Tiera: Towards Flexible Multi-Tiered Cloud Storage Instances[*]

Ajaykrishna Raghavan, Abhishek Chandra, and Jon B Weissman
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
{raghavan,chandra,jon}@cs.umn.edu

ABSTRACT

Cloud providers offer an array of storage services that represent different points along the performance, cost, and durability spectrum. If an application desires the composite benefits of multiple storage tiers, then it must manage the complexity of different interfaces to these storage services and their diverse policies. We believe that it is possible to provide the benefits of customized tiered cloud storage to applications without compromising simplicity using a lightweight middleware. In this paper, we introduce *Tiera*, a middleware that enables the provision of multi-tiered cloud storage instances that are easy to specify, flexible, and enable a rich array of storage policies and desired metrics to be realized. Tiera's novelty lies in the first-class support for encapsulated tiered cloud storage, ease of programmability of data management policies, and support for runtime replacement and addition of policies and tiers. Tiera enables an application to realize a desired metric (e.g., low latency or low cost) by selecting different storage services that constitute a *Tiera instance*, and easily specifying a policy, using *event* and *response* pairs, to manage the life cycle of data stored in the instance. We illustrate the benefits of Tiera through a prototype implemented on the Amazon cloud. By deploying *unmodified* MySQL database engine and a TPC-W Web bookstore application on Tiera, we are able to improve their respective throughputs by $47\% - 125\%$ and $46\% - 69\%$, over standard deployments. We further show the flexibility of Tiera in achieving different desired application metrics with minimal overhead.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Storage hierarchies*; H.3.2 [**Information Storage And Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms

Management, Performance

1. INTRODUCTION

Many cloud providers today offer an array of storage services that represent different points along the performance, cost, and durability spectrum. As an example, Amazon provides ElastiCache (a caching service protocol compliant to Memcached), Simple Storage Service (S3), Elastic Block Store (EBS), and Glacier as different cloud storage options[1]. A single service generally optimizes one metric trading off others. For example, Amazon ElastiCache offers low latency, but at high cost and low durability. Amazon S3 offers high durability and low cost but low performance. If the application is willing to use multiple cloud storage services, then it can realize composite benefits. For example, an application that requires low latency reads as well as durability, might choose to use a combination of Amazon ElastiCache and Amazon EBS, with most frequently accessed data being stored in ElastiCache and the rest in the EBS persistent store.

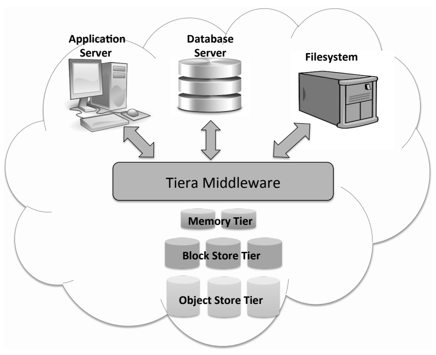


Figure 1: Tiera middleware enables applications to easily use multiple tiers to realize composite benefits

Accessing multiple tiers introduces significant complexity to the application. The application has to not only deal with different interfaces and data models of the storage, but at the same time, has to program policies to manage data across the different storage services to realize the desired metric. For example, two open source web applications that use multiple storage services when deployed in the cloud,

[*]This work was supported by NSF Grant CSR-1162405.

Middleware '14, December 08 - 12 2014, Bordeaux, France
Copyright 2014 ACM 978-1-4503-2785-5/14/12 ...$15.00
http://dx.doi.org/10.1145/2663165.2663333.

[1]Note that we use Amazon as an illustrative example in this paper, but similar issues also apply to other cloud providers.

WordPress [28] and Moodle [17], have $1000 - 5000$ additional lines of code to manage data across different storage tiers (WordPress across Memcached and S3, Moodle across Memcached, Local Disk, and MongoDB). The popular open source database, MySQL [18] has over 6000 lines of code to support memory and S3 as storage media. Hence we see that an application must make a choice: opt for simplicity by using one or a small number of storage tiers and live with the tradeoffs, *or* embrace complexity – be willing to use different interfaces, decide on the appropriate capacity for each kind of storage, and manage data placement and other policy requirements explicitly. We believe this is a false choice.

In this paper, we present *Tiera*, a middleware that enables the provision of flexible and easy-to-use multi-tiered cloud storage instances. A *Tiera instance* encapsulates multiple cloud storage services and enables easy specification of a rich array of data storage policies to achieve desired tradeoffs. The client of a Tiera instance is shielded from the underlying complexity introduced by the multi-tiered cloud storage services, and specifying a Tiera instance is simple and straightforward as we will illustrate. The novelty of Tiera lies in the first-class support for encapsulated tiered storage, ease of programmability of data management policies, and the ability to dynamically replace/add storage policies and tiers. In the long run, we envision Tiera instances that span a cloud and edge resources or multiple clouds (public or private) or data centers. Here, we focus on a single cloud implementation.

The key contributions of this paper are:

- We present the design and implementation of *Tiera*, a lightweight middleware that enables easy specification of multi-tiered cloud storage instances.
- We show how Tiera can support a rich array of storage policies to realize a desired metric (e.g., low latency or low cost) through a powerful event-response mechanism.
- We demonstrate the benefit of Tiera by deploying two *unmodified* applications, MySQL and the online bookstore application bundled with the TPC-W benchmark on Tiera in the Amazon cloud, yielding an increase in their respective throughputs by $47\% - 125\%$ and $46\% - 69\%$, over standard deployments.

The rest of the paper is organized as follows. Section 2 provides an overview of Tiera. This section also provides examples of Tiera instance specifications to demonstrate its power, flexibility, and ease-of-use. Section 3 explains the implementation details of a Tiera prototype, implemented in the Amazon cloud. Section 4 discusses the results of our experimental evaluation. The results demonstrate how an application can use primitives provided by Tiera to realize the composite benefits of multiple storage services without changes to the application logic itself. Section 5 describes related work. Section 6 concludes the paper and describes possible future research directions.

2. TIERA OVERVIEW

2.1 Data Model

Tiera implements 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 that constitute a Tiera instance. An object stored using Tiera can be accessed by the application using a globally unique identifier that acts as the *key* to access the corresponding *value* stored (as an object). It is left to the application to decide the keyspace from which to select this globally unique identifier. Tiera exposes a simple `PUT/GET` API to the application to store and retrieve data. An object stored into Tiera cannot be edited, though an application can choose to overwrite an object.

