Unifying the Data Center Caching Layer — Feasible? Profitable?

Liana V. Rodriguez  Alexis Gonzalez  Pratik Poudel  Raju Rangaswami  Jason Liu
Florida International University

ABSTRACT
Data centers today host a large number of workloads and many of these workloads consume significant storage resources. Given the long history of successes in storage caching, it is only natural such successes bear fruit in modern data centers, at scale. This paper presents CaaS, a generalized caching service for cloud data centers. Departing from existing application, storage, or data-type specific caches, CaaS unifies and abstracts data center caching resources making these available to any workload and for any data type. Also departing from past caching practices, CaaS is fault-tolerant allowing it to cache writes without risk of data loss. We expect that systems such as CaaS will help bridge the gap between heterogeneous and distributed cache resources and data-intensive applications in a data center.

CCS CONCEPTS
• Computer systems organization → Cloud computing; • Information systems → Web services.

KEYWORDS
Cloud data centers, Systems, Caching

ACM Reference Format:

1 INTRODUCTION
Cloud data center workloads use data stored in various forms, including files, blocks, key-value pairs, and objects. Caching is a common performance-acceleration technique used by applications either directly as a library or indirectly through system-level facilities. Examples of these include in-memory distributed caches [13, 14, 21, 22, 30, 58], host-side storage caches based on DRAM or SSD [17, 33, 56], and caches for key-value stores [64, 67, 70]. A typical software stack for cloud data centers consists of applications, user-level libraries, interfaces, and storage back-ends (see Figure 1).

The caches, located between the storage interface and storage systems, typically target a single store, and when they do target multiple stores, are often limited to the workloads running on a single host as in the case of a host-side cache [10, 43, 48, 54] and to a single store type. At the same time, the diversity of caching deployments is accelerating due to the introduction of 3D-XPoint based persistent memory devices [36–38] as well as the continued development of flash-based SSD technology with various cost-performance tradeoffs [61]. Furthermore, high-performance network fabrics (such as 40+ GigE, ROCE, Infiniband, etc.) are bridging the performance gap between accesses to local and remote devices. The compute stacks, in response, are becoming more efficient at utilizing the newer memory/storage devices [19].

The current siloed development and deployment of caches across store types and workloads introduce multiple challenges. First is the fragmentation of cache resource utilization across the data center and even at a single host when each cache implementation manages a distinct and dedicated subset of the data center’s cache resources; caches that are
incorrectly sized get either under-utilized or over-subscribed. A coordinated use of distributed caches across all hosts in the cloud data center can improve resource utilization. Second, caching solutions should be scalable. In siloed caching solutions, resources that are not accessible by a cache instance are simply not usable by its workloads; consequently, resource usage cannot scale with the increasing workload. Again, a coordinated use of distributed caches can expose and balance caching resources across physical hosts. Third, since there are diverse storage types that clients utilize, including block devices, file systems, object stores, and more, a caching solution that aims to optimize the global utilization of the cache layer must be able to support many such storage types. Fourth, workloads vary significantly across application and are dynamic in both temporal and spatial dimensions; a typical result is over-provisioning of resources to ensure worst-case performance which does not scale to cost-sensitive cloud data centers. Finally, given the increasing amounts of DRAM in hosts, it has been pointed out that production storage I/O workloads tend to be write dominant [42, 60]; a storage caching solution that is unable to cache writes is likely to be quite limited in its effectiveness.

Recent work on CacheLib [15] at Facebook is an effort to partially address the third challenge above through the unification of cache implementations across different storage types and applications. CacheLib rightly points out that although the many caching implementations serve different workloads, these implementations share many important development and engineering challenges. The goal of CacheLib is to simplify the development of stable, type-agnostic single instance caching systems via the use of the CacheLib caching library. CacheLib, however, does not address the remaining challenges including the creation of a fault-tolerant caching service that is able to utilize all of the available cache resources in the cloud.

In this paper, we demonstrate the significance of the above mentioned cloud caching challenges using state-of-the-art production storage workloads representing different types of storage. In response, we propose the design of CaaS, a generalized caching service intended to be deployed within a data center. Foremost, CaaS unifies caching across all types of storage systems with the goal of providing a general utility that abstracts and exposes all of the available cache resources in the cloud to each workload. This involves defining a standardized set of abstractions and API for caching data with attention to the division of responsibility of various interacting elements without loss of generality and without compromising flexibility. The CaaS service itself enables lightweight store-specific CaaS clients that drastically reduce the development complexity of cache implementations for new store types. Finally, CaaS optimizes the write path without loss of data freshness and data consistency which we demonstrate is imperative to achieve high overall cache performance.

