Exploiting Heterogeneous Resources in a Multi-cloud Environment

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Kwangsung Oh

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor Of Philosophy

Prof. Abhishek Chandra, Co-Advisor
Prof. Jon B. Weissman, Advisor

July, 2019

Acknowledgements

Many individuals helped me over past several years for me to finish this thesis. I would like to thank Ajaykrishna Raghavan for helping me to start my first research project, and Francis Liu and Albert Jonathan for discussing research together. Especially, I would like to thank Zachary Leidall for reviewing papers, this thesis, and many other writing materials which helped me a lot in revising them.

I owe a tremendous debt to my two great advisors, Prof. Abhishek Chandra and Prof. Jon Weissman. I am particularly grateful for my advisors' endless patience and unshakable calm. It was great luck for me to have the opportunity to work with them. I may not have been able to publish even a single paper without them. They always guided me in the right directions for my research without losing track of the big picture. I also appreciate Prof. Anand Tripathi for his consideration of my career.

On a personal note, I cannot thank my parents Yongwoon Oh and Jongsun Jung enough for their support in graduate school. Without their support, I could not even have started my PhD program. Finally, I would like to thank my wife Sunhye Park and my daughter Daah Oh, for always being there for me. Without them I could not have finished my thesis.

Dedication

Dedicated to my loves Sunhye, Daah, and our parents.

Abstract

Today, we increasingly rely upon Internet services and applications to automate many daily activities. As of 2019, for example, 118 million people enjoy watching Netflix in their free time, 150 million people find places to stay while they are traveling through AirBnB, and 75 million people rely on Uber to find a car to move around. For data locality, low latency, and availability, many applications utilize diverse cloud resources e.g., storage, network, and compute, in multiple geo-distributed data centers (DCs) of public cloud providers such as Amazon, Microsoft, and Google. In addition, applications can exploit greater cloud resource options if they consider using these cloud providers' DCs together.

Most cloud providers offer heterogeneous cloud resources e.g., memory, SSD, disk, and archival storage for cloud storage, that allow applications to choose a resource based on requirements and demands. Exploiting such heterogeneous resources, however, brings significant complexities to applications because each cloud resource option has different interfaces, data models, pricing policies, and geographical locations. The heterogeneities of cloud resources allow applications to trade off among different metrics, e.g., latency, availability, monetary cost and so on.

To maximize the benefits of heterogeneous cloud resources, applications must answer the question: "what is the best cloud resource configuration (which data centers and which cloud resources) to use to achieve our goals with minimized monetary cost?". Answering this question, however, is challenging because answers are different for each application based on their goals, e.g., SLA (performance), cost budget, consistency model, degree of fault tolerance and so on. Adding to the challenges, dynamics from a multi-cloud environment e.g., network outages and bandwidth/latency fluctuation, and from applications e.g., users' locations, demand, and data access patterns, make it near impossible to determine the best cloud resource configuration statically.

This thesis presents answers to these questions—*how* to exploit heterogeneous cloud resources easily, *how* to determine optimal cloud resource configurations, and *how* to handle dynamics—thereby addressing the challenges in a multi-cloud environment by building three novel and usable systems: a policy-driven geo-distributed cloud storage

system called Wiera, an automated multi-tiered geo-distributed data placement system called TripS, and a network cost-aware geo-distributed data analytics system called Kimchi.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today, we increasingly rely upon Internet services and applications to automate many daily activities. As of 2019, for example, 118 million people enjoy watching Netflix in their free time, 150 million people find places to stay while they are traveling through AirBnB, and 75 million people rely on Uber to find a car to move around. Many of these applications are interactive and their users are distributed geographically around the globe. Providing reduced user-perceived latency and higher service availability independent of user location is critical for these applications as failure to do so affects the revenue of service providers significantly.

To satisfy users, many applications utilize diverse cloud resources e.g., storage, network, and compute, in geo-distributed data centers (DCs) of public cloud providers such as Amazon, Microsoft, and Google, Amazon has DCs in 21 regions[1] [1], Microsoft has DCs in 54 regions [2], and Google has DCs in 20 regions [3] as of June 2019. This mode of deployment allows applications to not only reduce user-perceived latency by putting data and compute resource close to users but also to provide higher data availability and better fault tolerance by having redundant data on multiple DCs.

In addition to DC location options, most cloud providers offer heterogeneous cloud resource options with different characteristics and pricing policies that allow applications to choose which one based on needs. For storage services,[2] as an example, Amazon provides ElastiCache (a caching service protocol compliant with Memcached),

[1] We use the term region to represent a specific location e.g., US West, US East, and Europe West.
[2] In this thesis, we use the term storage tier and storage service interchangeably.

S3 (Simple Storage Service), EBS (Elastic Block Store), and Glacier (data archival service), which vary in their I/O latency, availability, durability, and cost. For networks between DCs (WAN), as another example, Amazon provides heterogeneous bandwidth capacities, latencies, and data transfer costs based on DC locations. In addition, Amazon provides a dedicated network connection service (Direct Connect) that increases bandwidth throughput with additional cost. Thus, it is common to see applications seeking to obtain composite benefits from heterogeneous cloud resources, e.g., for multiple storage tiers within a DC, putting hot data in memory using ElastiCache for better performance and cold data in S3 for higher durability and reduced cost but with much worse performance. That is, these heterogeneous cloud resources across multiple DCs provide applications with a further possibility of selecting one or more cloud resources that meet their goals, e.g., SLA (performance), cost budget, consistency model, degree of fault tolerance and so on.

In fact, applications can exploit even denser DC locations and greater cloud resource options if they consider using different cloud providers' DCs, i.e., applications can use multi-cloud resources of Amazon, Microsoft, Google, and so on together. Exploiting such diversity both within and *across* cloud providers can yield a greater cost-performance tradeoff space and therefore greater benefits [4]. Thus, it seems clear that applications will try to exploit diverse and heterogeneous cloud resources in a multi-cloud environment more and more. Many recent articles [5, 6, 7] show that this will be "true" soon, e.g., Gartner says that "a multi-cloud strategy will become the common strategy for 70% of enterprises by 2019, up from less than 10% today."[5]. We first discuss the opportunities that applications can realize from an evolving multi-cloud environment.

1.1 Opportunities in a Multi-cloud Environment

1.1.1 Increasing Data Locality

The geo-distributed nature of DCs implies that Internet service providers can exploit multiple DCs in different regions to provide reduced perceived latency to their geo-distributed users. For example, latency-sensitive services such as Netflix, Uber, and Airbnb utilize cloud resources in Amazon's DCs instead of having their own DCs to serve their globally distributed users. In a multi-cloud environment, those applications

Figure 1.1: Elapsed time to retrieve 100KB data in Virginia (US East)

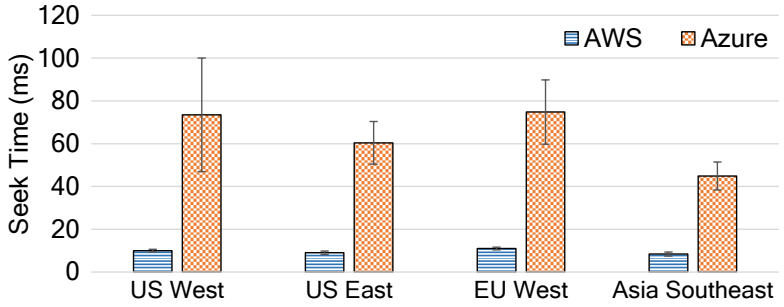


Figure 1.2: Disk seek time of VMs on each region.

can utilize Microsoft's DC in San Antonio (US South Central region) where Amazon does not have a DC, to provide even better data locality to users in that region.

1.1.2 Wider Locality Envelope

Along with the growing geo-distribution of DCs, however, another trend is the increasing density of DCs within a region. According to datacentermap.com [8], there were 171 DCs as of Sep 2014 and 232 as of June 2019 within the US West (California) region alone. This implies that multiple DCs of each cloud provider are co-located in the same region, e.g., both Amazon and Microsoft have DCs in California (US West), Virginia (US East), Ireland (Europe West), Singapore (Asia Southeast), Tokyo (Asia East), Mumbai (Asia South), and Sao Paulo (South America). The higher density of DCs within a region coupled with multiple-tiered storage provides a *wider locality envelope* for applications. The conventional wisdom is that co-location of computation and storage within the same DC is key to application performance, since applications running within a DC are

Table 1.1: Latency (ms) between DCs across different cloud providers

Region		Latency (ms)	
		AWS	Azure
US West (California)	AWS	-	3.84
	Azure	3.62	-
US East (Virginia)	AWS	-	1.97
	Azure	1.99	-
Europe West (Ireland)	AWS	-	17.58
	Azure	18.67	-
Asia Southeast (Singapore)	AWS	-	1.84
	Azure	1.98	-

often still limited to access local data. Given such a high density of DCs, the latency between *nearby* DCs across cloud providers (those within the same region) can be very small. Table 1.1 confirms that this is true: it shows that the latencies between DCs within a region is very small ($<$ 4 ms) for most regions, except the Ireland (Europe West) region. Given a denser-DC environment with multi-tiered storage support and low latency between DCs, applications may be able to find faster storage tiers from nearby DCs compared to local DC's. Figure 1.1 shows the elapsed time for clients running in AWS and Azure on Virginia (US East) to retrieve 100KB of data from different storage tiers on local and non-local DCs. The results show that clients can retrieve data in a nearby DC's memory 2.56 $\sim$ 3.94 times faster than the local DC's disk and 3.8 $\sim$ 5.34 times faster than the local DC's archival storage. This result shows that *non-local data access from faster tiers can be more efficient than local access to slower tiers.* Interestingly, our results also show that clients may be able to *retrieve data faster even from the same storage tier on a nearby DC.* E.g., clients running on Azure can retrieve data in AWS disk storage 1.34 times faster than local Azure disk, while clients running on AWS can retrieve data in Azure archival storage 1.23 times faster than local AWS archival storage. Note that we see a similar pattern of results using other regions and omit them for simplicity. Such performance differences may come from different

implementations of disk (archival) storage services as Figure 1.2 shows the disk storage tier performance (disk seek time) difference between AWS and Azure.

1.1.3 Opportunities

There are several possible opportunities that can be derived from using heterogeneous cloud resources and denser DC locations in a multi-cloud environment.

- **Simpler Consistency Policy:** With wider locality envelopes, we can replicate data to fewer DCs serving more clients across fewer replicas, thus reducing the overhead of maintaining consistency across them.
- **Hot and Cold Data:** If recently accessed ("hot") data is stored in fast memory in a nearby DC based on client access, an application can retrieve data from nearby DC's memory for better performance. At the same time, "cold" data could be moved to a cheaper storage tier in a nearby DC. Though accessing non-local DC's faster storage tiers may incur network cost, it may be desirable for latency-sensitive applications.
- **Higher Availability:** Each DC has independent failure domains so DC-level outages can be tolerated. Note, there might be correlated DC failures due to a natural disaster e.g., hurricane as we consider nearby DCs (those within the same region). However, 90% of DC failures happen independently without such correlation [9], thus our approach is still feasible.
- **Competitive Pricing:** Each cloud provider offers a heterogeneous pricing policy for cloud services so one may find similar performance at a lower cost from a tier in a different cloud provider's DC.
- **Richer Cost-performance Tradeoff Space:** Heterogeneous cloud resources' pricing policies and performance open up a richer cost-performance tradeoff space that applications can explore in a multi-cloud environment.

1.2 Challenges in a Multi-cloud Environment

Exploiting such diverse and heterogeneous cloud resources in a multi-cloud environment and maximizing the benefits from them are quite challenging due to several reasons.

- **Diverse Applications' Goals:** Each application has different goals and emphasizes different factors e.g., consistency model, degree of fault tolerance, expected access pattern, and users' locations, and metrics of interest such as user-perceived latency, cost budget and so on. This will create significant burdens for application developers to specify and implement and program diverse policies in order to achieve desired goals.

- **Complexities of Heterogeneous Cloud Resources:** Exploiting heterogeneous cloud resources within in a single DC introduces significant complexities to the application because each service has different interfaces, performance, and pricing policies. Exploiting heterogeneous cloud resources in multiple DCs across cloud providers brings even more complexity as each cloud provider also has different interfaces and pricing policies for their services.

- **Optimal Configurations of Diverse and Heterogeneous Cloud Resources:** While having more cloud resource options can be beneficial for applications to achieve desired goals by trading off among different metrics, e.g., latency and monetary cost, numerous and heterogeneous cloud resources make it hard for applications to determine the right cloud configuration because there can be countless combinations of cloud resources and DC locations in a multi-cloud environment. In addition, the fact that the number of cloud resources and DCs in growing implies that it will be more complex to find optimal configurations of cloud resources in the future.

- **Dynamics:** Most cloud providers do not guarantee consistent performance over time as cloud resources are shared by multiple users and thus applications may see worse performance than before from the cloud resources. In addition, applications may see changes such as users' locations, access patterns, and data popularity. Any of these dynamics may hurt applications' desired goals and thus static decisions or policies may not be effective.

1.3 Summary of Research Contributions

This thesis addresses the challenges—complexity to exploit heterogeneous cloud resources that have different characteristics, burdens for application developers to implement and program diverse policies that meet various applications' goals, dynamics from cloud infrastructure and applications, and complexity to determine the optimal solutions for data and task placement—in a multi-cloud environment. The key contributions of this thesis are:

- Allowing applications to exploit diverse and heterogeneous cloud resources easily by providing ways such as a high level language and simple parameters, to specify their desired goals, e.g., reduced cost, latency, consistency model, degree of fault tolerance, minimized monetary cost and so on.

- Modeling data placement and task placement problems as constrained optimization problems and solving them optimally using mixed integer programming (MIP) for Internet applications and geo-distributed data analytics applications.

- Handling long and short-term dynamics from both the network and the application with policy-driven, proactive, and heuristic approaches while achieving applications' desired goals.

- Designing and building novel and usable systems that address challenges in a multi-cloud environment: 1) Wiera: an integrated geo-distributed cloud storage system that runs both within and across data-centers owned by different cloud providers [10], 2) TripS: the first system that optimizes data placement with a consideration of both DC locations and storage tiers [11], and 3) Kimchi: the first geo-distributed data analytics (GDA) system that considers heterogeneous data transfer cost.

- Evaluating our systems empirically in the Amazon AWS and Microsoft Azure clouds, showing that 1) the use of multi-cloud resources can result in improved performance, reduced cost, and desired consistency at lower overhead, 2) automated data and task placement in a multi-cloud environment can help an application achieve desired goals with minimized cost even in the presence of dynamics, and 3) heterogeneous data transfer cost opens a greater cost-performance tradeoff space for GDA applications.

We also explore the challenges and opportunities of GDA systems that determine optimal task placement with a consideration of heterogeneous network and compute resources in a multi-cloud environment as future work.

1.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents a policy-driven geo-distributed storage system that provides an easy way to specify data placement policies exploiting heterogeneous storage services in a multi-cloud environment. Chapter 3 presents an automated multi-tiered data placement system that determines optimized data placement exploiting both DC locations and heterogeneous storage services at each DC in a multi-cloud environment. Chapter 4 presents a network cost-aware geo-distributed data analytics system that exploits heterogeneous WAN resources to help data analytics applications to meet cost-performance tradeoff preferences in a multi-cloud environment. Finally, Chapter **??** concludes.

Chapter 2

Wiera: Policy-driven Geo-distributed Storage System

2.1 Introduction

The use of multiple geo-distributed data centers (DCs) is commonly used to provide Internet services and applications to users that are distributed geographically. This mode of deployment not only reduces user-perceived latency by putting data close to users but also provides higher data availability and better fault tolerance by replicating data to multiple locations. Although this idea is simple, it introduces many complexities for the owner of the application and/or the data: 1) the number and location of replicas as a function of the desired consistency model, 2) degree of fault tolerance, 3) expected access pattern, and 4) metrics of interest such as user-perceived latency, cost, and so on. This is further complicated by the dynamics of the network environment, Internet services, and applications. Thus, static decisions or policies may not be effective. For example, application access pattern may vary over time with the storage system seeing a write-intensive pattern at first as new data is created and stored, followed by a read-intensive pattern as that data is retrieved. This pattern is common in many applications running on data analytics frameworks [12, 13]. Similarly, the location of active users or the demand for data may change over time based on changing popularity, trends and user interests, especially for Internet services.

While some geo-distributed storage systems [14, 15, 16] have been proposed, they

typically re-evaluate storage policies on very coarse time-scales such as hours-to-weeks and make assumptions that may not always be true (e.g., SPANStore [15] assumes users are static). This results in policies that may be inadequate in a wide-area multiple-tier environment that spans different storage providers and in which time-scales of change may be much shorter. Examples would include bursty demand due to flash crowds, temporary network outages, and changes in application access pattern type (reads vs. writes), all of which may occur at short time scales (seconds to minutes). Additionally, these systems generally do not exploit the wide diversity of storage characteristics available at different tiers of the cloud storage hierarchy both within and across data-centers and different providers. Different cloud providers offer multiple cloud storage services with different characteristics such as durability, performance, and cost across their constituent data centers (DCs). Thus, exploiting such diversity both within and *across* cloud storage providers can yield greater storage options and therefore greater benefits as explained in Section 1.1.

To address these challenges and opportunities, we present a new geo-distributed cloud storage system called Wiera (or Wide-area tIERA) that builds upon Tiera cloud storage system [17]. Tiera provides storage instances that span the storage hierarchy within a single data-center for a single cloud provider. Wiera exploits multiple storage tiers on the wide-area across different data-centers, across different cloud providers, and enables policies that can respond to dynamism at short time scales (seconds to minutes) with Tiera. The client is shielded from the underlying complexity introduced by multiple storage tiers across multiple DCs by a simple PUT/GET API and the encapsulation of storage policies. Wiera supports *global policies* by leveraging the local policy framework within each Tiera instance. A Wiera storage instance logically contains many Tiera instances distributed across the wide-area. We present the design and implementation of the Wiera system,[1] show how a rich array of policies can be easily expressed in Wiera, and evaluate its performance on a live multi-cloud system to show its potential.

The key contributions of this work are:

- The design and implementation of the Wiera system, an integrated geo-distributed cloud storage system that runs both within and across data-centers owned by different cloud providers.

[1] https://github.com/dcsg-umn/wiera

- Mechanisms for easily specifying a rich array of global storage policies across a geo-distributed multi-tiered cloud storage environment including several common policies from the literature.
- First-class support for handling network and application dynamics within the storage policies to achieve user metrics (e.g., reduced cost, latency, and so on).
- Flexibility that allows *unmodified applications* to further reap benefits by replacing data/storage policies externalized at run-time.
- An empirical evaluation of the Wiera prototype in the Amazon AWS and Microsoft Azure clouds, showing that the use of non-local data-center storage tiers can result in improved performance, reduced cost, and desired consistency at lower overhead.

2.2 System Model and Goals

In this section, we begin by describing our system model. We then present opportunities in using multi-tiered storage in a multi-cloud environment, along with the key goals of Wiera.

2.2.1 System Model

We consider a geo-distributed cloud environment consisting of multiple data centers (DCs) located across different geographic regions, each supporting multiple storage tiers. These DCs could belong to the same or different cloud providers. We next discuss the key aspects of this system model.

Geo-Distributed Data Centers

There are many cloud service providers publicly available and most of them have multiple data centers (DCs) geo-distributed around the world. For instance, as of Sep 2017, Amazon has DCs in 16 regions (and numerous Edge locations) [1] and Microsoft has DCs in 26 regions (and has announced plans for 8 additional regions) [2].

Density of Data Centers within a Region

Along with the growing geo-distribution of DCs, however, another trend is the increasing density of DCs within a region, e.g., 223 DCs in the California (US West region) alone as of Sep 2017. This implies that multiple DCs of each cloud provider are co-located in the same region. If we consider additional cloud providers such as Google and Rackspace, even more DCs are available within the same region. Given such a high density of DCs, the network latency between *nearby* DCs (those within the same region) can be very small as shown in Table 1.1.

Multiple Storage Tiers within a Data Center

Most cloud service providers offer multiple storage tiers with different characteristics, performance, durability, and cost for storage. For instance, Amazon provides ElastiCache (a caching service protocol compliant with Memcached), EBS (Elastic Block Store), S3 (Simple Storage Service), and Glacier (Data archival service) as different cloud storage options. Other cloud providers also offer similar storage services. These storage services generally optimize one metric trading off others. For instance, an application can get better performance from ElastiCache but at a higher cost and lower durability, compared to using S3. Thus, applications may need to use multiple storage tiers for their composite benefits to achieve their desired goals.

Dynamics

Dynamics are common in a multi-cloud environment. They commonly occur due to cloud infrastructures (network) e.g., network and data center failures and variations in network and storage performance and/or workload changes, e.g., access patterns, users' locations and data popularity. Any dynamics can hurt application's goals and thus it must be considered and handled for applications to achieve goals.

2.2.2 Goals of Wiera

Accessing multiple storage tiers within a DC introduces significant complexities to the application because different tiers have different interfaces, different data models and pricing policies. In addition, it can be burdensome to specify and program policies to

manage data across the different storage tiers to realize the desired metric(s). Using multiple DCs across cloud providers brings even more complexity as each cloud provider has different pricing policies and interfaces for their storage services. To maximize the benefits from multiple DCs, several challenges need to be overcome. By developing Wiera, we intend to enable applications to achieve their desired goals in a geo-distributed cloud environment with minimal effort.

The key goals of Wiera are:

- **Supporting flexible storage policies**: Each application has different goals and emphasize different factors. This will yield diverse policies that are complex to program. Thus, it is important to help application developers manage diverse data placement policies with minimal effort by providing an easy way to specify and apply policies.
- **Exploiting multiple storage tiers across DCs**: Each storage tier within a single DC has a different interface and a price policy based on its characteristics. If applications wish to use multiple storage tiers on across DCs and cloud providers, they can expect even more diverse storage interfaces, pricing policies, and performance variation. Abstractions should hide such complexities and yet be expressive enough to realize the benefits as shown in 1.1 and to capture user requirements.
- **Handling dynamics at run-time**: Many cloud storage services do not guarantee consistent performance over time. And the users' locations and their access patterns may change. Any of these dynamics can lead to a violation of applications' desired goals. These dynamics should be handled quickly to enable an application to achieve their desired goals in the presence of dynamics.

We show how Wiera helps applications achieve their desired goals in Section 2.5.

2.3 Wiera Overview

In order to handle data across multiple regions, we have designed Wiera to support several key requirements: data replication and consistency across multiple locations, load balancing, locality-awareness, and fault tolerance. In addition, Wiera has been implemented with scalability in mind from the start as it needs to exploit multiple

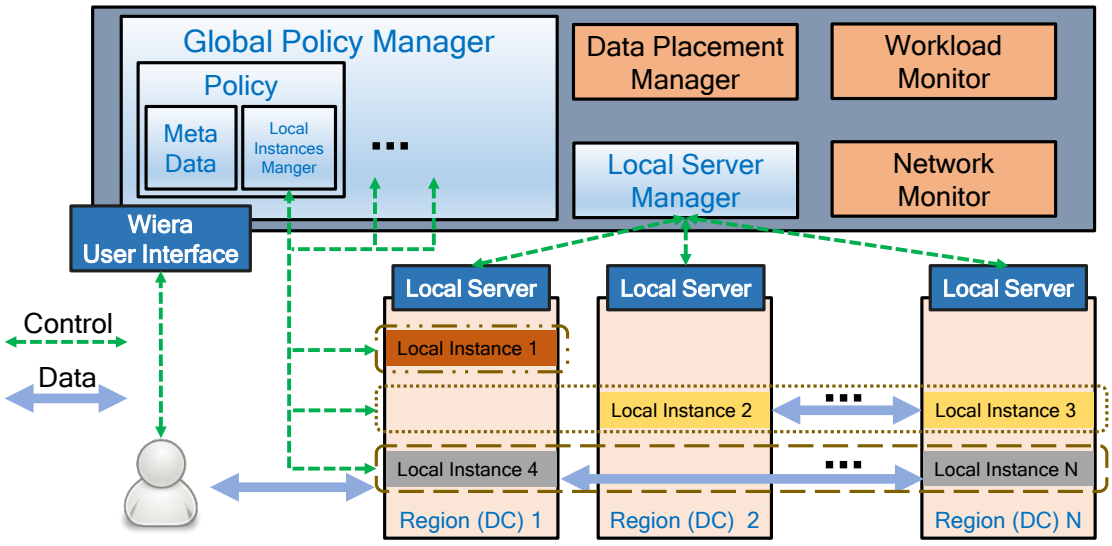


Figure 2.1: Wiera Architecture.

geo-distributed data centers and multiple storage tiers. In this section, we present an overview of Wiera, describing the Wiera architecture and data model, and mechanisms for defining global policies for managing data across multiple DCs.

2.3.1 Wiera Architecture

Figure 2.1 shows the Wiera architecture. Wiera builds on top of Tiera in a geo-distributed setting: Wiera employs Tiera instances as *local instances*[2] to exploit multiple storage tiers of each DC. A Wiera instance logically consists of multiple local instances running in multiple data centers.

Local Instance

As mentioned in Section 2.2.1, it is common to see applications seeking to obtain composite benefits from multiple cloud storage tiers to achieve their desired goals, e.g., putting hot data in memory for performance and cold data in object storage tier for

[2] Throughout this thesis, we use *local instance* to refer to a Tiera instance to distinguish clearly between *Tiera* and *Wiera*.

```
Wiera LowLatencyInstance = {
  tiers: [{id: memory, size:1G, t_type: ElasticCache}, {id: disk, size: 10G, t_type: EBS}],
  events: [{e_type: ActionPut,
    responses: [{r_type: Store, params: {to: memory}}]},
          {e_type: Timer , condition: {period: 30 secs},
    responses: [{r_type: Copy, params: {from: memory, condition: {dirty: true}, to: disk}}]}]}
```

Figure 2.2: LowLatency local instance.

reduced cost. However, accessing multiple storage tiers introduces significant complexities to the application because different tiers have different interfaces and different data models. At the same time, it creates a burden to specify and program policies to manage data across the different storage tiers to realize the desired metric(s). For example, popular open-source Internet applications e.g., WordPress [18] and Moodle [19], have thousands of lines of code to exploit multiple storage tiers e.g., ElasticCache and EBS (or S3), within a DC to provide low latency to users in a single region. To address these problems, a local instance encapsulates multiple cloud storage tiers and enables easy specification of a rich array of data storage policies to achieve desired tradeoffs. An *event-response* mechanism is used to express policies and manage data both within and across local instances. An event is the occurrence of some condition and a response is the action executed on the occurrence of an event. For local policies, a local instance supports different kinds of events such as *timer*, *threshold*, and *action* events (Get and Put) and responses such as *store*, *retrieve*, *copy*, *move*, *encrypt*, *compress*, *delete*, and *grow* to react to the events. For global policies, a local instance newly supports events *LatencyMonitoring*, *RequestsMonitoring*, and *ColdDataMonitoring* and responses *forward*, *queue*, and *change consistency* to support policies for handling dynamics in a multi-cloud environment, e.g., access pattern changes as we will explain more in detail in Section 2.3.3.

Figure 2.2 shows the local policy example that uses two storage tiers within a single DC, ElasticCache for memory performance and EBS (S3) for data persistence. For better performance, the instance will *put* data into memory first and then copy data back into EBS for persistence (*write-back* policy) responsive to a timer event (30 seconds). We will show how policies can be implemented and run on a local instance more in detail in Section 2.4.

Wiera Instance

While a *local instance* is responsible for managing data in multiple storage tiers *within* a single DC, a *Wiera instance* manages the data placement and movement *across* multiple local instances running on geo-distributed DCs using local instances event-response mechanism. A Wiera instance simplifies the global data access for applications by hiding the complexities of accessing multiple local instances. Wiera supports global policies by leveraging the local policy framework within each local instance. Applications can launch and manage local instances in multiple regions, and can enforce a global data management policy between them through Wiera, as we will explain more fully in Section 2.3.4.

Table 2.1: Wiera Instance Management API

API	Arguments	Function
startInstances	wiera_instance_id, policy	Launch instances
stopInstances	wiera_instance_id	Stop instances
getInstances	wiera_instance_id	Get instances list

Wiera Components

Wiera consists of the following main components:

- The Wiera User Interface (WUI) provides an API to applications to manage Wiera instances (Table 2.1). The API allows applications to: launch multiple local instances as part of a Wiera instance with a global policy specification, stop instances, and get the list of currently running instances.

- Local Server Manager (LSM) manages local servers at different locations, which spawn and remove local instances based on application requests. For instance, if the application calls *startInstances* through WUI to start local instances at Region 1 and Region 2, LSM will direct the local servers in Region 1 and Region 2 to each spawn a new local instance.

- Global Policy Manager (GPM) creates a new policy for a Wiera instance. It stores

metadata for the policy and executes a local Instance Manager (LIM) to manage the local instances which belong to the Wiera instance.

- Local Instance Manager (LIM) is executed for each policy and manages local instances spawned by LSM.

We will explain how these components work together in more detail in Section 2.4. Wiera also includes other components such as a network monitor, workload monitor, and data placement manager. The network monitor aggregates latency information for handling requests from each instance and latencies between instances. The workload monitor aggregates workload related information such as users' locations (number of requests from each instance), access patterns, and object sizes. Generating a dynamic global policy automatically based on this aggregated information could be done by a data placement manager but is left as future work.

In the current Wiera architecture, Wiera utilizes a centralized coordinator (master) for simplicity. However, failures may happen either due to Wiera component failures or cloud infrastructure failures (DC-level [9] or service-level [20]). For high availability in the face of failure, Wiera can utilize standard techniques and protocols including: fast recovery, checkpointing, replication, distributed consensus, and multiple masters with leader election as in many previous systems [21, 22, 23, 24]. In this chapter, we focus on support for handling different policies and a more fault tolerant implementation is left as future work.