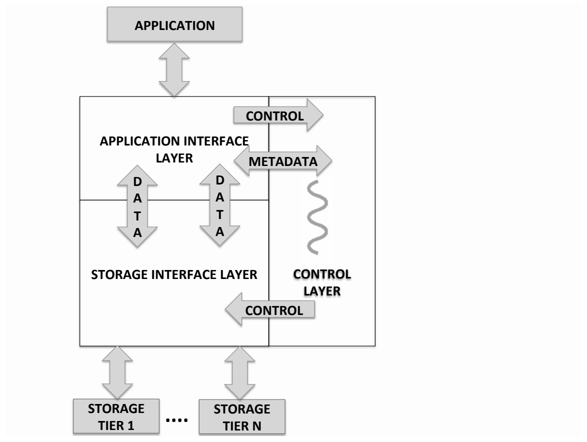


Figure 2: Three layers of a Tiera server

Tiera treats objects stored within it as an uninterpreted sequence of bytes that can be of variable size and represent any type of application data, e.g., text files, tables, images, etc. Tiera tracks the common attributes or metadata for each object: size, access frequency, dirty flag, location (i.e. which tiers), and time of last access. In addition, each Tiera object may also be assigned a set of tags. Tags are stored as part of object metadata and provide a method to add structure to the object name space. It 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. Tags may also be used to pass application "hints" to Tiera. For example, an application could add a "tmp" tag to temporary files and a policy could dictate that objects with "tmp" tag be stored in inexpensive volatile storage.

2.2 Architecture

An application interacts with Tiera by specifying the different cloud storage *tiers* it desires to use and their respective initial capacities. The application also specifies a policy that governs the life cycle of data stored through Tiera. A *tier* can be any source or sink for data with a prescribed interface. The storage tiers along with the *Tiera server*, constitute a *Tiera instance*. Once configured, the application interacts with the *Tiera instance* to store and retrieve data. It can also interact with the instance to alter its configuration and the governing policy.

The Tiera server has three primary roles: (1) to interface with applications to enable storage and retrieval of data, (2) to interface with different storage tiers to read/write data to them, and (3) to manage the data placement and movement across different tiers. These roles are performed by three layers: the application interface layer, the storage interface layer, and the control layer respectively (Figure 2).

The *application interface layer* exposes a simple `PUT/GET` API that allows an object to be placed/retrieved with respect to the Tiera instance. The client can merely call `PUT/GET` and let the Tiera server decide in which tier the object should be placed/retrieved (e.g., in the Amazon context: Memcached, S3 bucket, EBS volume, and so on) based on the control layer. Note that it is also possible to support the traditional POSIX file system interface to Tiera. We have developed such an interface using *Filesystem in User Space* (FUSE) [10] to run applications that require a POSIX interface to Tiera.

The *storage interface layer* interfaces with the storage services encapsulated by the Tiera instance. The Tiera server uses service-specific APIs to interact with the different storage tiers to carry out different operations such as object storage/retrieval, moving data across tiers, and resizing the storage tiers.

The *control layer* decides how data is to be placed and managed throughout the Tiera instance lifecyle. It provides two primary mechanisms—*event* and *response*—to manage the 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. Events can be combined such that a particular response is initiated only when all the conditions hold, and similarly multiple responses can be associated with a single event. These two mechanisms together form the primary building blocks for data management policies in Tiera. Events may be defined on individual named objects or object classes, the latter allowing a single policy to apply to object collections (sharing a common tag) and differential policies to be easily specified.

```
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);
 }
}
```

Figure 3: LowLatency Tiera instance

Tiera supports three different kinds of *events*: (1) *timer events* that occur at the end of a specified time period, (2) *threshold events* that can be based on attributes of data objects and of the tiers themselves, and occur when the value of the attribute reaches a certain value, and (3) *action events* that occur when actions such as data insertion or deletion are performed. Table 1 shows some of the responses currently supported by our Tiera implementation: `store` data in a tier, `retrieve` data from a tier, `move` data between tiers, `copy` data from one tier to another, and `delete` data in a tier. Tiera also supports advanced responses: `storeOnce`, `grow/shrink` , `compress/uncompress`, and `encrypt/decrypt`. Tiera's design is highly modular making it very easy to add a new response. Other responses will be added to Tiera in the future to support transactions, data snapshotting, and object versioning. In the next section (2.3) we will show how a rich array of data management policies can be easily constructed using these event-response mechanisms.

Response	Arguments	Function
`store`	Objects, Tiers	Stores objects in the specified tiers.
`storeOnce`	Objects, Tiers	Stores objects in the specified tiers. An object is stored only if its content is unique.
`retrieve`	Objects	Retrieves objects from an underlying tier.
`copy`	Objects, Destination Tiers, Bandwidth Cap	Copies objects to the specified tiers. Transfer speeds are throttled if a bandwidth cap is specified.
`encrypt`	Objects, Key	Encrypts objects with the specified key.
`decrypt`	Objects, Key	Decrypts objects with the specified key.
`compress`	Objects	Compresses the specified objects. The ZLIB compression library is used to perform compression.
`uncompress`	Objects	Inflates the specified objects.
`delete`	Objects, Tiers	Deletes objects from the specified tiers.
`move`	Objects, Destination Tiers, Bandwidth Cap	Moves objects to the specified tiers.
`grow`	Tier, Percent Increase	Expands tier capacity by the specified percentage.
`shrink`	Tier, Percent Decrease	Reduces tier capacity by the specified percentage.

Table 1: Supported responses in Tiera

2.3 Defining Tiera Instances

Tiera instance configuration, including policies are specified through an instance specification file. The instance specification provides the desired storage tiers to use, their capacities, and the set of events along with corresponding responses to be executed. An application realizes the tradeoffs it desires by (i) selecting different storage tiers that constitute the instance, and (ii) specifying the event-response pairs used to define the policy.

For example, consider the following specification for a Tiera instance called `LowLatencyInstance` (Figure 3). This instance uses two storage tiers and an event that specifies that a data object is to be placed in *tier1* (Memcached) when it is inserted into this Tiera instance (with `PUT`). The instance also implements a *write-back* policy by combining a *timer event* with the `copy` response, to write out any dirty data (i.e data added or modified in a tier since the last copy) to the persistent store at regular time intervals. It is assumed that the specific tier names (e.g. Memcached and EBS) are known to Tiera.

Tiera enables rich policies that can be specified easily to

```
Tiera PersistentInstance() {
 tier1: { name: Memcached, size: 200M };
 tier2: { name: EBS, size: 1G };
 tier3: { name: S3, size: 10G};

 % write-through policy using action event
 % 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);
 }
}
```

Figure 4: Persistent Tiera Instance

```
 % LRU Policy
 event(insert.into == tier1) : response {
    if (tier1.filled) {
     % Evict the oldest item to another tier
        move(what: tier1.oldest, to: tier2);
    }
    store(what: insert.object, to: tier1);
 }

 % MRU Policy
 event(insert.into == tier1) : response {
    if (tier1.filled) {
     % Evict the newest item to another tier
        move(what: tier1.newest, to: tier2);
    }
    store(what: insert.object, to: tier1);
 }
```

Figure 5: Implementing LRU and MRU in Tiera

realize a desired tradeoff (e.g., latency vs. cost or durability). For instance, for the LowLatencyInstance, by choosing to read/write data from/to Memcached, low latency will be achieved but at high monetary cost and reduced data durability. If an application desires better data durability, it could specify a smaller time value t for data write-back.

