Wiera: Towards Flexible Multi-Tiered Geo-Distributed Cloud Storage Instances

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

Geo-distributed cloud storage systems must tame complexity at many levels: uniform APIs for storage access, supporting flexible storage policies that meet a wide array of application metrics, handling uncertain network dynamics and access dynamism, and operating across many levels of heterogeneity both within and across data-centers. In this paper, we present an integrated solution called Wiera. Wiera extends our earlier cloud storage system, Tiera, that is targeted to multi-tiered policy-based single cloud storage, to the wide-area and multiple data-centers (even across different providers). Wiera enables the specification of global data management policies built on top of local Tiera policies. Such policies enable the user to optimize for cost, performance, reliability, durability, and consistency, both within and across data-centers, and to express their tradeoffs. A key aspect of Wiera is first-class support for dynamism due to network, workload, and access patterns changes. Wiera policies can adapt to changes in user workload, poorly performing data tiers, failures, and changes in user metrics (e.g., cost). Wiera allows *unmodified applications* to reap the benefits of flexible data/storage policies by externalizing the policy specification. As far as we know, Wiera is the first geo-distributed cloud storage system which handles dynamism actively at run-time. We show how Wiera enables a rich specification of dynamic policies using a concise notation and describe the design and implementation of the system. We have implemented a Wiera prototype on multiple cloud environments, AWS and Azure, that illustrates potential benefits from managing dynamics and in using multiple cloud storage tiers both within and across data-centers.

Keywords

Data Locality; Multi-DCs; Multi-tiered storage; Wide Area Storage; In Memory Storage

1. INTRODUCTION

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

DOI: http://dx.doi.org/10.1145/2907294.2907322

Today, the use of multiple geo-distributed datacenters (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 workload, 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, cloud services, and applications. Thus, static decisions or policies will not be effective. For example, application workload 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 data analytics applications. 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 [3, 22, 1] 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 assumes users are static). This results in policies that may be inadequate in a wide-area multiple-tier environment that spans different storage providers 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[1] with different characteristics such as durability, performance, and cost across their constituent DCs. Exploiting such diversity both within and *across* cloud storage providers can yield greater storage options and therefore greater benefits. Moreover, the number of DCs has been increasing continuously suggesting this opportunity will only grow. According to datacentermap.com [10], there were 171 DCs as

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

of Sep 2014 but 201 as of Dec 2015 within only the US West (California) region. So it seems clear that applications will have many more storage options to place data given the increased density of DCs. In a previous paper [15] we showed that applications can benefit by using multiple DCs within the same region.[2]

To address these challenges and opportunities, we present a new geo-distributed cloud storage system called Wiera (or Wide-area tIERA) that builds upon our Tiera cloud storage system [18]. Tiera provides storage instances that span the storage hierarchy within a single data-center for a single cloud provider. Wiera extends Tiera in multiple dimensions: to 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). As 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, 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 paper 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.
- 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 furhter 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.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of Tiera upon which Wiera is built. Section 3 provides an overview of Wiera including its architecture and examples of global policies. Section 4 explains the implementation details of a Wiera prototype in the AWS (Amazon) and Azure (Microsoft) clouds. Section 5 discusses the results of our experimental evaluation. The results demonstrates that Wiera handles dynamics and achieves benefits by utilizing disparate storage tiers across multiple DCs and cloud providers. Section 6 reviews related work. Section 7 concludes the paper.

[2]We use the term region to represent a specific location e.g., US West, US East, and Europe West.

2. BACKGROUND

Wiera extends Tiera in multiple directions: wide-area (e.g., multiple data-centers, regions, and providers), dynamism, and global policies. We provide a brief description of Tiera as a background for understanding Wiera. Readers may consult the Tiera paper [18] for additional technical details.

2.1 Tiera Instance

Cloud service providers offer multiple storage tiers with different characteristics, performance, durability, and cost for storage. For example, Amazon provides ElastiCache (a caching service protocol compliant to Memcached), S3 (Simple Storage Service), EBS (Elastic Block Store), and Glacier as different cloud storage options. These storage services generally optimize one metric trading off others. For instance, an application can get better performance from ElastiCache but at high cost and low durability, compared to using S3. Thus, it is common to see applications seeking to obtain composite benefits from multiple cloud storage tiers, e.g., putting hot data in memory using ElastiCache for better performance and cold data in S3 for higher durability and reduced cost with much lower performance. 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). To address these problems, a Tiera instance encapsulates multiple cloud storage tiers and enables easy specification of a rich array of data storage policies to achieve desired tradeoffs. An important property of Tiera is that it can be inserted into an existing application framework with minimal or no code changes.