Table 2.2: Wiera Object API

API	Arguments	Function
get	string key	Retrieve the latest version of object
getVer	string key, integer version	Retrieve specific version of object
getVerList	string key	Retrieve list of available version of object
put	string key, binary object	Store object
update	string key, binary object, integer version	Update specific version of object
remove	string key	Remove all version of object
removeVer	string key, integer version	Remove specific version of object

2.3.2 Data Model

Data in Wiera is stored as objects [25]. This model enforces an explicit separation of data and metadata enabling unified access to data distributed among the different storage services and DCs. While each object is immutable (i.e., cannot be modified), it can have multiple versions to support replications and consistency control in weak consistency model e.g., eventual consistency as a Wiera instance can replicate data across multiple local instances in order to support low latency and fault tolerance. Thus, modification of an existing object results in the creation of a *new version* of the object. Each object can be accessed through a *globally* unique identifier that acts as the *key* to access the corresponding *value* stored with Wiera API shown in Table 2.2. To support policy specification, Wiera provides several common attributes or metadata for each object such as: *size*, *access frequency*, *dirty bit*, *modified time*, *location* (i.e., which storage tiers), *version*, and *last access time*.

2.3.3 Events and Responses for Global Policy

In a geo-distributed setting, clients may access data from different regions. The placement and replication of data can have significant impact on the application's latency of access, load across different DCs, and consistency of data. Wiera provides a number of new events and responses to support different policies to manage data across multiple locations. Wiera adds three *monitoring* events: (1) *LatencyMonitoring* events that occur when data access requests take longer than a specified latency threshold (and thus, may violate an application's latency requirement), (2) *RequestsMonitoring* events that occur when a local instance gets more requests than other instances (and thus, may be overloaded), and (3) *ColdDataMonitoring* events that occur when certain data is not accessed more recently than a specified time threshold (and hence, is cold). To react to these newly added events, Wiera also adds new responses: (1) *forward* that forwards a request to another local instance (e.g., for load balancing), (2) *queue* that enqueues a request for lazy update to other locations (e.g., to reduce on update traffic), and (3) *change_consistency* that changes the consistency model between local instances at run-time to handle workload dynamics.

2.3.4 Defining Wiera Policies

Given data model and *events-responses* mechanism discussed above, applications can flexibly specify a number of global data management policies, including many that have been proposed in the literature [14, 15, 16, 26].

In this section, we show how diverse policies can be easily specified and implemented by providing examples written in a concise and expressive *high-level policy language*. Note, while a large set of events and responses is built in Wiera, it is not complete to support all different data placement policies. Applications can create own events and responses based on needs and Wiera allows applications plug them in to Wiera easily [27].

The application needs to specify the desired storage tiers and the regions where instances will be running, storage tiers' capacities, the set of events along with corresponding responses for each instance. Note that all global policies in this section are just examples to show how they can be easily specified. Applications can modify these policies or create a new policy based on their requirements.

Note that instances running at different locations can have different local policy specifications as well. In this chapter, however, we use the same specification everywhere for simplicity, unless noted otherwise. Further, for simplicity, we show the specification of *put* operations in our examples, and *get* operations also can be specified similarly.

Data Consistency Policy

Many recent Internet applications serve users around the world, thus data consistency is one of the most important metrics to be considered. For example, popular database systems, e.g., MySQL [28] and Cassandra [29] have thousands of lines of code for replicating data into multiple DCs using various consistency models, i.e., synchronous (multi-master) or asynchronous (eventual) replication in MySQL and quorum consistency in Cassandra. We begin by showing how a desired data consistency model between local instances can be easily specified in a Wiera Policy.

Figures 2.3(a), 2.3(b), 2.4, and 2.5 show four different consistency policies: *MultiplePrimariesConsistency*, *PrimaryBackupConsistency*, *QuorumConsistency*, and *EventualConsistency* respectively.

```
Wiera MultiPrimariesConsistency = {
  hosts: [{id: host_1, i_type:LowLatencyInstance, region: US-West, primary: true},
          ...
          {id: host_N, i_type:LowLatencyInstance, region: EU-West}],
  events: [{e_type: ActionPut ,
    responses: [{r_type: LockGlobalWrite , params: {key: obj.key}},
                {r_type: Store , params: {to: local_instance.memory}},
                {r_type: Broadcast , params: {to: all_hosts}},
                {r_type: UnLockGlobalWrite , params: {key: obj.key}}]}]}
```

(a) Multiple Primaries consistency policy.

```
Wiera PrimaryBackupConsistency = {
  events: [{e_type: ActionPut , condition: {primary: true},
    responses: [{r_type: Store , params: {to: memory}},
                {r_type: Broadcast , params: {to: all_hosts}}]},
           {e_type: ActionPut , condition: {primary: false},
    responses: [{r_type: ForwardPut , params: {to: primary}}]}]}
```

(b) Primary Backup consistency policy.

Figure 2.3: Primary-based consistency policies.

```
Wiera QuorumConsistency = {
  events: [{e_type: ActionPut,
    responses: [{r_type: LockGlobalWrite , params: {key: obj.key}},
                {r_type: GetLatestVer , params: {from: write_quorum}},
                {r_type: Store , params: {to: local_instance.memory, ver: obj.ver+1}},
                {r_type: Broadcast , params: {to: write_quorum}},
                {r_type: LockGlobalWrite , params: {key: obj.key}}]}]}
```

Figure 2.4: Quorum consistency policy.

In the *MultiPrimariesConsistency* policy (Figure 2.3(a)) specification, multiple locations maintain replicas of the data and every update to any replica is synchronously transmitted to all other replicas. This policy can be used for applications in which strong data consistency is more important than *put* operation performance, e.g., a flight booking system or a banking system. The figure shows how this policy is easily implemented using Wiera events and responses. Here, the same local instance (*LowLatencyInstance* from Figure 2.2) is created on multiple regions. When a local instance receives a *put* request from an application, it tries to get a *global lock* first for the key as specified in the Wiera policy. Once it gets the lock for the key, it stores the object into the local Memcached storage tier first as was explained in Figure 2.2. Then it distributes the update to other instances that are part of the same Wiera instance. The lock is released

```
Wiera EventualConsistency = {
  events: [{e_type: ActionPut,
    responses: [{r_type: Store, params: {to: memory}},
                {r_type: Queue, params: {what: obj}}]}
            {e_type: Timer, condition: {period: 5 secs},
    responses: [{r_type: DeQuque, what: obj},
                {r_type: Broadcast, params: {to: all_hosts}}]}]}
```

Figure 2.5: Eventual consistency policy.

upon getting a response from all other instances.

In the *QuorumConsistency* policy (Figure 2.4), strong consistency can be achieved without having replicas on all local instances by using a voting protocol that uses two constraints on the numbers of nodes needed for each read and write operation. Suppose N nodes are in the system, then we have 1) Nr + Nw > N and 2) Nw > Nr / 2, where Nr and Nw are numbers of read and write quorums respectively, to avoid both read-write and write-write conflicts. In the policy, a local instance that received a write request first gets the global lock and finds the latest version from local instances in a write quorum. Then, it increases the version and broadcasts the update to local instances in the quorum and releases the global lock. This policy requires one more step to get the latest version among instances for version consistency. However, it allows applications to achieve the strong consistency even with less number of replicas, which helps reduce the cost of network and storage.

In the *PrimaryBackupConsistency* policy (Figure 2.3(b)), there is only one primary replica. Here, if a local instance gets a *put* request from an application and the instance is not the *primary*, it will simply forward it to the *primary* instance. This policy is simpler than the *MultiPrimariesConsistency* policy and can provide better performance since no global lock is required, but the *primary* instance can be a bottleneck for overall performance. The application can trade off its desired consistency with performance in this policy. For instance, to minimize *get* latency, the primary can send updates to other instances synchronously by using a *copy* response, so that all replicas are up-to-date. On the other hand, to improve *put* latency, updates could be transmitted asynchronously by the primary using *queue* response.

The *EventualConsistency* policy (Figure 2.5) is desired for better PUT/GET latency, e.g., for social network services like Facebook and Twitter. Here, a *put* operation simply

```
Wiera ChangeConsistency = {
  events: [{e_type: LatencyMonitoring, condition: {latency: > 800, period: 30 secs},
    responses: [{r_type: ChangePolicy, params: {to: EventualConsistency}}]},
          {e_type: LatencyMonitoring, condition: {latency: <= 800, period: 30 secs},
    responses: [{r_type: ChangePolicy, params: {to: MultiPrimariesConsistency}}]}]}
```

(a) Changing consistency policy.

```
Wiera ChangePrimary = {
  events: [{e_type: RequestsMonitoring, condition: {period: 60 secs},
    responses: [{r_type: ChangePrimary, params: {to: instance_forward_most}}]}]}
```

(b) Changing the primary in Primary Backup policy.

Figure 2.6: Defining dynamic policies.

stores the object to the local replica first and then queues the update for distribution to other replicas later in the background. Applications can also specify how frequently queued updates need to be distributed. In this consistency model, there is no specific order of *put* operations from each instance, thus each instance needs to handle object version conflicts when update requests come in from other instances as we will explain in Section 2.4 in more detail.

Defining Dynamic Policies

Some Internet applications that tolerate relaxed (eventual) consistency e.g., social networking, shopping, entertainment, news, and messaging, can benefit from improved (strong) consistency e.g., better user experience and revenue increase, as shown in previous work [30]. While strong consistency is desirable for the benefits, achieving strong consistency can be expensive due to high WAN latency and dynamics e.g., network fluctuation and workload changes, in a multi-cloud environment. For example, in the MultiPrimariesPolicy (Figure 2.3(a)), the latency for a *put* operation will depend on the highest round trip latency from the primary initiating the update to any replica. For these applications that want to benefit from strong consistency, it would be desirable to have a dynamic policy that can change its actions at run-time. One example of such a dynamic policy would be one that can adjust the consistency model based on observed latencies of operations. That is, applications may want to use strong consistency for all data accesses when all operations can be performed with low latency. In the high latency case, eventual consistency can be used for better user-perceived latency.

Figure 2.6(a) shows how Wiera can specify such a dynamic consistency policy. In this figure, an application specifies the latency threshold (800 ms) and the duration (30 seconds) for which this latency threshold is exceeded. Once the *put* operations violates both conditions, Wiera changes the global consistency policy to *eventual consistency* at run-time for better *put* operation latency. Similarly, while using the *eventual consistency* model, once Wiera detects that the latency for *put* operations can satisfy the conditions for the strong consistency, it will switch them back to *strong consistency* policy at run-time. Both strong (i.e., *MultiPrimariesConsistency*) and weak (i.e., *EventualConsistency*) consistency implementations are made available to this policy. The change of consistency policy is done in a manner that allows all operations in progress (or queued) to be applied first. All new requests from applications that arrive when the consistency is being changed will be blocked and queued until the change takes effect.

Consider another case in which handling dynamics is required. Assuming a single primary, if the workload changes over time (e.g., client locations change with time of day), then moving the primary replica closer to the users might be desirable [14]. Figure 2.6(b) shows how this can be achieved with Wiera for the *PrimaryBackupConsistency* policy (Figure 2.3(b)). If the primary instance discovers that another instance received (and forwarded) more requests from an application than the primary, then Wiera will change the primary instance to the more heavily accessed replica. Once this change has been done, all requests will be forwarded to the new primary instance.

Achieving Desired Metrics

Applications can have different desired metrics such as performance, reliability, cost, etc. Wiera policies can be defined to achieve such desired metrics as well. While we focused on the consistency policies above to achieve desired latencies in the presence of replication, another important metric could be cost.

Many Internet applications have a large fraction of data which is accessed infrequently or not at all. For example, Facebook shows its data access patterns typically conform to a Zipfian distribution [31] in which only a small proportion of data is frequently accessed. One way for such an application to lower its cost could be to use cheaper but slower storage (e.g., Amazon S3 or Glacier) for its cold data while using more expensive, faster storage (e.g., MemCached or EBS) for hot data. Figure 2.7(a)

```
Wiera  ReducedCostPolicy = {
  events: [{e_type: ColdDataMonitoring, condition: {from: disk, period: 120 hours},
    responses: [{r_type: Move, params: {to: archival, bandwidth:100KB/s}}]}]}
```

(a) Reducing cost by moving cold data to cheaper storage.

```
Wiera SimplerConsistencyPolicy = {
  events: [{e_type: ActionPut, condition: {region: each_region, primary: true},
    responses: [{r_type: Store, params: {to: memory}},
                ...
                {r_type: Broadcast , params: {to: other_primaries}},
                ...
          {e_type: ActionPut, condition: {primary: false},
    responses: [{r_type: Forward, params: {to: region_primary}}]}]}
```

(b) Simpler consistency by using the fastest storage tier within the same region.

Figure 2.7: Achieving desired cost metrics.

shows how Wiera can allow applications to get benefits from such cheaper and durable storage tiers. In this policy, each instance has one cheaper storage tier. An application defines cold data by setting a threshold on elapsed time from the last access (120 hours). If an instance gets the event which notifies that there is any object has not been accessed for 120 hours, it is identified as cold and moved to the cheaper storage tier.

Another way to reduce cost could be by maintaining fewer replicas. This could reduce both storage costs as well as network bandwidth costs by reducing the update traffic, as cloud providers charge for all out-bound network traffic. As shown in Section 1.1.2, an application can achieve good performance even with fewer replicas by accessing nearby DCs' faster storage tier (e.g., Memcached) instead of a local slower tier (e.g., EBS or S3). Figure 2.7(b) shows how Wiera can enable the reduction in the number of replicas by using the fastest storage tier in the centralized DC in a region. In this policy, all local instances are running within the same region (US West), and are forwarding requests to a primary instance. Thus instances in this region need not be concerned about data consistency which can reduce network traffic and cost. All non-primary instances could then be used as caches (if data is *read-only* or data can be served under *eventual consistency*) or for load balancing if needed. An application can reduce cost further by maintaining a single replica for cold data on centralized cheaper storage tier. That is, if the application allows instances to share the centralized cheaper storage tier for cold data, it can save even greater storage cost. We will explain how this can be

```
Wiera EventualConsistency = {
  events: [{e_type: ActionPut,
    responses: [{r_type: Store, params: {to: cheapest, lat: 100 ms, period: 5 secs}},
                {r_type: Queue, params: {what: obj}}]}]}
```

Figure 2.8: Adaptive storage tier abstraction.

achieved in Section 2.5.4 in more detail.

Using remote storage tiers may incur monetary network cost which should be considered. Wiera provides the flexibility for users to choose the right point in the cost-performance tradeoff. In a hybrid cloud environment, an application may not need to worry about the network cost. If much of the data flow happens from the private DC *into* a nearby public cloud DC, one could acquire better performance without any network cost as network traffic into a DC is normally not charged.

2.3.5 Adaptive Storage Abstraction

Though Wiera enables applications to easily specify desired policies as we have shown so far, Wiera developers still need to be concerned with specifics such as storage cost, network cost, and storage characteristics of particular tiers for the optimized policy. To relax this burden, Wiera provides high-level adaptive storage abstractions (ASA) for the Wiera programmers (e.g., *cheapest (or fastest) storage tier* instead of a specific tier) either to handle requests locally or to forward them to another instance automatically to meet the goals.

Figure 2.8 is an example policy that shows how Wiera can handle put requests in eventual consistency (Figure 2.5) to achieve desired latency goal (100 ms). In this policy, a local instance handles put request locally as long as the desired latency is met as no network cost incurs. If the local instance observes a latency (100 ms) more than period threshold (5 seconds), from local storage tiers, it forwards requests to another instance which can achieve the desired latency goal using latency information. If there is no instance that can meet the latency goal, it simply chooses the cheapest local storage tier. Wiera supports two storage abstractions: 1) *cheapest* that finds the cheapest storage tier that meet the latency goals, and 2) *fastest* that finds the fastest local storage tier or another instance regardless of cost. We have a plan to extend this abstraction for Wiera developers to allow them to query desired storage tier with

more metrics e.g., durability, desired cost, and availability. Note, applications may not achieve latency goal for *put* requests if they want to achieve strong consistency e.g., MultiPrimariesConsistency policy (Figure 2.3(a)) in which updates are sent to all local instances synchronously in the presence of dynamics. In this case, applications may want to use ChangeConsistency policy 2.6(a) to meet the goals as shown in Section 2.3.4.

2.4 Implementation

We now describe our implementation of the Wiera prototype (under 1000 lines of code written in Python) and how Wiera components work together. We also describe additional features newly implemented beyond Tiera. To enable communication with applications, Wiera launches a Thrift [32] server, a remote procedure call framework, that enables applications written in different languages to communicate with each other. Since local instances now need to connect to Wiera and all other instances, we implement a communication component using Thrift in Tiera (under 500 lines of codes written in Java) while most of the Tiera code base remained unchanged. A global policy is implemented in the instance by hand-coding (as mentioned in Section 2.4.1, generating Java code from a specification will be done automatically as future work) the event-response pairs into Tiera's control layer. We implemented the Wiera policies that were explained in Section 2.3.4 (in 100 lines of codes written in Java per Wiera policy). Figure 2.9 shows a snippet Java code to implement the *MultiPrimariesConsistency* policy (Figure 2.3(a)) as an example. As shown in the figure, it is straightforward and easy to produce Java code[3] for a Wiera instance from its specification, e.g., storage tiers' properties can be set with a few function (*put*) calls. Note that Wiera mainly manages local instances and their policies but is not involved in data movement. *All data flow* happens directly between local instances as specified in the policies.

[3] In future work, we plan to provide an automated compiler which generates Java code from a Wiera specification. At present, the Java is hand-coded.

```
//Storage Tiers
memoryTier.put("tier_type", SingleTierInfo.TIER_TYPE.MEMORY);
memoryTier.put("tier_name", "Memcached");
memoryTier.put("tier_list", "localhost");
memoryTier.put("tier_size", "5G");
diskTier.put("tier_type", SingleTierInfo.TIER_TYPE.DISK);
diskTier.put("tier_name", "EBS" );
diskTier.put("tier_list", "ebs_cache/");
diskTier.put("tier_size", "5G");
tiers.put("tier1", memoryTier);
tiers.put("tier2", diskTier);
region1.put("name", "LowLatencyInstance");
region1.put("region", "US-West");
region1.put("tiers", tiers);
...
regionN.put("name", "LowLatencyInstance");
regionN.put("region", "EU-West");
regionN.put("tiers", tiers);
//Broadcasting updates to other instances synchronously
writeLock = getGlobalWriteLock(key);
writeLock.acquire(); //Get a global-lock
version = instance.put(key, value, tierName, tag);
broadcastToPeers(key, version, tierName, tag, 0, latencyInfo);
WriteLock.release(); //Release the global-lock
```

Figure 2.9: MultiPrimariesConsistency implementation in Java code.

2.4.1 Wiera Communication

As described in Section 2.3.1, Wiera is composed of multiple components. Whenever a local server (note, not a local instance) launches, it connects to the *Local Server Manager* (LSM) first to let Wiera know that it is ready to spawn instances. Note: instances run within the local server process for simplicity, but could easily run as a separate process for better fault tolerance. The LSM holds all information about local servers and periodically sends a "ping" message to check on their health. The steps to initiate local instances on multiple regions are as follows: 1) an application specifies the instances, their regions, and policies through the Wiera application interface, 2) when Wiera gets the request, the *Global Policy Manager* (GPM) creates a new policy with a *Wiera id* sent from the application and launches a new *Local Instance Manager* (LIM) to communicate with the local instances which will be created, 3) the LSM asks the local servers to spawn instances with storage tiers and local policy as specified in the

request, 4) a local server receives the request, spawns a new instance, and informs the instance about the LIM address to which the new instance will connect, 5) the new instance runs a server with a unique port number to communicate with other instances. It then connects to the LIM and sends its own server information (port number for the application and port number for communicating between instances), 6) when the LIM accepts server information from its instances, it propagates information to all instances, 7) Wiera returns the list of instances and the *Wiera id* to the application which sent the request, and 8) the application can connect to the closest instance (placed at the head of the list) and sends requests to it.

2.4.2 Global Lock and Conflicts Handling

If an instance is replicated it may need to obtain a global lock before distributing updates to all other instances. For example, if an application specifies the *MultiPrimariesConsistency* policy (Figure 2.3(a)), it should get the global lock first for data consistency. For the global lock, Wiera relies on Zookeeper [24], an atomic messaging system that keeps all of the servers in sync, and we use the Curator library [33] for using Zookeeper easily. When an instance gets updates from another instance, it will update the object as specified in the global policy. In the *MultiPrimariesConsistency*, as an example, if an instance receives an update from another instance, it will simply update the object because the instance that sent the update has a global lock for the object and thus it does not need to be concerned about data consistency. However, in the *EventualConsistency* policy (Figure 2.5), instances should check whether there is any write-write conflict between instances whenever they get updates from another instance. This is needed to avoid version conflicts because they do not hold the global lock for better write performance. To handle this, we add a new feature, which allows applications to have multiple object versions. Each object can have multiple versions with added metadata including version number, create time, access count, last modified time, and last accessed time. All object metadata is stored and persisted using BerkeleyDB [34]. When instances distribute updates to other instances, they also send metadata including object version and last modified time. Thus, each instance that receives an update can decide whether it will accept the update based on the metadata version and last modified time. In the current implementation, we choose a simple strategy, last write wins. That is, updates

will be accepted when they have a higher version number than the local object or when the update is newer (most recently written) than the local object if the versions are the same. We add new APIs for this feature as shown in Table 2.1.

2.4.3 Events and Responses for Wiera

As mentioned in Section 2.3.3, we added *events LatencyMonitoring*, *RequestsMonitoring* and *ColdDataMonitoring*, to Wiera to handle dynamics in the multiple cloud environment. *LatencyMonitoring* events are handled by a dedicated thread which waits to be signaled. The thread handling the application request will signal the dedicated thread to check the latency. The dedicated thread checks whether the conditions (a latency threshold and period of the violation) are met. If it is determined that all conditions are violated it will notify Wiera to handle it. In our example policy (Figure 2.6(a)), a *change_policy() response* request with a new desired consistency model will be issued to Wiera to change the consistency model.

RequestsMonitoring events are handled by the dedicated thread which waits to be signaled in the primary instance (or in all instances as specified by the policy). The thread which handles requests in the primary instance signals the dedicated thread to check the number of requests from both an application and other instances. If the thread detects that an instance has received more requests forwarded from other instances than it has directly received from the application, a *change_policy() response* request with a new *primary* instance will be issued to Wiera to change the primary instance.

ColdDataMonitoring events are handled by the dedicated thread in each instance. The dedicated thread will keep checking metadata to find any object not accessed for a specific amount of time. If it finds an object which has not been accessed, it will take the actions as specified in the policy. In our example policy, in Figure 2.7(a), it simply moves the object to the cheaper storage tier as a *response*.

2.4.4 Handling Failure

In the current implementation, an application can specify the required number of replicas to be available at all times. If a replica crashes, the system detects this via periodic heartbeat and creates a new replica if this threshold is not met similarly to Google File

System [21]. In addition, if the application observes that the closest instance is down then it tries to send requests to the second closest instance, and so on. In future work, we plan to develop mechanisms in Wiera in support of new reactive fault tolerance policies.

2.5 Experimental Evaluation

We evaluated the Wiera prototype in the Amazon cloud and Azure. Wiera and local instances were hosted on Amazon EC2 instances. For our experiments, we used EC2 t2.micro instances, 1 vCPU, 1GB of RAM, and 16GB of EBS storage for Wiera and local servers unless mentioned otherwise. Wiera is running on the US East (Virginia) region and Zookeeper is also running with Wiera on the same instance (for global locking purposes). Local servers are running on multiple regions depending on experiments. The client workloads were generated using Yahoo Cloud Serving Benchmark [35] (YCSB) and we use the popular open-source benchmark tools SysBench [36] and RUBiS [37]. We measure latency from the perspective of an application within a DC, with clients running on the same VM where the instances are running (thus no wide-area latency from users of applications). Our experiments illustrate the following: (1) it is easy to change the data consistency model and configuration using Wiera to handle dynamics from applications and cloud services, (2) Wiera can enable applications to optimize for a particular metric in multi-cloud environments, (3) Wiera can be easily used with an application without any modification and (4) Wiera is not a performance bottleneck even with an increasing number of local instances on multiple locations.

2.5.1 Cost-Performance Tradeoff for Strong Consistency

To achieve strong consistency for a better user experience, applications may want to use either *MultiPrimariesConsistency* (Figure 2.3(a)) or *QuorumConsistency* (Figure 2.4). In this experiment, we compare performance and cost between MutliPrimariesConsistency and QuorumConsistency with 8 DCs of AWS, e.g., Virginia, California, Toronto, Tokyo, Mumbai, Ireland, Frankfurt, and Sao Paulo. We examine four different quorum configurations for strong consistency, i.e., write quorum (W) and read quorum (R),

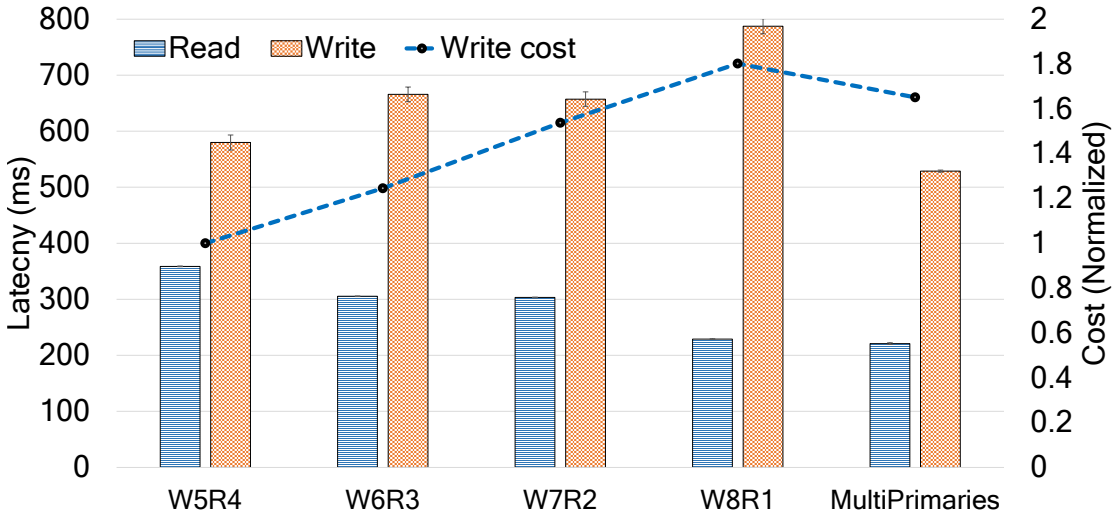


Figure 2.10: Performance and cost comparison between Quorum and MultiPrimaries consistencies.

W5R4, W6R3, W7R2, and W8R1. We use 128KB data for requests. During the experiment, we measure the latency and cost of storage and network bandwidth for both read and write on the California (US West) region.

Figure 2.10 shows that applications can tradeoff monetary cost for operation latency for strong consistency. All configurations of QuorumConsistency provide worse performance compared with MultiPrimariesConsistency but with cost saving up to 40%. For example, W5R4 case results in 40% cost saving due to less replicas (storage) and network traffic compared with MultiPrimariesConsistency but incurs additional write latency (50 ms) due to network traffic for consensus. Note, the W8R1 configuration has the same effect as MultiPrimariesConsistency but with additional network traffic (cost) for consensus. The figure also shows a tradeoff between read and write performance and monetary cost in QuorumConsistency. That is, having more replicas increases write latency and cost but decreases read latency. For instance, W5R4 case raises the read latency by 36% and overall cost by 80% but reduces the write latency by 36% compared with W8R1 case. The results show that *Wiera can provides flexibility in terms of cost and performance for strong consistency.*

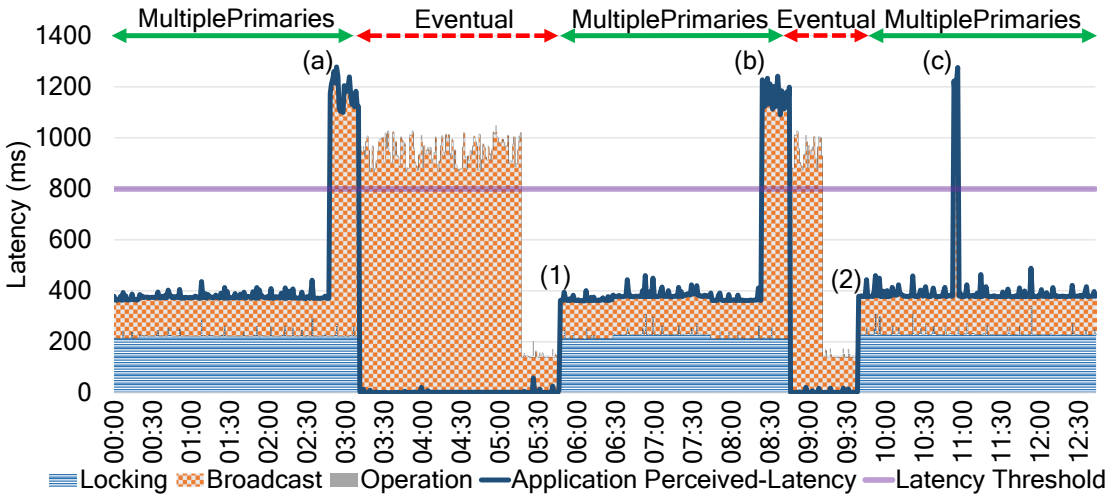


Figure 2.11: Changing consistency at run-time.

2.5.2 Changing Consistency

In this section, we show how Wiera changes consistency policy dynamically as specified in the *ChangeConsistency* policy (Figure 2.6(a)), using a *put* operation latency threshold of 800 ms and a period threshold of 30 seconds. In this experiment, instances are running in regions US West, US East, Europe West, and Asia East, and simulated applications send requests to instances in all the regions using workload A: an update heavy workload in YCSB [35].