As another example, the PersistentInstance (Figure 4) trades performance for better data durability. This instance uses a small Memcached tier to cache the most recently written data. It implements a *write-through* policy between *tier1* (Memcached) and *tier2* (EBS). A write-through policy can be specified using an *action event* with a copy response that causes an object to be inserted into *tier2* as soon as it is inserted into *tier1*.

Tiera also allows easy specification of object placement and caching policies through the use of object attributes such as access frequency and time of last access. For example, access frequency can be used for easy specification of *hot* and *cold* objects. Similarly, time of last access can be used to identify *old* and *new* objects, making it simple to implement *LRU* or *MRU* eviction policies in Tiera, as shown in Figure 5.

The event-response framework also allows for dynamic modification to the instance configuration to respond to changing workload. For example, consider a scenario where an application is using the PersistentInstance (Figure 4)

```
Tiera GrowingInstance(time t) {
   tier1: { name: Memcached, size: 200M };
   tier2: { name: EBS, size: 2G };

   % Placement Logic
   event(insert.into) : response {
         store(what: insert.object,
               to: tier1);
   }

   % Growing with workload, add as much Memcahed
   % storage as its current size everytime the tier
   % tier is 75% full
   event(tier1.filled == 75%) : response {
         grow(what: tier1, increment: 100%);
    }
   }

   % write-back policy
   event(time=t) : response {
         move(what: object.location == tier1,
              to: tier2);
   }
}
```

Figure 6: Expanding a tier

to store data, but its *working set* size is about to exceed $200MB$. To handle this increase in working set, a *threshold event* can be added to the instance specification to grow the Memcached tier when the amount of stored data reaches a cap. The event-response specification for doing this is illustrated in Figure 6.

3. TIERA IMPLEMENTATION

We now describe our implementation of a prototype Tiera server (which is under 4000 lines of code) in the Amazon public cloud using the following storage tiers: Memcached, Ephemeral Storage (Amazon EC2 local volumes), Amazon EBS, and Amazon S3. The Tiera server is deployed as a Thrift server [4] on an EC2 instance (can be co-located with the application on the same EC2 instance). Thrift is a remote procedure call framework, that enables applications written in different languages to communicate with each other. The use of Thrift makes it easy to interface applications written in different languages with Tiera. When the server starts up, it begins by reading the configuration file that is used to indicate the different tiers (and their capacities) that would constitute the instance, the size of the thread pool dedicated to service client requests, the size of thread pool dedicated to service responses and evaluate events, and the location to persistently store metadata and credentials for an Amazon Web Services account. All object metadata is stored and persisted using BerkeleyDB [19]. Once the tiers to be used are established, and the two thread pools and the metadata store are initialized, the instance is ready to serve client requests. When the instance receives a client request, it is serviced by a thread from the thread pool dedicated to service user requests. The thread servicing the PUT/GET requests takes an appropriate action, as dictated by the policy programmed on the instance.

The prototype supports the three types of events mentioned previously: *action*, *threshold*, and *timer*. The prototype currently supports all the responses listed in Table 1. A desired policy is implemented in the instance by hand-coding the event-response pairs into the control layer. Automated compilation and optimization of specification files will be addressed in future work. We next describe how different events are implemented in the prototype.

Timer events are handled by a dedicated thread in the control layer. This thread is responsible for examining if a *timer* event has occurred. Once this thread determines that an event has indeed occurred, it signals a free thread (part of the thread pool mentioned) to service the event by executing the response associated with the particular *timer* event. The original thread continues to check the occurrence of other *timer* events. At present Tiera allows *timer* events to be specified at the granularity of seconds.

Threshold events can be specified as *background* or *foreground* (default is *foreground*). Background events are evaluated by threads when actions effecting a variable on which the threshold is defined occur. For example, consider a threshold event being defined on the amount of data stored in a tier. Two actions effect this variable directly, (1) storing new data in the tier, and (2) deleting data stored in the tier. Both actions trigger the *threshold* event to be evaluated and check if the defined threshold has been reached. Background events are evaluated asynchronous to the actions mentioned and must be explicitly declared as such. Foreground events are evaluated synchronously and are presumed to be the default.

Action events are generally foreground events and are evaluated in the context of the thread servicing a client request. Responses associated are required to be fast since they effect latency of data access. If a slow response needs to be associated with an *action* event then it should be specified as a *background* event. The occurrence of the event will cause a thread to be signalled, which would wake up and service any response associated with the the *action* event.

4. EXPERIMENTAL EVALUATION

We evaluated the Tiera prototype in the Amazon cloud. The Tiera instance containers and the clients were hosted on EC2 instances. For our experiments we used EC2 t1.micro instances–1 ECU, 615 MB of RAM, and 8GB of EBS storage, and EC2 m3.medium instances–3 ECU, 3.75 GB of RAM, and 8GB of EBS storage to host the Tiera instances. The client workloads were generated using a combination of benchmarking tools: sysbench [23], TPC-W [24], Yahoo Cloud Serving Benchmark [7] (YCSB), fio [9], and our own benchmarks. These benchmark tools were themselves run on an EC2 t1.mirco, or m3.medium instances and all measurements were made from these clients running in the Amazon cloud (i.e., no wide-area latency). Our experiments illustrate: (1) how easy it is to run *unmodified* applications on Tiera, and provide them the composite benefits of multiple storage tiers, (2) how Tiera enables an application to optimize for a particular metric, and (3) the minimal overhead introduced by Tiera.

4.1 Case Study: MySQL and TPC-W On Tiera

In this section we present our experience running two applications on Tiera–(1) MySQL – a popular opensource database management system (DBMS), and (2) an online bookstore application (bundled with the TPC-W benchmark). We were able to run both applications on Tiera *without any modifications to the applications* themselves. Running these applications on Tiera, we are able to offer them composite benefits of using multiple tiers.

4.1.1 MySQL On Tiera

MySQL [18] is a popular open source DBMS used by many applications. Within the Amazon cloud, MySQL is typically deployed on an EBS volume (persistent block store) attached to an EC2 instance. This deployment performs reasonably well when the amount of data accessed isn't very large or when there aren't many concurrent requests, and so requests can be served from the local instance's buffer cache or MySQL's built-in caches. However, the throughput drops significantly and the response latency increases when data requested by the client can no longer be served from these caches. Hence, many techniques have been explored to maintain the throughput level and keep the response latency bounded. One such technique is to store the database completely in memory. MySQL has a special storage engine called *Memory Engine* that stores databases completely in memory. However this technique only works for non-transactional workload and when the database can completely fit in the node's memory. Also storing the entire database in a single node's memory makes the deployment vulnerable to failure[2].

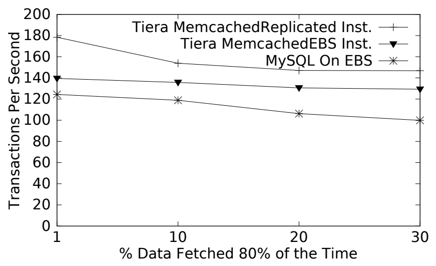
Another common technique is to modify the end application such that it caches the results of a database access in a memory storage system like Memcached [15]. When the application uses other storage services like Memcached to store database results, it has to deal with additional complexities such as being able to scale up and scale down the storage service with a change in the workload.