Two primary mechanisms—*event* and *response*—are used to express policies and manage data within the instance. An *event* is the occurrence of some condition and a response is the action executed on the occurrence of an event. Tiera supports different kinds of events such as *timer*, *threshold*, and *action* events. Tiera also supports *responses* such as *store*, *retrieve*, *copy*, *move*, *encrypt*, *compress*, *delete*, and *grow* to react to the events. New events and responses will be added in support of Wiera as we will show in Section 3.2.3. A Tiera instance is defined by specifying the following: the desired storage tiers, their capacities, and a set of events along with their responses. For example, Figure 1(a) and 1(b) show Tiera instances for low latency and for persistent data respectively. **LowLatencyInstance** (Figure 1(a)) uses two storage tiers, Memcached for performance and EBS 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. **PersistentInstance** (Figure 1(b)) trades performance for better data durability. This instance uses a small Memcached area to cache the most recently written data and copies data to EBS immediately when data is inserted into Memcached according to a *write-through* policy. This example also shows a simple backup policy which copies data into a more durable storage tier S3 if data in EBS is filled more than 50%. In this instance, an application may want to move data to Glacier instead of S3 not only for durable storage but also to reduce the price of cold data. Using these policies, the client of a Tiera instance is

```
Tiera LowLatencyInstance(time t) {
   % two tiers specified with initial sizes
   tier1: {name: Memcached, size: 5G};
   tier2: {name: EBS, size: 5G};

   % action event defined to always store data
   % into Memcached
   event(insert.into) : response {
      insert.object.dirty = true;
      store(what:insert.object, to:tier1);
   }

   % write back policy: copying data to
   % persistent store on a timer event
   event(time=t) : response {
      copy(what: object.location == tier1 &&
                 object.dirty == true,
           to:tier2);
   }
}
```

(a) LowLatency Tiera instance.

```
Tiera PersistentInstance(time t) {
   tier1: {name: Memcached, size: 5G};
   tier2: {name: EBS, size: 5G};
   tier3: {name: S3, size: 10G};

   % write-through policy using action event data
   % and copy response
   event(insert.into == tier1) : response {
      copy(what:insert.object, to:tier2);
   }

   % simple backup policy
   event(tier2.filled == 50%) : response {
      copy(what:object.location == tier2,
              to:tier3, bandwidth:40KB/s);
   }
}
```

(b) Persistent Tiera instance.

Figure 1: Tiera instance specifications.

shielded from the underlying complexity introduced by the multi-tiered cloud storage services.

2.2 Data Model

Data in Tiera is stored as objects[14] treated as an uninterpreted variable size sequence of bytes that can represent any type of application data, e.g., text files, tables, images, etc. Each object is immutable (i.e., cannot be modified) and can be accessed through a *globally* unique identifier that acts as the *key* to access the corresponding *value* stored. It is up to the application to decide the keyspace from which to select this globally unique identifier. Tiera exposes a simple PUT/GET API to allow applications to store and retrieve data. An object stored into Tiera cannot be edited, though an application can choose to overwrite an object. To support policy specification, Tiera provides several common attributes or metadata for each object such as: size, access frequency, dirty bit, modified time, location (i.e., which storage tier), and last access time. In addition, each object can be assigned a set of tags which enables an application to define object classes (those that share the same tag). The user can then easily specify policies that apply to all objects of a particular class. For example, an application could add a "tmp" tag to temporary file and a policy could dictate that objects with "tmp" tag are stored in inexpensive volatile storage. We have revised the Tiera data model to allow Wiera to manage multiple versions of an object as we will explain in Section 3.2.1.

3. WIERA OVERVIEW

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 data centers.