For preliminary evaluation, we model CaaS by using network latency of 40GigE and primary-backup replication for fault-tolerant writes. Analytical results shows that CaaS decreases write latencies by 9x, while reducing IO latency by 1.4x when compared to using a flash SSD based local cache when both caches are configured with space equivalent to 10% of the workload’s unique data accessed.

2 MOTIVATION

2.1 Separation of Concerns

Computing infrastructures often maintain several internal services that are commonly implemented using their own tailored cache [15]. For example, memcached is mostly used for read-heavy workloads but does not provide any fault-tolerance guarantees. EC-Cache [57], which implements an object cache, uses Alluxio [1] for cache data management, and motivates the use of erasure coding to better balance I/O and load across a cluster of cache servers. Figure 1 shows an ecosystem of software storage for cloud data centers, including our proposed solution, CaaS. Existing systems are focused on caching specific data types for single application instances.

To optimally utilize a data center’s cache resources, the ability to abstract and expose these resources to all applications irrespective of data type is valuable. CaaS targets a general distributed caching layer that provides caching as a service to different applications and supports multiple data types. While we focus on CaaS’ ability to abstract the commonly used data types as referenced by the workloads indicated in Table 1, CaaS is itself generic. CaaS achieves generalizability via separating the concerns of cache access from cache implementation. Applications use data-type specific CaaS clients to access the CaaS service. As we shall elaborate later (§3), CaaS clients transform data-type specific accesses into type-agnostic accesses to interact with the CaaS service implementation.
### Table 1: Data center workloads representing 4 different storage types.

<table>
<thead>
<tr>
<th>Store Type</th>
<th># Traces</th>
<th>Requests</th>
<th>Writes</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block [11, 41, 51, 63, 66]</td>
<td>363</td>
<td>86,171,476</td>
<td>71%</td>
<td>User home/project directories; Webpage; Web-based servers; Online course system; Hardware monitoring; Source control; Web staging; Terminal; Web/SQL; Media; Test web; Firewall/web proxy; VMware VMs.</td>
</tr>
<tr>
<td>File [18, 63]</td>
<td>58</td>
<td>4,442,610</td>
<td>9.8%</td>
<td>Undergrad/Grad/Staff/Faculty NFS/CIFS fileserver; Web/DB; Backup servers and Researcher 1, 2 desktops.</td>
</tr>
<tr>
<td>KV [63, 72]</td>
<td>24</td>
<td>98,042,332</td>
<td>40.5%</td>
<td>Memcached clusters at Twitter.</td>
</tr>
<tr>
<td>Object [25, 63]</td>
<td>36</td>
<td>15,409,158</td>
<td>22.5%</td>
<td>IBM object store traces.</td>
</tr>
</tbody>
</table>

Figure 3: Performance for different local/remote cache and back-end latencies. Y-axis is log scale.

2.2 A Plethora of Caches

Caches are critical to enabling cloud services that meet peak demands while using resources efficiently [20, 31]. These services are mainly deployed using VM/Container abstractions that consume storage through a single storage system. Increasingly, a diversity of devices, including DRAM, Optane, and flash, are now available as storage caches in the cloud. Ideally, all available cache resources within the data center should be shared across storage systems and applications. In the absence of sharing, caches typically get over-provisioned to accommodate the entire range of one application’s working set demand.

To better understand the impact of sharing, we calculate the cache demands of various workloads and data types so that consuming applications achieve an average of 80% hit-rate. For this experiment, we selected one workload for each store type shown in Table 1 (block: casa-7, file: usr2.0, object: Trace011, key-value: Memcached-cluster3). Next, we chose the cache demand using the workloads’ miss-ratio curve (MRC) [66]. Figure 2 illustrates how cache requirements vary over time for the four types of storage systems, which means that static cache partitioning strategies are likely to be inefficient.

2.3 The Importance of Caching Writes

Local and distributed caches are often used in a read-only mode where reads are optimized for better performance. Previous research points out that production storage workloads are often write-dominant due to the availability of increasingly larger local caches that can serve reads while writes must be written to a fault-tolerant storage layer to avoid data loss [42, 46]. Since the writes in such caching systems are always routed to back-end storage, the cache is virtually ineffective for servicing writes. To estimate the penalty of serving writes from the back-end storage, we created a simple model whereby an application accesses a local and a remote cache with block storage over the network. Figure 3 illustrates how caching writes improves throughput by 10x in local caches and 7x in remote caches with hit rate between 90% and 99% when the back-end storage accesses require 1ms. Furthermore, Figure 4 shows the performance increase.