Apart from the limitations mentioned above, we see that either MySQL needs to be modified heavily (the *Memory Engine* implementation is 4000 lines of code) or the end application needs to be modified to optimize for performance. Here, we explore the possibility of running **unmodified** MySQL on Tiera to overcome the limitations mentioned above. Using Tiera also enables a MySQL deployment to easily optimize other metrics such as cost or reliability. The benefits of the optimizations could be passed to the end applications, without the applications having to manage multiple storage services and deal with the associated complexities.

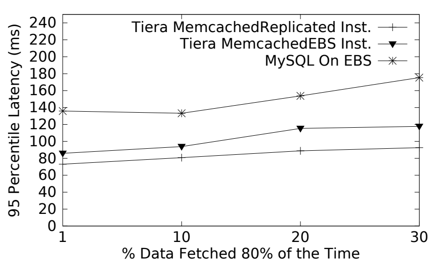
Performance Optimization:
For this experiment, we used the unmodified MySQL Community Edition [18] version 5.7. We hosted MySQL on an m3.medium EC2 instance. We generated OLTP workload using sysbench. We hosted the benchmark tool itself on a separate t1.micro instance. The OLTP workload followed the *special* distribution, that is a certain percentage of the data is requested 80% of the time. We varied this percentage of data requested from 1% to 30%. We also varied the concurrency of the workload. We first ran MySQL on a non-root EBS volume attached to the m3.medium EC2 instance, which is a standard way to deploy MySQL in the cloud. We then deployed MySQL on two different Tiera instances (described below) and subjected them to the same workloads. And last, we subjected the MySQL *Memory Engine* to similar workloads.

For the following experiment we defined two Tiera instances – `MemcachedReplicated` and `MemcachedEBS`. The `MemcachedReplicated` instance consists of two Mem-

[2]For better fault tolerance it is required to run MySQL in a special cluster mode which implies additional costs.

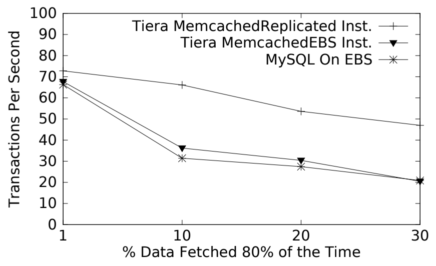


(a) Throughput

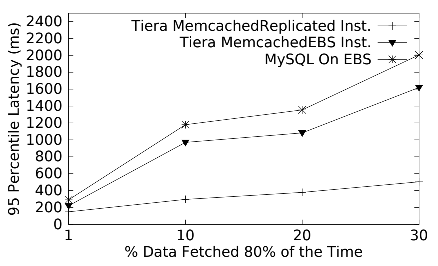


(b) 95 Percentile Response Latency

Figure 7: Throughput and 95 Percentile Response Latency For Read-Only Workload With 8 Threads



(a) Throughput



(b) 95 Percentile Response Latency

Figure 8: Throughput and 95 Percentile Response Latency For Read-Write Workload With 8 Threads

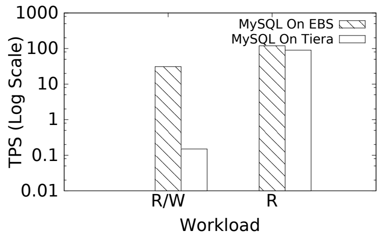
cached tiers: one Memcached tier in the same availability zone as the client and the other in a separate availability zone in AWS[3]. We defined a simple data management policy for this instance: on a `PUT` to the instance the data is written to both tiers before being acknowledged. This replication of data provides better fault tolerance than having just one copy in memory as in the *Memory Engine*. The `GET` request is served from the Memcached tier in the same availability zone as the client. The `MemcachedEBS` instance consists of two tiers as well: a Memcached tier and an EBS tier. The data management policy for this instance involved writing data to both the Memcached and EBS tier on `PUT` and serving data from Memcached for `GET`. The instance specification files for both these instances are under 15 lines each (in contrast to nearly 4000 additional lines of code needed to support MySQL directly over memory). Both these instances had Memcached tiers large enough to fit the entire database in memory. The `MemcachedEBS` instance has a lower cost of storage per GB compared to the `MemcachedReplicated` instance, since it has a lesser amount of Memcached storage. Since we need to provide a POSIX interface to MySQL, we used the FUSE filesystem interface we developed to interface MySQL with the Tiera instances. The FUSE filesystem we developed splits the database files into 4 KB objects (OS page size) and stores them in Tiera.

In Figures 7 and 8, we plot the throughput in terms of transactions per second and the 95 percentile response latency for read-only and read-write workloads with 8 threads. We see that for both read-only and read-write workloads the `MemcachedReplicated` instance performs the best, supporting the highest throughput and providing the lowest response latencies. The MySQL deployment on the Tiera `MemcachedReplicated` instance provides a 125% increase in throughput compared to the standard MySQL deployment on EBS for read-write workloads, and an increase in throughput of 47% for read-only workload. The increase in throughput is less for read-only workload due to the caching of data in the buffer cache of the EC2 instance. The `MemcachedEBS` instance provides similar performance levels as the `MemcachedReplicated` instance for read-only workloads, at a fraction of the cost. Note that even in a purely read-only transactional workload MySQL performs writes to its journal, which have to be persisted on to EBS in the `MemcachedEBS` instance (these writes have to be performed on both Memcached tiers in case of the `MemcachedReplicated` Tiera instance). These writes to EBS result in a lower performance for the `MemcachedEBS` instance compared to the `MemcachedReplicated` instance. We see that for a read-write workload the performance of the `MemcachedEBS` instance resembles that of MySQL running over EBS, due to writes to EBS (acting as a performance bottleneck). The MySQL deployment on Tiera `MemcachedEBS` instance provides a 29% increase in throughput over the standard MySQL deployment on EBS for the read-only workload, but their throughputs are nearly equal for the read-write workload.

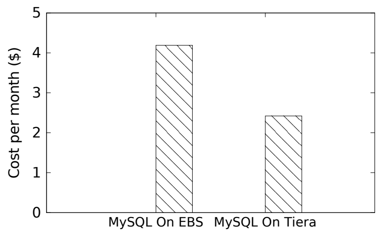
The experiment with MySQL *Memory Engine* yielded a throughput of ≈ 0.15 TPS for the different workloads. This is because the MySQL *Memory Engine* doesn't support transactions and only supports table level locks. Apart from offering poor performance the fault tolerance of this deployment is also poor since data is stored in a single node's memory. For better fault tolerance MySQL would have to be deployed in replicated *cluster mode*. This would imply a significant increase in the cost of deployment.