We set instances to use the *MultiplePrimariesConsistency* policy (Figure 2.3(a)) initially in which all *put* operations result in updates being distributed to all other instances synchronously. Figure 2.11 shows the latency for *put* operations in US West region. Note that we see a similar pattern of results from all regions, and we omit these results for simplicity. The bold line in the figure indicates the application-perceived latency. Initially, the application sees around 400 ms which includes time for getting (and releasing) the global lock for a key, broadcasting updates to all other instances synchronously, and internal operations (write to local storage). We inject delays into an instance to simulate network or storage delay. In the figure, we can see that there are 3 simulated delays from (a) to (c). All of these delays cause the operation latency violation (800 ms), but only delays (a) and (b) cause a period threshold violation (30 seconds).

For delays (a) and (b), Wiera detects that both thresholds are violated, so it changes the consistency to the *EventualConsistency* policy (Figure 2.5) to preserve application-perceived *put* operation latency which now becomes less than 10 ms. This is because instances now don't need to get the global lock for the key and broadcasting updates can be done in the background in the *EventualConsistency*. Note that Wiera identifies the last delay (c) as being transient and hence, ignores it. When Wiera detects that there is no additional delay during the period threshold (30 seconds) i.e., points (1) and (2) in the figure, it changes the consistency model back to the *MultiplePrimariesConsistency*. This result shows that *Wiera can adaptively change consistency models to handle dynamics at run-time.*

2.5.3 Changing Primary Instance

User location is another factor that may be important for data placement policy as shown in systems such as Tuba [14] and Volley [16] . Tuba shows that changing the storage configuration can improve overall resource utilization and user-perceived latency. In this section, we show how Wiera can easily achieve the same goals as Tuba. To do this, we implement one of Tuba's policies: changing the *primary* instance based on user location.

In this experiment, instances are running on three regions: US West, Europe West, and Asia East. 10 clients are running per each region and the number of active clients are modeled with a normal distribution to mimic the workload in different regions of the world. The mean of the normal distribution is 7.5 minutes and variance is set to 5 minutes. The number of active clients will increase and decease in the following order, Asia East, EU West and US West. Each simulated client sends requests to instances for each regions using workload B: Read mostly workload (5% put and 95% get) in YCSB [35]. We use the *queue response* mentioned in Section 2.3.4 to distribute updates asynchronously to other instances as Tuba does. We implement the *ChangePrimary* policy (Figure 2.6(b)). The difference as compared to Tuba is that Wiera changes the primary instance by comparing the number of *put* operations from clients and from other instances forwarded while Tuba used a cost model. Wiera could also adopt this cost model if desired. Initially, we set the primary instance to run on the Asia East region. The primary instance checks the *put* operation history (last 30 seconds) to

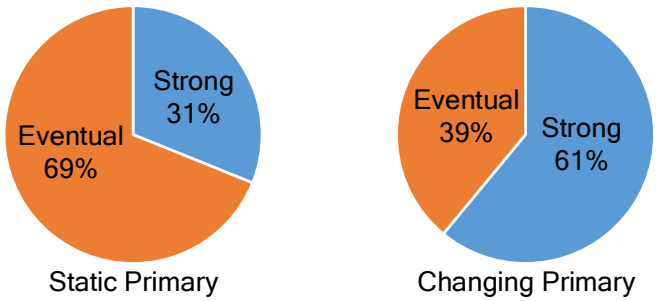


Figure 2.12: Percentage that applications can see the latest data (Strong) and outdated (Eventual) data.

find an instance which forwards more requests than the primary instance received from clients. We set the time period threshold to 15 seconds.

Figure 2.12 shows the probability that the clients will see the latest data (Strong) and outdated data (Eventual). With a static (non-changing) primary location, 69% of *get* operations can return outdated data: clients that are not close to the primary instance can see outdated data since the updates are distributed asynchronously. Wiera reduces this to 39% when the the primary instance location is changed dynamically. That is, more clients now have a greater chance to obtain the latest version of the data from their closest instance. This pattern is similar to that shown for Tuba.

In addition, the overall application-perceived *put* operation latency is also decreased by changing the primary instance. Table 2.3 shows an average *put* operation time for each region and overall average of all regions. With static primary location, the clients in Asia East can see low latency ($<$ 5 ms) since they are always close to the *primary* instance, but clients in other regions need to wait a long time until *put* operations are forwarded to the *primary* instance. With changing of the primary instance, clients in all regions can have a greater chance that their closest instance will become the primary instance, so that the overall *put* operation latency can be decreased. These results show that *Wiera can easily adopt policies hard-coded in other systems.*

2.5.4 Reducing Cost Using Multiple Storage Tiers

Many Internet services and applications have reported that their data access pattern follows Zipfian distribution e.g., Facebook [31], that is large portion of data is accessed

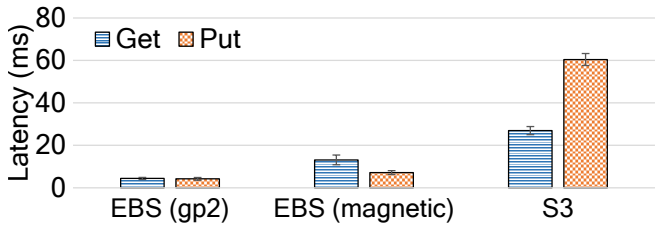


Figure 2.13: Operation Latency for 4KB in US East.

Table 2.3: Average put operation latency (ms)

	EU West	US West	Asia East	Overall
Static	216.61	105.26	< 5	105.18
Changing	95.19	72.20	40.60	68.13

infrequently or not at all. Applications using cloud storage services, however, have to pay for the storage provisioned even for cold data whether it has been accessed or not. Even worse, the size of data will keep increasing but never decreasing while a large fraction of data will not be accessed.

In this section, we will describe how an application can save the cost for storage with a new *ColdDataMonitoring* event explained in Section 2.3.3.

Within a DC, an application has various durable storage options with different performance. Figure 2.13 shows the latencies that the application can see from each storage tier through a local instance. Table 2.4 shows the prices for provisioned storage (per GB/Month), put/get requests (per 10,000 requests), and network cost (within a DC, between AWS DCs, and to Internet respectively). Unsurprisingly, we see clear evidence that applications can get better performance from more expensive storage tiers. That is, EBS SSD (gp2-general purpose) ($0.1/GB) provides the best performance and S3 ($0.03 or $0.0125 for S3-IA) provides the worst performance while EBS HDD (magnetic) ($0.05) is in between them. Note that since EBS uses the OS buffer cache, we see very low latency (< 1 ms) regardless of EBS type if there is enough memory on EC2. To see the native performance of EBS, we throttle the memory by running a memory-intensive application while doing the experiment.

Based on this cost and performance information, let's assume that an application

Table 2.4: Storage Tiers' Price in AWS (US East) as of Feb 2017

	EBS (SSD)	EBS (HDD)	S3	S3-IA
Storage	$0.1	$0.05	$0.03	$0.0125
Put request	$0	$0.0005	$0.05	$0.1
Get request	$0	$0.0005	$0.004	$0.01
Network	$0 / $0.02 / $0.09			

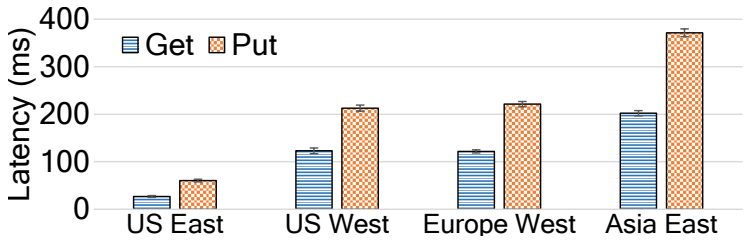


Figure 2.14: Operation Latency for S3 in US East from each region.

sees that 80% out of 10TB data in EBS have not been accessed for 120 hours, in Figure 2.7(a) as an example. As a *response* for a *ColdDataMonitoring* event, each instance will move 8TB data into S3-IA and the application will save $700 (if data was stored in SSD) and $300 (if data was stored in HDD) per month for each instance. Of course, the application will see higher latency for cold data in S3-IA and pay a more expensive request cost than EBS, but this will happen very rarely as the data is cold. For the high *put* operation latency from S3-IA, the application can ignore this since all *put* operations will be done in other faster storage tiers as specified in the policy. Thus, the application can save the storage cost by moving cold data into cheaper storage without much penalty.

The application can save even greater storage cost if it allows instances to share the storage tier where the cold data is stored. That is, when Wiera detects that data is getting cold from all regions, it will ask the instance running on a single centralized region to move cold data into local S3-IA and will ask other instances running on other regions to remove cold data as a *response* for the *ColdDataMonitoring* event. If an instance running on a region other than the centralized one needs to read cold data, it will access the S3-IA storage tier located at the centralized region. Since S3-IA is a durable storage tier, the application doesn't need to consider data durability even

with the reduced number of data replicas. Of course, the application needs to consider operation latency and network cost for the centralized storage tier. Figure 2.14 shows the operation latencies from all regions when all instances use S3-IA in US East region as a shared centralized storage tier. The highest *get* operation latency is around 200 ms when a request comes from Asia East. If this *get* latency is acceptable to the application, it can save $300 more (from our previous example, $100 per each region) by reducing the number of replicas for cold data. The high *put* operation latency also can be ignored since all *put* operations will be done in each region locally. In this example, the cost for requests becomes much more expensive by using a centralized storage tier, i.e., from free or $0.0005 to $0.01 per 10,000 *get* operation request, and from free to $0.09 (or $0.02 between AWS) per GB for network as shown in Table 2.4. However, by definition, the access to cold data will be rare.

2.5.5 Exploiting Non-Local DC's Storage Tiers for Better Performance

One of the benefits of Wiera is that it increases the range of storage tier options. In Section 1.1.2, we have shown that a *nearby* faster DC storage tier can provide better performance than a local but slower DC storage tier even with wide-area network latency. In this section, we will show how Wiera can let applications achieve better performance from non-local DC storage using a benchmark, SysBench, a system performance benchmark. Note that we have built our own POSIX-compliant file system using *Filesystem in User Space* (FUSE) [38] to run applications that require a POSIX interface to Wiera, so that all application requests are forwarded to Wiera through FUSE. Thus, applications that require a POSIX interface can run on top of Wiera *without any modification*.

In this experiment, we compare I/O performance between Azure's local disk without Wiera and AWS' memory with Wiera using SysBench. We use Azure instances, Basic A2 (2 CPU, 3.5 GB of RAM), Standard D1 (1 CPU, 3.5GB of RAM), Standard D2 (2 CPU, 7GB of RAM) and Standard D3 (4 CPU, 14GB of RAM), and AWS EC2 t2.micro instance for non-local memory storage. First, we measure the native disk performance attached to Azure VMs. To avoid any cache (memory) influence, we turn the host cache off for the disk attached and use the O_DIRECT flag for SysBench. This kind of setting is desired for some applications e.g., database systems (MySQL), to avoid double cache

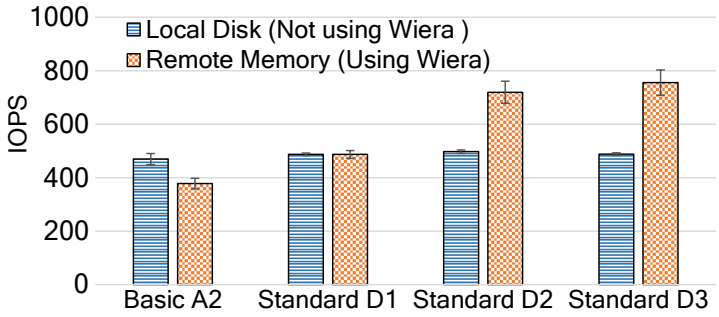


Figure 2.15: Performance (IOPS) comparison.

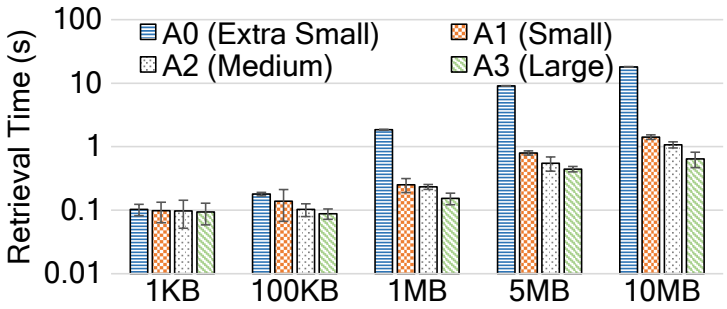


Figure 2.16: Elapsed time (log scale) for a client on AWS US East to tretieve different size of data on memory from various type of VM on Azure US East.

effects that may create cache misses. We then measure the *remote* memory (in AWS) performance through Wiera. In this setting, we deploy instances on AWS and Azure in the US East (Virginia) region where the latency between DCs is around 2 ms as shown in Table 1.1. We use the *PrimaryBackupConsistency* policy (Figure 2.3(b)) with synchronous update (*copy response*) and set an instance running on Azure to be the *primary* instance. We set the *primary* instance to have a disk storage tier only and set another instance on AWS to have memory storage tier. We set a *get* operation policy for all *get* operations to be forwarded to the instance on AWS. That is, if the *primary* instance receives *put* operations from SysBench, it puts data into local disk and sends the update to another instance on AWS synchronously. If the *primary* instance receives *get* operations from SysBench, it retrieves data from another instance on AWS i.e., *remote* memory instead of local disk.

We run the SysBench benchmark on Azure 10 times varying the VM size. Figure 2.15 shows results for each VM size. For the local disk performance, the figure shows the

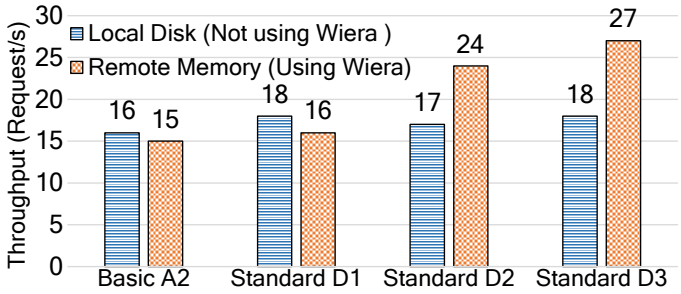


Figure 2.17: Throughput (request/s) comparison.

same performance ($\sim$ 500 IOPS) regardless of VM size. This is because Azure throttles the disk performance to 500 IOPS [39]. For the *remote* memory performance through Wiera, performance is sensitive to the VM size. Wiera can achieve a 44% performance improvement when the *primary* instance is running on Standard D2 and Standard D3 instances. Accessing non-local DC memory through Wiera may be affected by CPU performance but the fact that Basic A2 (2 CPUs) provides worse performance than Standard D1 (1 CPU) implies CPU is not a bottleneck in this experiment. This is because Azure also throttles the network performance based on VM type and size as Figure 2.16 shows.

This result shows that an application can achieve a desired goal (better performance) using *nearby* faster DC storage tiers through Wiera if network performance between DCs is not a bottleneck.

2.5.6 RUBiS on Wiera

We next explore running an *unmodified* web application, the popular open-source benchmark RUBiS, on Wiera to show that the benefits can be obtained with minimal impact to existing applications. RUBiS is a multi-component Web application that implements the functions of an auction site eBay.com, selling, buying, bidding, commenting and so on. We use Apache and PHP for the front-end Web server and MySQL for the back-end database. We use the same evaluation environment setup as in Section 2.5.5. All RUBiS components are hosted on an Azure VM.

For this experiment, MySQL uses two different storage settings: either *local* disk or *remote* memory through Wiera. We set the flag O_DIRECT (which prohibits MySQL

to use the OS buffer) and reduce MySQL internal buffer size to the minimum (16MB) to see the performance from the native attached disk. The database was populated with information for 50,000 items and 50,000 customers. 300 simulated clients are hosted on a separate t2.micro EC2 instance on the same region (US East). The benchmark is run for 300 seconds, with 120 seconds for ramp-up and 60 seconds for ramp-down. Likewise, we vary VM size from Basic A2 to Standard D3. Figure 2.17 shows the throughput from each VM size. Similar to the SysBench results in Section 2.5.5, we see low throughput from small instances (Basic A2 and Standard D1) and higher throughput (50% $\sim$ 80% improvement) from larger instances (Standard D2 and Standard D3) due to a reduction in network throttling. This experiment shows how easily an application can use Wiera to achieve desired (performance in this experiment) goal by accessing multiple storage tiers on multiple DCs *without any modification*.

2.5.7 Adaptive Storage Abstraction Policy

In this section, we show how Wiera handles dynamics with *adaptive storage abstraction* (ASA) explained in Section 2.3.5 as specified in the EventualConsistency (Figure 2.5), using the *cheapest* storage abstraction, a put operation latency threshold 100 ms and a period threshold of 5 seconds. For simplicity, instances are running in regions US East and US West and simulated applications send requests to instances in all the regions using workload A:an update heavy workload.

Figure 2.18 shows the latency for put operations in US East region. The bold line in the figure indicates the application-perceived latency. Initially, the application sees around 11 ms as it puts data into the local EBS storage with eventual consistency i.e., asynchronously. Similar to Section 2.5.2, we inject delays from (a) to (c) into an instance running on US East to simulate network or storage delay. From the figure, we can see that (b) and (c) cause the latency violation (100 ms), but only (b) causes a period threshold violation (5 seconds). When the local instance running in US East detects that both thresholds are violated i.e., point (1), it forwards put requests to the instance running in US West as it has the next cheaper storage tier that can achieve the latency goal even with WAN latency (~72 ms). When the local instance in US East detects that there is no additional delay during the period threshold (5 seconds) i.e., points (2), it stops forwarding put requests and start using the local storage tier again.

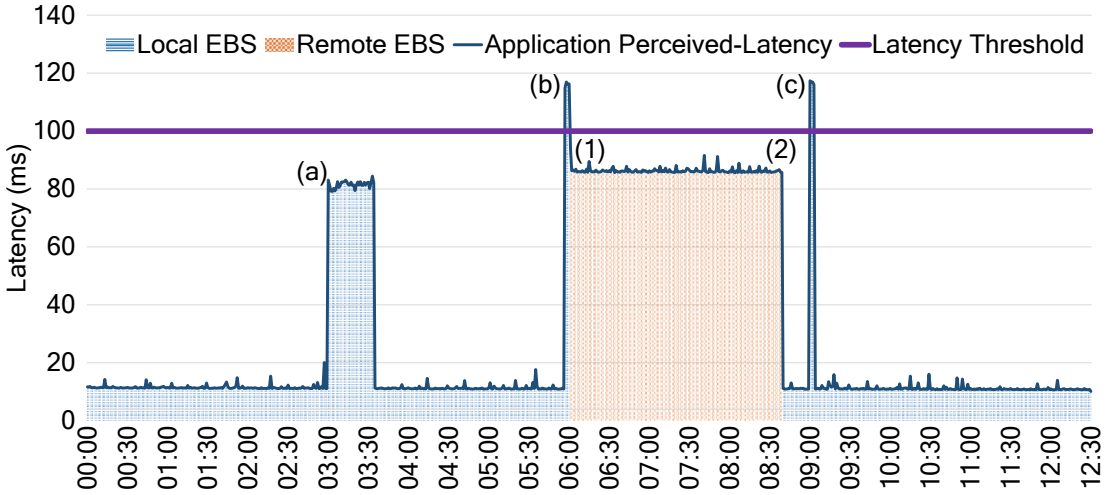


Figure 2.18: Adaptive storage (instance) binding to handle a dynamic.

Table 2.5: Network latencies (ms) in ascending order from US East

Region	Latency	Regions	Latency
US West	72.2	Europe West	81.6
Europe Central	89.4	US West 2	90.64
South America East	120	Asia Northeast	155
Asia Northeast 2	172	Asia South	210
Asia Southeast 2	226	Asia Southeast	228

This result shows that Wiera can adaptively forward requests to handle dynamics at run-time with a high-level abstraction.

Though accessing remote storage tiers may incur extra network cost as shown in Table 2.4, Wiera provides the flexibility for users to choose the right point in the cost-performance tradeoff as mentioned in Section 2.3.4.

2.5.8 Scalability of Wiera

We next conduct experiments to evaluate the scalability of Wiera with the number of local instances as well as their geographic distribution. We evaluate the scalability of Wiera both for control and data operations. We have deployed local instances on all DCs of AWS, i.e., US region (Virginia, California, and Oregon), Europe region

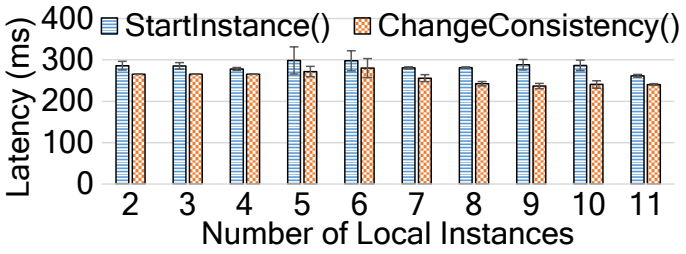


Figure 2.19: Broadcasting control messages latency to local servers (StartInstance) and instances (ChangeConsistency).

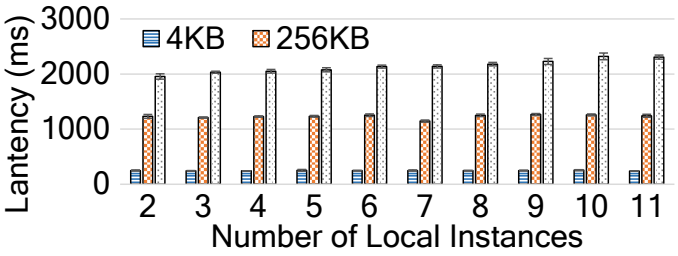


Figure 2.20: PUT operation latency with varying number of local instances.

(Ireland and Frankfurt), Asia region (Tokyo, Seoul, Sydney and Mumbai), and South America region (Sao Paulo). Table 2.5 shows the latencies from US East (Virginia), where Wiera manager is running, to all other regions where local instances are running. In our experiments, as we vary the number of local instances, we deploy additional instances in the inverse order of latency from US East (farthest first). This order is used to show that the performance overhead is bounded by the largest inter-tier latency and not Wiera itself. Thus, for the experiment with two instances, local instances are running on US East (Virginia) and Asia Southeast (Singapore), the farthest from US East. For the experiment with three instances, we add an instance on Asia Southeast 2 (Sydney), the second farthest from US East, and so on.

Wiera Control Message Latency

Figure 2.19 shows the latencies to broadcast control messages, *StartInstance*() to all local servers to spawn instances and *ChangeConsistencyPolicy*() to all local instances to change the consistency model among them. The results show that the latencies are close to the largest network latency from US East (Virginia) to Asia Southeast

(Singapore) (228 ms from Table 2.5) with a small additional latency for handing a message, regardless of the number of instances.

Data Operation Latency

As mentioned in Section 2.4, Wiera is not a bottleneck as it is not involved in the data path. Figure 2.20 shows the user-perceived latency for put operations with the *MultiPrimariesConsistency* policy (Figure 2.3(a)) when a client is running on US East (Virginia). It also confirms that the operation latency will be bounded by the largest latency but not the number of instances as the client can see the similar performance regardless of number of instances. It also shows that if the aggregated data size to be distributed is large, the network bandwidth of the VM can be a bottleneck. That is, there is no performance difference for small size data (4KB and 256KB) but slight performance degradation for 1MB data.

Overall, these results show that *Wiera is able to manage multiple local instances with tolerable performance overhead and that Wiera is not a performance bottleneck in data operations.*

2.6 Related Work

Data Locality: Recent research [40] has shown that data locality within a DC is irrelevant, given the bandwidth of current DC networks. They show that accessing data from a remote node's memory within a DC can provide better performance than reading data from local disk. In Section 1.1.2, we show that data locality may also be irrelevant in multiple DCs environment, and accessing data over the network from the same or faster storage resource in a nearby DC can be faster than using a slower local storage tier. Wiera realizes many opportunities for utilizing cross-DC storage as a complete system.

Policy-Driven Storage: The policy architecture for distributed storage systems (PADS) [41] was proposed for system designers to construct a new distributed storage system easily. Tiera [17], our previous work, provides an easy way to specify policies, allows applications to get composite benefits of multiple storage tiers within a single DC. Wiera exploits storage tiers across multiple cloud providers to get additional benefits such as

simpler consistency and reduced cost, and to handle dynamics from cloud infrastructures and applications at run-time by employing Tiera.

Wide Area Storage: Many previous storage systems [14, 15, 16, 42] utilize multiple DCs of same or different cloud providers. They show that automatic reconfiguration of the storage system can yield substantial benefits such as higher overall resource utilization and better user-perceived latency. However, these storage systems do not adequately handle dynamics from the cloud infrastructure and applications because of their design choices, most notably, a lazy data placement policy decision. Our work tries to handle such dynamics using a combination of local policy, global policy, and multiple storage tiers across multiple DCs. In addition, Wiera provides a flexible substrate that enables the implementation of such existing data policies easily.

Multi-Tiered Storage: Many previous works [43, 44, 45] try to utilize multiple tiers (e.g., HDDs vs. SSDs and local vs. cloud storage) for getting composite benefits of multiple storage tiers. While Tiera [17] focused on multiple storage tiers within a single DC, Wiera focuses on multiple storage tiers on multiple cloud providers to get greater benefits such as enhanced fault tolerance, simpler consistency, wider locality envelope, and reduced cost.

Storage Tiering Features on Cloud Providers: Some cloud providers offer similar features but with significant limitations. For example, AWS S3 provides storage tiering between S3 and Glacier with limitations e.g., only from S3 to Glacier, data size ($>$ 128 KB), and duration ($>$ 30 days) and more. Google only supports deletion of old objects. Wiera provides diverse policies and more flexible features through our own custom implementation but without such limitations while handling dynamics.

2.7 Conclusion

In this chapter, we introduced Wiera, an integrated geo-distributed cloud storage system that runs across multiple storage tiers, multiple data-centers, and multiple providers, to exploit storage options available to the application and user. The diversity of options

is exploited by a flexible storage policy framework that can optimize across a wide array of metrics such as performance, cost, durability, reliability, in the face of network and application dynamics. Wiera is built upon the Tiera storage system to achieve far greater flexibility and adaptability including support for multiple levels of consistency based on current SLAs or performance goals. The results indicate that metrics such as reduced cost and higher performance are obtainable by exploiting the larger set of storage options. Lastly, the benefits can be obtained with minimal impact to existing applications as demonstrated by the *unmodified RUBiS application*.

Although Wiera provides an easy way to specify diverse policies and helps applications handle dynamics at run-time through policies, applications developers still need to determine optimized data placement among countless number of storage combination of DC locations and storage tiers on DCs with a consideration many factors such as applications' goals, network (and storage) characteristics and pricing policies, and workload characteristics. This brings significant complexities to the developers and dynamics make even it impossible to find the static optimized data placement. In the next chapter, we discuss our data placement decision system to address this challenges.

Chapter 3

TripS: Automated Multi-tiered Data Placement in a Multi-cloud Environment

3.1 Introduction

As we have shown in Chapter 2, many cloud providers offer diverse storage services with different characteristics and pricing policies that can be used by applications to meet their storage needs. For example, Amazon Web Services (AWS) offers many storage services such as ElastiCache, S3, EBS, and Glacier. These services vary in their I/O latency, durability, and cost, providing cloud applications with multiple storage options to serve their users. In addition, there has been a growth in the number of data centers (DCs) being deployed in diverse geographical locations. Thus, besides offering multiple storage services, these geo-distributed DCs provide cloud applications with a further possibility of selecting one or more locations for storing their data. Many popular Internet services e.g., Twitter and Netflix have built multi-tiered storage systems (or components) running on multiple data centers [46, 47] to serve their users with such diverse storage options. In fact, applications can even exploit different cloud providers' storage services for reduced cost or better fault tolerance as presented in Section 1.1.

A key problem in a multi-DC, multi-tier environment is *data placement*: determining

which locations and which storage tiers to place data (replicas) on. Determining the "best" data placement in such an environment is challenging due to a large number of factors: 1) application's desired goals, such as cost, performance, and fault tolerance; 2) network characteristics, such as DC locations, inter-DC network latencies/bandwidths, and network pricing; 3) storage characteristics, such as data models, I/O performance, interfaces, and storage pricing policies; and 4) workload characteristics, such as number of requests and data popularity. As cloud providers offer even more DC locations and introduce new storage services, it will make data placement even more challenging. Further, dynamic changes to both workloads (e.g., changes in data access patterns and locations), and the environment (e.g., network and data center failures, variations in network and storage performance), make it impossible to determine the best data placement statically.

While several efforts have considered the data placement problem in a geo-distributed storage environment [14, 15, 16, 42], they have not considered the possibility of exploiting multiple storage tiers which can have a significant impact on metrics such as storage cost and performance. Recent work [26] has focused on data management across multiple storage tiers within a single DC, which may not be sufficient for a multi-DC environment, e.g., to achieve desired fault tolerance or to serve a dispersed set of end-users. We argue that data placement in a geo-distributed cloud environment must consider *both* multiple locations as well as multiple tiers together, to allow for a rich set of storage policies across cost-performance-reliability dimensions as shown in Chapter 1 and 2.

To address these problems, we present TripS (**S**torage **S**witch **S**ystem), a system that optimizes data placement by considering *both* DC locations and storage tiers. We have designed TripS to be lightweight so that it can be used with any storage system running in a multi-cloud environment [48, 49]. Applications that use a TripS-enabled storage system can make use of TripS by simply providing their high level goals, e.g., performance SLAs, consistency models, and desired degree of fault tolerance. TripS uses network and storage cost information, along with monitoring information about user access patterns, inter-DC network latencies and storage tier I/O latencies to optimize data placement. With given inputs, *the data placement problem is modeled as a constrained optimization problem* and solved using mixed integer programming (MIP) in TripS. While TripS can be programmed to optimize different metrics such as cost,

performance, or reliability, in this chapter, we focus on minimizing cost while satisfying latency bounds and fault tolerance requirements.