### Table 2: Typical latency for different devices used as local and remote caches. Average Load (L)/Store (S) or Read (R)/Write (W) latencies refer to AKB accesses and a QD=8 IOs. Remote cache access include network latency of 4μs corresponding to a 40GigE network.

<table>
<thead>
<tr>
<th>Devices</th>
<th>Local Cache (μs)</th>
<th>Remote Cache (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L or R</td>
<td>S or W</td>
</tr>
<tr>
<td>DRAM</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>Intel Optane PM</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>Intel Optane SSD [39]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>SSD [39, 68]</td>
<td>5.3</td>
<td>6</td>
</tr>
</tbody>
</table>

### Table 3: Latency for different back-end storage types.

<table>
<thead>
<tr>
<th>Store Type</th>
<th>Provider</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object [9]</td>
<td>Amazon S3</td>
<td>10 – 100ms</td>
</tr>
<tr>
<td>Block [6]</td>
<td>Amazon EBS</td>
<td>1 – 9ms</td>
</tr>
<tr>
<td>Block [29]</td>
<td>Google Persistent Disk</td>
<td>3.3ms</td>
</tr>
<tr>
<td>File [7]</td>
<td>Amazon EFS</td>
<td>10 – 100ms</td>
</tr>
<tr>
<td>KV [8]</td>
<td>Amazon Dynamo DB</td>
<td>1 – 9ms</td>
</tr>
</tbody>
</table>
when we increase the size of the remote cache. For instance, with queue depth (QD) of 8, doubling the remote cache size to (20GB) will increase the throughput 1.24x.

Table 1 shows the average read-write ratio across several workloads for four different types of storage systems. The high number of writes in cloud workloads coupled with the latency penalty for serving from the back-end storage (see Table 3) and the low values for today’s network fabrics (see Table 4) makes caching writes attractive from a performance standpoint. However, to be able to cache writes, the cache layer itself must be fault-tolerant, motivating a core CaaS design principle.

### 3 CAAS: A SHARED CACHE LAYER

CaaS is designed to aggregate caching as a service across applications and for multiple types of storage systems in the data center. In this section, we provide a preview of its basic design.

#### 3.1 Architecture

Figure 5 highlights the CaaS architecture. CaaS’ architecture follows a centralized design, where each store type has an associated CaaS Client library implementation that interposes between the Workload Generator and the Storage Back-end. The translation layer of the storage interposing CaaS Client converts storage-type-specific I/O requests into CaaS requests. The Workload Generators are applications consuming storage as a file system, object store, key-value store, or block store. The CaaS Coordination Service, implemented using a Zookeeper [34] cluster, manages all CaaS metadata, has a global view of the cache resources, and also interfaces with CaaS Clients to perform misses and writeback operations. A network of CaaS Data Servers export and provide access to their local cache device resources to CaaS Clients. Data Servers can be co-located in the same host as the CaaS Client or any remote host as a containerized service. The Coordination Service controls and implements dynamic cache space allocation and data distribution, load balancing, and fault-tolerance across the Data Servers.

#### 3.2 Interface

For CaaS to implement a shared abstraction layer that can support any store type and potentially multiple clients per store simultaneously, it needs an interface (API) that is simple and flexible. Furthermore, to support write caching, the API should be flexible to accommodate different write policies.

The API is designed around the concepts of Clices and Stores. A Clice (cache-slice) is the unit of caching in CaaS and Store refers to an instance of a given storage system that will consume storage using the CaaS Client. CaaS Clients initially register a store type with the Coordination Service and create a CaaS cache instance that will be used for subsequent requests to the store. Each request from the Workload Generator application is first broken up into one or more smaller clice granularity requests by the translation layer in the CaaS Client. To reduce traffic between CaaS Clients and the Coordination Service, CaaS Clients periodically download (clice → data-server) mappings from the Coordination Service. Outdated map entries get notified by the target Data Server. Outdated and missing mappings incur on-demand lookup requests at the Coordination Service. Data belonging to a single CaaS cache instance get distributed across multiple CaaS Data Servers. The CaaS Client communicates with specific CaaS Data Servers for clice read and writeClean/writeDirty operations. writeClean writes a previously non-existent clice while writeDirty updates a cached clice. The read/write operations specify the offset and size of data written to allow updating portions of clices. The Coordination Service invokes the writeback operation implemented by the CaaS Client to evict dirty data safely from the cache. The writeback operations are triggered based on the write policy specification that can control for data staleness of the Storage Back-end.

