TripS: Automated Multi-tiered Data Placement in a Geo-distributed Cloud Environment

Kwangsung Oh, Abhishek Chandra, and Jon Weissman
Department of Computer Science and Engineering
University of Minnesota Twin Cities
Minneapolis, MN 55455
{ohkwang, chandra, jon}@cs.umn.edu

ABSTRACT
Exploiting the cloud storage hierarchy both within and across data-centers of different cloud providers empowers Internet applications to choose data centers (DCs) and storage services based on storage needs. However, using multiple storage services across multiple data centers brings a complex data placement problem that depends on a large number of factors including, e.g., desired goals, storage and network characteristics, and pricing policies. In addition, dynamics e.g., changing user locations and access patterns, make it impossible to determine the best data placement statically. In this paper, we present TripS, a lightweight system that considers both data center locations and storage tiers to determine the data placement for geo-distributed storage systems. Such systems make use of TripS by providing inputs including SLA, consistency model, fault tolerance, latency information, and cost information. With given inputs, TripS models and solves the data placement problem using mixed integer linear programming (MILP) to determine data placement. In addition, to adapt quickly to dynamics, we introduce the notion of Target Locale List (TLL), a pro-active approach to avoid expensive re-evaluation of the optimal placement. The TripS prototype is running on Wiera, a policy driven geo-distributed storage system, to show how a storage system can easily utilize TripS for data placement. We evaluate TripS/Wiera on multiple data centers of AWS and Azure. The results show that TripS/Wiera can reduce cost 14.96% ∼ 98.1% based on workloads in comparison with other works’ approaches and can handle both short- and long-term dynamics to avoid SLA violations.

CCS Concepts
• Information systems → Cloud based storage; Hierarchical storage management;

Keywords
Data placement; Multi-tiered storage; Multi-DC storage

1. INTRODUCTION
Many cloud providers offer diverse storage options 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 services1, 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. For instance, as of Jan 2017, Amazon has DCs in 16 regions (and numerous Edge locations) [5] and Microsoft has DCs in 26 regions [6]. 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 [27, 18] 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 [20].

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 [10, 32, 2, 4], 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

1In this paper, we use the terms storage service and storage tier interchangeably.
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 [20, 19].

To address these problems, we present TripS (Storage Switch System), 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 [19, 8, 1]. 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 linear programming (MILP) in TripS. While TripS can be programmed to optimize different metrics such as cost, performance, or reliability, in this paper, 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 trade-off 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 [19], 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 paper 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 linear programming (MILP), 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.

2. SYSTEM MODEL

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 Wiera [19], SPANStore [32], SCFS [8], and RACS [1]. 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-iol 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 [20, 19]. 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 [16]. 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.

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 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 [32, 26]. Assigning user requests to appropriate application instances is out of scope for this work, and prior techniques [2, 4] could be utilized for this.

2.3 Data Placement Problem

We define a locale as a \{DC-location, storage-tier\} tuple, e.g., \{Amazon US East, S3\}². 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 paper, we consider the goal of minimizing the total cost (both storage and I/O costs).

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 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 3.1) that takes a set of inputs (Section 3.1.1) 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 3.2.1). At the same time, it provides the notion of Target Locale List (TLL) (Section 3.1.2) to adapt quickly to dynamics at short time scales (Section 3.2.2).

3.1 Data Placement Decision

3.1.1 TripS’ Inputs and Output

Inputs: TripS requires four types of inputs: applications goals, network and storage monetary cost, performance monitoring information, and workload information. Table 1 shows the inputs that TripS uses.

Application Desired Goals: TripS requires applications to provide three types of 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, \( \text{locale} \) count \( \text{(LC)} \), is the number of feasible placement options desired to handle short time-scale dynamics. TripS requires the following information.

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.

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.

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 paper, we evaluate TripS on a coarse placement of data (i.e., the entire data set for an application, as in other systems [2]), 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.

3.1.2 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 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 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 3.2.2, we will discuss in detail how the GDSS can use the multiple options in the TLL at runtime to avoid SLA violations.