3.1 Wiera Architecture

Wiera builds on top of Tiera in a geo-distributed setting: a *Wiera instance* consists of multiple Tiera instances running on multiple data centers. While *Tiera* is responsible for managing data on multiple storage tiers *within* a single DC, *Wiera* manages the data placement and movement *across* multiple Tiera instances running on geo-distributed DCs. A Wiera instance simplifies the global data access for applications by hiding the complexities of accessing multiple Tiera instances. Wiera can launch and manage Tiera instances in multiple regions, and can enforce a global data management policy between them, as we will explain more fully in Section 3.3.

Figure 2 shows the Wiera architecture. Wiera consists of the following main components:

- The Wiera User Interface (WUI) provides an API to applications to manage Wiera instances (Table 1). The API allows applications to: launch multiple Tiera instances as part of a Wiera instance with a global policy specification, stop instances, and get the list of currently running instances.

- Global Policy Manager (GPM) creates a new policy for a Wiera instance. It stores metadata for the policy and executes a Tiera Instance Manager (TIM) to manage the Tiera instances which belong to the Wiera instance.

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

We will explain how these components work together in more detail in Section 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. Based on this aggregated information, a data placement manager could generate a dynamic global policy automatically. In this paper, we focus on defining different policies, and such automated policy generation is left as future work.

3.2 Changes and New Features in Wiera

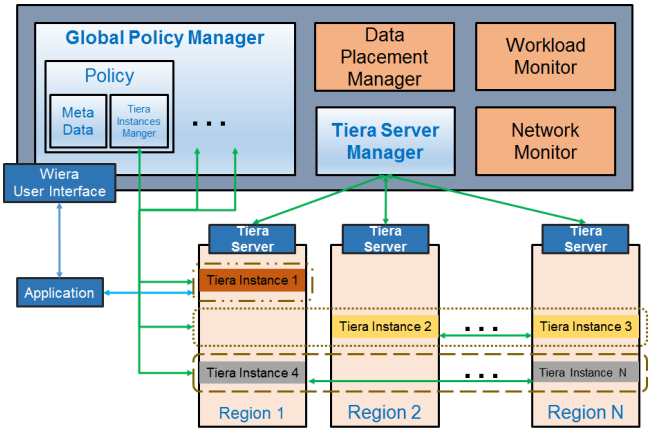

Figure 2: Wiera Architecture.

Table 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

As mentioned above, Wiera manages multiple geo-distributed Tiera instances. In order to handle data across multiple regions, we have extended the Tiera data model and policy mechanisms to support several key requirements: data replication and consistency across multiple locations; load balancing, locality-awareness, and fault tolerance; scalability of existing Tiera stores to multiple DCs; and modular construction of storage containers. In this section, we describe some of the changes and newly added features in Wiera.

3.2.1 Data Model Extension

In order to support low latency and fault tolerance, a Wiera instance can replicate data across multiple locations. Such replication could result in multiple copies of the same data object in different locations, each with a potentially different state depending on the consistency model being used. As mentioned in Section 2.2, an object stored in Tiera is considered immutable, so that it cannot be modified but only overwritten. In order to support replication and consistency control, we have extended the Tiera data model to allow maintaining multiple versions of an object. Thus, modification of an existing object now results in the creation of a *new version* of the object. By default, an application will be provided with an appropriate object version based on its consistency policy (e.g., its local version for eventual consistency, the latest version for a primary-based consistency, etc.). An application can specify the object version number if it needs to access old versions. Old versions of objects will be stored until they are required to be garbage collected in the policy specification. Wiera exposes an object versioning API to applications as shown in Table 2.

Table 2: Wiera Object Versioning API

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

3.2.2 Modular Instances

To provide *scalability and flexibility* to applications, a Tiera instance can specify another Tiera instance as a storage tier. This lets applications easily add pre-defined instances and easily extend Tiera instances to other regions where other Tiera instances are already running. For example, an application launches Tiera instances with a policy id (a) `RAW-BIG-DATA-INSTANCES`, for storing a big data size within a durable and cheap storage tier. Later, the application may launch other Tiera instances with a policy id (b) `INTERMEDIATE-DATA` which encapsulates `RAW-BIG-DATA-INSTANCES` as a read-only storage tier for retrieving raw data and local Memcached as another storage tier to store intermediate data for better performance. This can enable the modular assembly of complex storage containers.