TripS-enabled storage systems can handle network and workload dynamics at two levels. First, they can have TripS *recompute the optimal data placement* at coarse time granularities to incorporate long-term changes in system or workload characteristics. Second, to adapt quickly to dynamics as well as to handle short-term dynamics such as transient failures or overloads, we introduce the notion of *Target Locale List (TLL)*, a pro-active approach to avoid expensive re-evaluation of the optimal placement. A TLL is a list of multiple feasible placement options (those that satisfy the SLA requirements from any accessing location) computed *a priori* by TripS as part of its optimization. It uses the parameter *locale count* (LC) that enables applications to tradeoff cost with performance and/or fault tolerance by using faster storage tiers and/or having additional replicas. TLL allows applications to utilize (and switch between) these options at run-time to avoid SLA violations, without requiring the storage system to migrate data.

We evaluate the TripS prototype using Wiera explained Chapter 2, a policy-driven key-value storage system for multi-cloud environments on multiple AWS and Azure DCs to show its efficacy and benefits. We extended Wiera to use TripS and to apply the optimized data placement.

The main contributions of this work are:

- The design and implementation of TripS, the first system that optimizes data placement with a consideration of *both* DC locations and storage tiers in multi-cloud environments.
- Modeling and solving the data placement problem as a constrained optimization problem using mixed integer programming (MIP), enabling underlying storage systems to handle coarse time-scale dynamics through re-evaluation of the optimal placement.
- Introducing the notion of Target Locale List (TLL), a proactive approach that enables underlying storage systems to handle short time-scale dynamics without the need to re-evaluate data placement decision or move data at run-time.
- An empirical evaluation of TripS using the Wiera multi-cloud storage system, in an AWS and Azure cloud environment, showing that TripS can help an application

achieve desired goals with minimized cost even in the presence of dynamics e.g., lowering cost 14.96% ~ 98.1% based on workloads and significantly reducing SLA violations with minimal extra cost.

3.2 System Model

3.2.1 Storage System Model

We consider a federated cloud-based *geo-distributed storage system (GDSS)* spanning multiple data centers (DCs) located across different geographic regions. These DCs could belong to the same or different cloud providers. Further, each geographic region may contain multiple DCs (belonging to one or more cloud providers) located close to each other (i.e., having low inter-DC latency). Examples of a GDSS are SPANStore [15], SCFS [48], RACS [49], and our system Wiera explained in Chapter 2. Each DC supports multiple storage tiers with different characteristics in terms of performance, durability, and cost. For instance, in AWS, an application can get better performance from EBS-io1 but at a higher cost compared to other storage tiers, while S3 can provide cheaper storage but at a higher latency. Thus, applications may use multiple storage tiers for their composite benefits to achieve their desired goals. We assume the GDSS provides an interface to applications to access data from multiple DCs and tiers.

We consider an object storage model where data is managed as objects [25]. This model enforces an explicit separation of data and metadata enabling unified access to data distributed among the different storage services and DCs. We assume that the GDSS supports operations (Get and Put) to access objects using a globally unique identifier that acts as a key. In addition, we assume that a GDSS manages metadata for each object, e.g., size, access frequency, location/storage tier, and time of last access.

3.2.2 Application Model

We consider latency-sensitive applications that use a GDSS to provide reduced user-perceived latency and better data availability to users across different geographic regions. We assume application instances run on multiple geo-distributed DCs. We also assume that GDSS servers run on each DC to interface application data accesses with

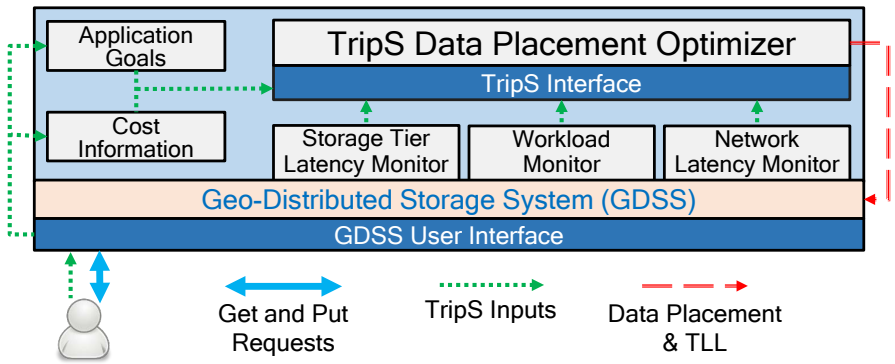


Figure 3.1: How TripS works with a GDSS.

the cloud storage services. An application instance can access data from the GDSS by connecting to any GDSS server (typically the closest one running on the same DC), that can provide access to the requested data (either directly if stored on the same DC, or indirectly from another DC). We assume that applications provide high-level goals, e.g., SLA, consistency model, and degree of fault tolerance to a GDSS through interfaces. In this work, we consider data access latencies between the application instances and the storage system instead of from the end-users as done in other systems [15, 26]. Assigning user requests to appropriate application instances is out of scope for this work, and prior techniques [14, 16] could be utilized for this.

3.2.3 Data Placement Problem

We define a ***locale*** as a {DC-location, storage-tier} tuple, e.g., {Amazon US East, S3}.[1]

The data placement problem consists of determining a set of locales (DC locations and corresponding storage tiers) where data should be placed (replicated) among all available *locales*, in order to satisfy the application requirements (SLA, degree of fault tolerance, etc.). In this work, we consider the goal of minimizing the total cost (both storage and I/O costs).

3.3 TripS Data Placement System

TripS is a lightweight data placement system that can support a GDSS that needs to make data placement decisions on behalf of its client applications. In principle, TripS can run with any GDSS that can provide the information needed to evaluate placement decisions. Figure 3.1 shows how TripS works with a GDSS. TripS makes data placement decisions, which are enacted by the GDSS which then places the data at the desired locales. Applications use the TripS-enabled GDSS for data access (storing and retrieving the data). Note that applications only interact with the GDSS, and do not communicate directly with TripS, so that Trips is not on the data path of application accesses. Unlike other systems, TripS tries to find an optimized data placement that considers *both* DC locations and its multiple storage tiers simultaneously across different cloud providers. TripS can be programmed to optimize for different objectives, e.g., minimizing cost or minimizing latency. In this work, we focus on the objective of minimizing cost while meeting an SLA (both performance and availability).

TripS models the data placement problem as a *constrained optimization problem* (Section 3.3.1) that takes a set of inputs (Section 3.3.1) based on application requirements and workload and environment characteristics. Given these inputs, the TripS optimizer outputs a desired data placement consisting of a list of locales ({DC-location,storage-tier} tuples) where data will be stored. TripS enables the GDSS to handle dynamics through re-evaluation of the optimal solution at coarse time scales (Section 3.3.2). At the same time, it provides the notion of *Target Locale List* (TLL) (Section 3.3.1) to adapt quickly to dynamics at short time-scales (Section 3.3.2).

3.3.1 Data Placement Decision

TripS' Inputs and Output

Inputs: Table 3.1 shows the inputs that TripS uses. TripS requires four types of inputs: applications goals, performance monitoring information, workload information, and network and storage monetary cost.

- **Application Desired Goals:** TripS requires applications to provide three types of

[1] In the rest of this chapter, we omit the name of the cloud provider unless required.

Table 3.1: Inputs for TripS

Input	Description
D	Set of DCs
D_iS	Set of storage tiers in DC i
$SLA^{get/put}$	Get/Put operation SLA from each DC
F	Minimum number of DC faults handled
$Consistency$	Consistency Model
$Center$	Centralized DC location for a global lock (if needed)
$LC\ (>\ 0)$	Locale count in the TLL
$L_{it}^{get/put}$	Get/Put latency for storage tier t in DC i
$L_{ij}^{network}$	Network latency from DC i to DC j
$A_i^{get/put}$	Number of Get/Put requests for DC i
$Size_i$	Average object size in DC i
$C_{ij}^{network}$	Network cost between DC i and DC j (static)
$C_{it}^{storage}$	Storage Tier t provisioned storage cost in DC i (static)
$C_{it}^{get\_/put\_req}$	Get/Put request cost for storage tier t in DC i (static)
$C_{it}^{ret/write}$	Data retrieval/write cost from/to storage tier t in DC i (static)

desired goals. First is an SLA consisting of average latencies for Get/Put operations. Second is the degree of fault tolerance F, i.e., the maximum number of simultaneous DC faults tolerated. Third is a consistency model. Currently, TripS supports only two consistency models: strong and eventual, and supporting other well-known consistency models is left as future work.

Another input parameter, *locale count* (LC), is the number of feasible placement options desired to handle short time-scale dynamics. TripS requires the following information.

- **Latency information:** The intra-DC latency of access to each storage tier, as well as the inter-DC network latency.
- **Workload information:** The access patterns (number of requests from each location) and average object size for requests.

- **Cost information:** Cost information consists of the pricing for network and storages services of all DCs that the GDSS may want to use, as well as the inter-DC network transmission costs. The costs are rarely changed and thus static in this work.

Output: Given these inputs, TripS computes the *data placement* consisting of the set of locales where data should be placed. In principle, TripS can determine data placement for any granularity of data (e.g., single data object to large data collections) and the overhead is tolerable. In this thesis, we evaluate TripS on a coarse placement of data (i.e., the entire data set for an application, as in other systems [16]), and leave fine-grained placement to future work e.g., data placements per *object* or *objects classification*. In addition, TripS also computes a *target locale list* (TLL) which we discuss next.

Target Locale List

We introduce the notion of target locale list (TLL) as a pro-active mechanism to handle dynamics in an agile manner at short time-scales. The main idea is for TripS to generate multiple feasible placement options (instead of just one placement) that all *nominally* satisfy application SLA requirements (based on the current or average dynamics, but are subject to future change). This enables the application to adapt quickly if one of the locales selected for data placement starts violating the SLA, without the need for a data re-placement or migration.

The target locale list (TLL) consists of multiple entries, one for each data access location (i.e., a DC location running application instances that will access data). Each entry in the list contains the set of locales that all meet the SLA from that DC access location. The number of locales specified per DC location is determined by the LC parameter. Thus, each DC location can have multiple choices of locales that can be accessed without SLA violation if LC > 1. Note that while the fault tolerance parameter F controls the number of replicas for availability, LC additionally controls the number of locales that all satisfy the SLA.

The application can use the value of LC to achieve a desired tradeoff between cost and the likelihood of meeting its SLA. For higher values of LC, data would have to be placed on more (or faster) locales to provide enough feasible options that satisfy the SLA from each DC location. This could result in higher cost, but the SLA will be

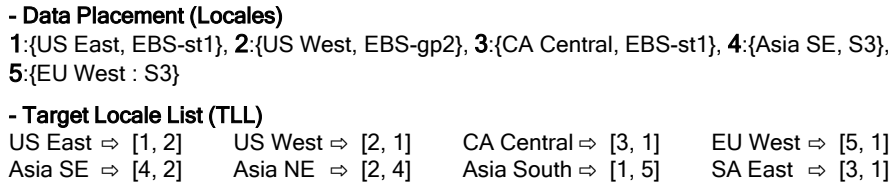


Figure 3.2: TripS output example with LC = 2.

satisfied more often and more consistently. On the other hand, lower values of LC (esp. LC = 1) will result in lower costs but might result in more frequent violations of the SLA.

Figure 3.2 shows an example of data placement and TLL (with LC = 2). The data placement consists of the locales where the data should be placed (replicated). Locales in the TLL provide a high degree of assurance that they will satisfy desired SLA for each DC location. For example, a GDSS server running on Asia NE can access data stored on Asia SE (S3) and US West (EBS-gp2) for both Get and Put requests without SLA violation. In Section 3.3.2, we will discuss in detail how the GDSS can use the multiple options in the TLL at runtime to avoid SLA violations.

Optimization Problem Formulation

Given the inputs, we formulate the data placement problem as a constrained optimization problem, which we solve using mixed integer programming (MIP). The details of the formulation are as follows.

Variables: We define three sets of output variables:

$$\forall i, j \in D, \forall t \in D_j S : T_{ijt}$$

T_{ijt} are binary variables (0 or 1): if 1, data can be retrieved from or written to storage tier t in DC j from DC i within SLA (with a consideration for extra latency for a global lock and data distribution for strong consistency).

$$\forall i \in D, \forall t \in D_i S : P_{it}$$

P_{it} are binary variables (0 or 1): if 1, data will be stored (replicated) in storage tier t in DC i.

$$\forall i, j, k \in D, \forall t \in D_k S : B_{ijkt}$$

B_{ijkt} are binary variables (0 or 1): if 1, DC j will send update to storage tier t in DC k

when DC i sent a Put request to DC j.

Objective: Minimize Total cost = Get cost + Put cost + Broadcast cost + Storage cost, where,

Get cost:

$$\sum_i \cdot A_i^{get} \cdot \sum_j \sum_t T_{ijt} \cdot (Size_i \cdot (C_{ji}^{network} + C_{jt}^{ret}) + C_{jt}^{get\_req})$$

Put cost:

$$\sum_i \cdot A_i^{put} \cdot \sum_j \sum_t T_{ijt} \cdot (Size_i \cdot (C_{ij}^{network} + C_{jt}^{write}) + C_{jt}^{put\_req})$$

Broadcast cost:

$$\sum_i \cdot A_i^{put} \sum_j \sum_k \sum_l B_{ijkt} \cdot (Size_i \cdot (C_{jk}^{network} + C_{kt}^{write}) + C_{kt}^{put\_req})$$

Storage Cost:

$$\sum_i \sum_t P_{it} \cdot Size_i \cdot C_{it}^{storage}$$

Here, we compute the Get and Put costs as the data retrieval and write costs based on the number of requests, the estimated object sizes, inter-DC network cost, and intra-DC storage tier access and request cost. Broadcast cost is the cost of broadcasting updates to all replicas and is based on the number of put operations along with the cost of propagating the writes to other DCs. The storage cost is the cost of storing data and is computed based on the storage price and amount of data stored at each storage tier.

Constraints:

Set number of locales in TLL:

$$\forall i, j \in D, \forall t \in D_j S \sum_j \sum_t T_{ijt} = LC$$

Set minimum number of replicas for availability:

$$\forall i \in D, \forall t \in D_i S \sum_i \sum_t P_{it} \geq F + 1$$

At most one storage tier in each DC:

$$\forall i \in D, \forall t \in D_i S \sum_t P_{it} \leq 1$$

Latency SLA constraint:

For eventual consistency:

$$\forall i, j \in D, \forall t \in D_j S (L_{ij}^{network} + L_{jt}^{get/put} \leq SLA^{get/put}) \; if \; (T_{ijt} = 1)$$

For strong consistency:

$$\forall i,j,l \in D, \forall t \in D_j S(L_{ij}^{network} + L_{jt}^{get/put} + 2 \cdot L_{jk(k=Center)}^{network} + \delta_{isput} \cdot max(L_{jl}^{network}) \leq SLA^{get/put})\ if\ (T_{ijt} = 1), \quad (3.1)$$

where, δ_{isput} indicates whether this is a Put request.

3.3.2 Handling Dynamics

Placement Re-evaluation

TripS may re-evaluate the optimal data placement if the GDSS detects sustained changes in application workloads (e.g., access patterns, location of request origins) or the environment (e.g., network latencies, failures) that can compromise the applications' goals. Alternately, TripS could periodically re-evaluate the data placement to handle potential dynamics, as done in other systems [15, 16]. Re-evaluating a new data placement can be expensive as solving the optimization problem incurs additional overhead. In addition, frequent re-evaluation of data placement can cause unnecessary data migration which is expensive. To prevent TripS/GDSS from thrashing in response to short-time dynamics, a GDSS can set a period threshold to determine whether to re-evaluate data placement ensuring the dynamics are not transient. The handling of short-term dynamics is discussed below.

When a new data placement is very different from the previous one, data migration might be required for minimized cost. However, migrating old data can lead to significant cost in a GDSS. In this work, we assume that data migration is handled by the underlying GDSS. That is, GDSS will determine whether it migrates data or not when it gets a new data placement from TripS. In our prototype, we use lazy reactive data migration in which migration is triggered when data stored on the previous locale is accessed and leave proactive data migration strategies to future work.

Locale Switching with TLL

While placement re-evaluation handles dynamics at coarse time-scales, it is desirable to achieve SLA goals even in the presence of dynamics in short time-scales. As discussed in Section 3.3.1, a TLL consists of locales that can all nominally satisfy the SLA. The

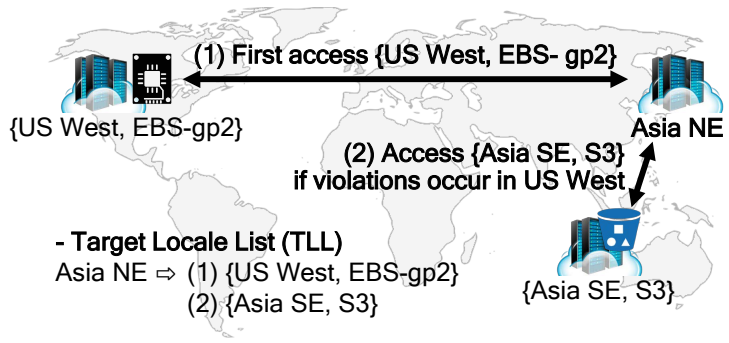


Figure 3.3: Locale switching example.

GDSS can thus switch locales to avoid SLA violations due to short-term dynamics at run-time. When a request arrives to a GDSS, it finds the cheapest (minimum monetary cost) locale to minimize cost using TLL and cost information. If it detects that a violation could happen using this tier based on latency information, it then finds the next cheapest locale in the TLL, and so on. For example, figure 3.3 shows how the GDSS server running on Asia NE (from the figure 3.2) can access data without SLA violations in the presence of dynamics. To handle requests, the server first accesses US West EBS gp2 that leads the cheapest cost due to cheaper outbound network of US West DC and zero request cost of EBS gp2. If the server detects SLA violations from US West EBS gp2 due to dynamics, it now accesses Asia SE S3 to avoid SLA violations. Note, applications cannot avoid the penalty introduced by dynamics for Put requests if they want to achieve strong consistency i.e., data needs to be updated synchronously to all locales. This problem for Put requests can be relaxed by changing the consistency model to a weaker one e.g., eventual consistency as shown in Section 2.3.4.

Having multiple locales in the TLL allows applications using TripS-enabled GDSS to trade off cost with performance in the presence of dynamics over short-time scales. One benefit is that this pro-active placement can reduce the cost of dynamic re-evaluation of placement. This is particularly true for transient dynamics.

3.4 TripS Implementation

We have implemented TripS on top of the Wiera GDSS presented in Chapter 2. Wiera manages the underlying storage and interacts directly with applications to provide data access. Wiera relies upon TripS to make automated data placement decisions that Wiera enacts. Note that Wiera is just an example of GDSS to show how GDSS can utilize and interact with TripS for data placement decision. Any GDSS or applications can utilize TripS based on their requirements as mentioned in Section 3.3.

Table 3.2: TripS API

API	Arguments	Functions
`set_cost`	cost_information	Set cost information
`set_goals`	application_goals	Set applications' goals
`evaluate`	monitoring_information	Evaluate dataplacement

3.4.1 TripS Interfaces and Execution

Table 3.2 shows the TripS API. In our prototype implementation, Wiera sends cost information, applications' goals and monitoring information e.g., network latency, storage latency and access pattens through TripS API that is declared and implemented with Thrift [32] to make a new data placement decision. TripS can be executed as a standalone server but it runs alongside the global Wiera instance server in this work. TripS uses PuLP [50] (toolkit for linear programming in Python) to model a data placement problem as a mixed integer programming (MIP) and uses solver CPLEX [51] to solve the optimization problem.

3.4.2 Wiera Extensions

We have added a few monitoring components (for monitoring information) and additional events/responses (for handling dynamics) to Wiera to enable it to enact the data placement via the TripS API. Wiera now exposes APIs e.g., *set_cost()* and *set_goal()* that forwards the cost information and the applications goals to TripS. As discussed above, a Wiera global instance consists of multiple local instances. Based on the data

placement decision, it's possible that only a subset of these local instances may store data at any point of time. In what follows, we use *active instance* to refer to a local instance that is participating in the current data placement and an *inactive instance* to one that does not (i.e., it is available but is not currently chosen to store data).

Monitoring Components

A few monitoring components have been added to Wiera to utilize TripS. Note that these could be provided either by a GDSS (as now in Wiera) or an external monitoring service that a GDSS relies upon.

- **Network Latency Monitoring between DCs:** For network latency information between DCs, local instances send "ping" messages periodically to each other to estimate the network latency between them.
- **Storage Latencies and workload information:**Wiera has a monitor to check latency for each Get/Put request and number of requests for each object. To handle short and coarse time-scale dynamics, each instance needs to know other instances' storage tier latency and number of requests. To this end, the local storage tier latency information of an instance and number of requests are exchanged and piggybacked on the response for the "ping" message.
- **Background Storage Latency Monitoring:** For TripS to work well, the storage latency of **all** tiers must be updated. In Wiera this is done automatically for all active instances that are used and accessed (thus, the monitoring suffers no additional cost). However, inactive instances (and tiers) will not have a chance to be accessed as requests are handled by other instances (and storage tiers). To avoid outdated latency information for inactive instances, a dedicated thread in each local instance periodically checks the latency for storage tiers by sending empty Put and Get requests to them. Since some storage tiers are charged for requests, e.g., S3, Wiera needs to set this period carefully to reduce the cost for monitoring.

```
Wiera UpdateDataPlacement = {
  events: [{e_type: RequestsMonitoring, condition: {forwarded: > 20%, period: 30 secs},
    responses: [{r_type: UpdateDataPlacement, params: {goals_and_cost_info}}]}]}
```

Figure 3.4: A new response updating data placement.

New Response for Updating Data Placement

Figure 3.4 shows a policy that handles coarse time-scale dynamics using *RequestsMonitoring* event in TripS-enabled Wiera. We let *RequestsMonitoring* monitor the number of application requests from each local instance and notify the global Wiera instance if there is a substantial change. Specifically, a change to the data placement is triggered when more than 20% of requests are forwarded within a specific time period. When the instance monitoring *RequestsMonitoring* event sees the changes sustained for a time period greater than a threshold, it asks the global Wiera instance to re-evaluate data placement through the newly added *UpdateDataPlacement* response. Then, the global Wiera instance calls the *evaluate()* function of TripS. We will show how short time-scale dynamics can be handled in Section 3.4.3.

3.4.3 Handling Requests

In this section, we describe how Wiera works with the data placement and the TLL generated by TripS to handle Put and Get requests, and to adapt to short time-scale dynamics.

Get Requests

When a local instance receives a Get request, it finds the cheapest locale from the TLL. Typically, it retrieves data from the local storage tier if the instance has data stored locally, otherwise, it simply forwards the request to a locale in the TLL that offers the minimum cost.

Put Requests

In native Wiera, any local instance that receives a Put request distributes the update to all other instances. In TripS-enabled Wiera, only selected active instances need to store

```
Wiera SwitchLocales = {
  events: [{e_type: ActionPut,
    responses: [{r_type: Forward, params: {to: cheapest, from: TLL}}]}]}
```

Figure 3.5: Switching locale policy.

data to minimize cost and hence, only these need to be updated. All Put requests are handled by locales in the TLL. When an instance receives a Put request from an application, it checks the TLL to find the cheapest locale to store the data. This initial locale selection considers the subsequent costs that must be paid to propagate updates from this initial locale. When an active instance handles a Put request (from applications or other instances), it distributes the update to all other instances. It is possible that it may forward the request to another instance in the TLL if doing so is cheaper than writing locally and distributing the update to other instances, i.e., when the local DC's outbound network cost is expensive.[2] To minimize network cost, only metadata–including key, size, access frequency, locale information, version (if supported), and last access time—is sent to inactive instances as they do not need to store data but need to know data locations to redirect their Get requests.

Switching Locales

Locales in a TLL nominally satisfy SLA goals as mentioned in Section 3.3.2. Figure 3.5 shows a policy that handles short time-scale dynamics by switching locales. To handle requests with minimum cost, local instances use the *cheapest* adaptive storage abstraction (ASA) explained in Section 2.3.5. The *cheapest* abstraction finds the cheapest locale in the TLL using cost information. If a local instance detects an SLA violation for the cheapest locale, it finds (switches to) the next cheaper locale (possibly a nearby DC's storage tier) at run-time to avoid SLA violation. It can switch back to the cheaper locale based on updated monitoring information later on. Since Wiera migrates data lazily, after a placement re-evaluation, some Get requests may initially be served from locales in the old TLL, while Put requests are always served from locales in the current TLL. Figure 3.4 shows a policy that handles coarse time-scale dynamics using

[2] This is similar to *"relayed update propagation"* in SPANStore.

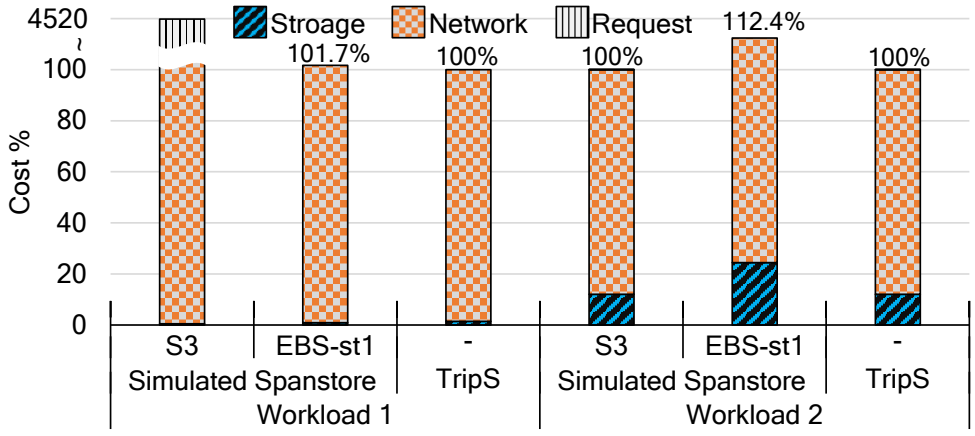


Figure 3.6: Optimized storage tiers selection by TripS with minimized cost. Costs are normalized to the TripS cost.

RequestsMonitoring event in TripS-enabled Wiera.

3.5 Experimental Evaluation

We evaluated the TripS prototype on Wiera in Amazon AWS and Microsoft Azure. For AWS, we used DCs across 8 regions: US East (Virginia), US East 2 (Ohio), US West (North California), US West 2 (Oregon), CA Central (Montreal), Europe West (Ireland), Asia Southeast (Singapore), and Asia Northeast (Tokyo). For Azure, we used DCs at 3 regions: US East, US West, and EU South. All application instances (clients) run in AWS. Due to network cost difference, e.g., Amazon charges $0.02 / GB for outbound network to other Amazon's DCs and $0.09 / GB to the Internet, and Microsoft charges $0.087~ $0.181 / GB based on destinations, we find that TripS typically chooses to store data in Amazon's DCs. Therefore, we show results that include only Amazon's DCs, except for one scenario where we are able to utilize both AWS and Azure together. TripS and global Wiera instance servers are running on Amazon US East (Virginia) region while local Wiera instances are running in all the regions. We used AWS t2.medium (2 vCPU 4 GB of RAM) for TripS/Wiera to have more CPUs for CPLEX and MIP solver. For local Wiera instances, we used EC2 t2.micro instances, 1 vCPU, 1 GB of RAM, 16 GB of EBS storage, 500 GB of EBS-st1 and 2 GB of EBS-gp2 unless mentioned

otherwise. Note that, TripS does not cause any overhead to underlying GDSS as it is not involved in the data path. The time for computing data placement has a negligible impact on the overall cost as we can see that TripS can solve the optimization problem in 1.3 seconds with t2.medium (2 vCPU 4 GB of RAM) for our experiment setting of 8 locations and 3 storage tiers per DC.

For the workloads, we use both workload A: An update heavy workload (50% Put and 50% Get) and B: Read mostly workload (5% Put and 95% Get) derived from the Yahoo Cloud Serving Benchmark [35]. We mainly show the result with workload B as we can see the similar pattern of results from both. Likewise, we mainly show the results with eventual consistency due to space constraints. For EBS-st1, to avoid the OS cache buffer effect, we assign a latency penalty (10 ms) as reported by others [52]. This is a reasonable penalty as we can see that the average disk seek times are 13.38 ms (29.51 ms 95 percentile) and 16.29 ms (38.09 ms 95 percentile) for random read and write (9:1 and 5:5 ratios respectively) with the system performance benchmarking tool [36] for EBS-st1. For EBS-gp2, we do not assign any latency penalty as its seek time is less than 1 ms. For comparison purposes, we simulate Spanstore with TripS by allowing TripS to use only a single storage tier from each DC, e.g., either only S3 or only EBS-st1. Lastly, all cost information we use in this thesis is as of Feb 2017.

3.5.1 Optimizing Data Placement

In this section, we show how TripS chooses locales for a diversity of access patterns and data sizes. In this experiment, we consider two scenarios: 1) latency-sensitive Web applications that use mostly small and frequently accessed data and 2) data analytic applications that mostly use large and infrequently accessed data. We use two simulated workloads with eventual consistency: 1) 8 KB average data size and 10,000 Get accesses and 1,000 Put accesses from each of the 8 DC locations for the Web application scenario and 2) 100 MB average data size and 1,000 Get accesses and 100 Put access from each of the 8 DC locations for the data analytic framework scenario. For storage cost, we use daily cost for workload 1 and monthly cost for workload 2. We use 200 ms for Get SLA and 350 ms for Put SLA for workload 1 and 500 ms for Get SLA and 850 ms Put SLA for workload 2.

