Caching for Object Stores in Data Centers

(Prospectus)

Emine Ugur Kaynar
Department of Computer Science
Boston University

Chapter 1

Introduction

Big data and its analysis are at the center of many businesses today. Organizations collect all data that could be of future value. The rise of big data and the growth of very large-scale distributed applications (e.g. Google search, Facebook, Amazon) have radically changed the way data is stored and the computing environments on which they are implemented. More and more organizations are creating data lakes, low-cost object-storage repositories that can store vast volumes of data [6, 45]. In data lake realization, the fundamental concept of the storage system is no longer a file, but an immutable object, as a result, data management and the relationship of users with data in storage becomes fundamentally different.

In data centers, data lakes are often shared across entire companies but processed by independent entities (e.g. business units within an enterprise). In these environments, a major bottleneck is getting the data from a large shared data lake due to limitations both in data lake performances (such as slow disks) and data center network. Organizations accelerates data lake access by deploying cache solutions [32, 8, 17, 5]. These solutions generally moves frequently-used datasets into RAM or SSD. With data caching and analysis spread across a large number of machines, the network connections between these machines are a critical determinant of performance.

This dissertation investigates caching infrastructure for data lakes deployed as immutable object stores to speed up the performance of big data analytic workloads. In this work, we ask the following research question: how can we design a cache infrastructure to provide reliable and efficient storage stack in a network constrained environment, specifically in private data centers, by exploiting the object stores' immutability? To answer this question, prior caching solutions for file and block based storage need to be re-evaluated; and new caching solutions need to be designed by considering the characteristics of data lakes and big data analytics. This thesis presents the design and implementation of a novel cache architecture for object stores to address network limitations and to improve the observed throughput of the storage stack.

A fundamental goal of our caching prototype is to allow simplified integration into existing data lakes to enable caching to be transparently introduced into data centers, to support efficient caching of objects widely shared across clusters deployed by different organizations, to improve the performance of big data analytic workloads and reduce the contention in data center network and to avoid the complexity of

managing a separate caching service on top of the data lake.

Chapter 2

Background and Motivation

We describe a typical data center topology with network imbalances due to over-subscription and incremental networking upgrades and discuss why big data analytic workloads have performance degradation in such environments. We then explain the characteristics of data lakes, vast storage repositories deployed as immutable object stores, and close the section by highlighting the characteristics of big data analytic worklods.

2.1 Network Imbalances

The performance of a distributed storage system is limited by that of its underlying network. Although full-bisection-bandwidth networks for cloud data centers have been popular in recent literature[37, 27], in practice cost prevents their widespread deployment; nor are they required by many data center workloads [11, 40, 43]. In even the best-designed modern data centers, rack *uplinks*—i.e. the connections between top-of-rack switches and the rest of the data center—are over-subscribed, with e.g. a ratio of 3:1 in Google's Jupiter [44] interconnect. This is a natural consequence of today's technology, where servers and cost-effective switches often have interfaces of the same or nearly the same speed. (e.g. 100 Gbit/s uplinks vs. dual 40 Gbit/s NICs). In this environment, the cost, complexity, and even physical volume of cabling required for full bisection bandwidth (i.e. 1 rack uplink per server NIC) puts it out of reach of all but the most extreme applications.

This over-subscription is compounded by connectivity restrictions due to organic growth: in many organizations and co-location facilities, clusters are deployed one by one over time, with individual and separate sources of funding. Shared facilities (e.g. data center-wide networking) cannot be upgraded each time a new increment of equipment is deployed; at best they are upgraded at regular intervals, and are thus on average a year or two out of date; at worst these facilities languish for much longer. For example, the MCHPCC (a shared institutional data center) [36] recently deployed a rack of servers with 100 Gbit NICs; however only two 40 Gbit uplinks were available to the rest of the network, for an over-subscription of as much as 50:1.