3.2.3 New Events and Responses

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 new *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 requirements), (2) *RequestsMonitoring events* that occur when a Tiera 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 Tiera 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 Tiera instances at run-time to handle workload dynamics.

3.3 Defining Global Policies

The data model extensions as well as the new *events* and *responses* discussed above give more flexibility to applications to specify a number of global data management policies, including many that have been proposed in the literature [22, 3, 1]. In this section, we explain how various global

policies can be specified by showing some examples. Wiera extends Tiera instance specification to define data placement policy *across* multiple Tiera instances. The desired storage tiers, their capacities, and the set of events along with corresponding responses for each instance are now specified using a Wiera policy. The application needs to specify the regions where instances will be running. 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 paper, however, we use the same specification everywhere for simplicity, unless noted otherwise. Further, due to space constraints, we show the specification of *put* operation in our examples, and *get* operation also can be specified similarly.

3.3.1 Data Consistency Policy

We begin by showing how a desired data consistency model between Tiera instances can be easily specified through in a global Wiera policy. Figures 3(a), 3(b), and 4 show three different consistency policies: Multiple Primaries Consistency, Primary Backup Consistency, and Eventual Consistency respectively.

In the MultiPrimariesConsistency policy (Figure 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 services in which strong data consistency is more important than *put* operation performance, e.g., flight booking system and banking system. The figure shows how this policy is implemented using the Wiera events and responses. Here, the same Tiera instances (LowLatencyInstance from Figure 1(a)) are created on multiple regions. When an application puts an object into an instance (normally the closest Tiera instance), it tries to get a *global lock* first for the key as specified in the global 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 1(a). Then it distributes the update to other instances that are part of the same Wiera instance. The lock is released upon getting a response from all other instances.

In the PrimaryBackupConsistency policy (Figure 3(b)), there is only one primary replica. Here, if a Tiera 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 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.

An EventualConsistency policy (Figure 4) is desired for better PUT/GET operations latency, e.g., for social network services like Facebook and Twitter. Here, a *put* operation simply stores the object to the local replica first and then queues the update for distribution to other replicas later in the background. Applications can specify how frequently queued updates need to be distributed. In this consistency

```
Wiera EventualConsistency() {
   ...

   %Eventual Consistency
   event(insert.into) : response {
      store(what:insert.oject, to:local_instance)
      queue(what:insert.object, to:all_regions)
   }
}
```

Figure 4: Eventual consistency policy.

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 4 in more detail.

3.3.2 Defining Dynamic Policies

Since the network, workload, and placement of replicas can be dynamic, 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. While strong consistency is desirable for better user experience, achieving strong consistency can be expensive in a geo-distributed cloud environment due to high WAN latency. For example, in the MultiPrimariesPolicy (Figure 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 instance, consider a policy that maintains strong consistency as long as the latencies are low, but switches to a weaker consistency model such as eventual consistency if the latencies become high. Some shopping web applications (like amazon.com) may get benefits by having different policies, strong consistency for more important data (purchase transactions), eventual (or causal) consistency for browsing data. When all operations can be performed with low latency, strong consistency can be used for all data access. In the high latency case, browsing data can use eventual consistency for better user-perceived latency.

Figure 5(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 runtime 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. 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 arrived at 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 [3]. Figure 5(b) shows how this can be achieved with Wiera for the PrimaryBackupPolicy. 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

```
Wiera MultiPrimariesConsistency() {
   Region1 = {name:LowLatencyInstance, region:US-West,
      tier1 = {name:LocalMemory, size=5G},
      tier2 = {name:LocalDisk, size=5G} }
   Region2 = {name:LowLatencyInstance, region:US-East
      tier1 = {name:LocalMemory, size=5G},
      tier2 = {name:LocalDisk, size=5G} }
   ...
   RegionN = {name:LowLatencyInstance, region:EU-West
      tier1 = {name:LocalMemory, size=5G},
      tier2 = {name:LocalDisk, size=5G} }

   %MultiPrimaries Consistency
   event(insert.into) : response {
      lock(what:insert.key)
      store(what:insert.object, to:local_instance)
      copy(what:insert.object, to:all_regions)
      release(what:insert.key)
   }
}
```

(a) Multiple Primaries consistency policy.

```
Wiera PrimaryBackupConsistency() {
   % Same Tiera instances configuration
   % Primary instance is running on Region1
   Region1 = {name:LowLatencyInstance, region:US-West,
             primary:True}
   ...

   %PrimaryBackup Consistency
   event(insert.into) : response {
      if(local_instance.isPrimary == True)
         store(what:insert.object, to:local_instance)
         copy(what:insert.object, to:all_regions)
      else
         forward(what:insert.object, to:primary_instance)
   }
}
```

(b) Primary Backup consistency policy.