Table 3.3: Cost comparison (Costs are normalized to "US East and Asia SE")

DC locations	Storage	Network	**Total**
US East only	45.5%	263.6%	**263.1%**
US East and Asia SE	100%	100%	**100%**

Table 3.4: Data placement and cost comparison

LC	Data Placement	Storage	Network	**Total**
1	US East (EBS-st1), US East 2 (EBS-st1), US West 2 (EBS-st1)	100%	100%	**100%**
2	US East (EBS-st1), US East 2 (**EBS-gp2**), US West 2 (EBS-st1)	140.7%	100%	**105.3%**
3	US East (**EBS-gp2**), US East 2 (EBS-gp2), **US West** (EBS-st1)	188.1%	100%	**111.5%**
4	US East (EBS-gp2), US East 2 (EBS-gp2), US West (EBS-st1), **CA Central (EBS-gp2)**	269.6%	166.7%	**180.1%**

Figure 3.6 shows the cost comparison between simulated Spanstore (considering only a single storage tier) and TripS (considering multiple storage tiers). From the figure, we can see that TripS can minimize cost for both workloads by exploiting multiple storage tiers. For workload 1, TripS mainly chooses EBS-st1 as it does not charge for requests. For Workload 2, TripS chooses only S3 as the storage cost is a non-negligible portion of the overall cost. This pattern of results is similar to Grandet [26] that considers multiple storage tiers within a single DC. Yet, our results consider multiple DCs while Grandet only considers a single DC that is insufficient for a multi-DC environment. For example, if data is placed only in US East, then applications running in Asia SE cannot meet any SLA lower than the inter-DC latency between the 2 DCS, which is more than 220 ms. In addition, even if we had a high latency requirement, using a single DC can lead to higher total cost due to network cost. Table 3.3 shows the cost comparison between using a single centralized DC US East (as in Grandet) and using 2 DCs US East and

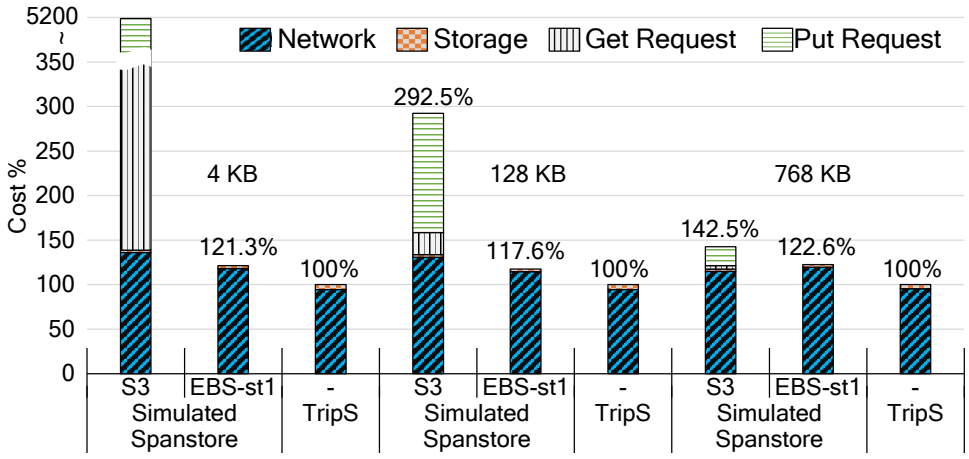


Figure 3.7: Comparing Storage Cost. Costs are normalized to the TripS cost.

Asia SE (2 replicas) with Workload 1. While using a single DC can reduce storage cost, it leads to extra network cost to access data from a remote region that is more expensive in a multi-cloud environment. These results show that both DC locations and storage tiers should be considered for optimal data placement and that TripS chooses DC locations (as in Spanstore) and storage tiers (as in Grandet) based on workloads and access patterns while minimizing overall cost in a multi-DC environment.

3.5.2 Dynamic Data Placement

Access pattern (reads vs. writes and user location) is an important factor to be considered as shown in many previous systems such as Tuba [14] and Volley [16]. In this section, we show how TripS can minimize storage cost by handling access pattern changes while achieving applications' goals. In this experiment, local Wiera instances are running on 8 regions. 10 clients are running per region. The number of active clients is increased and decreased in a cyclic manner from Asia Northeast to US West to mimic a diurnal access pattern. We calculate provisioned storage cost on a day (24 hours) basis as we simulated a daily access pattern. Simulated clients send requests to instances for each region using YCSB workload B: Read mostly workload (5% Put and 95% Get). We use the *UpdateDataPlacement* policy (Figure 3.4) in which a new data placement request is sent to TripS/Wiera as a *response* for *RequestsMonitoring* which monitors the number of requests sent from simulated clients at each instance. We use

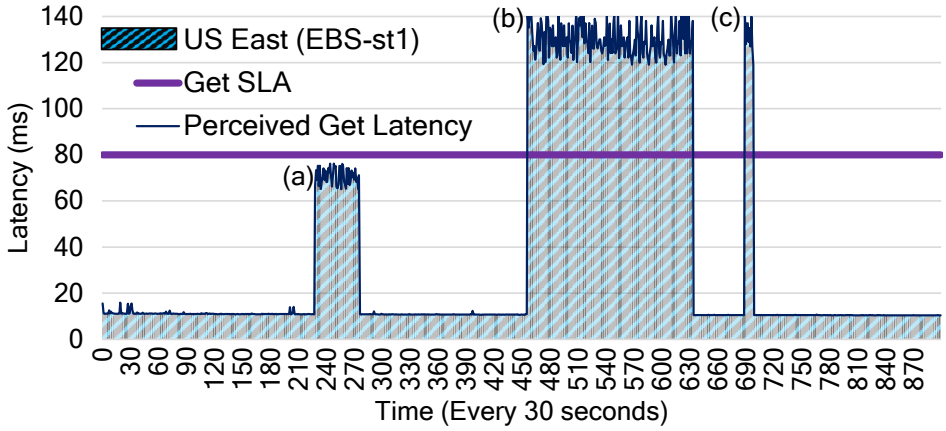
varying data sizes 4 KB, 128 KB, and 768 KB to mimic a photo sharing application's workload based on real-world statistics as in previous work [26]. We set 80 ms for the Get SLA, 200 ms for the Put SLA and LC = 1.

Figure 3.7 shows the cost benefit compared to simulated Spanstore settings that are limited to a single storage tier at each DC. Note that for the single storage tier cases, i.e., S3 and EBS-st1, we also re-evaluate data placement to handle changes like in Spanstore. For the 4 KB size, TripS can reduce overall cost 98.1% compared to S3 only cases and 17.5% for EBS-st1 only. The results show that using block storage (EBS-st1) can reduce overall cost significantly for small and frequently accessed data as S3 charges for each request, e.g., $0.0000004 and $0.000005 for a single Get and Put request respectively in the US East region while EBS-st1 does not charge for requests. This corresponds to the result in Section 3.5.1. From the experiment log, we can see that TripS avoids using S3 and chooses EBS-st1 and EBS-gp2. Even with the expensive storage cost for EBS-gp2, we can see that TripS chooses EBS-gp2 when it can reduce the number of replicas in order to reduce network traffic for distributing updates. For 128 KB data, TripS can reduce storage cost 65.8% and 14.96% for S3 and EBS-st1 only cases respectively. For 768 KB, TripS can reduce storage cost 29.8% and 18.4% as in other experiments. The results confirm that TripS can provide reduced overall cost by exploiting multiple storage tiers in multiple DCs in comparison with single storage tier GDSS's such as Spanstore even in the presence of changing workload patterns.

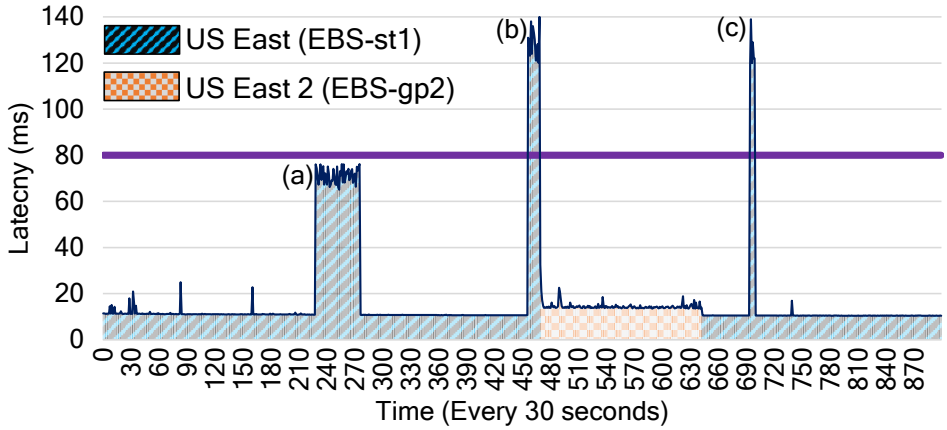
3.5.3 Short-Time Scale Dynamics

Next, we show how TripS enables the underlying GDSS to handle short time-scale dynamics by switching locales at run-time as specified in the *SwitchLocales* policy (Figure 3.5), using 100 ms for Get SLA and 200 ms for Put SLA and a period threshold of 10 seconds. In this experiment, instances are running in North America region, US East (Virginia), US East 2 (Ohio), US West (North California), US West 2 (Oregon) and CA Central (Montreal), and simulated applications send requests to instances in all the regions using workload B in YCSB. We use 8 KB data size in this experiment.

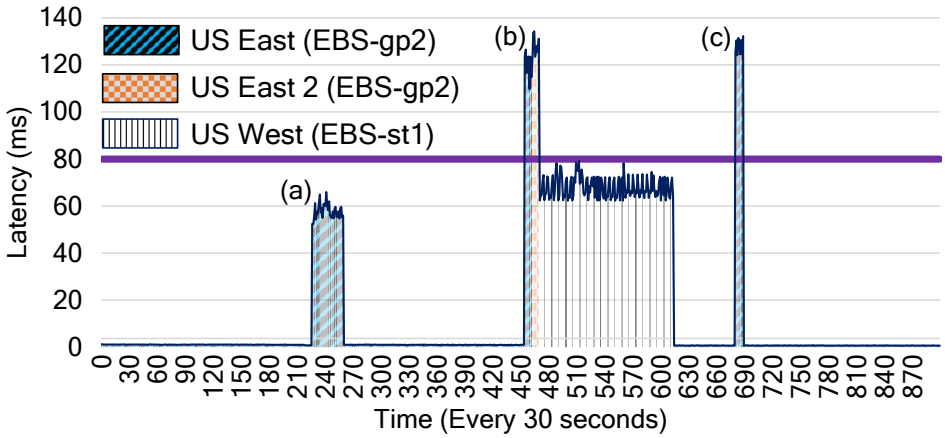
Initially, TripS evaluates the data placement with an assumption that all instances receive the same number of requests from each location. Table 3.4 shows the data placement evaluated with LC = 1, LC = 2, LC = 3, and LC = 4. The table also shows



(a) LC = 1 in the presence of dynamics in US East.



(b) LC = 2 in the presence of dynamic in US East only.



(c) LC = 3 in the presence of dynamics in both US East and US East 2.

Figure 3.8: Applications-perceived latency running on US East.

the extra cost as LC is increased. We can see that TripS chooses faster (expensive) storage tier (EBS-gp2) with extra cost to satisfy LC constraints. However, the total cost is dominated by network cost rather than storage cost in a multi-cloud environment. So as shown in the table, increased total cost is 5.3% and 11.5%. For LC = 3, there is no network cost change even with a DC location change from US West 2 to US West as both DCs have the same network cost policy. Lastly, the table shows that LC = 4 increases the number of replicas that leads to a higher network cost. Thus, applications can trade off cost with performance in the presence of dynamics using the LC parameter.

Figures 3.8(a) and 3.8(b) show the latency for Get operations in US East when LC is set to 1 and 2 respectively. The bold line in the figure indicates the application-perceived latency. For LC = 1 and LC = 2, the application sees around 12 ms as it retrieves data from local (US East) EBS-st1. We inject delays into the US East instance to simulate network or storage delay. In the figure, we can see that there are 3 simulated delays (a) 60 ms delay for 30 seconds, (b) 120 ms delay for 180 seconds, and (c) 120 ms delay for 5 seconds. In both cases, delay (a) does not cause any Get SLA violation. For the delay (b), applications suffer a Get SLA violation at around 180 seconds when LC = 1. However, for LC = 2, TripS/Wiera switches locales to retrieve data from US East 2 EBS-gp2 storage to avoid the violation. For the delay (c), TripS/Wiera does not switch the locales because the delay occurred less than period threshold (10 seconds).

Figure 3.8(c) shows the latency for Get operations in US East for LC = 3. Here, the application sees less than 1 ms as it retrieves data from local (US East) EBS-gp2. We inject the same delays into *both* the US East and US East 2 instances simultaneously. When the instance in US East detects a delay from local (EBS-gp2), it first switches to remote (US East 2) EBS-gp2 which also leads to a Get SLA violation. Once the instance detects a delay from US East 2, it switches to US West EBS-st1 to avoid SLA violation.

For both Figures 3.8(b) and 3.8(c), there is additional network cost as the instance in US East has to access non-local storage. Figure 3.9 shows that the extra cost reduces the rate of Get violations by 91.72% with 12.8% extra cost for dynamic in US East and by 91.14% with 19.7% extra cost for concurrent dynamics in US East and US East 2. We can see a similar pattern of results from all locations with varying latency from the second cheapest locale based on the network latency between DCs. These results show

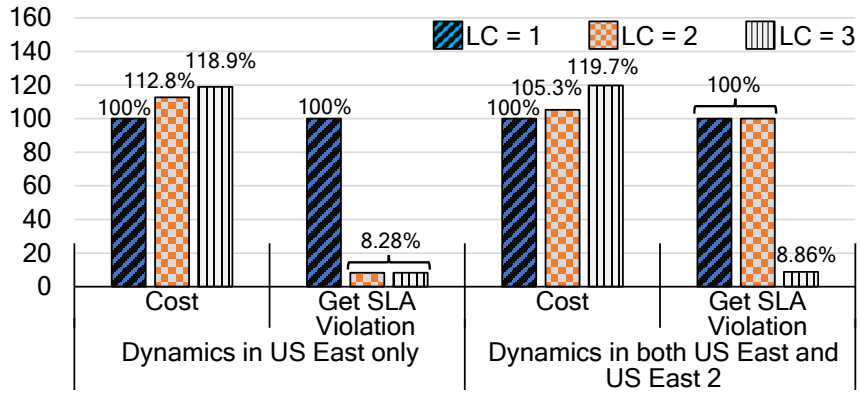


Figure 3.9: Get SLA violation rate and cost increasing rate. All values are normalized to LC = 1.

that TripS can enable a GDSS to adaptively switch locales to handle short time-scale dynamics, with a slightly higher cost.

Using Multiple Providers

Due to the long-haul network latency between DCs, it may not be possible to achieve low latency SLA in a single provider multi-DC environment, if an application desires a TLL with LC>1. However, TripS can exploit multiple DCs (possibly belonging to multiple cloud providers) within a geographic region to achieve these constraints. To show this, we use the exact same experiment setting as the previous section but with a much lower SLA, e.g., Get SLA of 10 ms, Put SLA of 20 ms and LC = 2. In this experiment, instances are running in 6 DCs of AWS and Azure, US East, US West, and EU South i.e., there are 2 DCs in each region. We also inject delays into the US East instance. We measure the latency for Get operations in AWS US East.

Figure 3.10 shows applications can avoid 88% Get violations yet with 585% extra cost due to the network traffic for distributing updates to all 6 locations. To relax the cost issue, TripS may enable applications to set LC only for a specific region e.g., only US East region needs to meet LC constraint. This result shows that TripS can exploit multiple providers' DCs to achieve very stringent SLAs and availability constraints albeit with higher cost.

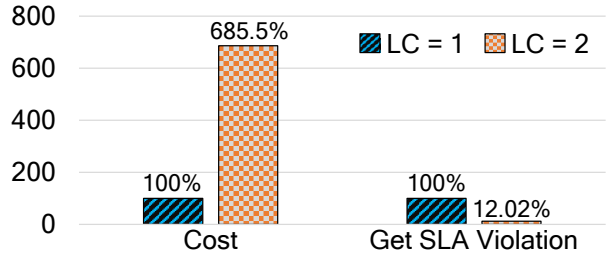


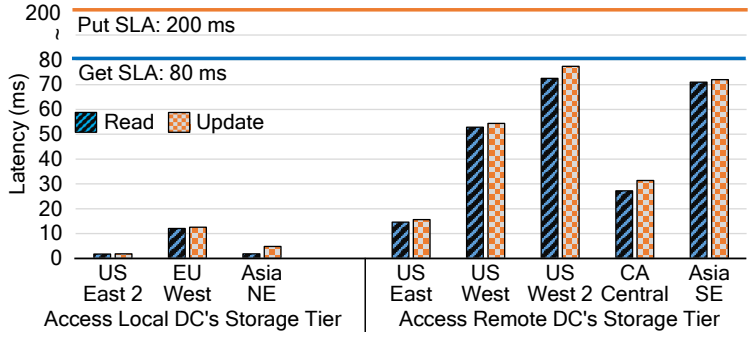
Figure 3.10: Get SLA violation rate and cost increasing rate for latency-critical applications. All values are normalized to LC = 1.

3.5.4 Benchmark and Application Scenario

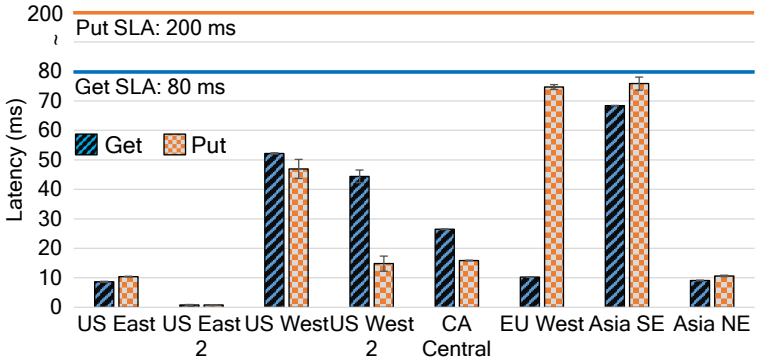
To see that TripS can help real applications achieve SLA goals, we ran the open-source YCSB Benchmark, and a Web application, Retwis on TripS/Wiera. Since both use Redis [53] as a backend storage, we implement Redis functions e.g., *lpush*, *lrange*, *sadd*, and *srem* wrapper interfaces on top of Wiera and modify less than 10 lines of code of the Redis YCSB module and Retwis to enable them to use TripS/Wiera instead of Redis. We ignore the overhead of the wrapper class as we can see that less than 2 ms is required for transforming data from binary format in TripS to the Redis supported data set (hash, map, list and so on).

YCSB Benchmark

To see that local instances can access data within the SLA, we ran the YCSB benchmark client from all 8 locations. We use the same experimental setting as in Section 3.5.2, i.e., 80 ms for Get SLA, 200 ms for Put SLA, 1 for LC, and eventual consistency without changing data placement. YCSB client sends 1,000 operations with YCSB workload B (95% read, 5% write) to Wiera from each DC location. Read and update operations in the YCSB client correspond to the Get and Put operation of Wiera. Initially, TripS chooses US East 2 EBS-gp2, EU West EBS-st1, and Asia NE EBS-gp2 for data placement. Figure 3.11(a) shows the average read and update latency. YCSB clients can see lower latency than the desired SLA latency. For those YCSB clients running on US East 2, EU West, and Asia NE, they can see lower latency than other instances as they have data in the local DC. The YCSB client running on US East also can see lower latency as it is close to US East 2 in terms of network latency (<12 ms).



(a) Average read and update latency for YCSB.



(b) Average Get and Put latency for Retwis.

Figure 3.11: Applications-perceived latency running on US East.

We can see similar results for workload A with strong consistency. This result shows that TripS helps applications achieve desired SLA goals with minimized storage cost.

Retwis

Retwis is a simple Web application that implements the functions of Twitter (loading timelines, posting, following and so on) that perform Gets and Puts on Redis. We use Retwis-py (Retwis written in Python) [54] and modify it to use TripS with a Python wrapper interface instead of Redis. We use the same experiment setting for the YCSB benchmark but with changing data placement. For the workload, we generate 1K users and assign 125 users to each DC of 8 locations. The number of active users increases and decreases from Asia Northeast to US West to mimic a diurnal access pattern as done in Section 3.5.2. Active users randomly choose between a few operations, i.e.,

read timeline (77%), post a new twitter (15%), follow another user (5%), and unfollow a user (3%). We run this experiment using the *UpdateDataPlacement* policy (Figure 3.4) to handle users access pattern changes. Note that each Retwis function requires multiple Get and Put operations, e.g., *following* requires 2 Get and 2 Put requests to TripS/Wiera. In our experiments, we measure and report Get and Put latencies.

Figure 3.11(b) shows the average Get and Put latency that Retwis can see from each location. Similar to the YCSB benchmark in Section 3.5.4, we see both Get and Put average latency are lower than the desired SLA goals. Latency can be different for Get and Put from a few locations as those locations have a chance to access either local or a remote storage tier based on the data placement and TLL. Even with this variation, the results show that Retwis can achieve the desired SLA while handling access pattern changes. This result shows Web applications can get the benefit of TripS with minimum effort.

3.6 Related Work

Automatic Data Placement Storage Systems: Many previous storage systems [14, 15, 16], re-evaluate data placement to achieve applications' goals. However, these storage systems do not consider multiple storage tiers but only DC locations. Since cloud providers are offering numerous storage tiers, it is not an easy task for applications to choose optimized storage tiers in multiple DCs. TripS considers both data locations and storage tiers to generate data placement while achieving applications' desired goals.

Multi-Tiered Storage: Tiera [17] and Grandet [26] are two systems that exploit multiple storage tiers within a single DC. Additionally, Grandet has an optimizer for storage tier selection to minimize cost. While Wiera presented in Chapter 2 exploits multiple storage tiers in multiple DCs, Wiera developers have to choose DC locations and storage tiers which is not an easy task. TripS solves this problem for Wiera (and for any GDSS) by evaluating optimized data placement and TLL that can be applied at runtime to minimize storage cost while achieving an application's desired performance goal.

Policy-Driven Storage: The policy architecture for distributed storage systems (PADS)

[41] was proposed for system designers to construct a new distributed storage system easily. In Chapter 2, we explored building a storage framework called Wiera that helps applications specify a tiered storage instance consisting of multiple storage tiers and DCs in a policy. In this chapter, we extended the Wiera policy specification to minimize storage cost automatically while achieving desired goals that exploits multiple storage tiers and DC locations with the aid of TripS.

Geo-distributed Data Analytics: Many works [55, 56, 57, 58] were proposed to overcome the limits of popular centralized data analytic frameworks e.g., Hadoop [59] and Spark [13] that perform poorly in geo-distributed setting in which network bandwidth between DCs is the most expensive and scarce resource. All of these works show that optimized data placement (minimized network traffics) is important for geo-distributed data analytics to improve overall performance. While we mainly focus on latency-sensitive applications in this work, enabling those geo-distributed data analytic frameworks i.e., network bandwidth-intensive applications, to obtain the benefits of TripS is left as future work.

3.7 Conclusion

In this chapter, we introduced TripS, a system that determines the data placement automatically for geo-distributed storage systems on behalf of applications in a multi-tiered, multi-cloud environment. TripS considers both data center locations and storage tiers to minimize overall cost while satisfying applications' goals and constraints. TripS evaluates the *data placement* by solving a constrained optimization problem formulation with given inputs. In addition, it also generated *target locale list* to adapt quickly to dynamics. To validate the benefits of TripS, we have implemented TripS on our GDSS called Wiera. We illustrated how TripS enables a GDSS to handle both coarse and short time-scale dynamics by enhancing the *event* and *response* mechanism of Wiera. The experimental results across a geo-distributed storage cloud consisting of AWS and Azure storage tiers showed that TripS can lower cost $14.96\% \sim 98.1\%$ based on workloads and significantly reduce SLA violations with minimal extra cost. We presented a novel tuning knob, the LC parameter, that can allow the user to tradeoff cost with performance

and/or fault tolerance. We showed that TripS enables applications to achieve SLA goals with a popular benchmark tool for cloud storage (YCSB). Lastly, we showed that a web application (Retwis) can get the benefits of TripS (e.g., handling dynamics with reduced cost) with minimal modification.

Chapter 4

Kimchi: A Network Cost-aware Geo-distributed Data Analytics System

4.1 Introduction

In Chapters 2 and 3, we presented storage systems supporting latency-sensitive Internet applications that want to exploit heterogeneous storage tiers and to determine optimal data placement in a multi-cloud environment. In this chapter, we study different type of applications that has different workloads compared to Internet applications and show how they can exploit heterogeneous network resources to achieve their desired cost-performance tradeoff preferences in a multi-cloud environment.

4.1.1 Motivation

Recently, geo-distributed data analytics (GDA) has become a popular method for mining valuable information from globally distributed big data generated by users and systems in a multi-cloud environment,[1] in areas as diverse as querying global trend detection on social network data, and log monitoring of geo-distributed CDN servers [60, 61, 62].

[1] We use the term multi-cloud to refer to both a single cloud provider that spans multiple DCs as well as multiple DCs that span multiple cloud providers.

One simple approach is to aggregate all data within a single data center (DC) and then analyze the data using a data analytic framework, e.g., Hadoop [59] and Spark [13]. However, this requires a significant amount of time for migrating large amounts of data into a centralized DC via a scarce and expensive resource, WAN bandwidth. Another alternative is to process data in-place, but it is well-known that Hadoop and Spark perform poorly in a geo-distributed setting due to the large overhead of inter-DC traffic in the shuffle stages [63].

To address this network overhead, numerous approaches have been proposed [55, 56, 57, 58, 64, 65, 66] that attempt to minimize network usage and consider heterogeneous network bandwidth in their scheduling and data placement decisions. However, much of the existing work focuses primarily on how to efficiently use the WAN for performance but does not address data transfer cost ($),[2] one of the most expensive and heterogeneous resources in a multi-cloud environment. This cost can be significant for continuous queries that require large data transfer between DCs. Recent works [67, 68] confirmed that the cost of WAN bandwidth makes up a significant fraction of the overall cost.

One may think that minimizing WAN usage results in minimized cost. Yet, this is not always true due to heterogeneous cloud pricing policies, e.g., up to an 8X inter-DC transfer cost difference even within the same cloud provider (AWS), and a 12.5X cost difference across cloud providers (AWS and Azure), as shown in Table 4.1. Since a large amount of data transfer in a GDA occurs between DCs [58, 69, 70], a GDA may encounter the *cost-bottleneck* due to a cost-agnostic approach that may significantly inflate operational cost.

In this chapter, we argue that *data transfer cost* must be a first-class consideration for a GDA running in a multi-cloud environment to avoid this cost-bottleneck. To consider cost, we are motivated by the following questions.

- What is the minimal query execution time given a target cost budget ($)?
- What is the feasible cost range to execute a query?
- How can a GDA achieve the desired cost-performance tradeoff in a multi-cloud environment?

[2] We use the term cost to refer to monetary cost of data transfer unless mentioned.

Table 4.1: Heterogeneous data transfer cost (per GB) between DCs (as of Jan, 2019) - SA: South America, AP: Asia Pacific.

Origin \ Destination		AWS			Azure		
		US East	SA East	AP NE	US East	SA East	AP NE
AWS	US East	$0	**$0.02**	**$0.02**	$0.09	$0.09	$0.09
	SA East	$0.16	$0	$0.16	**$0.25**	**$0.25**	**$0.25**
	AP NE	$0.09	$0.09	$0	$0.126	$0.126	$0.126
Azure	US East	$0.087	$0.087	$0.087	$0	$0.087	$0.087
	SA East	$0.181	$0.181	$0.181	$0.181	$0	$0.181
	AP NE	$0.138	$0.138	$0.138	$0.138	$0.138	$0

• How can a GDA handle dynamics for better performance during query execution without additional cost?

To answer these questions, we have designed and implemented Kimchi, a cost-aware GDA system. The goal of Kimchi is to explore a richer cost-performance tradeoff space and to achieve the best performance within a desired cost budget. To this end, Kimchi solves a constrained MIP (mixed integer programming) task placement problem that meets a desired tradeoff preference.

One significant challenge to cost-reduction is dynamics that are common in a multi-cloud environment [68, 71, 72], e.g., network contention and bandwidth changes, but most GDA systems [56, 55, 57, 58, 64, 65] ignored dynamics during query execution. While handling dynamics is important for performance, a large data migration may occur for handing dynamics that may lead to a cost-bottleneck. To adapt quickly to dynamics and avoid the cost-bottleneck, Kimchi uses a heuristic that adjusts task placement with a cost-awareness at run-time. Finally, Kimchi considers an asynchronous model, i.e., *push-based shuffle* mechanism [73, 74, 64], that overcomes the barrier synchronization of MapReduce programming model for cost-aware performance to avoid the possible cost-bottleneck.

A prototype implementation of Kimchi is built on the Spark [13] framework and we support new Spark properties that control Kimchi settings, so Spark applications can

utilize Kimchi *without any modification*. We evaluate Kimchi on AWS using the well-known benchmarks TPC-DS [75] and TeraSort [76] to show its benefit. Experimental results show that Kimchi reduces cost by up to 24% without impacting performance and reduces query execution time by up to 70% without impacting cost compared to a centralized approach, the vanilla Spark scheduler, and a bandwidth-aware approach, e.g., Iridium [58]. In addition, the results show that Kimchi can handle dynamics during query execution without additional cost. More importantly, Kimchi allows applications to explore a richer tradeoff space between cost and performance given different data distribution in a multi-cloud environment.

Table 4.2: Feature comparison with state-of-the-art. △ indicates that metric is considered but with limitations.

