D3N: A multi-layer cache for the rest of us

Emine Ugur Kaynar ⋆, Mania Abdi ◇, Mohammad Hossein Hajkazemi ◇, Ata Turk †
Raja R. Sambasivan §, David Cohen ◁, Larry Rudolph ‡, Peter Desnoyers ◇, Orran Krieger ⋆
⋆*Boston University*, ◇*Northeastern University*,†*State Street*, §*Tufts University*, ◁*Intel*, ‡*Two Sigma*

Current caching methods for improving the performance of big-data jobs assume high (e.g., full bi-section) bandwidth; however many enterprise data centers and co-location facilities have large network imbalances due to over-subscription and incremental networking upgrades. We describe D3N, a multi-layer cooperative caching architecture that mitigates network imbalances by caching data on the access side of each layer of a hierarchical network topology, adaptively adjusting cache sizes of each layer based on observed workload patterns and network congestion. We have added (and submitted upstream) a 2-layer D3N cache to the Ceph RADOS Gateway; read bandwidth achieves the 5GB/s speed of our SSDs, and we show that it substantially improves big-data job performance while reducing network traffic.

I. Introduction

In today's world, data is king [29]; organizations' success or failure depends on the amount and variety of data they collect and the speed of gleaning critical insights from it. As a result, many datacenters include low-cost, centralized storage repositories, called *data lakes*, to store and share vast datasets. This data may then be analyzed by big-data frameworks such as Hadoop [28] and Spark [50]), with data access accelerated by caching [4], [6], [15], [35] to move 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. Data access may be significantly constrained by over-subscribed network links both within and between clusters— constraints due to both design decisions and the piecemeal process (which we term *organic growth*) by which compute and networking equipment is deployed in non-hyper-scale datacenters.

In even the best-designed modern datacenters, rack *uplinks*—i.e. the connections between top-of-rack switches and the rest of the datacenter—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 oversubscription 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. datacenter-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. As an example, the authors recently deployed (in a shared institutional datacenter) 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.

To address these network limitations we present a multi-layer throughput-oriented "cooperative caching" architecture, Datacenter-Data-Delivery Network (D3N). D3N uses high-speed storage (e.g. NVMe flash or DRAM) to cache datasets on the access side of links in a hierarchical network, dynamically allocating cache space across layers based on observed workload patterns and link speeds, so that cache capacity is preferentially used for traffic crossing the most over-subscribed links.

A fundamental goal of D3N is to allow simplified integration into existing data lakes [18], [47] to enable caching to be transparently introduced into datacenters, to support efficient caching of objects widely shared across clusters deployed by different organizations, and to avoid the complexity of managing a separate caching service on top of the data lake. Moreover, integration into the data lake (rather than a separate service) offers opportunities for continuous enhancement of functionality and performance as the data lake software evolves, e.g., improved meta-data handling or simplifying consistency.

To facilitate integration into a data lake, D3N today makes all policy decisions based purely on information local to cache servers and is designed to only rely on interfaces typically available in data lakes. We have developed an implementation in Ceph [47], an object-based storage system [18] commonly used to implement data lakes. Using high-speed SSD for caching, we modify the Ceph RADOS gateway (RGW), which provides Ceph-backed Swift [8]

978-1-7281-0858-2/19/$31.00 ©2019 IEEE

and S3-compatible [5] interfaces, protocols supported by most big data frameworks. D3N has required no changes to the interfaces of any Ceph services, involves no additional meta-data services (e.g, to locate cached blocks), and all policies are implemented based purely on local information.

The most similar previous work is Alluxio [4], a caching service that can be deployed above existing data lakes and is accessed via a client library rather than a network protocol. In addition to the fundamentally different design goals, D3N differs from Alluxio in borrowing heavily from the rich research in *cooperative caching* [7], [17], [19], [32], [49] systems which explore how to create larger shared aggregate caches using local caches; Alluxio policies for distribution and cache replacement are dictated purely by the dataset user.

The main contributions of our work are:

1) We demonstrate that an effective cooperative cache can be integrated into a production data lake in a practical fashion without requiring changes to client or server interfaces and without introducing additional meta-data services. D3N (with static cache partitioning[1]) is currently being productized and integrated into upstream Ceph by an industry development team.
2) We demonstrate that an implementation can be highly efficient. We show with micro-benchmarks that our implementation introduced into the production Ceph code base can support per-cache read speeds of 5 GB/s, fully exploiting the SSDs and NICs in our system. We also demonstrate with large scale datacenter traces that D3N achieves significant performance improvements for realistic workloads—up to a 3x reduction in runtime vs. uncached when bandwidth-constrained.
3) We present and evaluate a novel adaptive cache partitioning algorithm using observed throughput and access patterns to optimize cache capacity division between rack-local and cluster-wide data, showing gains of up to 30% vs. fixed allocation in adapting to different access patterns and in responding to network contention.