Cost Optimization:
To show the flexibility of Tiera in supporting cost opti-

[3]Availability zones are isolated locations (i.e independent fault domains) connected via low latency links in the same geographic area.



(a) Throughput



(b) Cost of storage

Figure 9: Throughput measured in TPS (log scale) and the total cost of storage per GB

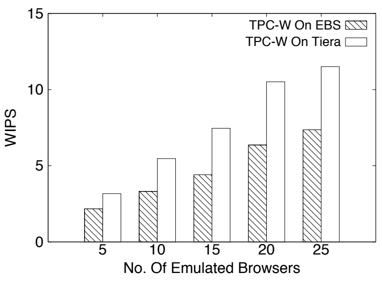


Figure 10: Average Throughput using TPC-W

mization, we next deployed MySQL on a Tiera instance, `MemcachedS3` that uses Memcached and S3 tiers. In contrast to the two Tiera instances described previously, the goal of this instance is to reduce the cost of deployment. The Memcached tier in this instance wasn't large enough to store the entire database. Portions of the database are cached in the Memcached tier using an LRU policy to manage the tier. We subjected this instance to read-only and read-write transactional workload with 10% of the data requested 80% of the time with 8 threads. In Figure 9, we compare the throughput and also the total cost of storage[4] for using MySQL on EBS against using *unmodified* MySQL on the `MemcachedS3` Tiera instance. We see that the deployment on the Tiera instance costs a fraction of the cost of deployment on EBS, and still provides comparable performance for a read-only workload, while sacrificing performance for the read-write workload. We note that the application can choose an appropriate tradeoff between cost and performance by easily varying the size of the Memcached tier in the Tiera instance.

4.1.2 TPC-W On Tiera

We next explore running an *unmodified* web application (end-to-end) on Tiera. For our experiment, we use an online bookstore application that comes bundled with an implementation of the TPC-W [12] benchmark. The online bookstore application uses MySQL on its backend and serves dynamic and static content through Apache Tomcat web server. In a typical deployment both the database files and the static HTML files and images would be stored in a EBS volume attached to an EC2 instance. This, as mentioned previously, provides poor throughput when data requested by the clients cannot be served from the local caches on an EC2 instance.

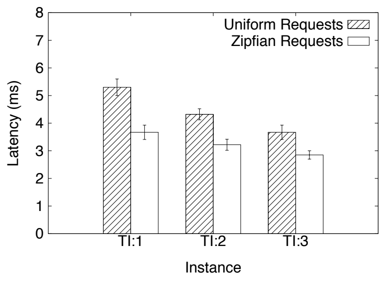
We deployed the online bookstore application on EBS and on the `MemcachedEBS` Tiera instance. Running the application on the Tiera instance involved storing the database records as well as the HTML pages and images served by the web server on the instance. The objective was to see if this could help improve the performance of the online bookstore application without having to change the application logic in any way.

We measured the performance of the two deployments using the TPC-W benchmark. The TPC-W benchmark is designed to exercise the web server and transaction processing system of the online bookstore application. The primary metric the benchmark measures is the throughput in terms of web interactions per second (WIPS). Web interactions performed by the benchmark range from simple requests for static content to interactions which require significant server side processing. These interactions are performed by emulated browsers. There are three different mixes of interactions that the benchmark supports, with different percentages of reads and writes. We used the shopping mix that is read dominant and also emulates typical shopping scenarios.

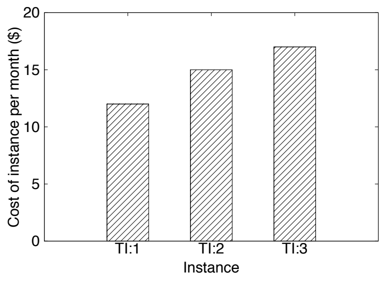
For our experiment we hosted the online bookstore on an m3.medium EC2 instance. The web and database servers were co-located on the EC2 instance. We reduced the amount of available memory on the EC2 instance to 1 GB by setting a boot time flag. This was done to ensure both MySQL and the web server performed sufficient IO and didn't serve all requests from the instance's buffer cache. The database was populated with information for $10,000$ items and $100,000$ customers. The emulated browsers were hosted on a separate m3.medium EC2 instance. The benchmark was run for 600 seconds, with 100 seconds each for ramp-up and ramp-down, for different number of emulated browsers. We varied the numbers of emulated browser from 5 to 25 (in steps of 5) and noted the WIPS over a period of 400 ($t = 100$ to $t = 500$) seconds. In Figure 10 we compare the average WIPS for when the online bookstore was run on EBS against when the bookstore was run on the `MemcachedEBS` Tiera instance. From the figure, we observe that we were able to scale up the performance for different levels of browser concurrency. The increase in throughput ranged from a minimum of 46% with 5 emulated browsers to a maximum of 69% for 15 emulated browsers. A similar scale up in performance could also be achieved by using Memcached to cache the HTML, image, and database files, but would require changes to the application and MySQL. We were able to provide a scale up in performance without any change in the application logic whatsoever.

4.2 Realizing Desired Metrics

[4]Source: `https://aws.amazon.com/ec2/pricing/`

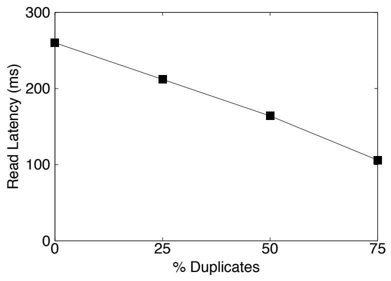


(a) Read Latencies

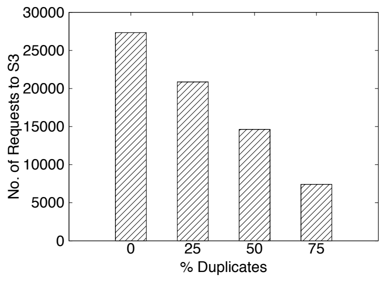


(b) Cost

Figure 11: Average latencies for instances and their respective total cost of storage per GB of Data



(a) Read Latencies



(b) Percent reduction in Requests to S3

Figure 12: Average read latencies and total number of PUT/GET requests to S3

Instance	Configuration
TI:1	50% Memcached, 30% EBS, 20% S3
TI:2	60% Memcached, 20% EBS, 20% S3
TI:3	70% Memcached, 10% EBS, 20% S3

Table 2: Tiera instances used to evaluate performance-cost tradeoff

Our next set of experiments demonstrate how Tiera can be used to easily achieve different desired application metrics.

4.2.1 *Optimizing For Performance And Cost*

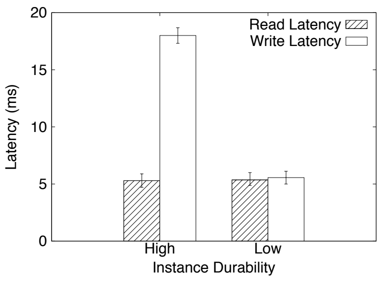
In this section we show two ways by which an application can configure an instance for performance: (1) using larger capacity of a fast tier, and (2) effectively using a limited amount of a fast tier to cache the working set.