	Clarinet	Iridium	Tetrium	Kimchi
Heterogeneous network B/W	✓	✓	✓	✓
Heterogeneous network cost				✓
Cost-performance tradeoff		△	△	✓
Handling dynamics			△	✓

4.1.2 Research Contributions

The main contributions in Kimchi are:

• Designing and implementing Kimchi, the first GDA system (to the best of our knowledge) that optimizes task placement with a consideration of heterogeneous data transfer cost ($) in a multi-cloud environment.

• Observing that minimizing data transfer size may not lead to the minimum cost.

• Formulating and solving the cost-aware task placement problem that allows applications to explore this *richer cost-performance tradeoff space*.

• Handling dynamics during query execution for cost-aware performance that avoids expensive re-evaluation of global task placement.

• Applying our solution to a push-based shuffle mechanism that maintains low cost and

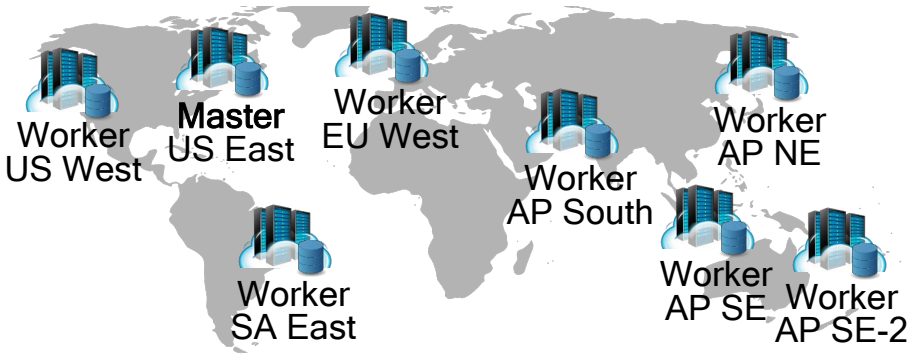


Figure 4.1: An example of geo-distributed DCs where a GDA is running to analyze geo-distributed data.

improves query performance.

Table 4.2 shows the comparison between state-of-the-art solutions and Kimchi. While all prior approaches consider heterogeneous network bandwidth, they do not consider heterogeneous data transfer cost. This network cost-agnostic approach can lead to a cost-bottleneck. For the cost-performance tradeoff, Iridium [58] and Tetrium [66] offered a knob to explore the trade-off space by limiting WAN usage. These systems, however, may not achieve a desired tradeoff due to heterogeneous data transfer cost, i.e., *minimizing data transfer size does not necessarily yield minimized data transfer cost* as we will show. To handle dynamics, Tetrium [66] re-evaluates the global optimized task placement decision. However, it may encounter a cost-bottleneck due to a cost-agnostic approach for handling dynamics. In addition, frequently re-evaluating global task placement can incur performance overhead. We will show how Kimchi handles dynamics while avoiding the cost-bottleneck and performance overhead of re-evaluation in Section 4.4.2.

4.2 System Model and Problem Statement

4.2.1 System Model

Data Center (DC) Setting: We consider geo-distributed data analytics (GDA) running in a multi-cloud environment. Figure 4.1 shows the DC locations where a GDA is running e.g., the master is in US East and the workers are in the other regions. Each

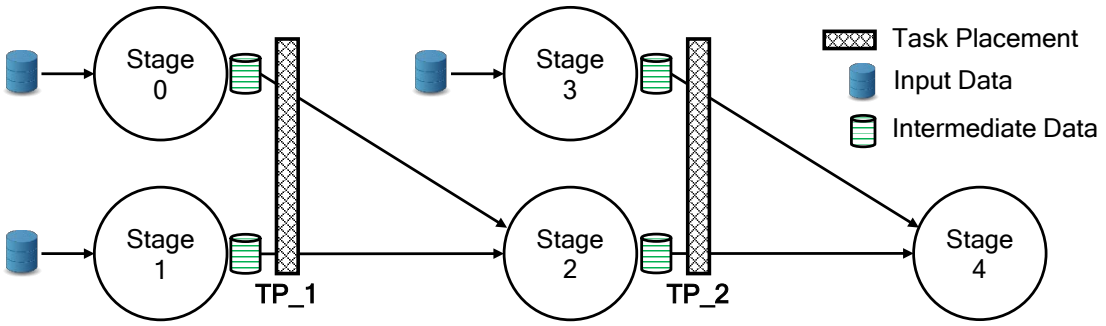


Figure 4.2: A DAG (Directed Acyclic Graph) example for a job with 5 stages.

DC has heterogeneous data transfer cost policies based on geographical locations and providers as shown in Table 4.1. Cloud providers only charge for outbound data transfer while inbound data transfer is free of charge. The network bandwidth between DCs is highly heterogeneous due to different bandwidth capacities and can fluctuate due to dynamics e.g., network contention on shared network links and bandwidth changes by cloud providers. While computational resources in each DC are finite due to a cost constraint and thus can be a performance bottleneck, we attack the WAN bandwidth as this can significantly inflate cost as well as degrade performance for a large class of GDA applications, the latter as noted in many previous works [56, 58, 65, 68, 72].

Applications: Applications generate network bandwidth-intensive MapReduce like queries to a GDA master that will assign tasks to workers that spawn executors to execute tasks. While applications have different cost-performance tradeoff preferences, achieving reduced query latency within their desired target budget is desirable for these applications. We believe that our approach can be applied to any application that needs to transfer large data frequently between DCs, where cost and performance are important.

Queries and Data: Figure 4.2 shows a job (query) example that has several stages, i.e., three map stages (0, 1, and 3) and two shuffle stages (2 and 4). Each stage accesses *geo-distributed* input data, e.g., stage 0 may access input data from all DCs in Figure 4.1. Map tasks access input data locally and output the results (intermediate data) locally. Shuffle tasks access intermediate data from all DCs. Intermediate data can be accessed multiple times for a single query, e.g., self-join operation. We make the

following assumptions.

- Map tasks are executed in-situ with their input data.
- Intermediate data size is large [58, 69, 70].
- The amount of data drops off quickly in subsequent stages for queries that have many sequential stages [58, 77].

Given these assumptions, map stages are not a performance bottleneck due to data locality [40, 78, 79] and in-memory caching [13, 80]. In shuffle stages, however, large intermediate data transfer occurs via all-to-all communication among DCs using WAN bandwidth, the main performance bottleneck in a GDA [55, 56, 57, 58, 64, 65]. For cost, accessing data (input or intermediate) within a DC does not incur cost. Shuffle stages, however, require large intermediate data transfer via WAN. This incurs cost, that can cause a cost-bottleneck. In short, shuffle stages are the bottleneck for both performance and cost and we therefore focus on shuffle stages in this work. While we consider multiple stages, determining optimal task placement for multiple shuffle stages results in a non-convex optimization [55, 58, 66]. Instead we adopt a greedy approach that determines task placement independently for each shuffle stage. This approach may not be optimal for a query but will work well under our assumption: rapid data size reduction in subsequent stages, i.e., a few shuffle stages are significant for overall cost and performance. For data distribution, we will discuss a richer cost-performance tradeoff space with varying data distribution in Section 4.6.5.

Barrier Synchronization: In a MapReduce programming model, stages cannot start until all their dependences are resolved due to barrier synchronization, e.g., stage 2 cannot start until stages 0 and 1 are done. To overcome this limitation, intermediate data can be pushed to target DCs in the background (asynchronously) in map stages, i.e., *push-based shuffle* [64, 73, 74]. We consider these two common GDA implementation models (barrier and push-based) and show how they work in our system in Section 4.4.

Task Placement Problem: The task placement problem consists of determining a set of tuples (tasks and corresponding DC locations) in order to satisfy the application's desired cost-performance tradeoff preference. Task placement is determined for shuffle (or result) stages that need to shuffle intermediate data. For example, there are 2 task placement problems i.e., **TP_1** and **TP_2**, in Figure 4.2. Note, input data is computed

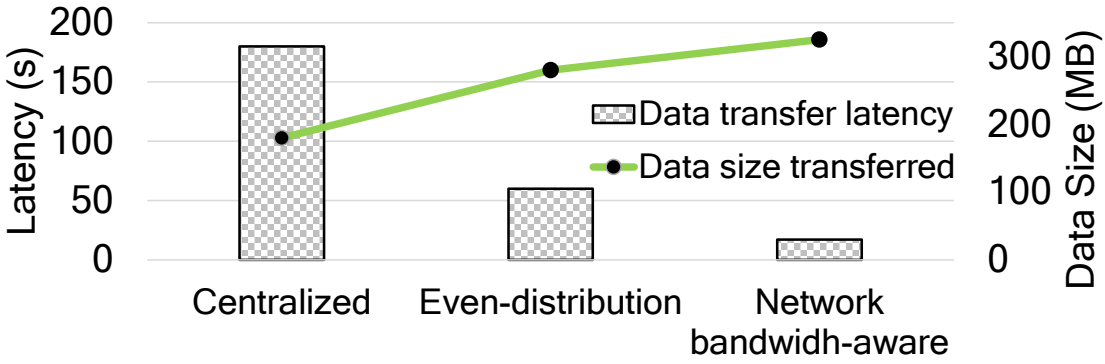


Figure 4.3: Data transfer latency and data size transferred.

in-situ as assumed thus task placement decisions for map stages are not considered in this work.

4.2.2 Illustrative Example

In this section, we illustrate how data transfer cost can affect overall operational cost by applying three different task placement approaches: centralized, even distribution (scheduling for load balancing), and network bandwidth-aware e.g., Iridium [58], to an example GDA scenario. We use the example in Iridium for comparison using the intra-AWS costs as shown in Table 4.1. As an example, consider an application consisting of a MapReduce query to a GDA that must process data contained in three DCs. The network environment is as shown in Table 4.3 in which DC A's downlink is a significant bottleneck in terms of bandwidth.

Table 4.3: 3 DCs Example.

	DC A	DC B	DC C
Intermediate Data	240MB	120MB	60MB
Uplink BW	10MB/s	10MB/s	10MB/s
Downlink BW	1MB/s	10MB/s	10MB/s

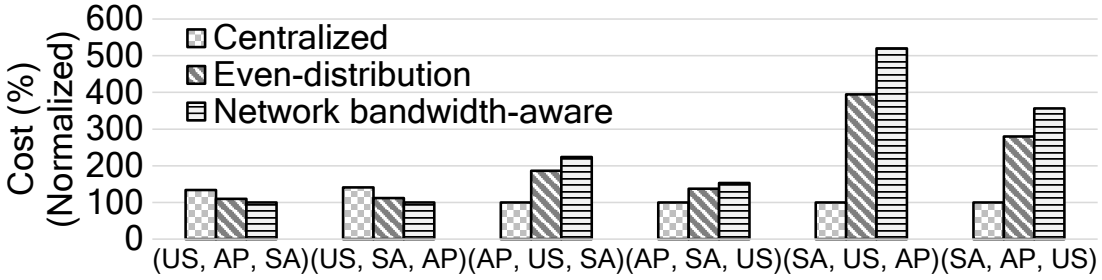


Figure 4.4: Data transfer cost comparison with varying AWS DC locations, US East, AP NE, and SA. Costs are normalized to the minimum cost for each configuration.

Figure 4.3 shows the data transfer latency and data size transferred for each approach. While the centralized approach minimizes data size transferred (180MB: 120MB from B and 60MB from C to A), it increases data transfer latency significantly (180 secs) due to the network bottleneck in DC A (1MB/s downlink). The bottleneck link is avoided in the network bandwidth-aware approach in which tasks are assigned based on given bandwidth, approximately (A: 5% (1MB/s), B: 47.5% (10MB/s), C: 47.5% (10MB/s)) to minimize data transfer latency (17.1 secs) with more data transferred (322.5MB: 9MB from B and C to A (9 secs), 142.5MB from A and C to B (14.2 secs), and 171 MB from A and B to C (17.1 secs)). The even-distribution approach offers performance somewhere between other approaches as shown in Iridium [58].

Heterogeneous data transfer cost: Figure 4.4 shows the cost comparison with varying DC locations. Interestingly, the first two left-most cases, (US, AP, SA) and (US, SA, AP), show that *minimizing data transfer size does not necessarily lead to minimized data transfer cost*. That is, the centralized approach results in 131% ~ 140% cost compared to the network bandwidth-aware approach even with less data transferred, i.e., 180MB (centralized) vs. 322.5MB (bandwidth-aware). This is because a large portion of data needs to be sent from DCs where data transfer cost is expensive i.e., SA and AP NE, in the centralized approach. For these cases, cost can be minimized if the centralized DC is determined based on both cost and data size rather than just data size e.g., choosing AP and SA as the centralized DC for each case respectively can minimize cost, i.e., *cost-aware centralized*. Other cases show that the network bandwidth-aware approach

can significantly increase the cost up to 5.2X, showing that a consideration of both heterogeneous network bandwidth and cost can open up a richer cost-performance tradeoff space. However, this example also shows the optimization problem to be complex.

Dynamics during query execution: For dynamics, assume that DC B's downlink becomes 1MB/s during query execution and the bandwidth-aware approach is used. This increases data transfer latency significantly (142 seconds: 142.5MB from A and C to B at worst case) if dynamics are not handled. To avoid this, tasks need to be re-assigned (A: 8.3% (1MB/s), B: 8.3% (1MB/s), C: 83% (10MB/s)) to minimize performance degradation (299.9MB from A and B to C (30 secs) at worst case). In this case, network cost can be either increased up by 30% or reduced by 13% based on DC locations compared to the original task placement, that opens additional tradeoffs.

4.3 Cost-aware Task Placement

4.3.1 Cost and Performance Tradeoff

In a multi-cloud environment, applications may have different cost-performance tradeoff preferences. For applications that want to have fast query response irrespective of cost, bandwidth-aware approaches e.g., Iridium [58], would be preferable. On the other hand, for applications that want to minimize operational cost, the cost-aware centralized approach e.g., the four right-most cases in Figure 4.4, would be preferable. However, most applications likely want to achieve cost and performance somewhere between these two approaches.

Figure 4.5 shows extreme possibilities of two approaches in terms of cost, i.e., cost-aware centralized for *Min cost* and bandwidth-aware for *Best performance*. Applications cannot reduce cost below Min cost and cannot improve performance above Best performance even by paying more than the *Max cost* that is associated with *Best performance*. The figure also shows a tradeoff space between the two extremes. Our goal is to allow applications to explore this tradeoff space by providing their desired tradeoff preference (C_pref), on the continuum between the two extreme cases.

To this end, we determine feasible cost boundaries or ranges i.e., Min cost and Max cost, to estimate a target cost budget that corresponds to the desired cost-performance tradeoff preference C_pref. With estimated Min and Max costs, the target budget can

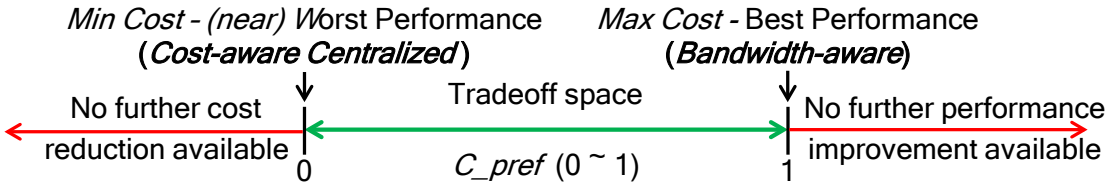


Figure 4.5: Possible tradeoff space between two extremes (centralized and bandwidth-aware approaches) in terms of cost.

be estimated as follows.

$$target\_budget = Min\ Cost + C\_pref \cdot (Max\ Cost - Min\ Cost) \quad (1)$$

For example, if Min cost ($100) and Max cost ($500) are estimated with given inputs, and C_pref is set to 0.5, $300 is used as a target budget. The Min cost can be estimated with a task placement that minimizes cost without considering network bandwidth (performance) and the Max cost can be estimated with a task placement that minimizes query execution time without considering cost.

4.3.2 Task Placement Problem Formulation

Given a target_budget, we formulate the task placement problem as a *budget-constrained optimization problem* that outputs a desired task placement consisting of a list of tuples {task, DC-location}. We use mixed integer programming (MIP) to optimally solve this problem.

Inputs and Outputs

Table 4.4 shows the inputs to the our model. Note, the inputs, NB, T, and I are continuously updated for each stage.

Tradeoff Preference: Applications need to provide their cost-performance tradeoff preference, C_pref. Applications can set it to 0 for minimized cost (cost-aware centralized approach), 1 for minimized latency (bandwidth-aware approach), or any number between 0 and 1 to specify a cost-performance tradeoff preference. C_pref will be converted to the target cost budget that is used as a cost constraint. This is the only input

Table 4.4: Inputs for task placement.

Input	Description
C_pref	Desired cost preference (0 ~ 1)
D	Set of DCs
C_{ij}	Data transfer cost from DC i to DC j
NB_{ij}	Network Bandwidth (MB/s) from DC i to DC j
T	Set of Tasks
I_{ij}	Intermediate Data size for shuffle task i in DC j

that applications need to provide for task placement.

Data Transfer Cost: Data transfer cost information is available from cloud providers web pages [81, 82] and is changed rarely (static in this work).

Network Bandwidth Information: The latest information for inter-DC bandwidth (bytes/s) is estimated by executors running on each DC when they transfer data between DCs.

Data Size for Shuffle Tasks: Input (intermediate) data size for all shuffle tasks stored in each DC is required. This information is available from the MapOutputTracker in a GDA.

Output: Given these inputs, we compute task placement consisting of set of pairs ({task, DC}). Output includes expected latency and network cost for each task. We will show how these values are used by a scheduler to handle dynamics during query execution in Section 4.4.2.

Optimization Problem Formulation

We solve three optimization problems to determine the task placement.

Determining the Target Budget: As shown in Equation 1, to determine the target budget, we solve two sub-problems to get *Min cost* and *Max cost*. The following variable and constraint (Equation 2) are used in this formulation:

$$\forall i \in T, \forall j \in D : A_{ij}$$

A_{ij} are binary variables (0 or 1): if 1, task i is assigned to DC j.

$$\forall t \in T, \forall i \in D : \sum_{i} A_{ti} \ = 1 \tag{2}$$

This is a constraint that a task can be assigned only to a single DC.

Min cost: The first sub-problem is to determine the lowest data transfer cost (lower cost bound).

Objective: Minimize total data transfer cost.

$$\forall t \in T, \forall i, j \in D : \sum_{t} A_{ti} \cdot I_{tj} \cdot C_{ji} \tag{3}$$

We found that all tasks are assigned to a single DC to minimize cost in the task placement by solving this sub-problem. The Min cost can be computed by using cost information and data size of each DC.

Max cost: The second sub-problem is to determine the upper cost bound for the lowest data transfer latency.

Objective: Minimize maximum data transfer latency.

$$\forall t \in T, \forall i, j \in D : Max(\frac{\sum_{t} A_{ti} \cdot I_{tj}}{NB_{ji}}) \tag{4}$$

In Equation 4, *we only consider the maximum data transfer latency between DCs i.e., the main bottleneck, that determines overall latency (performance).* With the task placement for the minimized network latency (best performance), the upper-bound data transfer cost (maximum cost) can be computed with given cost information. Finally, we can determine the *target_budget* with Min and Max cost with C_pref as shown in Equation 1.

Solving Task Placement with a Target Budget: Once the *target_budget* is determined, we solve the minimum data transfer latency problem (Equation 4) again with a *target_budget* as a constraint.

Objective: Minimize maximum latency, Equation 4 with the following constraint:

$$\forall t \in T, \forall i, j \in D : \sum_{t} A_{ti} \cdot I_{tj} \cdot C_{ji} <= target\_budget \tag{5}$$

By solving Equation 4 with Equation 5 as a constraint, *we minimize the highest data transfer latency i.e., query execution time, given target_budget.*

For the *target_budget*, we provide a tool that uses Equations 3 and 4 for applications to estimate the cost boundaries i.e., Min cost and Max cost, by providing inputs i.e., expected data size and locations before sending a query. Given cost boundaries, applications can set the desired target cost budget directly without setting C_pref.

4.4 Cost-aware Task Scheduling

In this Section, we will present how Kimchi uses cost-aware task placement to help applications to tradeoff between cost and performance. We will also show how Kimchi handles dynamics using a heuristic algorithm for cost-aware performance. Lastly, we will present how Kimchi determines task placement by a cost-aware push-based mechanism that utilizes WAN bandwidth efficiently.

4.4.1 Task Scheduling

When a shuffle stage is ready to be executed, Kimchi estimates a task placement for the stage at *query execution time* (run-time) by solving a constrained MIP task placement problem as explained in Section 4.3. Note, Kimchi can get the latest inputs including exact intermediate data size and locations, due to a barrier synchronization. Once a task placement is determined, the Kimchi scheduler starts scheduling tasks as determined by the task placement optimization that meets cost-performance preferences. Note, we only consider shuffle stages for cost-aware scheduling in this work and tasks of map stages are scheduled using data locality-aware scheduling, e.g., Hadoop [59] and Spark [13].

Algorithm 1 outlines how task placement is performed by the Kimchi scheduler. The task scheduling function will be called repeatedly whenever the scheduler receives a DC offer to execute one of the tasks in a stage (line 1). The scheduler finds a task for the DC from the task placement list and assigns the task to the DC after checking dynamics for the DC (line 2 $\sim$ 5). If dynamics are detected, the scheduler finds another DC among the idle DCs for the task to avoid performance degradation (line 7). An idle DC is defined as one for which there are no tasks remaining in the current stage (line 9). We will show how Kimchi detects and handles dynamics in the next section.

Algorithm 1 Cost-aware Task Scheduling

```
1: procedure CostAwareScheduling(taskPlacement tp, dc d, adjustOption o, dcs
   [] idle)
2:     t_id = DequeueTask (tp, d)
3:     if t_id >= 0 then
4:         if CheckDynamics (tp, d, t_id) == false then
5:             ExecuteTask(t_id, d)
6:         else
7:             AdjustTask(d, t_id, o, idle)
8:     else
9:         idle += d
```

4.4.2 Cost-aware Task Adjustment

A static optimal task placement may be sub-optimal due to mis-estimated bandwidth or may become sub-optimal due to dynamics that are common in a multi-cloud environment [68, 71, 72], e.g., network contention and bandwidth changes. Since a sub-optimal task placement can affect overall cost and performance, it needs to be handled. While most GDA systems [55, 56, 57, 58, 65] ignored dynamics during query execution, Tetrium [66] re-evaluated task placement to handle dynamics at run-time. However, Tetrium may encounter a cost-bottleneck due to its cost-agnostic approach for handling dynamics that may requires large data transfer. For performance, re-evaluating an optimal task placement can incur additional overhead, especially for large-scale jobs. In addition, frequent re-evaluation may occur if dynamics are frequent and/or the sensitivity trigger for re-evaluation is improperly tuned, leading to significant performance overhead.

To avoid the cost-bottleneck and performance overhead of frequent re-evaluation, Kimchi uses a heuristic algorithm that re-assigns tasks to other idle DCs in the presence of dynamics if doing so does not incur additional cost. The intuition behind this heuristic is that dynamics will lead to congestion at one or more DCs, leaving others, to complete their tasks much more quickly, and become idle. Kimchi utilizes these idle DCs with an expectation that the latency for the task can be amortized by running other tasks in parallel. That is, Kimchi tries to utilize bandwidth completely to improve performance. Note, idle DCs are tracked by Algorithm 1 and treated as idle only

Algorithm 2 Checking and Handling Dynamics

```
1:  procedure CheckDynamics(taskPlacement tp, dc d, taskId t_id)
2:      curLat = GetLatForTask(t_id, d)
3:      expectedLat = tp[t_id].expectedLat
4:      if curLat > expectedLat + LatThreshold then
5:          return true
6:      return false
7:  procedure AdjustTask(dc d, taskId t_id, dcs [] idle, adjustOption o)
8:      chosen_d = d
9:      candidates = []
10:     for each idle_d in idle do
11:         if CompareLat(idle_d, d, t_id) <= 0 then
12:             if o.always_adjust_tasks == false or
                CompareCost(idle_d, d, t_id) <= 0 then
13:                 candidates += idle_d
14:     if candidates.size > 0 then
15:         chosen_d = Choose(candidates, t_id, o.pref)
16:     ExecuteTask(t_id, chosen_d)
```

within the current stage. To avoid additional cost by task adjustment, Kimchi adjusts task placement such that they do not incur additional cost, i.e., cost-aware task adjustment. For better performance, Kimchi offers a scheduling option *always_adjust_tasks* that allows the scheduler adjusts tasks without cost-awareness, i.e., cost-agnostic task adjustment.

Algorithm 2 outlines how Kimchi detects and handles sub-optimal task placement due to dynamics. The Kimchi scheduler estimates a latency for a task using the latest bandwidth information before it assigns the task to the designated DC (line 2). The scheduler compares the estimated latency with the expected latency computed by the optimizer as in Section 4.3.2. If the difference is larger than a threshold (*LatThreshold*), Kimchi concludes that dynamics have occurred (lines 3 ~ 5) and tries to assign the task to another DC if doing so improves performance (line 7). To pick a new DC location that best meets application preferences e.g., cost or performance, Kimchi traverses idle DCs and executes the task on the chosen one. If no DC is chosen, the task can still be assigned to the designated DC (lines 10 ~ 16).

A low *LatThreshold* value will trigger re-scheduling and additional scheduling overhead to search for idle DCs. But this heuristic is fast and the search overhead is much lower than re-evaluating global task placement, hence we set the threshold to a small value (3 seconds by default). We will show how cost-aware task adjustment helps Kimchi remedy the sub-optimal task placement for better cost-aware performance in the presence of dynamics in Section 4.6.3.

4.4.3 Cost-aware Push-based Shuffle

Previous works [64, 73, 74] showed that a *push-based shuffle* mechanism can improve performance by pushing intermediate data in the background (asynchronously) in map stages. These systems, however, may encounter a cost-bottleneck because they did not consider data transfer cost heterogeneity to determine the target DCs to push intermediate data.

To improve performance while avoiding the cost-bottleneck, Kimchi adapts the push-based shuffle to use a cost-aware task placement. To push intermediate data, Kimchi needs to determine child stages' task placement in parent stages (including map stages) with imperfect input data information. For example (Figure 4.2), **TP_1** needs to be evaluated before stages 0 and 1 start. To estimate intermediate data sizes, Kimchi relies on input data information of all parent stages e.g., input information of stages 0 and 1, with an assumption that intermediate data size is *proportional* to input data size at a configurable rate (R), as done in previous works [57, 58, 64, 74]. While this assumption may not always be accurate, this simple approach yields cost and performance benefits as we will show in Section 4.6.4. With predicted intermediate data information and other inputs, Kimchi can estimate task placement for a child stage, e.g., **TP_1** for stages 0 and 1.

Algorithm 3 outlines how push-based shuffle works in Kimchi. Before Kimchi executes a stage, Kimchi determines task placement for a child stage (line 2). If the stage has a child stage, Kimchi checks if a task placement for a child stage is available (lines 10 $\sim$ 12). If there is no task placement for the child stage, Kimchi collects the MIP inputs, i.e., *expected* intermediate data sizes, tradeoff preference, and others as before, and runs the optimization (lines 13 and 14). Task placement for the child stage is stored for later use (lines 6, 15, and 17). The tasks in given stage are scheduled based on stage

Algorithm 3 Cost-aware Push-based Shuffle

```
1:  procedure ExecuteStage(stage s)
2:      c_tp = GetChildStageTaskPlacement(s)
3:      if s == MapStage then
4:          LocalityAwarePushBasedScheduling(c_tp)
5:      else
6:          tp = GetTaskPlacement(s)
7:          CostAwarePushBasedScheduling(tp, c_tp)
8:  procedure GetChildStageTaskPlacement(stage s)
9:      c_tp = []
10:     if hasChildStage(s) then
11:         c_s = getChildStage(s)
12:         if IsTaskPlacementAvailable(c_s) == false then
13:             inputs = GetInputs(s, c_s)
14:             c_tp = Evaluate(inputs)
15:             SetTaskPlacement(c_s, c_tp)
16:         else
17:             c_tp = GetTaskPlacement(c_s)
18:     return c_tp
```

type, i.e., data locality-aware scheduling for map stages and cost-aware scheduling for shuffle stages. In either case, the task placement for a child stage is used for pushing intermediate data in parent stages (lines 3 ~ 7). By using the push-based mechanism, both cost reduction and better performance can be achieved. That is, the pushed intermediate data can serve as a *cache* that prevents repetitive remote access to the same intermediate data in a query, e.g., self-join.

Note, activating both the task adjustment (Section 4.4.2) and the push-based shuffle mechanism can lead to cost inflation due to duplicated intermediate data transfer, i.e., intermediate data pushed may not be accessed if a task is re-assigned. In this work, we adjust tasks only when the push-based shuffle is not activated to avoid cost inflation. We plan to explore both options as future work for better cost-aware performance.

4.5 Prototype Implementation

We have implemented a Kimchi prototype on Apache Spark (2.2.1)[13]. Thus, applications can submit jobs through the same interfaces provided by Spark to benefit from

Table 4.5: Kimchi (Spark) property examples

Property Name	Description
spark.kimchi.taskScheduling	Using Kimchi (True or False)
spark.kimchi.costPreference	Desired cost preference (0 ~ 1)
spark.kimchi.adjustTask	Adjusting task placement (True or False)
spark.kimchi.pushShuffling	Using push-based shuffle (True or False)

Kimchi *without modification*. Table 4.5 shows properties that applications can set to utilize Kimchi easily.

In our prototype implementation, the task placement optimizer API is implemented with Thrift [32], a remote procedure call framework. The optimizer is written in Python (~ 700 lines of code) and receives inputs in a JSON format [83]. Kimchi uses PuLP [50] to model a task placement problem as a mixed integer programming (MIP) and uses CPLEX [51] to solve the optimization problem. The scheduler sends inputs via an API for estimating an optimized task data placement.