Even though data center networks are getting faster and networking becomes cheaper, we believe that the rapid increase in network demand to continue and therefore the network imbalances will exist in the future. In such environments, we believe

that it is crucial for storage systems to be network-aware, able to place storage on the access side of network bottlenecks and to cache, tier, or migrate it when performance benefits warrant.

2.2 Big Data Processing in Data Centers

Figure 2·1 shows a simplified view of the data center topology: racks of servers connected by top-of-rack (ToR) switches, an over-subscribed inter-rack network within clusters, and a further over-subscribed data center network between clusters and an enterprise data lake. Clusters may be deployed at different times, with different network technologies, and may be owned or deployed by different entities: sub-units of a corporation, research groups in a university, or entire companies in a co-location facility.

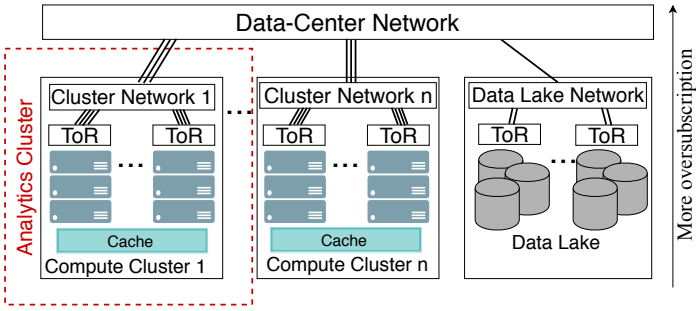

Figure 2·1: A typical data center topology that contain multiple clusters connected via an over-subscribed data center network. Data lakes are usually introduced as a separate cluster with all accesses going over the data center network.

Individual clusters are frequently installed or upgraded in a single deployment, allowing ToR and inter-rack bandwidth to be sized appropriately. In contrast, differing upgrade schedules and split ownership typically prevent inter-cluster networks from being upgraded at the same time, resulting in significant bandwidth mismatches across compute clusters.

In Figure 2·1 one of the clusters is a *data lake*, storing datasets used for analysis by multiple compute clusters. In an enterprise this may be akin to the classic data warehouse; in a scientific environment, a repository for shared datasets. Such data lakes are typically implemented by object stores,which satisfy the key requirements of high capacity, economical storage for unstructured, read-mostly data, with fine-grained access control to provide varied level of access to datasets owned by different entities.

2.3 Workload Characterization in Immutable Object stores

Data lakes are typically implemented as immutable (write-one) object stores such as Ceph [45] or AWS's S3 [6] which are implemented as scale-out services, typically to run over thousands of cheap commodity hardware with locally attached disks. They have different characteristics and access patterns than both traditional distributed file systems or block storage, where these differences are crucial for accelerating the distributed storage stack.

Object stores such as [26, 6, 45] discard many of the semantics of traditional file systems (e.g. POSIX consistency, locking) and are designed to be extremely scalable. Objects within the object store are stored in a flat namespace and are immutable write-once where objects can never be modified or re-written again, therefore object stores don't need to provide any type of distributed locking mechanisms like file systems. Their eventual consistency model and write-one characteristic allow data to be highly cacheable and accelerating performance. Due to the unique characteristic of object stores, existing cache approaches for file-based [41, 28] or block-based [12] storage systems do not match with the requirements of object stores.

In addition, object stores use larger object sizes (Ceph uses 4MB object size), which can provide opportunities for investigating strategies that have not been feasible in the past for block storage. Previous caching approaches used in different domains such as multiprocessor, operating systems that deal with much smaller block granularity than object stores, use traditional cache management policies like LRU or its variants due to their performance constraints. In contrary, more complex algorithms can be used for cache managements for objects stores with large object accesses.