3.1.3 Optimization Problem Formulation

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

**Variables:** We define three sets of output variables:

\[ \forall i,j \in D, \forall t \in D, S : T_{ijt} \]

\[ T_{ijt} \] 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).

\[ \forall i \in D, \forall t \in D, S : P_{it} \]

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

\[ \forall i, j, k \in D, \forall t \in D, S : B_{ijkt} \]

\[ B_{ijkt} \] 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,

\[ \sum_{i} A_{i}^{get} \sum_{j} \sum_{t} T_{ijt} \cdot (\text{Size}_{t} \cdot (C_{ij}^{net} + C_{ij}^{cst}) + C_{ij}^{get}) \]

Get cost:

\[ \sum_{i} A_{i}^{put} \sum_{j} \sum_{t} T_{ijt} \cdot (\text{Size}_{t} \cdot (C_{ij}^{net} + C_{ij}^{write}) + C_{ij}^{put}) \]

Put cost:

\[ \sum_{i} \sum_{j} \sum_{k} \sum_{t} B_{ijkt} \cdot (\text{Size}_{t} \cdot (C_{jk}^{net} + C_{kt}^{write}) + C_{kt}^{put}) \]

Broadcast cost:

\[ \sum_{i} \sum_{t} P_{it} \cdot \text{Size}_{t} \cdot \text{C}_{it}^{storage} \]

Storage Cost:

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:

\[ \forall i, j \in D, \forall t \in D, S : \sum_{S} T_{ijt} = \text{LC} \]

Set minimum number of replicas for availability:

\[ \forall i \in D, \forall t \in D, S : \sum_{j} P_{it} \geq F + 1 \]

At most one storage tier in each DC:

\[ \forall i \in D, \forall t \in D, S : \sum_{j} P_{it} \leq 1 \]

Latency SLA constraint:

For eventual consistency:

\[ \forall i, j, t \in D, i \neq j, t \in D, S : \text{Latency}_{ijt} \leq \text{SLA}_{ijt} \text{Get/put} \text{ if } (T_{ijt} = 1) \]

For strong consistency:

\[ \forall i, j, t \in D, i \neq j, t \in D, S : \text{Latency}_{ijt} \leq \text{SLA}_{ijt} \text{Get/put} + 2 \cdot \text{Network}_{ijt} \cdot \text{Latency}_{Center} \in \text{Max}(\text{Network}_{ijt}) \text{ if } (T_{ijt} = 1) \]

where, \( \delta_{input} \) indicates whether this is a Put request.

3.2 Handling Dynamics

3.2.1 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 [32, 2]. 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.

### 3.2.2 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 3.1.2, a TLL consists of locales that can all nominally satisfy the SLA. The GDSS can thus switch locales to avoid SLA violations due to short-term dynamics at runtime. 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 shows how the GDSS server running on Asia NE (from the figure 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 our previous work [19].

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.

### 4. TRIPS IMPLEMENTATION

We have implemented TripS on top of the Wiera [19] GDSS. 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. We begin by providing a brief description of Wiera as background. Readers may consult the Wiera paper [19] for additional details. 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.

#### 4.1 Wiera Geo-distributed Storage System

Wiera is a policy-driven key-value storage system for a multi-cloud environment. Wiera provides a flexible framework for application developers to specify storage policies easily with which applications can exploit multiple storage tiers across multiple DCs (even across different providers). The client of Wiera is shielded from the underlying complexity introduced by multiple storage tiers of multiple cloud providers by a simple Get/Put API and the encapsulation of storage policies. In Wiera, an application can create a global Wiera storage instance encompassing multiple DCs. Each Wiera instance is comprised of several local Wiera instances, each running within a DC.

**Local Wiera instance**: The local instance [22] encapsulates multiple cloud storage tiers within a DC 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 within a local instance. An event is the occurrence of some condition and a response is the action executed on the occurrence of an event. A local instance supports different kinds of events such as timer, threshold, and action events (Get and Put). It supports responses such as store, retrieve, copy, move, encrypt, compress, delete, and grow to react to the events.

**Global Wiera instance**: While the local instance is responsible for managing data on multiple storage tiers within a single DC, the global Wiera instance manages the data placement and data movement across multiple local instances running on geo-distributed DCs. 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. Wiera 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.