The simplest way for an application to realize good performance is by specifying a larger capacity for a faster tier. This will typically imply that the application will have a higher total cost of storage. To illustrate this strategy, we define three instances, with increasing amounts of Memcached tier. The instances, in addition to Memcached, also use EBS, and S3 (Table 2). Memcached tier is used to store the most recently accessed data, EBS is used to hold objects evicted from the Memcached tier, and similarly S3 holds objects evicted from EBS. The data is stored in an exclusive manner across the tiers (i.e., no copies of a data object reside in multiple tiers at the same time). These instances can easily be created using the `PersistentInstance` as a template (Figure 4), and replacing the event-response pairs in it with 3 event-response pairs (one pair per tier) to implement LRU (Figure 5). To measure the read latencies, we use a benchmark that generates two kinds of workloads: Uniform and Zipfian (with default $\theta = 0.99$). Both workload generators are derived from YCSB, with no changes. The benchmark simulated read requests from 14 clients, each requesting 4KB of data per request from the Tiera instance. We also estimated the cost per month for each Tiera instance. We do not show the additional cost due to data access requests to S3 since it is the same across all instances.

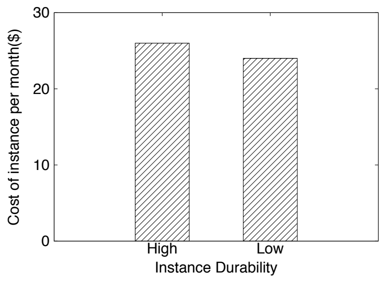
In Figure 11 we plot the average latency (averaged across 5 runs) observed by the clients, and the approximate cost of the various configurations. We see a clear trend in the results, each configuration successively trading lower read latency for higher usage cost, indicating that a Tiera instance with desired performance-cost tradeoff can be constructed using the corresponding specification.

Next we show how an application using Tiera can effectively utilize the existing capacity of a fast tier rather than expanding its capacity (at higher cost). This is achieved by reducing the size of the application's working set. Such a reduction allows for more data to be placed in a faster tier, maximizing its utilization and as a result ensuring a low average latency. One way to do this is to eliminate any redundant data. The application can use the `storeOnce` primitive to do just this. This primitive stores data in a tier only if it's content is unique. This resulting policy will now enable an application to realize the benefits of data de-duplication.

For this experiment, we modified the popular open source cloud backed file system S3FS [22] to use a Tiera instance



(a) Read/Write Latency



(b) Cost

Figure 13: Average latencies and total cost of storage per GB of data

as the backend. S3FS is a user space file system that can be mounted on a desktop or EC2 instance to read/write data from/to an S3 bucket. In our case we modified S3FS to read/write data from/to a Tiera instance using the `storeOnce` response in its policy. The instance was configured to use S3 as the persistent store and Memcached tier to cache recently accessed data (20% Memcached and 80% S3). This instance can be constructed using the `GrowingInstance`(Figure 6) as a template, replacing the EBS tier with S3 tier and replacing the event-response pair in it with two event-response pairs. One event-response pair to manage Memcached as an LRU cache and another event-response pair to invoke `storeOnce` on a `PUT`.

We populate the Tiera instance with data having a varying percentage of redundancy (i.e from 0 to 75%). We use fio to generate read requests following a Zipfian distribution (with default $\theta = 1.2$) on data stored in the Tiera instance. In Figure 12 we plot the latency of access and also the total number of requests to S3. We see that with a decreasing percentage of unique data, more data can be cached in the same amount of Memcached tier resulting in better read latencies. The read latencies observed here are much higher than those observed in the previous experiment due to the size of the Memcached tier being significantly smaller and the absence of the EBS tier. Hence this instance would cost less than the ones described in the last experiment. This instance has the added benefit of reducing the total cost of storage by not only reducing total space consumed in Memcached and S3, but also by reducing the number of `PUT/GET` requests made to S3 which are also charged.

4.2.2 *Optimizing For Durability*

In this section we present three experiments that illustrate different ways by which an application can achieve desired durability. In the first experiment, Tiera is configured to use a storage tier that provides a durability guarantee itself (but at the expense of another metric like performance). In the second and third experiments, we show how to construct policies that replicate data across multiple tiers to achieve desired durability, while minimizing the penalty on other metrics.

In Table 3 we illustrate two instances with different kinds of storage and policies that achieve a different trade-off between durability and performance/cost. Each of these instances can be realized by making simple modifications to the specification illustrated in Figure 3. They each try to maintain a low read latency but each chooses a different

Instance	Configuration	Policy
`High Durability`	100MB Memcached, 100MB EBS, 100MB S3	Immediately backup data to EBS, and push to S3 every 2 mins
`Low Durability`	100MB Memcached, 100MB S3	Backup data in Memcached to S3 every 2 mins

Table 3: Tiera instances used to evaluate performance-durability trade-off

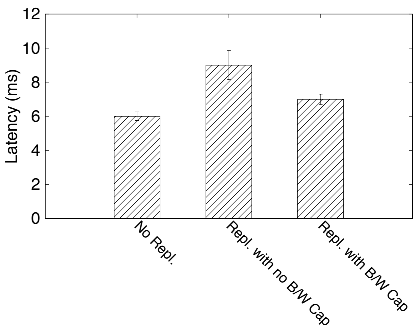
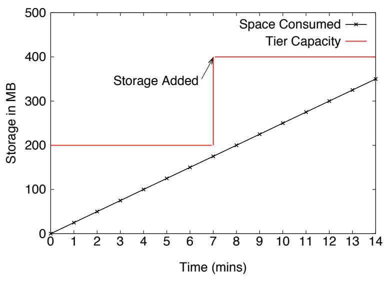


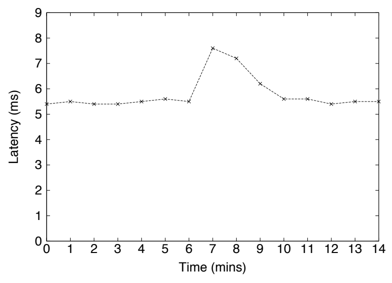
Figure 14: Reducing latency by throttling background replication

point along the durability and cost axes. Using YCSB we ran a mixed workload, with equal proportions of reads and writes of 4KB sizes. We measure the latencies for the configurations described, for a Uniform workload. Figure 13 illustrates the average latencies for reads and writes and also the estimated monetary cost for each configuration. The `High Durability` instance tries to minimize read latency by keeping the data in Memcached, but also achieves high durability by immediately backing up to an EBS volume, thus incurring higher monetary cost (and also higher write latency). It further periodically pushes data to S3, which is a more durable store. The `Low Durability` instance trades-off durability and cost for better write latency by writing data only to Memcached and backing up to S3 with lower frequency. This reduced frequency lowers the durability - in the worst case, an application can lose the most recent 2 minute window of data updates.