For network bandwidth and cost information, we modified the executor to estimate network bandwidth while it fetches data from other DCs and to track the number of remote bytes read by each executor for each origin and destination. This aggregated executor information is sent to the scheduler by piggybacking on heartbeat messages and used as input to solve the placement problem. We also modified the executor to use Wiera (Chapter 2), a geo-distributed policy driven storage system, to push and fetch intermediate data as the Spark API does not fully support functions for the push-based shuffle.

4.6 Evaluation

Experimental Setting: We deployed and evaluated Kimchi across 8 AWS regions (outbound network cost per GB): US East-Virginia ($0.02), US West-California ($0.02), EU West-Ireland ($0.02), AP SE-Singapore ($0.09), AP SE-2-Sydney ($0.14), AP North-Tokyo ($0.09), AP South-Mumbai ($0.086), and SA East-Sao Paulo ($0.16). We used AWS t2.medium (2 vCPU cores and 4GB of RAM) for workers and AWS t2.large (2

vCPU cores and 8GB of RAM) for both the Spark master and the Spark driver for job submissions. For workload, we used TPC-DS [75] a standard decision support benchmark, and TeraSort [76] a standard sorting benchmark, to benchmark the performance of GDA systems. For input data, 40GB data is evenly distributed and used for TPC-DS queries that produce large intermediate data transfers in shuffle stages. We use 10GB input data for TeraSort. While 10GB is relatively small for a GDA, TeraSort produces large intermediate data that is sufficient to show the availability of a richer tradeoff space between cost and performance for different data distributions. We use the Hadoop Distributed File System (HDFS) [84] as an underlying storage system to fetch input data. We use Wiera (Chapter 2) to push and fetch intermediate data only when the push-based shuffle is activated (Section 4.6.4).

We use different baseline approaches: 1) data locality-aware (vanilla Spark), 2) centralized (minimized network usage), and 3) bandwidth-aware, e.g., Iridium. The centralized approach minimizes network usage without considering bandwidth and cost heterogeneity in which the centralized DC is the one that has the largest portion of intermediate data. The Iridium approach is equivalent to the $C\_pref = 1$ without task adjustment, which does not consider data transfer cost but only heterogeneous bandwidth for performance reasons.

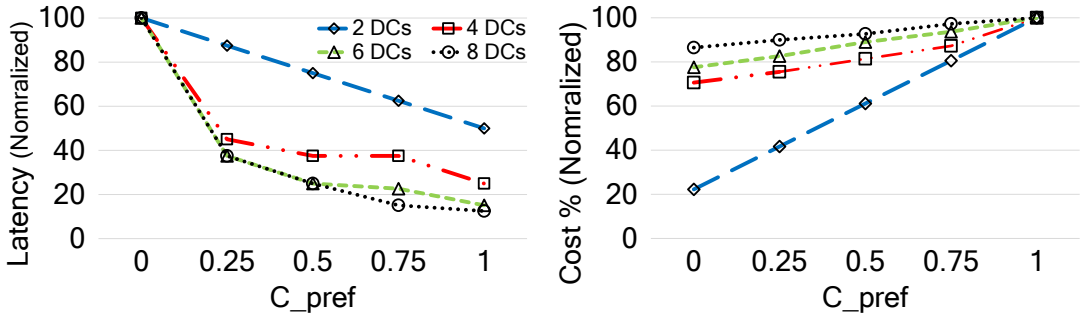
All experimental results are an average of 10 runs, plotted with 95% confidence intervals. The option to handle dynamics is deactivated if not mentioned for comparison purpose.

Overhead of Solving The Task Placement Problem: Though MIP is not efficient, the time for computing task placement has a negligible impact on the overall performance as Kimchi can solve the optimization problem for each stage approximately in 2 seconds with t2.medium (2 vCPU 4 GB of RAM) for our experimental setting of 8 DC locations.

4.6.1 Illustrating Cost-performance Tradeoff

In this section, we show the cost-performance tradeoff space in terms of cost and performance with a varying number of DCs and their locations using a simple synthetic workload e.g., wordcount and sort, that has 2 stages including 1 shuffle stage.

In this experiment, we use 4 different DC configurations: 1) 2 DCs in US East and



(a) Latency comparison.

(b) Cost comparison.

Figure 4.6: The highest data transfer latency of tasks in a shuffle stage and data transfer cost for each C_pref and DC configuration. Values of Figure 4.6(a) and Figure 4.6(b) are normalized to $C\_pref = 0$ case and $C\_pref = 1$ case respectively.

SA East, 2) 4 DCs in US East, US West, AP SE-2, and SA East, 3) 6 DCs in US East, US West, EU West, AP SE, AP SE-2, SA East, and 4) all 8 DCs. We assume that the intermediate data is evenly distributed and use measured network bandwidth between DCs.

Figure 4.6(a) shows the highest data transfer latency of tasks in the shuffle stage (normalized to the maximum latency) for each DC configuration. Not surprisingly, $C\_pref = 0$ case provides the highest latency (worst performance) and $C\_pref = 1$ case provides the lowest latency (best performance) for all configurations. Results show that if C_pref is increasing, the highest latency is decreasing and thus performance is improving. While the latency decreases smoothly from $C\_pref = 0.25$ to $C\_pref = 1$, $C\_pref = 0$ case shows steep latency increase as all data is sent to a single DC (or a few DCs), i.e., network contention. The results show that Kimchi can control overall query performance by adjusting the highest data transfer latency.

Figure 4.6(b) shows that cost is increased as C_pref is increased, i.e., performance improvement with additional cost. The results also show a trend that the available cost reduction range decreases as the number of DCs increases. For example, DC 2 case has 77.5% cost reduction opportunity but DC 8 case only has 13.5%. This is because the cost variance is getting smaller as the number of DCs increases in our DC configurations i.e., 2 DCs case for the biggest cost variance and 8 DCs case for the smallest cost variance, and we consider even data distribution. We predict even

greater cost reduction opportunities with more complex cost policies e.g., both AWS and Azure, but omitted for space reasons. We will show how data distribution can affect cost reduction opportunities in Section 4.6.5.

Table 4.6: DCs to execute tasks and number of tasks for the DCs with a 8 DCs configuration.

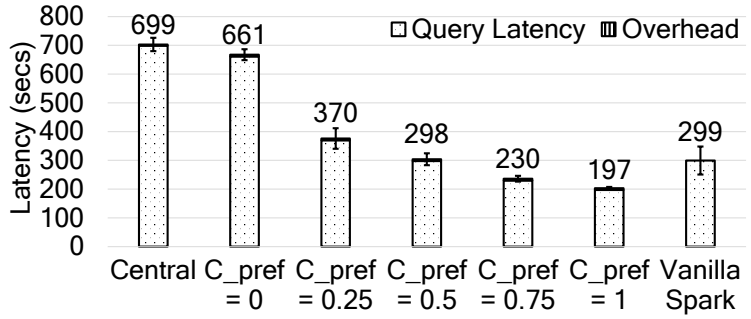
C_pref	DC selections (the number of tasks for the DC)
1	US East (1), US West (1), EU West (1), SA East (1), AP SE (1), AP SE-2 (1), AP NE (1), AP South (1)
0.75	US West (1), AP SE (1), AP SE-2 (2), AP NE (2), AP South (1), SA East (1)
0.5	SA East (2), AP SE (2), AP SE-2 (2), AP NE (2)
0.25	SA East (3), AP SE (1), AP SE-2 (4)
0	SA East (8)

Table 4.6 shows which DCs are chosen for tasks for each C_pref value with a 8 DCs configuration. For example, Kimchi assigns tasks to all DCs if C_pref is set to 1 and all tasks are assigned to a single DC (SA East) if C_pref is set to 0. From the table, we can see that *Tasks are assigned to DCs that have expensive outbound data transfer cost to reduce cost for a small* C_pref *value by avoiding data transfer out from those DCs.* For example, tasks are mainly assigned to SA East, AP SE, and AP SE-2 with small C_pref values.

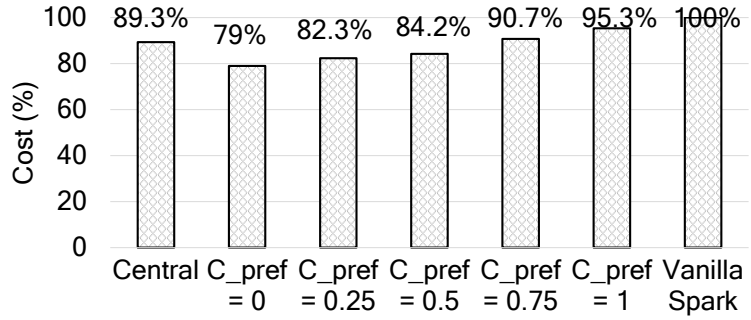
Overall, these results show that Kimchi can determine task placement to meet a desired cost-performance tradeoff by adjusting the highest data transfer latency, i.e., minimizing the highest data transfer latency (best performance) with a cost constraint.

4.6.2 Cost and Performance Comparison

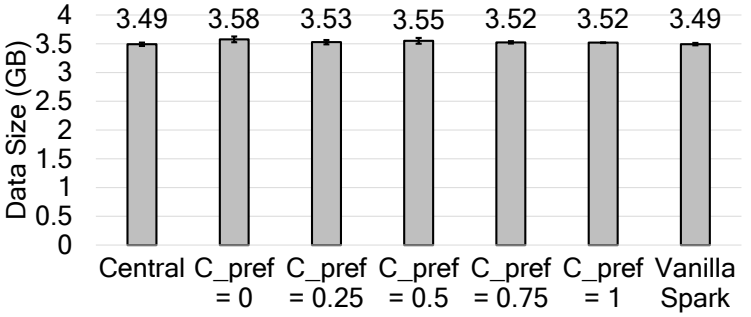
In this experiment, we show the benefit of the cost-aware task placement for task scheduling. We compare the query execution time and data transfer cost using different baseline approaches, data locality-aware (vanilla Spark), centralized (minimized network usage), bandwidth-aware (Iridium), and our approach with varying C_pref ($0 \sim 1$) for one of the TPC-DS queries (query 95), consisting of several shuffle stages



(a) Query latency comparison.



(b) Cost comparison.



(c) Data transfer size comparison.

Figure 4.7: Query latency, cost, and data transfer size comparisons. Costs are normalized to the vanilla Spark case.

with large intermediate data, i.e., a bandwidth-intensive workload. Note, the $C\_pref = 1$ case is the Iridium approach as it considered only map stages for cost-performance tradeoff. The $C\_pref = 0$ case can be considered another centralized approach as all tasks are assigned to a single DC, but it considers cost heterogeneity, i.e., cost-aware centralized.

Figure 4.7(a) shows the query execution time for the TPC-DS query. The results show that the $C\_pref = 1$ case provides the best performance due to network bandwidth-aware task placement. Figure 4.7(b) shows the cost comparison. Interestingly, the $C\_pref = 1$ case provides slightly cheaper (5%) cost compared to the vanilla Spark case even with better performance because *Max cost* is fixed as a cost constraint in $C\_pref = 1$, i.e., cost-aware scheduling in Kimchi. Note, we see a large performance variance from the vanilla Spark case, but with higher cost, due to its data locality-aware scheduling that does not work well in terms of both performance and cost in a multi-cloud environment. The $C\_pref = 0.5$ case shows a 15% cost reduction without impacting performance compared to the vanilla Spark case. For the centralized approaches both minimized data transfer and the $C\_pref = 0$ case, the query latency steeply increases due to network contention as a result of data being sent to a single centralized DC. The $C\_pref = 0$ case reduces cost by 10% compared to the centralized approach due to cost-aware scheduling. Lastly, the results show that the query latency decreases as cost increases and this agrees with our results shown in Section 4.6.1.

Figure 4.7(c) shows data transfer size for each approach. Interestingly, we cannot see any clear relationship with cost. However, the results clearly show that *minimizing data transfer size does not necessarily lead to cost reduction.* That is, minimized network usage does not provide the minimum cost among approaches even with the minimum data transfer size. Instead, the $C\_pref = 0$ case provides the minimum cost among the approaches even with the largest amount data transfer. In addition, the data size difference between the $C\_pref = 0$ and centralized is only 2.5% but the cost difference is 21%. This confirms that the cost heterogeneity can significantly affect overall cost. The vanilla Spark case shows a similar data transfer size to the minimized network approach due to its data locality-awareness but with 10% more cost. These results agree with the results in Section 4.2.2.

Overall, this experimental results shows that *Kimchi can reduce query execution*

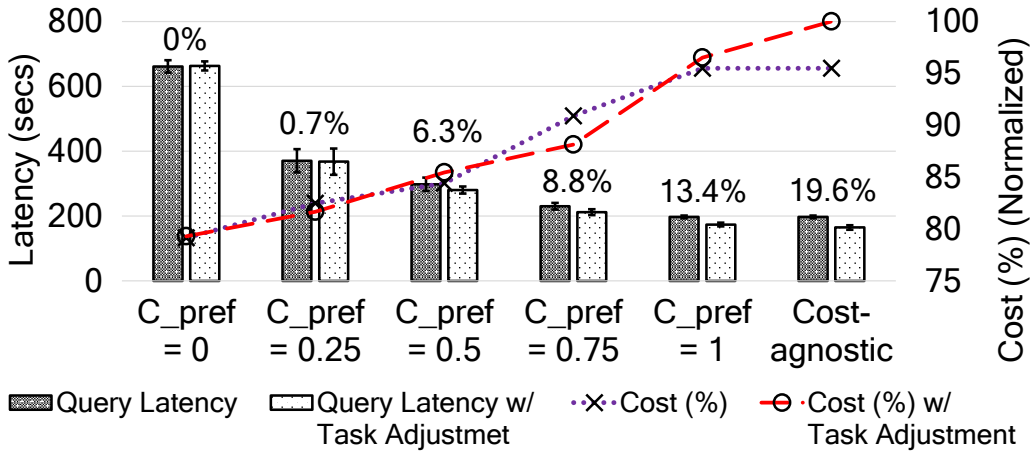


Figure 4.8: Cost and performance comparison by adjusting task placement. The percentage numbers show the performance improvement.

time without impacting cost compared to other approaches and can allow applications to explore richer cost-performance tradeoff options.

4.6.3 Cost-aware Task Adjustment

As explained in Section 4.4.2, the task placement for a stage may be sub-optimal due to mis-estimated bandwidth or become sub-optimal during query execution due to dynamics e.g., network contention and bandwidth fluctuation. In this section, we show benefits of cost-aware task adjustment.

We use the same setting and environment used in Section 4.6.2, but we activate the option to adjust task with cost-awareness to change task placement at query execution time. With this option, tasks will be re-assigned only if doing so does not incur additional cost, i.e., cost-aware task adjustment. Additionally, we set *always_adjust_tasks* option to be activated to allow the scheduler to re-assign tasks without a consideration of cost for the $C\_pref = 1$ case, i.e., cost-agnostic task adjustment. To be responsive to dynamics, we set the latency threshold explained in Section 4.4.2 to 1 second.

Task Adjustment at Run-time

Figure 4.8 shows the cost and performance comparison with the results of Section 4.6.2 for each C_pref value. In terms of performance, adjusting task placement helps to reduce query execution time up to 19.6% compared to not adjusting task placement. The results show that adjusting task placement improves performance for large C_pref values, e.g., 1 and 0.75. But it does not improve performance for small C_pref values, e.g., 0.25 and 0. This is because, tasks are assigned to a few specific DCs to minimize cost with a small C_pref value as shown in Section 4.6.1. Thus, the scheduler has less opportunity to find DCs that offer the same or cheaper cost for re-assigning tasks. For example, if C_pref value is set to 0, tasks cannot be re-assigned even if there are available (idle) DCs because none of DCs can provide cheaper cost.

For cost, little difference was observed because tasks are re-assigned only when doing so does not incur additional cost, i.e., cost-aware task adjustment. However, for the $C\_pref = 1$ case with cost-agnostic task adjustment, the cost increases by 5% for 6.2% performance improvement. This is because task re-assignment does not consider cost but only network bandwidth, i.e., tasks are always re-assigned to idle DCs as long as performance improvement is expected.

Comparison with baselines: The $C\_pref = 1$ with cost-agnostic task adjustment case incurs similar cost to the vanilla Spark case but it provides 45% performance improvement. The $C\_pref = 0.75$ case provides 70% performance improvement without impacting cost compared to the centralized case. The $C\_pref = 1$ with cost-aware task adjustment case improves performance by 13.4% without additional cost compared to the static approach e.g., Iridium.

Handling Network Bandwidth Change

In this section, we show the benefit provided by Kimchi in the face of significant dynamics during query execution. While the same experiment setting is used, we randomly throttle one of the links between DCs (3 Mbit/s) using Linux Traffic Control [85] during query execution. Note, we could observe lower network bandwidth than 3 Mbit/s during our experiment even between AWS DCs, e.g., from SA East to AP SE-2.

Figure 4.9 shows the cost and performance comparison with a static approach (the

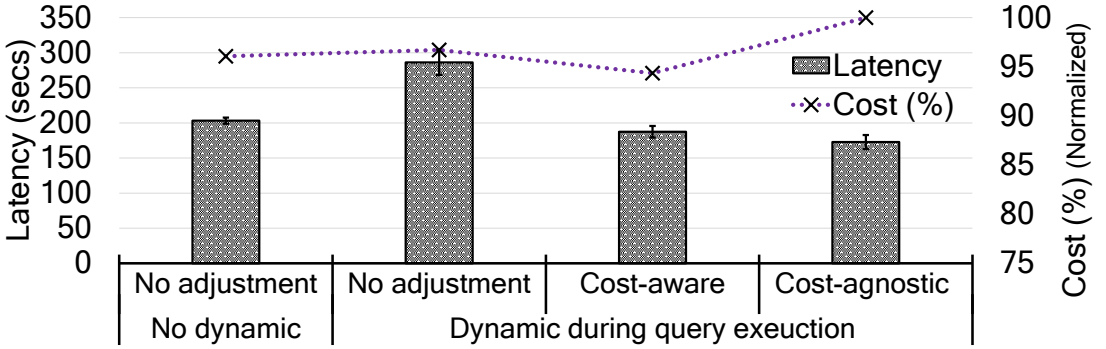


Figure 4.9: Cost and performance comparison when one of DCs becomes a bottleneck. For all cases, C_pref is set to 1.

two left-most cases in the figure), e.g., Iridium, and our non-static approach, both cost-aware and cost-agnostic task adjustments. In terms of performance, the figure shows that the injected network throttling causes 41% additional latency in the static approach as it does not consider dynamics during query execution. The cost-aware task adjustment case shows that it can improve performance by 35% compared to static approach in the presence of dynamics. In addition, the cost-agnostic task adjustment case can further improve performance (40%) with additional cost. Interestingly, the cost-aware case task adjustment provides better performance than the static approach without dynamic. This is because Kimchi uses network efficiently by adjusting tasks as shown in the previous experiment (Section 4.6.3).

For cost, the static approach shows a similar cost regardless of dynamics as expected. That is, the tasks are assigned as specified in task placement even in the presence of dynamics. The cost-aware case reduces cost by 2% compared to the static case. This is because tasks are re-assigned to other DCs only when by doing so improves performance without additional (or with less) cost in the cost-aware case. Cost-agnostic case improves performance by 8% at 5% additional cost compared to the cost-aware case.

Results show that any single link that becomes a bottleneck during query execution can significantly affect overall performance, thus needs to be considered. Kimchi can adjust task placement to handle dynamics e.g., network contention and network fluctuation during query execution time to achieve better cost-aware performance.

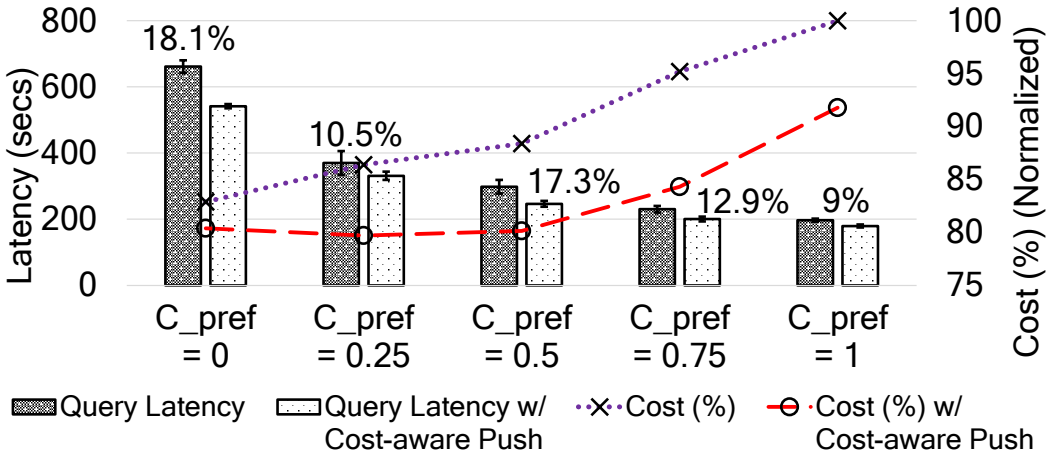


Figure 4.10: Cost and performance comparison using push-based shuffle. The percentage numbers show the performance improvement.

4.6.4 Cost-aware Push-based Shuffle

In this section, we show benefits of cost-aware push-based shuffle in terms of both cost and performance as explained in Section 4.4.3. In this experiment, we use the same setting and environment used in Section 4.6.2, but we set the push-based shuffle to be activated. We set the R value explained in Section 4.4.3 to 0.8, e.g., 8GB intermediate data will be given with 10GB input data. Kimchi estimates task placement of child stages with this *expected* intermediate data size and locations before executing parent stages.

Figure 4.10 shows the cost and performance comparison with the results of Section 4.6.2 for each C_pref value. In terms of performance, push-based shuffle reduces query execution time by 9% ~ 18.1%. The results show that we could get moderate performance improvement even with a simple assumption (fixed R). We believe even further performance improvement is possible with more precise estimation using historical information and recent machine learning techniques.

For cost, we would expect to see similar cost regardless of using push-based or barrier-based shuffle. Yet, the figure shows that the cost can be reduced by 2.5% ~ 10.9%. This is because the same intermediate data is accessed several times within a query, e.g., self-join. By using a pushed-based mechanism, repetitive remote access can be avoided, that leads to both cost reduction and performance improvement as

Table 4.7: Data and cost saving by accessing local pushed intermediate data.

C_pref	1	0.75	0.5	0.25	0
Data Saving	192MB	249MB	257MB	290MB	379MB
Cost Saving	8.2%	10.9%	8.3%	6.7%	2.5%

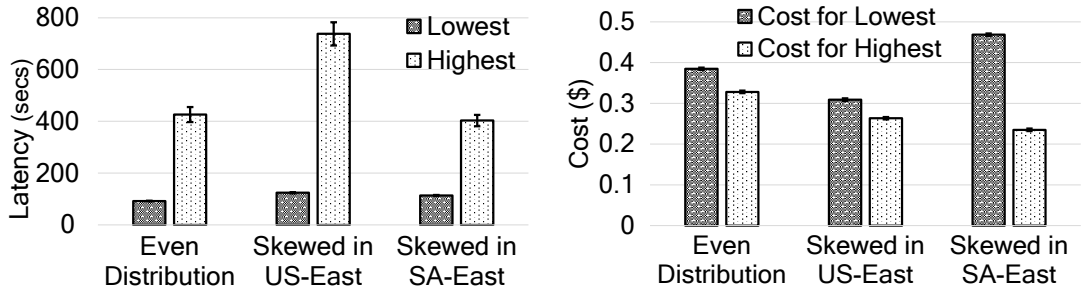
presented in Section 4.4.3.

Table 4.7 shows data savings by accessing pushed intermediate data locally and its corresponding cost saving. The table shows a trend that saved data transfer size increases as C_pref value approaches 0. This is because a small number of DCs are chosen for shuffle stages with a small C_pref value, and this leads to a greater amount of intermediate data fetched remotely. The table also shows that more data saving does not necessarily lead to more cost saving. This is because DCs chosen with a small C_pref value have greater opportunity to fetch data from DCs that have cheaper data transfer cost.

Comparison with baselines: The result shows that the $C\_pref = 0.5$ with push-based shuffle case reduces cost by 23.6% without impacting query performance compared to the vanilla Spark case. The $C\_pref = 0$ case shows 14% cost reduction without impacting performance compared to the centralized approach. The $C\_pref = 1$ with cost-aware push case improves performance by 9% compared to the $C\_pref = 1$ without push case with reduced cost (8%).

4.6.5 Impact of Varying Data Distribution

In this section, we show the tradeoffs between cost and performance based on different data distributions of intermediate data. We use the same setting and environment as in previous sections. For simplicity, we use TeraSort to clearly show the tradeoffs for a single shuffle stage. For input data, we use 10GB and we distribute data in three different ways: 1) even distribution, 2) 1/3 data stored in US East and 2/3 data evenly distributed in the rest of DCs, and 3) 1/3 data stored in SA East and 2/3 data evenly distributed in the rest of DCs. Note, US East has the cheapest data transfer cost ($0.02/GB) while SA East has the most expensive data transfer cost ($0.16/GB).



(a) Latency range.

(b) Cost range.

Figure 4.11: The lowest and highest latencies and cost for each latency with varying data distribution.

Figure 4.11(a) shows the lowest latency ($C\_pref = 1$ with a cost-agnostic task adjustment case) and the highest latency ($C\_pref = 0$ case), and Figure 4.11(b) shows corresponding cost for each latency with different data distribution. Note, we could see similar pattern of results shown in TPC-DS with different C_pref values and options in this experiment, but omitted for space reasons. For the first case (even distribution), we can see 78.5% performance improvement (lowest latency/highest latency) and 15% cost reduction (lowest cost/highest cost) opportunities and this agrees with our results shown in Section 4.6.1. For the second case (1/3 input data in US East), skewed data increases the highest latency significantly, which opens up a 83.2% performance improvement opportunity without reducing the cost reduction opportunity (15%). This is because a large portion of intermediate data in US-East ($0.02/GB) needs to be sent to another centralized DC that has a more expensive network cost, to minimize cost due to data transfer cost heterogeneity. In this case, applications may be willing to spend additional cost for greater performance improvement compared to an even data distribution. For the last case (1/3 input data in SA East), skewed data increases cost reduction opportunities (50%) with similar performance improvement opportunities as the even distribution. This is because a large portion of intermediate data in SA East ($0.16/GB) needs to be migrated to other DCs to improve performance, which causes a cost increase. In this case, applications may be willing to bear additional latency for more cost reduction compared to other cases.

The results show that different data distributions open up a diverse tradeoff landscape and thus a GDA should consider the nature of the data distribution in order to meet desired tradeoff preferences.

4.7 Related Work

Geo-distributed Data Analytics: To overcome the limitation of the WAN, numerous approaches have been proposed [55, 56, 57, 58, 65, 66] that attempt to minimize network usage and deal with heterogeneous network bandwidth for reduced query execution time. Another set of previous works [64, 73, 74] have introduced the push-based shuffle to reduce query latency. While recent works [67, 68] showed that the cost of WAN bandwidth makes up a significant fraction of the overall cost, no existing GDA considered heterogeneous data transfer cost. To the best of our knowledge, Kimchi is the first GDA that considers heterogeneous data transfer cost to avoid a cost-bottleneck in a multi-cloud environment.

Tradeoff between Cost and Performance: Recent GDA systems [58, 66] offered a knob with which applications can tradeoff the WAN usage and query latency, similar to *C_pref* in Kimchi. These systems, however, may not achieve desired tradeoff due to the cost-agnostic approach, i.e., *reduced data transfer size does not necessarily lead to reduced data transfer cost* as we have shown throughout this chapter.

Handling Dynamics: Most GDA systems [55, 56, 57, 58, 65] do not handle dynamics during query execution. While Tetrium [66] considered dynamics, it can encounter cost- and performance-bottlenecks due to the cost-agnostic approach and the overhead of re-evaluating global task placement. While Kimchi adjusts tasks that are pending to handle dynamics, tasks are running in bottleneck DCs can be detected and re-assigned to other DCs as done in Lube [86].

Multi-Stage Jobs: For multi-stage jobs, we adopt a greedy approach to find optimized task placement independently for each stage due to non-convex optimization [55, 58, 66]. We plan to use historical information, e.g., intermediate data, reduction rate, type of operators for estimating the global solution for jobs. This approach would be highly efficient if queries are recurring as assumed in other works [56, 58, 77, 87].

Computing Resources: In this work, Kimchi focuses on exploring the tradeoff space

given computing resources (VM instances) deployed by other resource configurations systems [88, 89, 90]. That is, our work is complementary to these works. Tetrium [66] considered computing resources heterogeneity but monetary cost. We plan to extend Kimchi to consider computing resources and cost.

Rearrange Input (Intermediate) Data: Previous works [57, 58, 91] proposed to reduce query latency by re-arranging input data before query execution. This is complementary to our approach. For network cost, migrating input data may significant inflate overall cost if input data is bigger than intermediate data. We plan to re-arrange input and intermediate data for exploring a richer cost-performance tradeoff space.

4.8 Conclusion

In this chapter, we show that data transfer cost can significantly affect overall operational cost for geo-distributed data analytics (GDA), and thus should be considered. We present Kimchi, a GDA system that determines task placement with a consideration of data transfer cost, network bandwidth, data size and locations, and applications' desired cost-performance preference. Kimchi improves query performance without additional cost by a cost-aware task placement, a cost-aware task adjustment, and a cost-aware push-based mechanism. Experimental results on AWS show that Kimchi enables applications to reduce cost without impacting performance, improve performance without impacting cost, and enables the user to explore a richer cost-performance tradeoff space given different data distributions in a multi-cloud environment.