There is also a difference between object storage and file/block based-storage in the way the data is accessed. A file is read via the GET HTTP function which returns the entire or the certain portion of the file. Objects are typically large, accessed in their entirety and sequentially. In this work, we focus on data access for analysis by jobs (e.g. bigdata frameworks) running on analysis clusters. Recent studies demonstrate that there is good temporal locality in big data workloads [15, 39, 7, 16]. Studies also show that most objects are written once and never read after certain amount of time, and there is repeated accesses to small number of files.

To highlight workloads' characteristics, we analyze the access pattern of big data analytic workloads. We use the 2010 publicly available trace from Facebook [15], Yahoo [48] as well as a 2019 trace from Two Sigma, and made the following observations; (i) there is sufficient re-use of data from a data lake to make a caching solution valuable, (ii) the re-use pattern is complex enough that manual copying of data into local storage is inefficient, (iii) a number of files are accessed early and remain popular throughout the trace while other files become hot for periods of time at intermediate points in the trace; suggesting the value of a dynamic caching mechanism, (iv) datasets are accessed repeatedly by the same analytic clusters and between different analytic clusters. Our observations and existing literature suggest to exploit

the locality to cache datasets in memory or SSDs, to improve performance for hot datasets.

Chapter 3

Related Work

In this section, we will briefly describe related works about caching and why these solutions fall short in improving efficient caching for object stores in data centers.

Caches for file and block based storage: Early file system solutions such as NFS [41] and AFS [28] cache data at the client-side to reduce the access latency for subsequent accesses issued by the same client and propose a consistency protocol. Since the data in object stores are immutable and most of the object stores adopt the eventual consistency model, much simpler cache solutions that do not require a complicated cache consistency protocol can be implemented for object stores. Researchers also focus on acceleration of the guest VMs performance [12, 10, 35] in data centers. However, these solutions designed for non-shared data such as virtual disk images where each virtual disk image is stored separately. These

Caches for Bigdata Analytic: Several projects have explored caching (mostly in-memory) for big data analytics [25, 49, 8, 32, 33, 5, 19]. For instance [8, 33, 25, 19] provide an in-memory cache run within a single cluster and can not be applied as a data center scale cache solution. Among these works, Pacman [8] proposes a task-aware allocation of caches and Quartet [19] relies on achieving memory locality by reordering tasks based on the cache status. The most mature and relevant project, Alluxio is formerly known as Tachyon [32], a standalone caching service that can be deployed above existing data lakes.These solutions are designed for specific big data framework and design to be a cache for a single cluster. However, we would like to design a data center scale cache solution which is shared among many computer cluster and is design to address the network congestion problem in data centers.

Cooperative Caching: There is a rich research in *cooperative caching* [18, 9, 47, 20, 18, 30] that ranges from CDNs [30, 47, 22, 9], to network file systems [18, 42, 23], and multiprocessors architectures [20, 14, 21]. These systems explore how to create larger shared aggregate caches using local caches, and these solutions can provide a benefit in a bandwidth constrained data centers. [18, 42] reduce the latency by introducing a dynamic cache management algorithm to find the optimal cache size allocation for dynamic caching demands of local/global accesses. The rich research in cooperative caching can help us to explore how to create larger shared aggregate caches using local caches for entire data centers.

Improving applicability of SSDs for caching and tiering Data centers incorporate higher speed storage devices such as SSDs into the existing hierarchy of a

storage system either as tiered storage or caching within the existing storage clusters to improve the I/O performance. (Add FlashTier,DIDACache, mercury). Some vendors provide a transparent flash layer for high-end storage server to reduce the disk latency. Enterprises like Facebook [29], Google [4] have deployed SSD-based either as caching approaches or as a storage tier [13] which migrates data between two tiers automatically on their production clusters. These solutions do not focus on network congestion but to slow disk speeds.