Figure 3: Primary-based consistency policies.

```
Wiera DynamicConsistency() {
   % In Multiple-Primaries Consistency
   % Put operation spends more time than
   % threshold required for specific amount of time
   event(threshold.type == put) : response {
      if(threshold.latency > 800 ms
         && threshold.period > 30 seconds)
         change_policy(what:consistency,
                       to:EventualConsistency);
      else if (threshold.latency <= 800 ms
               && threshold.period > 30 seconds)
         change_policy(what:consistency,
                       to:MultiPrimariesConsistency);
   }
}
```

(a) Changing consistency policy.

```
Wiera ChangePrimary() {
   % In Primary-Backup Consistency
   % If there is an instance which received more
   % requests than primary received from application.
   event(threshold.type == primary) : response {
      if(forwarded_reqeusts_per_each_instance
         >= updates_from_primary
         && threshold.period = 600 seconds)
         chage_policy(what:primary_instance,
                      to:instance_forward_most)
   }
}
```

(b) Changing the primary in Primary Backup policy.

Figure 5: Defining dynamic policies.

more heavily accessed replica. Once this change has been done, all requests will be forwarded to the new primary instance.

3.3.3 *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 consistency policies above to achieve desired latencies in the presence of replication, another important metric could be cost.

Many internet applications see huge 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 [11] 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., MemCache or EBS) for hot data. Figure 6(a) 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 our prior work [15], 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., S3 or EBS). Figure 6(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 Tiera 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 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 achieved in Section 5.3 in more detail.

Using remote storage tiers may induce 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 pub-

```
Wiera ReducedCostPolicy() {
   ...
   RegionN = {name:PersistanceInstance, region:US-West,
      tier1 = {name:LocalDisk, size=5G}}
      tier2 = {name:CheapestArchival, size=5G}}

   %Data is getting cold
   event(object.lastAccessedTime > 120 hours) : response {
      move(what:object.location == tier1
           to:tier2, bandwidth:100KB/s);
   }
}
```

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

```
Wiera SimplerConsistency() {
   Region1 = {name:LowLatencyInstance, region:US-West-1,
             primary:True
      tier1 = {name:LocalMemory, size=30G}
      tier2 = {name:LocalDisk, size=30G}}
   Region2 = {name:ForwardingInstance, region:US-West-2}
   ...
   RegionN = {name:ForwardingInstance, region:US-West-N}

   %PrimaryBackup Consistency
   event(insert.into) : response {
      if(local_instance.isPrimary == True)
         store(what:insert.object, to:local_instance)
      else
         forward(what:insert.object, to:primary_instance)
   }
}
```

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

Figure 6: Achieving desired cost metrics.

lic cloud DC, one could acquire better performance without any network cost as network traffic into DC is normally not charged.

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 and with Tiera servers. We also describe additional features newly implemented in Tiera. To enable communication with applications, Wiera launches a Thrift server [4], a remote procedure call framework, that enables applications written in different languages to communicate with each other. Since Tiera 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 the event-response pairs into the Tiera's control layer. We implemented global policies that were explained in Section 3.3 (under 100 lines of codes written in Java per global policy). Note that Wiera mainly manages Tiera instances and their policies but is not involved in data movement. All data flow happens directly between Tiera instances as specified in the policies.