Another common way to achieve durability is by replication. However, background replication impacts the read/write



(a) Capacity and Space Consumed



(b) Read Latency

Figure 16: Amount of space occupied and tier capacity during the 10 minute period and read latencies in that time period

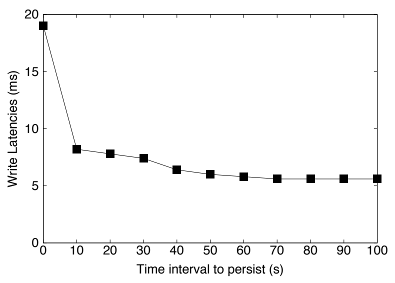


Figure 15: Average write latencies for different time interval values to persist data

latency that an application experiences. We illustrate this with a simple experiment. We construct an instance that consists of two separate EBS volumes. One could imagine a scenario where an application wants to maintain two copies of data in separate volumes for better data durability. The application can choose an eventual consistency model and write data to just one volume and configure the Tiera instance to copy the data to the second EBS volume after some time period or after certain amount of new data has been written into the first volume. We configured our instance to implement the later policy, i.e copy data from one volume to the other after 50 MB of new data had been written to the first volume. We observed that the read/write latencies increased (by $\approx 50\%$) during the periods data was being copied from one volume to the other. This increase in latency is due to the contention between client requests and background replication for disk bandwidth. By putting a cap on the bandwidth allotted to this background replication to 40 KB/s (passed as a parameter to the `copy` response) we observed we were able to reduce the impact of the background replication process (Figure 14) on the foreground IO requests. Note that this improvement in performance doesn't come for free, limiting the bandwidth for background replication reduces data durability as it takes longer to backup data. However, throttling replication ensures that an application experiences uniform latencies throughout, and not latency values with high variation during replication.

The time interval at which data in a fast tier (e.g. Memcached) is backed to a persistent store (e.g EBS) provides another method to achieve a desired tradeoff between data durability and performance. Consider the instance described in Figure 3. We show how changing the value of the time interval provided to the timer event can result in improved client perceived write latencies. We measure the write latencies, while running a write-only workload using YCSB, for increasing values of the time interval that triggers a backup of data in Memcached to EBS. The Memcached tier behaves as a write-through cache when this time interval is zero (client pays the latency of synchronous writes to the persistent block store in this case), and write-back cache when this time interval is set to a large value. From Figure 15 we see that the write latencies decrease as the value of this time interval increases. Further, we note that as the time interval to trigger data persistence increases, the durability reduces. A desired policy on this continuum can be defined based on preferences for performance and durability.

4.2.3 Dynamic Instance and Policy Change

An important aspect of Tiera's novelty lies in the ability to dynamically modify, add, or replace policies while running. Dynamic policies are useful when a Tiera instance needs to be externally reconfigured due to unanticipated events, e.g. unforeseen access patterns or demand, failures, availability of new resources, to name a few.

Adapting to Changing Workload Pattern:
In this experiment, we illustrate how a Tiera instance can adapt to the workload. The policy implemented by this instance is illustrated in Figure 6. The instance is subjected to a write heavy workload inserting 4KB objects for a period of 14 minutes. The instance expands the Memcached tier to accommodate a possibly growing working set to ensure that the client perceived average read latency is bounded (bound = 6ms) value. In Figure 16 we plot the amount of storage consumed and the capacity of the tier within the Tiera instance as a function of time and also the read and write latencies during that time period. In this experiment at time t = 6 mins, the space consumed in the Memcached tier reaches the threshold set in the policy i.e 150 MB. At this time a new EC2 instance was spawned, which took approximately 1 minute to complete. We see that at time t = 7 mins the read latency goes up and remains high until t = 10 mins. This spike in latency is due to a high number of cache misses. The latency finally settles down to its original value once the cache is warmed up.

Adapting to Failures:
In the next experiment we illustrate this ability by enabling

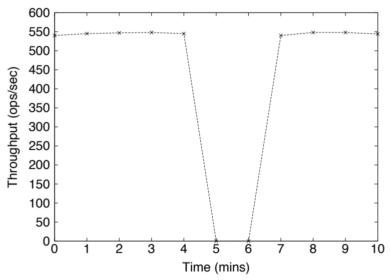


Figure 17: Throughput over a 10 minute window, during which EBS failure is simulated

an application to cope with an unexpected storage service failure. We are aware of at least two separate incidents [2, 1] where an Amazon storage service failed for multiple hours.

For this experiment we start of with a simple instance consisting of two tiers, Memcached and EBS. The instance implements a *write-through* policy, that is data is written to both Memcached and EBS. We simulate a failure to EBS and reconfigure the instance to use Ephemeral Storage and S3. Ephemeral Storage provides performance comparable to EBS (i.e read and write latencies similar to EBS), but data stored in Ephemeral Storage is not durable[5] and hence needs to be backed to a durable store like S3. We also deployed an external monitoring application that detects a storage failure and will reconfigure the instance if this occurs. The monitoring application writes data to the Tiera instance on a 2 minute schedule. It assumes a storage service has failed if the attempt to write data (after successive retries) fails. The instance is reconfigured with two new tiers (Ephemeral Storage and S3) and two new event-response pairs. One event-response pair enables data to be stored in Ephemeral Storage and the other enables data backup from Ephemeral Storage to S3 every 2 mins. To show the benefit of this feature, we subject the instance to a write only workload using YCSB, and measure the IO throughput during a 10 minute window. At the start of the experiment, the instance writes data to both Memcached and EBS before acknowledging the request. We simulate a failure in EBS (similar to [1]) by timing out writes around t = 4 mins. The monitoring application discovers the failure at around t = 6 mins and requests instance reconfiguration as just described. In Figure 17 we plot the operations per second for this 10 minute window. We see that throughput drops to zero between t = 4 mins to t = 6 mins. The throughput is subsequently restored back to its original value by t = 7 mins.

4.3 Characterizing Overhead

In the last experiment we measure the overhead added by the Tiera control layer. The Tiera instance in this experiment implements the write-through policy described in Figure 4. We used the YCSB workload generator to generate a Zipfian workload inserting 4KB objects, making on average 200 requests per second to the instance. We compared two setups for this experiment, one with the Tiera control layer enabled, and one without (where the application directly accessed each of the storage tiers). We measured the latencies from the time the `PUT/GET` request was received to the time the request was acknowledged. The overhead introduced by the control layer corresponds to evaluating and executing the action event to determine the tiers to place data. We begin by simulating requests from one client and gradually increase the number of clients, this has the effect of causing the event to fire multiple times. In Figure 18 we plot the read and write latencies as the number of clients increases, thus increasing the frequency of event firings. We see that the performance overhead introduced by Tiera is very low (under 2%).

[5]Data stored in Ephemeral Storage is lost if the EC2 instance to which it is attached reboots.