Cache replacement policies Cache management is a heavily studied area of research. Most of the applied systems today use LRU and LFU for their simplicity. Among these works ARC [34] is a prominent example that leverages shadow queues to dynamically allocate memory between LRU and LFU queues. Cliffhanger [17] also uses shadow queues to adaptively sizes per-workload cache sizes to increase hit rates. Pacman [8] is specially designed for big data analytics workload which increases cache efficiency by guaranteeing that all data needed to start a new job is read into its caches at the same time. These solutions could be use in the proposed cache solution to improve the performance of applications.

Chapter 4

Existing Work

This section presents the design of a data center-scale caching architecture based on an immutable object abstraction. We will review the design goals of the cache architecture, the initial prototype, its implementation, and the dynamic cache size management algorithm. We close the section by discussing the proposed research needed to be accomplished for the completion of the thesis.

4.1 Fundamental Design Goals and Principles

Here we describe the design goals of data center scale cache architecture which mitigates network imbalances by caching data on the access side of bottlenecks. To improve the throughput of widely-shared immutable object stores in bandwidth constrained data centers, our cache architecture is designed in a way that is;

- **Extension of the data lake:** A real working caching mechanism which is easy-to-implement and integrated into an object storage solution. We would like to avoid the complexity of deploying and managing separate caches on top of the data lake, and efficiently use the available cache resources by sharing the cache among many clusters.
- **Improving the Performance:** Speed up run-time of the workloads with locality, support both reads and writes, and increase the throughput of object stores.
- **Reduce demand on network:** We want to design a system which improves the storage throughput and reduces the contention on data center network.

All these goals have implications on our design, and we list the design principles as follows;
(i)Transparency: For integrating the cache into the data lake easily, cache should be transparent to client and should offer the same network interface (S3) as the unmodified system, allowing access from unmodified clients.
(ii)Scalability: Cache bandwidth and storage should scale naturally with the number of clients and cache blocks should be identified without a need of centralized metadata service.
(iii)Adaptability: Data placement, eviction, etc. should be determined without

user input, based on measured run-time behavior such as access patterns and observed network bottlenecks.

(iv)Resiliency: Failures and recovery or resource addition should be handled automatically, minimizing disruption to clients

These requirements and principles shape the initial prototype *"D3N"* architecture which is an extension to the Ceph RADOS Gateway (RGW) and is targeted to analysis frameworks such as Hadoop and Spark, caching objects accessed via the S3A and similar connectors. In the first implementation, for simplicity and resiliency, as well as for integration in existing storage solutions, all caching and routing decision are based on local information rather than central coordination.

4.2 Initial Read Cache Prototype

We initially implement the first version of the cache prototype as a proof of concept to show that; (i) the caching system can be integrated into an existing storage solution, (ii) improves the storage throughput and (iii) is transparent to the client.

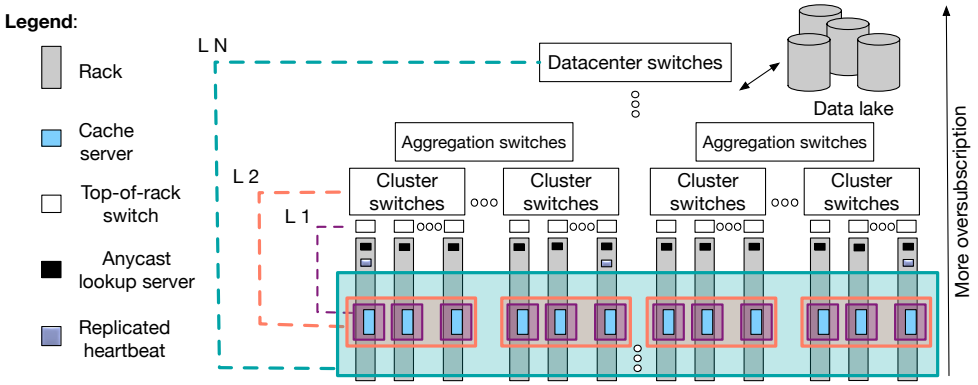

Figure 4·1: D3N architecture: Cache services (light purple) serve requests from nearby servers (blue); data blocks are distributed across pools of caches via consistent hashing. Lookup servers (black) identify L_1 caches to clients.