#### 3.3 Design Outline

The Coordination Service allocates cache resources to a particular CaaS cache instance. The basic CaaS design supports fault-tolerant writes using a primary-backup scheme [5, 45, 62], which assigns CaaS data servers to a replication group. Each replication group has a leader responsible for serving read and write requests that are mapped to it. Write requests are not completed at the client until they are replicated and acknowledged by the CaaS Data Server(s) acting together.
as the leader of the replication group(s) backing the clices corresponding to the request. The leaders assign consecutive serial/version numbers to every write request of a single clice. All versions are initially logged at the replication group leader to eventually be applied persistently after receiving acknowledgment from all the followers. Previous writes to the same clice are reflected in the persistent state of leader. Read requests that arrive at non-leader servers are discarded and each leader maintains a leasing period to be able to respond to read requests even during reconfiguration of the replication group [28]. We anticipate using one of the many established protocols to handle failures for leaders and followers that ensure zero data loss and serializability [34, 47, 65]. Finally, writeback operations need careful handling to ensure that the backend storage layer can perform consistent snapshots [41, 55]. In addition to exploring consistent writeback [42] operations, we anticipate utilizing the fault-tolerant cache layer to implement idempotent checkpoints of consistent states in storage.

4 PRELIMINARY STUDY

We developed a simple model to estimate the performance of CaaS relative to a local cache. First, we used the 363 block storage workloads from Table 1 and ran an ARC cache simulator [59] against each to obtain the number of Read hits. We simulated write-through for the local cache and write-back for CaaS since CaaS is fault-tolerant and thereby safe to absorb writes.

Next, we computed the average read and write I/O latencies by attributing operation latencies as follows. We assumed a back-end storage latency of 2ms representing Amazon’s EBS solution (Table 3). For the local cache solution, we assumed a flash-based local SSD cache and estimated the average read and write I/O latencies using the corresponding numbers from Table 2. CaaS reads require two network hops between the CaaS client and the replication group leader corresponding to request and return the clice respectively. CaaS writes, on the other hand, simulate four network hops, two each between the CaaS client and group leader, and group leader and group follower, respectively. We use the 40Gbit values from Table 4 as the network hop latency. Figure 6 shows the latency models for CaaS and Local Cache including also SSD read/write latency in CaaS. Our model does not take into account the specific differences in software overheads which are subject to various degrees of optimization in either system, focusing instead on network-facing latencies that are unavoidable.

Figure 7 shows the preliminary results for CaaS when using the ARC [49] replacement policy and varying the cache size. We varied the size for the local cache up to maximum of 10% of the workload’s footprint (all unique data items accessed) to show the differences in read latency with respect to a CaaS remote cache which is also varied up to 20% of workload’s footprint. For the case of reads, CaaS achieves similar latencies when is sized equal to a flash SSD based local cache while reducing values up to 6% as the remote cache size increases beyond 10%. For the case of writes, CaaS significantly reduces latency by approximately 9x. Finally, CaaS reduces the average I/O latencies by 1.4x with respect to SSD based local cache, when both caches are sized at 10% relative to the workload’s footprint.

5 RELATED WORK

Distributed caching services have been extensively used in the past to improve performance in production systems. Previous solutions include Memcached, Redis and its variants [13, 14, 22, 26, 30, 58, 73], Facebook’s caching systems [12, 16, 52], Twitter’s optimizations of Memcached, Twemcache [72], and Microsoft’s FaRM [23] system. These designs are focused on key-value and/or object stores and most are optimized for reads using DRAM-based caches.

In building solutions specific to object stores and file systems, Alluxio [1] and EC-Cache [57] developed a distributed cluster cache that accounts for different types of cache resources and leverages erasure coding for load balancing. On the other hand, CacheLib [15] argues for a general-purpose caching engine used in many user-cases at Facebook. CacheLib offers functionalities in a library rather than a caching service agnostic of the storage system as we present in CaaS.
The idea of caching writes for block stores has been previously explored in the literature to increase performance especially in the context of non-fault-tolerant local SSD caches [42, 56]. The Ceph Block Device [2] storage system also uses a persistent write-back policy that guarantees consistent writes on top of a Persistent Memory or SSD cache device [3]. CaaS, in contrast, supports fault-tolerant writes in a distributed caching service that can support a variety of storage systems. CaaS’ initial fault-tolerant write design is based on classic primary-backup replication [5, 28, 45, 62, 65].

