrowski, Christopher. Comparing vm-placement algorithms for on-demand clouds. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (2011), IEEE, pp. 91–98.

[58] Nan, Xiaoming, He, Yifeng, and Guan, Ling. Optimal resource allocation for multimedia cloud based on queuing model. In *Multimedia Signal Processing (MMSP), 2011 IEEE 13th International Workshop on* (2011), IEEE, pp. 1–6.

[59] of Electrical Engineering. Computer Science Laboratory, Princeton University. Department, and Ullman, JD. *The performance of a memory allocation algorithm*. 1971.

[60] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks.

[61] Ostermann, Simon, Prodan, Radu, and Fahringer, Thomas. Extending grids with cloud resource management for scientific computing. In *Grid Computing, 2009 10th IEEE/ACM International Conference on* (2009), IEEE, pp. 42–49.

[62] Ousterhout, Kay, Rasti, Ryan, Ratnasamy, Sylvia, Shenker, Scott, and Chun, Byung-Gon. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), pp. 293–307.

[63] Palkar, Shoumik, Lan, Chang, Han, Sangjin, Jang, Keon, Panda, Aurojit, Ratnasamy, Sylvia, Rizzo, Luigi, and Shenker, Scott. E2: A framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 121–136.

[64] Rajagopalan, Shriram, Williams, Dan, Jamjoom, Hani, and Warfield, Andrew. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 227–240.

[65] Rizzo, Luigi. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 9–9.

[66] Rizzo, Luigi, and Lettieri, Giuseppe. VALE, a switched ethernet for virtual machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2012), CoNEXT '12, ACM, pp. 61–72.

[67] Rodero, Ivan, Jaramillo, Juan, Quiroz, Andres, Parashar, Manish, Guim, Francesc, and Poole, Stephen. Energy-efficient application-aware online provisioning for virtualized clouds and data centers. In *Green Computing Conference, 2010 International* (2010), IEEE, pp. 31–45.

[68] Salfner, Felix, Tröger, Peter, and Polze, Andreas. Downtime analysis of virtual machine live migration. In *DEPEND 2011, The Fourth International Conference on Dependability* (2011), pp. 100–105.

[69] Sekar, Vyas, Egi, Norbert, Ratnasamy, Sylvia, Reiter, Michael K., and Shi, Guangyu. Design and implementation of a consolidated middlebox architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 323–336.

[70] Sharma, Prateek, Lee, Stephen, Guo, Tian, Irwin, David, and Shenoy, Prashant. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems* (2015).

[71] Sherry, Justine, Hasan, Shaddi, Scott, Colin, Krishnamurthy, Arvind, Ratnasamy, Sylvia, and Sekar, Vyas. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 13–24.

[72] Shi, Xuelin, Xu, Ke, Liu, JiangChuan, and Wang, Yong. Continuous double auction mechanism and bidding strategies in cloud computing markets. *arXiv preprint arXiv:1307.6066* (2013).

[73] Shieh, Alan, Kandula, Srikanth, Greenberg, Albert, Kim, Changhoon, and Saha, Bikas. Sharing the data center network. In *Proc. of NSDI* (2011).

[74] Singh, Rahul, Irwin, David E, Shenoy, Prashant J, and Ramakrishnan, Kadangode K. Yank: Enabling green data centers to pull the plug. In *NSDI* (2013), pp. 143–155.

[75] Song, Yang, Zafer, Murtaza, and Lee, Kang-Won. Optimal bidding in spot instance market. In *INFOCOM, 2012 Proceedings IEEE* (2012), IEEE, pp. 190–198.

[76] Sugerman, Jeremy, Venkitachalam, Ganesh, and Lim, Beng-Hong. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track* (2001), pp. 1–14.

[77] Trivedi, Animesh, Stuedi, Patrick, Pfefferle, Jonas, Stoica, Radu, Metzler, Bernard, Koltsidas, Ioannis, and Ioannou, Nikolas. On the [ir] relevance of network performance for data processing. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[78] Velte, Anthony, and Velte, Toby. *Microsoft Virtualization with Hyper-V*, 1 ed. McGraw-Hill, Inc., New York, NY, USA, 2010.

[79] Wang, Guohui, Ng, T.S. Eugene, and Shaikh, Anees. Programming your network at run-time for big data applications. In *Proc. of HotSDN* (2012).

[80] Williams, Dan, Jamjoom, Hani, and Weatherspoon, Hakim. The xen-blanket: virtualize once, run everywhere. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 113–126.

[81] Wood, T., Ramakrishnan, K.K., Shenoy, P., and der Merwe, J. Van. Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines. In *Proc. of ACM VEE* (March 2011).

[82] Xiong, Pengcheng, Hacigumus, Hakan, and Naughton, Jeffrey F. A software-defined networking based approach for performance management of analytical queries on distributed data stores. In *Proc. of SIGMOD* (2014).

[83] Xiong, Pengcheng, He, Xin, Hacigumus, Hakan, and Prashant, Shenoy. Cormorant: Running analytic queries on mapreduce with collaborative software-defined networking. In *Proc. of HotWeb* (2015).

[84] Yap, Kok-Kiong, Huang, Te-Yuan, Dodson, Ben, Lam, Monica S., and McKeown, Nick. Towards software-friendly networks. In *Proc. of APSys* (2010).

[85] Yi, Sangho, Kondo, Derrick, and Andrzejak, Artur. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* (2010), IEEE, pp. 236–243.

[86] Zafer, Murtaza, Song, Yang, and Lee, Kang-Won. Optimal bids for spot vms in a cloud for deadline constrained jobs. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (2012), IEEE, pp. 75–82.

[87] Zaharia, Matei, Borthakur, Dhruba, Sen Sarma, Joydeep, Elmeleegy, Khaled, Shenker, Scott, and Stoica, Ion. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems* (2010), ACM, pp. 265–278.

[88] Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J., Shenker, Scott, and Stoica, Ion. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[89] Zhang, Irene, Garthwaite, Alex, Baskakov, Yury, and Barr, Kenneth C. Fast restore of checkpointed memory using working set estimation. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 87–98.