4·1 shows the architecture of the caching mechanism, which is deployed across a set of racks (or cluster), caching data from a remote data lake. D3N has three components: *cache servers*, to which client requests are directed; *a lookup service*, used by clients to locate their "local" cache, and a *heartbeat service* to track the set of active caches. Cache servers act as proxies for the back-end object store, storing data locally for reuse. The preferred storage media for these servers is high-speed SSD, providing a combination of high capacity and sufficient bandwidth to saturate network links; Clients access the lookup service (implemented as a DNS server) via IP anycast [1], and query it both periodically and upon request timeout to handle events such as recovery of failed cache servers, client VM migration, or other non-failure

events which might affect optimal client-to-cache pairing. The heartbeat protocol is used by the cache lookup service to find active caches, as well as by the cache servers themselves to determine how data is to be distributed.

To scale caching resources with client demand we deploy one or more cache servers in each rack; we have found one per rack of 1U servers to be adequate for Java-based analytics workloads, but more may be needed for more I/O-intensive applications. Each cache server acts as an local L_1 cache for its local clients, caching data they have requested, while L_2 is formed by aggregating resources across multiple caches and store data sets for entire data center. This aggregation requires additional hops across the intra-cluster network to resolve local L_1 misses; however the result is a reduction in load on the external network and data lake. Each chunk has a *"home location"* within the L_2 cache, and L_1 misses are forwarded to the chunk home location. Only in the event of a miss at the home location is a request forwarded to the data lake, the results of which are cached at both the home (i.e. L_2) and client-serving (L_1) locations.

4.2.1 Implementation

Our prototype implements the cache architecture and consists of modifications to the Ceph RADOS Gateway(RGW) [38], allowing use by any framework which supports the S3 or Swift object interfaces (e.g. Hadoop, Spark, Storm, and Flink). It stores cached data in 4 MB blocks as individual files on an SSD-backed file system. The implementation added or modified 2,500 lines of code, or about 3% of the 68,000-line RGW code base. As part of our collaboration with Red Hat [2], the first version of read cache is currently being integrated into the standard Ceph distribution (upstreamed) by Red Hat engineers.

In a typical Ceph/RGW deployment, client requests (e.g. using the S3A HDFS compatible connector) are load-balanced across multiple RGW instances. Each S3/Swift object is divided into fixed-sized blocks (4 MiB by default) which are stored as individual RADOS objects; in the typical configuration each block is replicated on three physical storage devices on independent hosts, chosen by the CRUSH hash algorithm [46].

In the D3N implementation, all client requests are routed to a rack-local cache server, where each block in a request is identified by its object ID and offset, and stored as a file on a local file system backed by striped SSDs. Each cache server independently uses LRU-based replacement policies to determine when to add or evict a block. Cache hits are read from files; misses are redirected to another cache server via consistent hashing [31], forming a cooperative cache. Misses in the cooperative cache are in turn fetched from the back-end storage via the unmodified RGW logic.

4.2.2 Dynamic Cache Management

In a data center, big data workloads change constantly throughout the day, as a result, the constant demand for network and storage fluctuates. To react to these fluctuations cache and network resources must be allocated carefully based on the demand. For instance, the network to back end traffic might be congested, and in such a case the cache should store more data for global accesses, or if workloads repeatedly process the same data sets and there is limited sharing among workloads, then data sets should cache in rack local cache servers. To fully utilize bandwidth within each layer under dynamic conditions, we present an algorithm that dynamically adjusts cache sizes of each layer based on observed workload patterns and network congestion.