4.1 Wiera Communication

As described in Section 3.1, Wiera is composed of multiple components. Whenever a Tiera server (note, not a Tiera instance) launches, it connects to the *Tiera Server Manager* (TSM) first to let Wiera know that it is ready to spawn instances. Note: instances run within the Tiera server process for simplicity, but could easily run as a separate process for better fault tolerance. The TSM holds all information about Tiera servers and periodically sends a "ping" message to check on their health. The steps to initiate Tiera 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 *policy id* sent from the application and launches a new *Tiera Instance Manager* (TIM) to communicate with the Tiera instances which will be created, 3) the TSM asks the Tiera servers to spawn instances with storage tiers and local policy as specified in the request, 4) a Tiera server receives the request, spawns a new instance, and informs the instance about the TIM 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 TIM and sends its own server information (port number for the application and port number for communicating between instances), 6) when the TIM accepts server information from its instances, it propagates information to all instances, 7) Wiera returns the list of instances and global policy 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 as in Tiera.

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 MultiPrimariesConsistency (Figure 3(a)), it should get the global lock first for data consistency. For the global lock, Wiera relies on Zookeeper [13], an atomic messaging system that keeps all of the servers in sync, and we use Curator library [8] for using Zookeeper easily. When an instance gets updates from another instance, it will update the object as specified in the global policy. In 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 EventualConsistency (Figure 4), 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. As in Tiera, all object metadata is stored and persisted using BerkeleyDB [16]. 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 has 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.

4.3 New Events and Responses

As mentioned in Section 3.2.3, 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 5(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 6(a), it simply moves the object to the cheaper storage tier as a *response*.

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. 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.

5. EXPERIMENTAL EVALUATION

We evaluated the Wiera prototype in the Amazon cloud and Azure. Wiera and Tiera 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 Tiera 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). Tiera servers are running on multiple regions, US East (Virginia), US West (North California), Europe West (Ireland) and Asia East (Tokyo). The client workloads were generated using Yahoo Cloud Serving Benchmark [6] (YCSB) and our own benchmarks. 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, and (3) Wiera can be easily used with an application without any modification. As mentioned in Section 4, Wiera is not a bottleneck in the data path. The performance overhead introduced by Tiera is very low (under 2%) as shown in the Tiera paper [18].

5.1 Changing Consistency

In this section, we show how Wiera changes consistency policy dynamically as specified in the *DynamicConsistency* policy (Figure 5(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 [6].

We set instances to use *MultiplePrimariesConsistency* (Figure 3(a)) initially in which all *put* operations result in updates being distributed to all other instances synchronously. Figure 7 shows the latency for *put* operations in US West region[3]. 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 sec). For delays (a) and (b), Wiera detects that both thresholds are violated, so it changes the consistency to *EventualConsistency* (Figure 4) to preserve application-perceived *put* operation latency which now becomes less than 10 ms. This is because instances don't need to get the global lock for the key and broadcasting updates is done in the background in *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 *MultiplePrimariesConsistency*. This result shows that Wiera can adaptively change consistency models to handle dynamics at run-time.

5.2 Changing Primary Instance

User location is another factor that may be important for data placement policy as shown in systems such as Volley [1] and Tuba [3]. Tuba shows that changing 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

[3]Note that we see a similar pattern of results from all regions, and we omit these results due to space constraints.

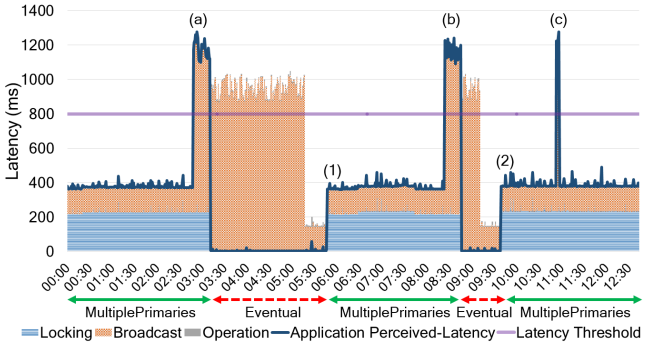

Figure 7: Changing consistency at run-time.

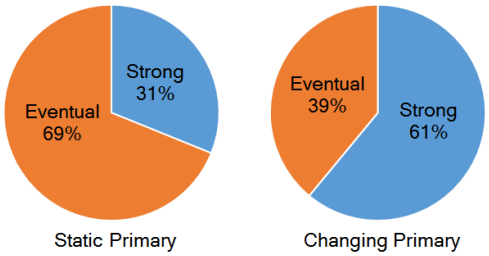

Figure 8: Percentage that applications can see the latest data (Strong) and outdated (Eventual) data.