#### 4.2 TripS Interfaces and Execution

Table 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 patterns through TripS API that is declared and implemented with Thrift [3] 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 [17] (toolkit for linear programming

<table>
<thead>
<tr>
<th>API</th>
<th>Arguments</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_cost</td>
<td>cost_information</td>
<td>Set cost information</td>
</tr>
<tr>
<td>set_goals</td>
<td>application_goals</td>
<td>Set applications goals</td>
</tr>
<tr>
<td>evaluate</td>
<td>monitoring_information</td>
<td>Evaluate data placement</td>
</tr>
</tbody>
</table>

Figure 3: Locale switching example.
in Python to model a data placement problem as a mixed integer linear programming (MILP) and uses solver CPLEX [11] to solve the optimization problem.

4.3 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).

4.3.1 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.

4.3.2 Event and Response

Figure 4 shows policies to handle dynamics in TripS/Wiera. We use two Wiera events 1) RequestsMonitoring and 2) LatencyMonitoring to handle both coarse and short time-scale dynamics. 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 the instance receiving the highest number of requests within a specific time period has changed. 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 request_data_placement response. Then, the global Wiera instance calls the evaluate() function of TripS.

To handle short time-scale dynamics, all local instances monitor local storage tiers latency via LatencyMonitoring. We added a set_tier_violation response which marks a storage tier that currently causes SLA violations greater than a specific period threshold. When instances handle requests, they avoid accessing the storage tiers that have a mark. Those storage tiers can be (re-)accessed when the mark is removed by the background storage latency monitor.

4.4 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 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. 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 Storage Tiers: Locales in TLL nominally satisfy SLA goals as mentioned in Section 3.2.2. To handle requests with minimum cost, local instances find and use the cheapest locale in TLL using cost information. If a local instance detects an SLA violation (marked by set_tier_violation) for the cheapest locale, the instance 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 monitor-

3This is similar to “relayed update propagation” in SPANstore.
**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 MILP 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 [9]. 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 [30]. 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 [25] 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 paper is as of Feb 2017.

### 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.

Figure 5 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 shows the cost comparison between using a single centralized DC US East (as in Grandet) and using 2 DCs US East and 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.

### 5.2 Dynamic Data Placement

Access pattern (reads vs. writes and user location) is an important factor to be considered as shown in many previ-
Table 4: Data placement and cost comparison

<table>
<thead>
<tr>
<th>LC</th>
<th>Data Placement</th>
<th>Storage</th>
<th>Network</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>US East (EBS-st1), US East 2 (EBS-st1), US West 2 (EBS-st1)</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>US East (EBS-st1), US East 2 (EBS-gp2), US West 2 (EBS-st1)</td>
<td>140.7%</td>
<td>100%</td>
<td>105.3%</td>
</tr>
<tr>
<td>3</td>
<td>US East (EBS-gp2), US East 2 (EBS-gp2), US West (EBS-st1)</td>
<td>188.1%</td>
<td>100%</td>
<td>111.5%</td>
</tr>
<tr>
<td>4</td>
<td>US East (EBS-gp2), US East 2 (EBS-gp2), US West (EBS-st1), CA Central (EBS-gp2)</td>
<td>269.6%</td>
<td>166.7%</td>
<td>180.1%</td>
</tr>
</tbody>
</table>

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 `SwitchingStorageTier` policy (Figure 4), 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 4 shows the data placement evaluated with LC = 1, LC = 2, LC = 3, and LC = 4. The table also shows 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 7(a) and 7(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 7(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...
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 [23] 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).

5.4.1 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 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 10(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). 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.

5.4.2 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
Multi-Tiered Storage:

Many previous works \[14, 31, 28\] while achieving applications’ desired goals. Data locations and storage tiers to generate data placement.

Optimized storage tiers in multiple DCs. TripS considers both data center and DC locations with the aid of TripS.

Geo-distributed Data Analytics: Many works \[15, 29, 13, 21\] were proposed to overcome the limits of popular centralized data analytic frameworks e.g., Hadoop \[12\] and Spark \[33\] 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.

7. CONCLUSION

In this paper, 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% ~ 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.

8. ACKNOWLEDGMENT

We acknowledge grant NSF CSR-1162405 and CNS-1619254 that supported this research.
9. REFERENCES