The algorithm dynamically adapts the fraction of cache devoted to local vs. global requests at each cache server, with a per-layer eviction algorithm (e.g. LRU) used within each pool; chunks shared between L_1 and L_2 are purged when they have been evicted from both. In particular, at each layer caches tracks both miss overhead and the Miss Ratio Curve (MRC), using a shadow LRU list for MRC tracking. This information allows periodic adjustment of cache allocation: the MRCs may be used to predict the change in L_1 and L_2 hit rates if a specific amount capacity is moved from L_1 to L_2 or vice versa, and mean response times for L_1 and L_2 misses can then be used to gauge the overall impact of such a change. This approach naturally adapts the size of L_1 and L_2 based on the working set of the applications and any network bottlenecks that arise.

Chapter 5

Proposed Research

The initial prototype does not separate the policy from the mechanism and does not provide a write cache. We focus on these two limitations of the initial prototype and will explore potential ways to improve these limitations. The initial prototype combines its caching mechanism with a specific caching policy, determining where to cache data and where to request it from if not found locally. Like many other cache solutions [3, 24], our initial prototype uses consistent hashing [31] which determines the policy for where to place and retrieve data. Although consistent hashing evenly distributes data and does not depend on a centralized lookup server, it brings certain limitations. For instance, consistent hashing does not consider the load imbalances or bandwidth capacity of servers, and it does not handle well dynamic and skewed workloads. In the current design, there is no global view of the cache status, and each cache server manages its own cache based on the local information. To improve the performance, future work will add an interface for separating mechanisms from policy, and allow more flexible object placement strategies based on the global cache state.

Secondly, we want to provide a proper persistence layer in the RGW that could be used for a write-back cache. This layer will improve the storage throughput for write-request which can be critical for certain workloads. We would like intermediate data to be created, written, read and deleted without ever going back to the data lake.

5.1 Directory-based Approach

The second version of the cache architecture introduces *"the directory-based approach"* to locating data replicas and investigate whether the benefits in flexible object placement would outweigh the overhead of directory access. The directory will allow us to separate the policy from the mechanism and it provides more flexible data placement. We propose developing a globally shared directory that would maintain information about the location of each S3 object and corresponding blocks and information about the cache servers such as to request rate, eviction rate which can be used by cache management decisions. We will benefit from the directory to maintain any critical information about object accesses. This allows us to learn from the past access patterns (and cache behavior) to predict future accesses that may repeat and result in

increasing the overall storage throughput of the data center.

Our main goal with the directory-based approach is to (i) improve the performance of big data analytics jobs by reducing the traffic to the data lake and reducing the redundancy in the cache if it's keeping other things from being cached, (ii) learn from past caching behavior to predict future caching priorities, (iii) balance the request load across cache servers, (iv) enable whole S3 object level operations, (v) enable write-back caching, where the directory will need to indicate if the object is cached in the write-back cache or back-end data lake. (vi) provide a platform to explore different algorithms and policies.

This thesis should address the following questions regarding the directory implementation: What are the consistency and resilience requirements for the directory?, What protections are needed if client systems are to access it? To address these questions, this thesis will presents our design choices for the consistency and resilience requirement for the shared directory, which will be implemented using a shared reliable database. For the write-back cache, the directory must provide strong reliability guarantees under failures to prevent unrecoverable loss of written data. For the read cache, "cold start" is an option during the failures, however, applications will experience a significant performance drop in both latency and bandwidth with a cold cache. In order to prevent performance degradations and reduce back-end load, we propose to implement a durable and highly-available directory for read-cache that is needed to be brought back to the warm state after a failure. We improve the durability of the directory in the data center by either replicating directory, store the directory persistently in SSD or disks, or reconstructing the directory of objects that had been present in the write-back cache must by scanning the whole cache servers and collecting the file system information. In this thesis, in particular, we explore the advantages (and challenges) of the directory implementation and show the durability and resilience requirements of it by considering the immutable objects.