II. Motivation

In Figure 1 we see a simplified view of the datacenter topology motivating D3N: racks of servers (dark blue) connected by top-of-rack (ToR) switches, an over-subscribed inter-rack network within clusters, and a further over-subscribed datacenter 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.

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.

[1]The adaptive partitioning algorithm is still experimental research.

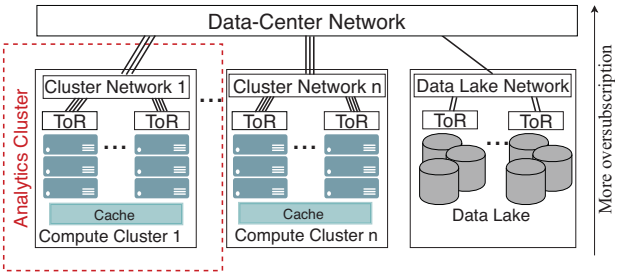


Fig. 1: A typical private datacenter. Servers installed within racks are connected by a hierarchical, over-subscribed data-center network. The layers correspond to ToR, cluster, and datacenter-network-wide switches. Sets of racks and their networking elements form compute clusters with the data lake in a separate cluster.

In Figure 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 data sets. Such data lakes are typically implemented by object stores, such as Ceph [47] or AWS S3 [5], 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.

We focus on data access for analysis by jobs (e.g. big-data frameworks) running on *analysis clusters*; comprised of single, partial, or multiple of the physical clusters depicted in Figure 1. Analysis frameworks may run directly on cluster servers, or on virtual machines [9] or containers [45] allowing migration within or across compute clusters. We specifically focus on workloads which directly access data via connectors such as S3A [42] for Hadoop, which provides HDFS-compatible access to S3 and compatible object stores. In such cases local storage is typically used for ephemeral data, with the data lake for long term storage of inputs and outputs.

A simple numerical model can be used to show the value of D3N's multi-level cache in such a bandwidth constrained environments. With D3N, we would place a cache server in each rack, and share the SSD (1 TB in this case) between L_1 cache dedicated to caching rack-local accesses and L_2 cache dedicated to caching remote accesses. The model is:

$$
\begin{aligned}
r &= \frac{1}{\frac{1-m1}{r_{L1}} + \frac{1-m1-m2}{r_{L2}} + \frac{m2}{r_{DL}}} \\
m1 &= MRC(F_{L1} \cdot C) \\
m2 &= MRC((1-F_{L1})C \cdot N)
\end{aligned}
\tag{1}
$$

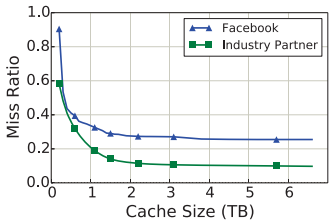
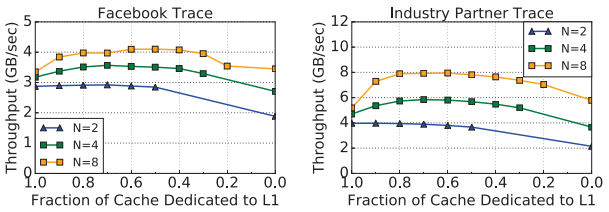


Fig. 2: Miss ratio curves of Facebook & industry trace.



(a) Facebook trace. **(b)** Industrial partner trace.

Fig. 3: Modeled throughput of storage cluster for varied numbers of caching nodes (N), ratio of L_1 and L_2 cache. 1 TB cache per node, 40 Gbit cache / 20 Gbit for storage traffic inter-rack / 10 Gbit to data lake.

where r_{L1}, r_{L2}, r_{DL} are L_1 hit, L_2 hit and L_2 miss bandwidths, F_{L1} is the fraction of cache devoted to L_1, C and N are the capacity of a single cache and number of caches, and *MRC* is the Miss Ratio Curve (MRC) for the workload.

Figure 3 shows the storage throughput for a two level cache for 2, 4, and 8 cache nodes, as we vary the L_1 fraction of the cache from 100% (fully-local) to a minimum of $\frac{1}{N}$ (due to unified caching). We set r_{L1} r_{L2} and r_{DL} assuming a datacenter with 40 Gbit ToR switches, 40 servers per rack and 100 Gbit uplinks, resulting in a 16:1 oversubscription[2] The MRCs (Figure 2) are calculated from Facebook and an industry partner traces (see Section V-A).