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 A: Read mostly workload (5% put and 95% get) in YCSB [6]. We use the *queue response* mentioned in Section 3.3.2 to distribute updates asynchronously to other instances as Tuba does. We implement the Wiera ChangePrimary policy (Figure 5(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 find an instance which forwards more requests than the primary instance received from clients. We set the time period threshold to 15 seconds.

Figure 8 shows the chance that the clients will see the latest data (Strong) and outdated data (Eventual). With a static (no 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 3 shows that an average *put* operation time for each region and overall average of all regions. With static

Table 3: Average put operation latency (in 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

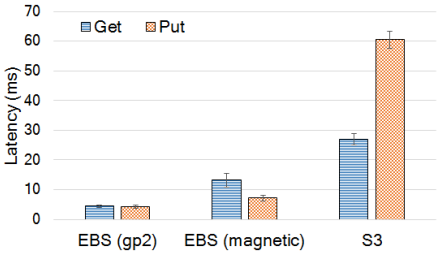

Figure 9: Operations Latencies for 4KB in US East.

primary location, the clients in Asia East can see low latency (<5ms) 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.

5.3 Reducing Cost Using Multiple Storage Tiers

Many internet services and applications have reported that their data access pattern follows Zipfian distribution e.g., Facebook [11], that is huge portion of data is accessed 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 3.2.3.

Within a DC, an application has various durable storage options with different performance. Figure 9 shows the latencies that the application can see from each storage tier through a Tiera instance. Table 4 shows the prices for provisioned storage, put/get requests, and network cost. 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

Table 4: Storage Tiers' Price in AWS (US East)

	EBS (SSD)	EBS (HDD)	S3	S3-IA	Cost/Unit
Storage	$0.1	$0.05	$0.03	$0.0125	GB/Month
Put req	$0	$0.0005	$0.05	$0.1	10,000 reqs
Get req	$0	$0.0005	$0.004	$0.01	10,000 reqs
Network	$0	$0	$0	$0	GB/Within a DC
Network	$0.09	$0.09	$0.09	$0.09	GB/To Internet

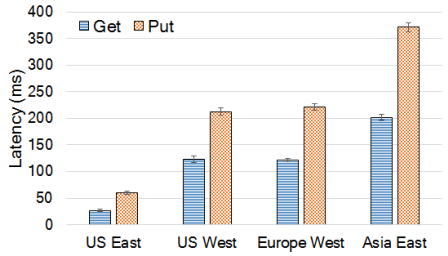

Figure 10: Operation Latency for S3 in US East from each region.

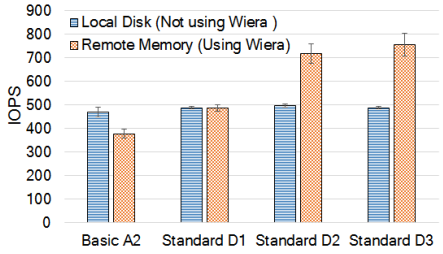

Figure 11: Performance (IOPS) comparison.

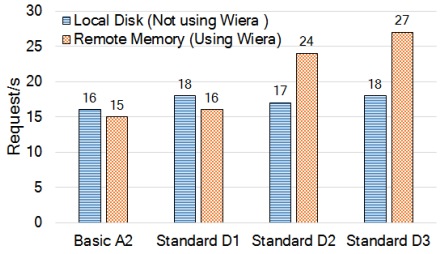

Figure 12: Throughput (request/s) comparison.

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 (<1ms) 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 sees that 80% out of 10TB data in EBS have not been accessed for 120 hours, in Figure 6(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 10 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 4. However, by definition, the access to cold data will be rare.

5.4 Exploiting Remote Storage Tiers

One of the benefits of Wiera is that it increases the range of storage tier options. In our previous work [15], 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 both a benchmark (SysBench) and a real application case (RUBiS). Note that we have built our own POSIX-compliant file system using *Filesystem in User Space* (FUSE) [12] 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*.