6 DISCUSSION AND FUTURE WORK
Several questions challenge the success of CaaS and guide our future work. Foremost is the performance loss at the caching layer when it must be accessed over a network instead of locally. Remote accesses over 40/100Gbit networks add less than 10μs of additional latency to individual cache accesses. When compared to the 1 – 100ms of back-end storage accesses (Table 3), this still represents at least two orders of magnitude latency improvement, implying that there is significant I/O performance to be gained even when a storage cache is accessed over the network.

A second challenge is the complexity of managing a fault-tolerant, writable cache in comparison to simpler write policies such as write-through and write-around. CaaS exposes a cache layer that is designed to provide the same durability, consistency, and fault-tolerance properties as the underlying storage system. Utilizing CaaS is as simple as registering with the Coordination Service for an underlying data store and specifying the store type and a write policy. The write policy would provide the client control over data staleness at the underlying data store while a carefully constructed writeback process in CaaS can ensure that the storage back-end can perform point-in-time consistent snapshots for backup [41, 55].

Finally, similar to storage, CaaS allows workloads to detach (get unregistered) from the cache — either voluntarily or forcibly upon client failure, possibly migrated, and re-attached (registered) to CaaS via its API.

A significant design decision in CaaS is the size of its unit of access, the clice. The number of records needed to keep track of clice locations increases proportionately with the number clices. In general, smaller clice sizes will increase metadata overhead at the CaaS Coordination Server, but reduce overall cache space fragmentation. On the other hand, larger clice sizes will increase resource fragmentation at the Data Servers. A possible choice is the size of an OS page (4KB) which is also the unit at which files and block storage typically get accessed. However, an OS page may be too small for object stores that typically are accessed at much larger granularities. Thus, choosing the clice size represents important trade-offs and it may be beneficial to support multiple clice sizes at the CaaS layer.

A well-designed cache allocation scheme is key to reducing the level of resource fragmentation in CaaS. Previous works employed slab-based allocators with different initial sizes and growth factors [22, 72]. We anticipate using a similar design, albeit with a reduced challenge for handling large requests since the CaaS requests are pre-split into smaller clices. To adapt to the request size distributions, CaaS will need to include a dynamic strategy to move slabs across different slab classes that serve different ranges of clice sizes. In addition, state-of-the-art cache eviction policies for uniform sizes need to be modified to work at the slab granularity. Furthermore, cache allocation and eviction are distributed data placement problems in CaaS. Any clice data placement policy must go hand-in-hand with dynamic load balancing across CaaS Data Servers.

Another important consideration in CaaS is the support for a hierarchy of different cache devices within the data servers such as DRAM, Persistent Memory, Optane SSDs, and flash-based SSD. Efforts at understanding the performance properties of potential caching devices have underscored their diversity, prompting a “no single technology wins for every workload” approach [68, 71]. Recently, some studies have designed and incorporated dynamic cost-performance analysis in heterogenous caches [24, 69]. For CaaS to exploit heterogeneity, it needs to develop intelligence about device performance characteristics and workload demands.

Given the unified nature of the caching layer, another challenge involves respecting the data isolation requirements across different clients. We anticipate that CaaS will follow well-established protocols related to providing authorized and secure cloud data access to client applications [27, 32, 40].

Finally, in the long term, service level agreements (SLAs) are likely to be a key indicator of CaaS’ success. A distributed cache resource allocation algorithm that guarantees a user-defined level of quality of service (QoS) would be critical. With such an ability, CaaS clients representing latency and/or throughput sensitive applications would be able to specify a cache instance’s QoS requirement at creation time. Prior work has developed solutions for partitioning a host’s local cache based on generating a miss-ratio curve (MRC) per workload and transforming them to latency and throughput curves [43]. Such approaches may lead to joint optimizations considering the requirements of a shared distributed cache with multiple types of devices being used by a large number of workloads and for multiple types of data stores.

ACKNOWLEDGMENTS
We would like to thank the reviewers of this paper and our shepherd Andromachi Chatzieleftheriou for their feedback. This work was supported by NSF grants CCF-1718335, CNS-1563883, and CNS-1956229.
REFERENCES