5. RELATED WORK

Multi-tiered storage systems are an area of active research [13, 26, 11]. The storage tiers in these systems exhibit device heterogeneity (e.g., SSDs vs. HDDs) with different performance characteristics and costs. We believe our work is complementary to this research, as Tiera could model each distinct storage device as a separate tier. This would allow Tiera to optimize device-based metrics such as energy. For instance, one can imagine a *green* Tiera instance that favors low energy consumption and selects an SSD tier.

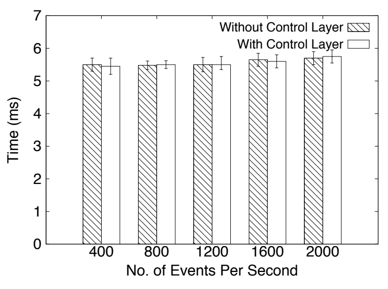
Our work is similar in spirit to the FleCS containers [29], but Tiera treats multiple tiers as first class objects and therefore enables a richer set of tradeoffs and composite benefits. It is possible that FleCS containers could be realized using Tiera instances. The policy architecture for distributed storage systems (PADS) [5], proposes an architecture similar to Tiera. Tiera differs from PADS and FleCS containers in its support for dynamically modifying policies at runtime. Tiera also supports the addition/removal of tiers at runtime. Overall, our work differs from others as we allow dynamic policies (e.g. the adapting to failures example 4.2.3) by adding new events and responses at runtime, and expose storage primitives like data de-duplication, compression, encryption etc for applications to use.

Object storage systems [16] store data as objects, and store data and metadata separately. Introduced in the late 1990's, object storage architecture has gained popularity recently and is implemented in most cloud storage systems such as Amazon S3 [3], Windows Azure Store [27], and OpenStack SWIFT [20]. This architecture provides unified access to data that is potentially distributed across many nodes, similar to how Tiera provides unified access to data stored in different storage tiers (potentially even edge storage). The architecture allows more intelligent management of data through policies defined at either the object level or container level. Tiera enables such capability as it can be used to define policies in terms of either thresholds or actions, on either individual objects or object classes.

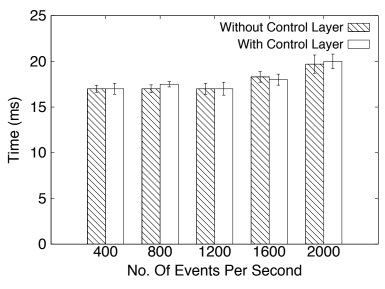
There have been a few projects/products [25, 21] that integrate cloud storage like Amazon S3 with edge storage (on a proxy server or on the desktop). They have primarily explored techniques to reduce the client latency and cost of storage. These systems adopt a particular storage policy instead of providing applications the ability to pick and tune the policy to suit the tradeoffs it is willing to make.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented Tiera, a middleware that enables the specification of multi-tiered cloud storage instances that are flexible and easy-to-use. We showed how Tiera can enable a rich array of storage policies and desired metrics to



(a) Read Latency



(b) Write Latency

Figure 18: Average latencies with increasing number of events per second

be realized through a powerful event-response mechanism and support for runtime replacement of Tiera policies. We illustrated how benefits of using multiple storage services can be provided to applications using Tiera without modifications to the application's logic. The experimental results also showed how complex policies can be easily expressed in Tiera allowing the user to make reasoned choices regarding tradeoffs. Lastly, we provided evidence that Tiera overhead is modest.

In the future we plan to explore techniques for generating appropriate instance configuration and data management policies using abstract application requirements and workload characteristics, e.g. 99 percentile read latency < 10 ms with read requests following a uniform distribution. We also plan to employ horizontal scaling to scale Tiera control layer to be able to store very large number of objects. There are many large-scale distributed storage systems [8, 6, 14] that employ horizontal scaling for scalability, and we plan to leverage similar techniques. A distributed control layer architecture also provides metadata management scalability and better fault tolerance. Another possible future direction would be exploring multi cloud deployments. This use case poses many interesting challenges for Tiera in terms of rethinking abstractions, system scalability, ensuring data and metadata consistency, predictive data and migration/prefetching.

7. REFERENCES

[1] Amazon EBS outage 2011. `http://tinyurl.com/p46yuvy`.

[2] Amazon S3 outage 2008. `http://www.zdnet.com/blog/btl/amazon-explains-its-s3-outage/8010`.

[3] Amazon Simple Storage Service - S3. `http://aws.amazon.com/s3/`.

[4] Apache Thrift. `http://thrift.apache.org/`.

[5] Belaramani, N. M., Zheng, J., Nayate, A., Soulé, R., Dahlin, M., and Grimm, R. PADS: A Policy Architecture for Distributed Storage Systems. In *NSDI* (2009), pp. 59–74.

[6] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[7] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *SOCC* (2010), ACM, pp. 143–154.

[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007), vol. 7, pp. 205–220.

[9] FIO - Flexible IO Tester. `http://freecode.com/projects/fio/`.

[10] FUSE - Filesystem in User Space. `http://fuse.sourceforge.net/`.

[11] Guerra, J., Pucha, H., Glider, J. S., Belluomini, W., and Rangaswami, R. Cost Effective Storage using Extent Based Dynamic Tiering. In *FAST* (2011), pp. 273–286.

[12] Implementation of TPC-W Benchmark. `https://github.com/jopereira/java-tpcw`.

[13] Kim, Y., Gupta, A., Urgaonkar, B., Berman, P., and Sivasubramaniam, A. HybridStore: A cost-efficient, high-performance storage system combining SSDs and HDDs. In *MASCOTS* (2011), IEEE, pp. 227–236.

[14] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010).

[15] Memcached. `http://memcached.org/`.

[16] Mesnier, M., Ganger, G. R., and Riedel, E. Object-based storage. *Communications Magazine, IEEE 41*, 8 (2003), 84–90.

[17] Moodle. `https://moodle.org/`.

[18] MySQL Community Server. `http://dev.mysql.com/downloads/mysql/`.

[19] Olson, M. A., Bostic, K., and Seltzer, M. I. Berkeley DB. In *ATC* (1999), USENIX, pp. 183–191.

[20] OpenStack SWIFT. `https://wiki.openstack.org/wiki/Swift`.

[21] Panzura. `http://panzura.com/`.

[22] S3FS - FUSE-based file system backed by Amazon S3. `http://code.google.com/p/s3fs/`.

[23] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net/`.

[24] TPC-W Benchmark. `http://www.tpc.org/tpcw/`.

[25] Vrable, M., Savage, S., and Voelker, G. M. BlueSky: A Cloud-backed File System for the Enterprise. In *FAST* (2012), USENIX Association, pp. 19–19.

[26] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS) 14*, 1 (1996), 108–136.

[27] Windows Azure Store. `www.windowsazure.com/en-us/store/`.

[28] WordPress. `http://wordpress.org/`.

[29] Yoon, H., Ravichandran, M., Gavrilovska, A., and Schwan, K. Distributed Cloud Storage Services with FleCS Containers. In *Opencirrus* (2011).