5.4.1 Better Performance from Non-Local DC

In this experiment, we compare I/O performance between Azure's local disk without Wiera and AWS' memory with Wiera using SysBench [21], a system performance benchmark. 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 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 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. We use *PrimaryBackup*

consistency policy (Figure 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 11 shows results for each VM size. For the local disk performance, the figure shows the same performance ($\sim$ 500 IOPS) regardless of VM size. This is because Azure throttles the disk performance to 500 IOPS [4]. 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. We think that this is because Azure throttles the network performance between instances based on VM type and size as we have shown in our previous work [15]. These results 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.

5.4.2 RUBiS on Wiera

We next explore running an *unmodified* web application, the popular open-source benchmark RUBiS [20], on Wiera. RUBiS is a multi-component web application that implements 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 5.4.1. 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 12 shows the throughput from each VM size. Similar to the SysBench results, 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.*

6. RELATED WORK

Data Locality: Recent research [2] 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 our previous work [15], 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.

In-Memory Storage: Many previous works utilize memory to improve performance. Cooperative Caching [9] tries to use idle remote node's memory to improve file system performance. Many recent storage systems, like Redis [19] and RAMCloud [17] aggregate memory resources from many nodes and present it as a common storage pool to applications. In this paper, we show that *unmodified* applications can get a performance benefit from these in-memory storage systems even in multi-DC environment through Wiera.

Wide Area Storage: Many previous storage systems utilize multiple DCs. Volley [1] performs automated data placement across distributed DCs using diurnal and weekly users' data access patterns to reduce user perceived-latency and to minimize costs associated with inter-DC traffic. Spanner [7] manages cross-data center replicated data and implements database operations while maintaining externally-consistent distributed transactions for their internal applications. These systems use a single cloud storage provider. In contrast, SPANStore [22] tries to utilize multiple cloud provider DCs rather than a single provider to get a higher DC density to deliver data closer to users with reduced cost, much like a content delivery network. Tuba [3] tries to achieved applications' desired goals while maximizing the utility delivered to read operations. 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 e.g., Wiera can support a time-varying user-specified data consistency model based on changes to access patterns and network conditions while most previous works only support a few hard-coded data consistency models.

Policy-Driven Storage: The policy architecture for distributed storage systems (PADS) [5] was proposed for system designers to construct a new distributed storage system easily. PADS provides a data plane that is a fixed set of mechanisms for storing, transmitting, and consistency information and control plane policy that specifies the system-specific policy for orchestrating flows of data among nodes. In our previous work Tiera [18], we explored building

a storage framework that helps applications build a tiered storage system consisting of local DC memory resources for better performance, and persistent storage services like S3 or EBS for durability. Tiera supports dynamic policy modification e.g., addition/removal of tiers, adding new events and responses, at run-time. In this paper, we extend and utilize Tiera which focused on a single DC. We use 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.

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. Moreover, cloud providers don't provide guarantees on the consistent performance of their services and it is left to the applications to handle dynamics. Wiera makes this much easier for applications through a simple interface and support for changing policies.

7. CONCLUSION

In this paper, 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.*

8. ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd, Thilo Kielmann, for their helpful comments. We acknowledge grant NSF CSR-1162405 that supported this research.

9. REFERENCES

[1] S. Agarwal et al. 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.

[2] G. Ananthanarayanan et al. 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.

[3] M. S. Ardekani and D. 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.

[4] Azure Virtual Machine. http://alturl.com/vzmuw/.

[5] N. Belaramani et al. 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.

[6] B. F. Cooper et al. 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.

[7] J. C. Corbett et al. Spanner: Google's globally distributed database. volume 31, pages 8:1–8:22, New York, NY, USA, Aug. 2013. ACM.

[8] Curator. http://curator.apache.org/.

[9] M. D. Dahlin et al. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

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

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

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

[13] P. Hunt et al. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

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

[15] K. Oh et al. 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.

[16] M. A. Olson, K. Bostic, and M. 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.

[17] J. Ousterhout et al. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

[18] A. Raghavan, A. Chandra, and J. 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.

[19] Redis. http://redis.io/.

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

[21] sysbench: multi-threaded system evaluation benchmark. https://github.com/akopytov/sysbench/.

[22] Z. Wu et al. 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.