5.2 Prefetching

As we mentioned in the previous section 5.1, the directory will allow us to separate policy and mechanism. We would like to use the directory implementation for defining new cache management policies that may lead to more effective caching than traditional cache management strategies. We would like to determine better cache insertion and cache update policies by considering the load on cache servers, locality of accesses and popularity of the object when storing objects in the cache. Moreover, by using the information stored in the directory we would like to enable prefetching to improve the performance of workload by fetching useful data in advance. The implementation of the directory will focus on identifying what information has to be stored in order to improve the cache management policies and accuracy of the prediction for future cache accesses.

5.2.1 Write-Back Cache

Writing data back to original data lake works well for read-heavy workloads, however, write-heavy workloads observe network latencies and contention on the storage side. Besides, caching the output and intermediate data sets can be crucial for the performance of certain big data workloads (e.g batch analytics, iterative jobs). For instance, a great number of big data workloads in data centers are batch analytic where multiple jobs are processed in a sequence. In the batch analysis, the output becomes the input to the consecutive stage. Moreover, the output of a job can be re-accessed across multiple workloads. Slow writes can significantly hurt the performance of job pipelines, where one job consumes the output of another. Therefore, caching output or intermediate data sets is crucial for the performance of certain big data workloads.

To speed up the access time for output and/or intermediate data sets, we propose to design and implement a write-back cache mechanism without compromising fault-tolerance, which allows clients to continue as soon as data is written to the cache. Write-back cache design needs to address all the issues of ensuring redundancy before responding to write operations and efficiently recovering from failures. Similar to read cache, the design of the write-back cache should take into account the unique characteristic of immutable object stores, the underlying storage system media and workloads pattern.

Chapter 6

Thesis Progress and Timeline

Table 6.1 presents a coarse breakdown of my timeline for completion of this dissertation. A significant portion of the remaining time is dedicated to the implementing and evaluating the directory based approach and write-back cache. While I have prepared some writing for previous paper submission, the proposed improvement of the initial implementation included in this proposal have not yet been described at length.

Work	**Completion Date**
Read cache	completed
Dynamic cache size management	completed
Directory-based approach	Mar. 2020
Write cache	May. 2020
Thesis writing	July 2020
Thesis defense	Aug. 2020

Table 6.1: Thesis Progress and Timeline

Bibliography

[1] Operation of anycast services. `https://tools.ietf.org/html/rfc4786`.

[2] Red hat. `https://www.redhat.com/en`.

[3] Varnish http cache. `https://varnish-cache.org/`.

[4] Albrecht, C., Merchant, A., Stokely, M., Waliji, M., Labelle, F., Coehlo, N., Shi, X., and Schrock, E. Janus: Optimal flash provisioning for cloud storage workloads. In *Proceedings of the USENIX Annual Technical Conference* (2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA, 2013), pp. 91–102.

[5] Alluxio - open source memory speed virtual distributed storage. `https://www.alluxio.org`.

[6] Amazon Web Services, I. Amazon Simple Storage Service (S3) — Cloud Storage — AWS. available at `aws.amazon.com/s3/`.

[7] Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., and Harris, E. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 287–300.

[8] Ananthanarayanan, G., Ghodsi, A., Warfield, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 267–280.

[9] Annapureddy, S., Freedman, M. J., and Mazières, D. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 129–142.

[10] Arteaga, D., Cabrera, J., Xu, J., Sundararaman, S., and Zhao, M. Cloudcache: On-demand flash cache management for cloud computing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 355–369.

[11] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2010), IMC '10, ACM, pp. 267–280.

[12] Byan, S., Lentini, J., Madan, A., and Pabn, L. Mercury: Host-side flash caching for the data center. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)* (April 2012), pp. 1–12.

[13] Cache tiering. `http://docs.ceph.com/docs/master/rados/operations/cache-tiering/`.

[14] Chang, J., and Sohi, G. S. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2006), ISCA '06, IEEE Computer Society, pp. 264–276.

[15] Chen, Y., Alspaugh, S., and Katz, R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1802–1813.