Chapter 5

Conclusion

Many recent Internet services and applications e.g., AirBnB, Nexflix, and Uber, rely on diverse and heterogeneous cloud resources in a multi-cloud environment to serve their geographically distributed users around the globe while achieving their needs e.g., less capital expenditure, easy maintenance, flexibility and so on. Exploiting such cloud resources in a multi-cloud environment can bring many opportunities e.g., simpler consistency policies, better performance, higher availability, and reduced cost, to applications. However, there are several challenges—complexities from diverse and heterogeneous cloud resources, burdens for application developers to implement numerous data and task placement policies, dynamics from infrastructure and users, and difficulties for determining optimized solutions (e.g., data and task placement)—that applications must overcome to get benefits from the environment.

In this thesis, we presented our novel and usable systems that 1) address the challenges, 2) realize the opportunities, and 3) help applications to achieve desired goals easily in a multi-cloud environment.

5.1 Policy-driven Storage System

Accessing heterogeneous storage options i.e., storage tiers and DC locations, introduces significant complexities to Internet applications because each option has different interfaces, different data models, and pricing policies. In addition, it can be burdensome to specify and program data placement policies to manage data across the different

storage options based on applications' desired metric(s) e.g., desired consistency model, SLA, degree of fault tolerance, desired cost and so on. This is further complicated by the dynamics from 1) cloud infrastructures (network): cloud providers do not guarantee consistent performance over time and 2) applications: users access patterns and locations keep changing. Thus, static decisions or policies may not be effective.

To address these challenges, we introduced Wiera, an integrated geo-distributed cloud storage system that runs across multiple storage tiers, multiple DCs, and multiple cloud providers, to exploit storage options available to the application and user. Wiera provides a flexible storage policy framework which allows applications to exploit diversity of storage options for optimizing across a wide array of metrics such as performance, cost, durability, reliability, in the face of network and application dynamics. We also presented several policy examples that show how easily data placement policies can be specified. We evaluated Wiera with given policy examples and the results indicated that metrics such as reduced cost and higher performance are obtainable by exploiting the larger set of storage options. Lastly, the benefits can be obtained with minimal impact to existing applications as demonstrated by the *unmodified* RUBiS application.

5.2 Automated Data Placement System

While exploiting multiple storage options in a multi-cloud environment can offer several benefits, a key problem left in that environment is *data placement*: determining storage options to place data replicas on. Though many previous works considered the problem of data placement in a geo-distributed cloud environment, they only considered DC locations but not storage tiers which can have a significant impact on metrics such as storage cost and performance. In addition, previous works are limited to handling long-term dynamics (from hours to weeks) such as changes in data access patterns and locations, by recomputing optimal data placement. Thus, short-term dynamics (from seconds to minutes) such as transient failures or overloads, cannot be handled.

To address these challenges, we introduced TripS (**S**torage **S**witch **S**ystem), a system that determines the data placement automatically in a multi-cloud environment. We designed TripS to be lightweight so that it can be used with any storage system running in that environment. TripS considers both storage tiers and DC locations to

minimize overall cost while satisfying applications' goals and constraints. TripS models and solves the data placement problem as a constrained optimization problem with mixed integer programming (MIP) with given inputs from an underlying storage system. We introduced the new notion of Target Locale List (TLL) to handle short-term dynamics proactively without the need to reevaluate data placement decisions. We evaluated the TripS prototype on Wiera in Amazon AWS and Microsoft Azure and the results indicated TripS helps applications 1) to lower overall cost by exploiting both multiple storage tiers and DC locations, and 2) to significantly reduce SLA violations by handling long- and short-term dynamics with minimal cost. We also showed that a web application (Retwis) can get the benefits of TripS with minimal modification.

5.3 Network Cost-aware Geo-distributed Data Analytics System

Finally, we addressed data analytics applications in a multi-cloud environment in which WAN bandwidth is a significant performance bottleneck. Geo-distributed data analytics (GDA) has become a popular method for mining valuable information from globally distributed data generated by users and systems. While many previous GDA systems have focused on improving query performance by efficiently managing the network performance bottleneck that is inter-DC bandwidth, these systems, unfortunately, may encounter a *cost bottleneck* because they have not considered the data transfer cost ($), one of the most expensive and heterogeneous resources in a multi-cloud environment. We found that minimizing WAN traffic does not necessarily lead to reduced data transfer cost due to the heterogeneous outbound data transfer cost that opens up a richer tradeoff space between monetary cost and query execution time.

To explore the tradeoff space, we have designed and implemented Kimchi, a network cost-aware GDA system for data analytics applications to meet cost-performance tradeoff preferences by considering data transfer cost, network bandwidth, data size and a desired tradeoff preference. A Kimchi prototype has been implemented on Spark and our experiments show that it reduces cost ($) 14% $\sim$ 24% without impacting performance and reduces query execution time by 45% $\sim$ 70% without impacting cost compared

to other baseline approaches (centralized and vanilla Spark). More importantly, Kimchi allows applications to explore a much richer cost-performance tradeoff space in a multi-cloud environment.

Chapter 6

Future Research Directions

While we have mainly focused on heterogeneous storage and network cloud resources in previous chapters, heterogeneous compute resources and costs are also important and are open problems worthy of future research. Here we briefly describe several possible research directions.

6.1 Geo-distributed Data Analytics with Multiple Resources

Many previous geo-distributed data analytics (GDA) systems mainly focused on heterogeneous WAN bandwidth[1] capacities to improve query performance with two assumptions: 1) GDA systems are running on multiple centralized clouds where compute capacities at sites are infinite and thus are not a performance bottleneck and 2) GDA applications have bandwidth-intensive workloads and thus data transfer time via WAN is dominant to overall performance (query execution time).

However, a newly emerging cloud infrastructure called edge cloud, invalidates the first assumption because edge clouds may have limited compute capacity compared to centralized clouds, while being more distributed to provide greater locality to applications. In addition, applications may use limited compute capacities even at centralized clouds due to a cost budget constraint. Thus, if compute capacity at each site is finite or limited, the compute resource may become a performance bottleneck.

For workloads, some GDA applications run iterative algorithms such as PageRank,

[1] In this chapter, we use the term bandwidth to refer network bandwidth unless mentioned.

Connected Components, and Machine Learning, that require much more compute capacities i.e., compute-intensive workload, and thus the second assumption may also not always be "true". Thus, GDA systems must consider not only heterogeneous WAN bandwidth capacities but also heterogeneous compute capacities to improve query performance as done in the recent work, Tetrium [66]. That is, Tetrium uses *bandwidth and compute capacities-aware approach* to improve query performance.

Table 6.1: AWS heterogeneous compute cost per hour (Snowball provides VM instances in Edge cloud)

Compute / Locations	Centralized				Edge	
	c5.xlarge	t3.2xlarge	g3s.xlarge	Lambda	Lambda	Snowball
	4 CPUs 8GiB	8 CPUs 32GiB	4 & 1 GPU 30.5GiB	128MiB ~ 2GiB	128MiB ~ 2GiB	-
US East	$0.17	$0.3328	$0.75	$0.0075 ~ $0.12	$0.0225 ~ $0.36	$2.08
SA	$0.262	$0.5376	N/A	$0.0075 ~ $0.12	$0.0225 ~ $0.36	$5
Asia NE	$0.214	$0.4352	$1.04	$0.0075 ~ $0.12	$0.0225 ~ $0.36	$3.33

While the bandwidth and compute capacities-aware approach can improve query performance, applications may encounter a *cost bottleneck* if heterogeneous network costs are not considered as shown in Chapter 4. That is, large intermediate data needs to be shuffled in reduce stages and the network cost-agnostic approach can increase data transfer cost significantly. Unfortunately, even more additional costs may be incurred because input data also needs to be migrated in map stages with the compute capacity-aware approach to improve performance.

The compute capacity-aware approach may deal with a compute-intensive workload very well. Applications that have compute-intensive workloads, however, may encounter another cost bottleneck if heterogeneous compute costs is not considered. Like heterogeneous data transfer cost, cloud providers offer very different pricing policies for compute resources based on DC locations and types, e.g., CPU and/or GPU within VM instances

and serverless computing. Table 6.1 shows some heterogeneous compute resources and their costs in AWS DC locations. Note, as of June 2019, AWS offers 168 VM instance types in the US East region alone with different pricing policies based on compute capacities i.e., number of CPU and GPU, and memory sizes. This table clearly shows that applications have numerous compute resource options, e.g., VM instance types (c5.xlarge or g3s.xlarge for GPU), compute resource types (vm instance or serverless), and geo-graphical locations (centralized or edge), to choose from based on needs. The table shows that each compute resource has very different pricing policies e.g., compute resources in edge clouds are very expensive compared to compute resources in centralized clouds, that opens another greater cost-performance tradeoff space along with heterogeneous data transfer cost shown in Kimchi (Chapter 4). Since compute time can be dominant in overall job execution time in compute-intensive workloads, GDA systems must consider heterogeneous compute capacities and their costs for applications to achieve desired cost-performance tradeoff preferences. The next section will show how heterogeneous data transfer and compute resource costs can affect overall cost for GDA applications with bandwidth-intensive and compute-intensive workloads.

6.2 Illustrative Example

In this section, we illustrate how heterogeneous data transfer and compute costs can impact previous approaches by discussing an example. To this end, let's consider an application executing a query on three sites as shown in Table 6.2. Note, this example is used in Tetrium [66] and we use it for comparison purposes. Let's assume a bandwidth-intensive workload first in which each compute slot can handle 100MB of data in 2 seconds and 1 second for map and reduce tasks respectively, and the size of intermediate data is halved from the size of input data.

The following are the performance and cost comparisons between two state-of-the-art approaches.

- **Bandwidth capacity-aware:** Many previous works e.g., Iridium [58], focused on heterogeneous bandwidth capacities to improve performance. That is, the numbers of compute slots at each site are not considered but only bandwidth. With an assumption that there is infinite compute capacities at each site, input data is processed

Table 6.2: 3 Sites example.

	Site A	Site B	Site C
Number of Compute Slots	40	**10**	20
Input Data	20GB	30GB	50GB
Uplink Bandwidth	5GB/s	1GB/s	2GB/s
Downlink Bandwidth	5GB/s	1GB/s	5GB/s

in-situ. However, if compute capacities are finite, as shown in Table 6.2, site B will become a performance bottleneck (60 seconds for map stage) due to lack of compute slots (10) and large input data size (30GB). This will result in 60 seconds to finish the map stage. In shuffle stages, intermediate data at each site (site A: 10GB, site B: 15GB, site C: 25GB) will be shuffled based on bandwidth capacities (10.5 seconds) and reduced with given compute slots at each site (18 seconds).

- **Bandwidth and compute capacities-aware:** Recent work [66] focused on heterogeneous bandwidth and compute capacities to improve performance. In this approach, input data needs to be migrated based on compute slots and bandwidth capacities. This will result in input data migration from site B (15.7 GB) and C (21.4GB) to site A to avoid the performance bottleneck from lack of compute slots at site B and A. Input data migration requires additional data transfer time (15.7 seconds from 0 second) but reduces compute time a lot (30 seconds from 60 seconds) in map stages compared to the bandwidth capacity-aware approach. The intermediate data are shuffled (6.13 seconds) and reduced (8 seconds) in shuffle stages based on given compute slots and bandwidth capacities.

Table 6.3: Summary for two approaches

	Map stage	Reduce stage	Total
Bandwidth capacity-aware	60 secs	28 secs	88 secs
Bandwidth and compute capacities-aware	45.7 secs	14.14 secs	59.84 secs

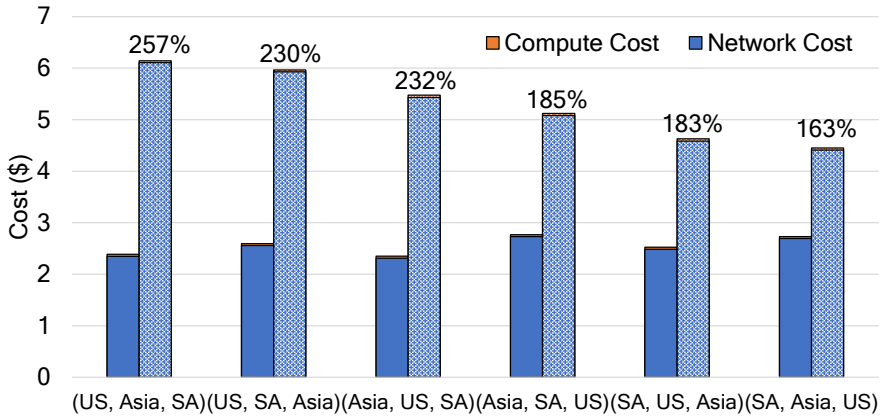


Figure 6.1: Cost comparison between the bandwidth capacity-aware approach (left) and the bandwidth and compute capacities-aware approach (right) for bandwidth-intensive workload. The percentage numbers show the cost increase for the bandwidth and compute capacities-aware approach compared to the bandwidth capacity-aware approach.

Table 6.3 shows the performance comparison between two approaches. The result obviously shows that considering heterogeneities of bandwidth and compute capacities can improve overall performance. That is, the bandwidth and compute capacities-aware approach reduces query execution time 33% compared to the bandwidth capacity-aware approach. Unfortunately, these approaches have not considered heterogeneous data transfer costs (Table 4.1) and thus applications may see inflated overall cost as shown previously in Chapter 4.

Figure 6.1 shows the cost comparison when we apply varying combinations of site locations to the example (Table 6.2) for each approach. Note, the left bars show costs for the bandwidth capacity-aware approach and right bars show costs for the bandwidth and compute capacities-aware approach. We use inter-AWS data transfer and compute (t3.3xlarge case) costs shown in Tables 4.1 and 6.1 respectively with an assumption that the application uses AWS for simplicity. The figure shows that the bandwidth and compute capacities-aware approach can incur additional cost (up to 2.57X) based on site locations and thus it may result in a cost bottleneck and open a greater cost-performance tradeoff space.

While compute cost is included in Figure 6.1, it has a negligible portion of overall cost because the result is derived with a bandwidth-intensive workload in which compute

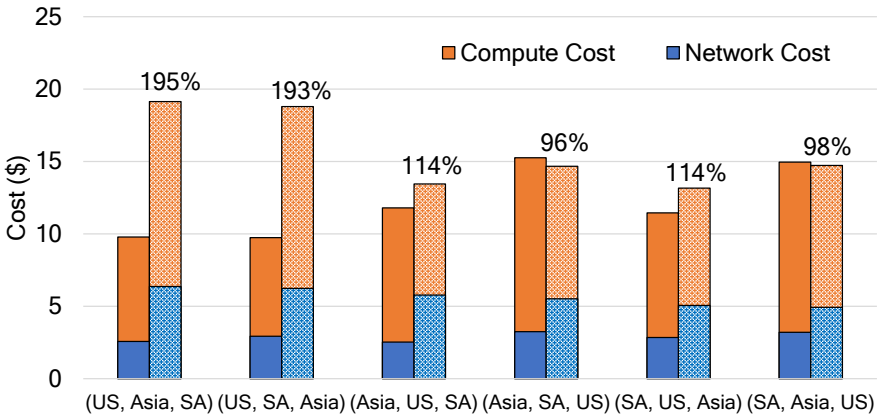


Figure 6.2: Cost comparison between the bandwidth capacity-aware approach (left) and the bandwidth and compute capacities-aware approach (right) for compute-intensive workload. The percentage numbers show the cost increase for the bandwidth and compute capacities-aware approach compared to the bandwidth capacity-aware approach.

time for map and reduce is very small (2 seconds for map task and 1 second for reduce task). So if the compute time increases significantly i.e., a compute-intensive workload, the portion of compute cost would also increase. Let's consider an iterative query using the PageRank algorithm with 3 iterations as an example of a compute-intensive workload in which map stages require 200 seconds and reduce stages require 100 seconds. Note, these values are estimated based on previous work [92]. In the query, there will be only one map stage that represents data structures for graphs and ranks by reading data from underlying storage system (HDFS) and there will be three reduce stages for the 3 iterations. For heterogeneous compute cost, we assume that site A is an edge cloud with more expensive compute resources (up to 3X) compared to other sites as shown in table 6.1.

Figure 6.2 shows the cost comparison between approaches when we apply these new values to the example (Table 6.2). Note, the left bars show costs for the bandwidth capacity-aware approach and right bars show costs for the bandwidth and compute capacities-aware approach. The result clearly shows that compute cost has a larger portion of overall cost compared to a bandwidth-intensive workload (Figure 6.1) while there is not much difference for network cost. The result also shows that compute cost varies based on site configurations, i.e., more compute cost is incurred when site A is in

the US (two left-most cases) region. While Figure 6.1 shows that the bandwidth and cost-aware approach always increases overall cost, interestingly, the result shows that performance improvement and cost reduction can be achieved with the bandwidth and compute capacities-aware approach, i.e., (Asia, SA, US) and (SA, Asia, US) cases. This result clearly shows that considering heterogeneous compute cost opens another greater cost-performance tradeoff space along with data transfer cost for GDA applications and thus GDA systems must consider all heterogeneous cloud resources together for applications to achieve their goals.

6.3 Our System Design and Challenges

Having shown the potential opportunity of a greater cost-performance space for geo-distributed analytics (GDA) applications in a multi-cloud environment, we now present our system design vision and research challenges to realizing a practical deployment.

6.3.1 System Vision

As we have shown, considering diverse and heterogeneous cloud resources capacities and costs would open a greater cost-performance tradeoff space. We envision a GDA system that exploits such diverse and heterogeneous cloud resources e.g., compute resources: VM instances (with GPU) and serverless services (e.g., AWS Lambda and Lambda Edge), and their monetary costs for GDA applications to explore a richer tradeoff space easily by specifying desired tradeoff preferences, as we have done in Kimchi (Chapter 4).

For optimal task placement, the compute capacity-aware approach considers *static* heterogeneous compute capacities i.e., available CPU slots at each site, to avoid a compute capacity bottleneck by sending input data to other sites, that can incur significant additional data transfer cost. Instead, we consider *adaptive* compute capacities to avoid the bottleneck while handling jobs. That is, if the given compute capacity at a site is likely to become a performance bottleneck for jobs, a GDA system may use additional compute capacities e.g., spawning new VM instances in centralized cloud or using serverless computing services such as AWS Lambda Edge in an edge cloud, to

avoid the performance bottleneck at the site instead of migrating input data. Likewise, when the compute resources are not (or are less) utilized, the resources may be revoked to save cost. While this adaptive approach would lead to better performance (reduced query execution time) in a cost efficient way that meets applications' goals, it also introduces many challenges that need to be overcome to realize the benefits as we will discuss in the next section. Note, the adaptive approach may not work well in an edge cloud that has limited compute resources but that limitation will be automatically handled by cloud providers using multiple edge cloud locations, e.g., AWS has 187 edge locations around the globe and 7 edges locations within Chicago alone as of June 2019. In the case when additional compute resources are not available in an edge cloud, the tasks may be offloaded to the closest centralized cloud location if doing so can help applications to achieve their goals. In the next section, we present possible challenges to the design and implementation of the systems we envisioned.

6.3.2 Challenges

Designing and building adaptive GDA systems that consider diverse and heterogeneous cloud resources are challenging due to several reasons.

- **Determining Optimal Task Placement:** To determine optimal task placement decisions for GDA applications, many factors that affect task placement must be considered: 1) applications' tradeoff preferences, 2) the cost of cloud resources, 3) cloud resource capacities, 4) overall data sizes for queries, and 5) expected workloads, e.g., bandwidth-intensive or compute-intensive (or both). Considering all these factors will bring significant complexities for GDA systems to determine global optimal solutions.

- **Predicting Workloads and Execution Times of Jobs:** To determine optimal task placement and manage cloud resources adaptively during query executions, GDA applications should be able to predict workloads and execution times of jobs. This is because optimal task placement decisions and required cloud resources will be determined based on workloads of jobs and their execution times. While many previous works [88, 89, 90] focused on automatic prediction of the performance of a target application to select the right cloud configuration, these systems have not considered the possibility of sharing cloud resources among jobs and the adaptability of cloud

resources during query execution. That is, each job will have independent and static cloud resources during query execution. This independent and static approach may miss out on sharing cloud resources that can reduce cost by using resources in a more efficient way, and may result in poor performance if any dynamics e.g., network delay and variation of compute performance, occur during query execution.

- **Determining Optimal Cloud Configuration:** Even if the GDA systems could predict expected workloads and execution times of jobs, determining optimal cloud configurations for jobs is still challenging because the possible combinations of resources will increase significantly as we consider diverse and heterogeneous cloud resources. That is, applications must choose from a variety of VM instances (or serverless compute) to determine the right number of CPU cores and memory, the number of VMs, and their network bandwidth. Note, AWS offers 168 VM instance types in the US East region alone as of June 2019.

- **Designing for Hybrid Compute Resource Model and Implementation:** In our system design, different compute resources e.g., VM instances and serverless, need to be chosen automatically and used together for handling jobs. This will bring significant complexities to GDA systems because compute resources have different characteristics such as different interfaces and provisioning times. In addition, while a serverless compute service provides an easy way to run code without deploying VM instances, it has several limitations, e.g., memory size, deployment package (program) size, and execution time. Recent work [93] shows that significant effort is required to use serverless compute resources in GDA systems. To maximize benefits, GDA systems need to think carefully about the design for addressing these complexities and managing heterogeneous compute resources in an efficient way.

- **Handling Dynamics:** As many previous works [64, 94, 95, 96] showed, cloud providers do not guarantee consistent performance over time for their cloud resources. Thus, WAN bandwidth fluctuation or compute performance variation may occur after the optimal solutions e.g., cloud resource configurations and task placement decision, are determined. One simple, possible solution to handle such dynamics could be monitoring tasks' status to detect tasks running within bottleneck resources as done in the recent work, Lube [86]. This allows GDA systems to cancel the delayed tasks and

reassign them to other sites (or nodes) to avoid performance bottlenecks.

References

[1] AWS Global Infrastructure. https://goo.gl/M0ddoz.

[2] Azure regions. https://goo.gl/EesP16.

[3] Google data center locations. https://cloud.google.com/about/locations.

[4] Kwangsung Oh, Ajaykrishna Raghavan, Abhishek Chandra, and Jon Weissman. Redefining data locality for cross-data center storage. In *Proceedings of the 2Nd International Workshop on Software-Defined Ecosystems*, BigSystem '15, pages 15–22, New York, NY, USA, 2015. ACM.

[5] Gartner Study Recommends Multicloud Approach for Enterprise IT. https://www.io.com/colocation-gartner-enterprise-multicloud.

[6] CenturyLink acquires multi-cloud platform startup ElasticBox. https://news.centurylink.com/news/centurylink-acquires-multi-cloud-platform-startup-elasticbox.

[7] The future isn't cloud. It's multi-cloud. https://www.infoworld.com/article/3165326/cloud-computing/the-future-isnt-cloud-its-multi-cloud.html.

[8] Data Center Map. http://www.datacentermap.com/.

[9] DC Outages Report. https://goo.gl/pDr7or.

[10] Kwangsung Oh, Abhishek Chandra, and Jon Weissman. Wiera: Towards flexible multi-tiered geo-distributed cloud storage instances. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 165–176, New York, NY, USA, 2016. ACM.

[11] Kwangsung Oh, Abhishek Chandra, and Jon Weissman. Trips: Automated multi-tiered data placement in a geo-distributed cloud environment. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, pages 12:1–12:11, New York, NY, USA, 2017. ACM.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[13] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[14] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 367–381, Berkeley, CA, USA, 2014. USENIX Association.

[15] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 292–308, New York, NY, USA, 2013. ACM.

[16] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.

[17] Ajaykrishna Raghavan, Abhishek Chandra, and Jon B. Weissman. Tiera: Towards flexible multi-tiered cloud storage instances. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 1–12, New York, NY, USA, 2014. ACM.

[18] WordPress. http://wordpress.org/.

[19] Moodle. http://moodle.org/.

[20] AWS outage. https://goo.gl/N1LqZr.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[23] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[24] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX-ATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[25] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *Comm. Mag.*, 41(8):84–90, August 2003.

[26] Yang Tang, Gang Hu, Xinhao Yuan, Lingmei Weng, and Junfeng Yang. Grandet: A unified, economical object store for web applications. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 196–209, New York, NY, USA, 2016. ACM.

[27] Wiera Homepage. https://github.com/dcsg-umn/wiera/.

[28] MySQL [Online]. Available: http://dev.mysql.com.

[29] Apache Cassandra. http://cassandra.apache.org/.

[30] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[31] Flashcache at Facebook: From 2010 to 2013 and beyond. `http://alturl.com/us4fi/`.

[32] Apache Thrift. `https://thrift.apache.org`.

[33] Curator. `http://curator.apache.org/`.

[34] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[35] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[36] SysBench. `https://github.com/akopytov/sysbench`.

[37] RUBiS Web site. `http://rubis.ow2.org`.

[38] FUSE - Filesystem In User Space. `https://github.com/libfuse/libfuse/`.

[39] Azure Virtual Machine. `https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-sizes/`.

[40] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, Berkeley, CA, USA, 2011. USENIX Association.

[41] Nalini Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Mike Dahlin, and Robert Grimm. Pads: A policy architecture for distributed storage systems. In

Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI'09, pages 59–73, Berkeley, CA, USA, 2009. USENIX Association.

[42] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[43] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. Hybridstore: A cost-efficient, high-performance storage system combining ssds and hdds. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 227–236, Washington, DC, USA, 2011. IEEE Computer Society.

[44] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 20–20, Berkeley, CA, USA, 2011. USENIX Association.

[45] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Bluesky: A cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association.

[46] The Infrastructure Behind Twitter: Scale. `https://goo.gl/HNYQYz`.

[47] Netflix - Data Persistence. `https://netflix.github.io`.

[48] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical*

Conference, USENIX ATC'14, pages 169–180, Berkeley, CA, USA, 2014. USENIX Association.

[49] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 229–240, New York, NY, USA, 2010. ACM.

[50] Stuart Mitchell, Stuart Mitchell Consulting, and Iain Dunning. Pulp: A linear programming toolkit for python, 2011.

[51] CPLEX Optimizer. `https://goo.gl/6BKOZY`.

[52] Well-known disk seek time. `https://goo.gl/30EjTX`.

[53] Redis. `http://redis.io`.

[54] Retwis-py. `https://github.com/pims/retwis-py`.

[55] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. Pixida: Optimizing data parallel jobs in wide-area data analytics. *Proc. VLDB Endow.*, 9(2):72–83, October 2015.

[56] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 323–336, Berkeley, CA, USA, 2015. USENIX Association.

[57] Benjamin Heintz, Abhishek Chandra, Ramesh K. Sitaraman, and Jon Weissman. End-to-end optimization for geo-distributed mapreduce. *IEEE Transactions on Cloud Computing*, 4(3):293–306, 7 2016.

[58] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *SIGCOMM Comput. Commun. Rev.*, 45(4):421–434, August 2015.

[59] Hadoop. `http://hadoop.apache.org`.

[60] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 49–, Washington, DC, USA, 2006. IEEE Computer Society.

[61] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 275–288, Berkeley, CA, USA, 2014. USENIX Association.

[62] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. Mesa: A geo-replicated online data warehouse for google's advertising system. *Commun. ACM*, 59(7):117–125, June 2016.

[63] Michael Cardosa, Chenyu Wang, Anshuman Nangia, Abhishek Chandra, and Jon Weissman. Exploring mapreduce efficiency with highly-distributed data. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 27–34, New York, NY, USA, 2011. ACM.

[64] Shuhao Liu, Hao Wang, and Baochun Li. Optimizing shuffle in wide-area data analytics. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 560–571, 2017.

[65] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. CLARINET: wan-aware optimization for analytics queries. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 435–450, 2016.

[66] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *Proceedings of*

the Thirteenth EuroSys Conference, EuroSys '18, pages 12:1–12:16, New York, NY, USA, 2018. ACM.

[67] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, December 2008.

[68] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 629–647, Berkeley, CA, USA, 2017. USENIX Association.

[69] Apache Spark @Scale: A 60 TB+ production use case. `https://code.facebook.com/posts/1671373793181703/apache-spark-scale-a-60-tb-production-use-case/`.

[70] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 301–316, Berkeley, CA, USA, 2014. USENIX Association.

[71] Fan Lai, Mosharaf Chowdhury, and Harsha Madhyastha. To relay or not to relay for inter-cloud transfers? In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, 2018. USENIX Association.

[72] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 236–252, New York, NY, USA, 2018. ACM.

[73] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[74] Yanfei Guo, Jia Rao, and Xiaobo Zhou. ishuffle: Improving hadoop performance with shuffle-on-write. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 107–117, San Jose, CA, 2013. USENIX.

[75] TPC-DS. `http://www.tpc.org/tpcds`.

[76] http://sortbenchmark.org/YahooHadoop.pdf. `http://sortbenchmark.org/YahooHadoop.pdf/`.

[77] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[78] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[79] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[80] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[81] AWS Pricing. `http://aws.amazon.com/ec2/pricing/`.

[82] Azure Pricing. `https://azure.microsoft.com/en-us/pricing/details/bandwidth/`.

[83] Introducing JSON. `http://www.json.org/`.

[84] Hadoop Distributed File System. `http://http://hadoop.apache.org/`.

[85] Linux Traffic Control. `http://lartc.org/manpages/tc.txt`.

[86] Hao Wang and Baochun Li. Lube: Mitigating bottlenecks in wide area data analytics. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, Santa Clara, CA, 2017. USENIX Association.

[87] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[88] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 469–482, Berkeley, CA, USA, 2017. USENIX Association.

[89] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 363–378, Berkeley, CA, USA, 2016. USENIX Association.

[90] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, 2018. USENIX Association.

[91] Hangyu Li, Hong Xu, and Sarana Nutanong. Bohr: Similarity aware geo-distributed data analytics. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, Santa Clara, CA, 2017. USENIX Association.

[92] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13):2110–2121, September 2015.

[93] Qubole Announces Apache Spark on AWS Lambda [Online]. Available: `https://www.qubole.com/blog/spark-on-aws-lambda/`.

[94] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[95] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[96] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association.