[16] Chen, Y. L., Mu, S., Li, J., Huang, C., Li, J., Ogus, A., and Phillips, D. Giza: Erasure coding objects across global data centers. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 539–551.

[17] Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 379–392.

[18] Dahlin, M. D., Wang, R. Y., Anderson, T. E., and Patterson, D. A. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1994), OSDI '94, USENIX Association.

[19] Deslauriers, F., McCormick, P., Amvrosiadis, G., Goel, A., and Brown, A. D. Quartet: Harmonizing task scheduling and caching for cluster computing. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016), USENIX Association.

[20] Dybdahl, H., and Stenstrom, P. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture* (Feb 2007), pp. 2–12.

[21] Dybdahl, H., and Stenstrom, P. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture* (Feb 2007), pp. 2–12.

[22] Fan, L., Cao, P., Almeida, J., and Broder, A. Z. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw. 8*, 3 (June 2000), 281–293.

[23] Feeley, M. J., Morgan, W. E., Pighin, E. P., Karlin, A. R., Levy, H. M., and Thekkath, C. A. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 201–212.

[24] Fitzpatrick., B. Memcached. available at `http://memcached.org/`.

[25] Floratou, A., Megiddo, N., Potti, N., Özcan, F., Kale, U., and Schmitz-Hermes, J. Adaptive caching in big sql using the hdfs cache. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 321–333.

[26] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[27] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Vl2: A scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev. 39*, 4 (Aug. 2009), 51–62.

[28] Howard, J. H. An overview of the andrew file system. In *in Winter 1988 USENIX Conference Proceedings* (1988), pp. 23–26.

[29] Inc., F. Facebook flashcache. `https://github.com/facebookarchive/flashcache`.

[30] Iyer, S., Rowstron, A., and Druschel, P. Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), PODC '02, ACM, pp. 213–222.

[31] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[32] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. *Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks.* ACM SoCC'14, Nov. 2014.

[33] McCabe, C., and Wang, A. Hdfs read caching. `http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching/`.

[34] Megiddo, N., and Modha, D. S. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 115–130.

[35] Meng, F., Zhou, L., Ma, X., Uttamchandani, S., and Liu, D. vcacheshare: Automated server flash cache space management in a virtualization environment. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 133–144.

[36] The massachusetts green high performance computeing center (mghpcc). `https://www.mghpcc.org`.

[37] Nightingale, E. B., Elson, J., Fan, J., Hofmann, O., Howell, J., and Suzue, Y. Flat datacenter storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 1–15.

[38] Ceph object gateway.

[39] Ren, K., Kwon, Y., Balazinska, M., and Howe, B. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow. 6*, 10 (Aug. 2013), 853–864.

[40] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev. 45*, 4 (Aug. 2015), 123–137.

[41] Sandberg, R., Golgberg, D., Kleiman, S., Walsh, D., and Lyon, B. Innovations in internetworking. Artech House, Inc., Norwood, MA, USA, 1988, ch. Design and Implementation of the Sun Network Filesystem, pp. 379–390.

[42] Sarkar, P., Sarkar, P., and Hartman, J. H. Hint-based cooperative caching. *ACM Trans. Comput. Syst. 18*, 4 (Nov. 2000), 387–419.

[43] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J.,

Tanda, E., Wanderer, J., Hölzle, U., stuart, S., and Vahdat, A. *Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network*, vol. 45. ACM, Sept. 2015.

[44] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hlzle, U., Stuart, S., and Vahdat, A. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM SIGCOMM Conference* (2015), SIGCOMM '15, ACM, pp. 183–197.

[45] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 307–320.

[46] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[47] Yadgar, G., Factor, M., and Schuster, A. Cooperative caching with return on investment. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2013), pp. 1–13.

[48] S2 - yahoo! statistical information regarding files and access pattern to files in one of yahoo's clusters. `https://webscope.sandbox.yahoo.com/catalog.php?datatype=s`.

[49] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.