Our simple numeric model underestimates the value of the two-level cache in that it assumes that the bandwidth between L_2 caches is not contended, and that there is no locality in requests from the same rack or cluster. Even with these pessimistic assumptions we see that the multi-level approach offers better performance than either a pure L_1 or pure L_2 approach for 4 cache servers or more. The industry trace, with its 90% reuse rate at eight cache nodes shows a 40% improvement over a pure L_1 cache and a 25% improvement over a pure L_2 cache.

III. D3N Architecture

D3N is a caching architecture for large immutable objects accessed from multiple client machines. It caches data on the access side of potential network bottlenecks, e.g. for data analytics over source data residing in a large remote object store with S3-like [5] semantics. We assume that objects are large and accessed in their entirety, possibly across multiple clients, making throughput the dominant performance metric.

D3N was designed to be a transparent extension of the data lake, with design goals including:

- **Scalability:** Cache bandwidth and storage should scale naturally with the number of clients.
- **Adaptability:** Data placement, eviction, etc. should be determined without user input, based on measured run-time behavior.
- **Transparency:** D3N should offer the same network interface (S3) as the unmodified system, allowing access from unmodified clients
- **Resiliency:** Failures and recovery or resource addition should be handled automatically, minimizing disruption to clients.

These requirements shape the D3N architecture: a multi-level cooperative cache, where levels correspond to wider domains of cooperation. Cache servers are deployed alongside clients on a per-rack basis, with a discovery service used by clients to locate their "nearest" cache; as the number of racks accessing the cache increases, the number of cache servers naturally scales. The layer where data is cached adapts automatically to the access patterns of the applications and the bandwidth available between cache servers. 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.

A. D3N Components and Organization

D3N has 3 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, 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; however other media (RAM, NVM) may be used. 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 caches, client VM migration, or other events which might affect optimal client-to-cache pairing.

The heartbeat protocol is used by the lookup service to find active caches, as well as by the cache servers themselves to determine how data is to be distributed. (If desired, the heartbeat protocol may be replaced by another coordination service, such as Zookeeper [31].)

These components may be seen in Figure 4. Here D3N is deployed across a set of racks (or *cluster*), caching data from a remote data lake. We assume that the resources within these racks may be optimized for their tasks,

[2] We set r_{L1} to 40 Gbit, r_{L2} to 20 Gbit and r_{DL} to 10 Gbit/s; with a 16:1 oversubscription it seems optimistic to assume 20 Gbit of the 100 Gbit uplink is available for storage traffic, and 10 Gbit/s is roughly the bandwidth we have seen from our 90 spindle Ceph cluster.

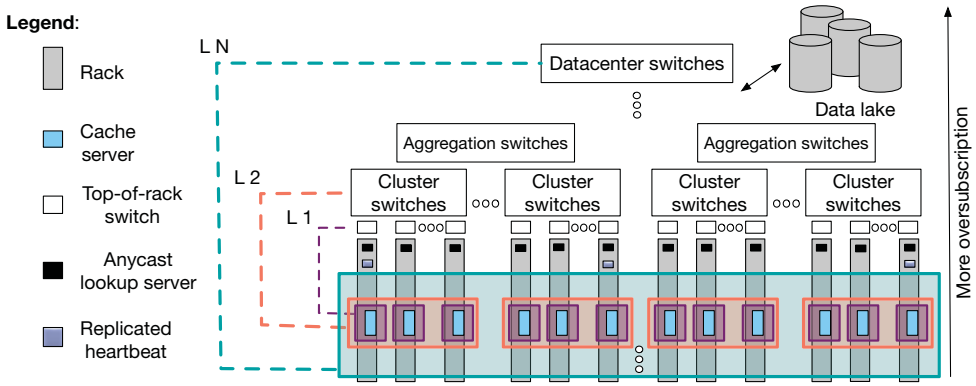


Fig. 4: D3N architecture. Cache services (light purple) serve requests from nearby servers (blue); data blocks are distributed across pools of caches via consistent hashing. Lookup servers (black) identify L_1 caches to clients.

incorporating sufficient caching and network resources to support the expected demand from the client machines in the cluster. Conversely, we assume that bandwidth may be limited between this installation and the data lake, or that the data lake itself may have intrinsic performance limits (due to e.g. disk bandwidth), and that addressing these limits may be outside the control of the parties responsible for installing this cluster.

To scale caching resources with 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 L_1 cache for its local clients, caching requested data, while successive cache layers are formed by aggregating resources across multiple caches. 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.

Since the target applications often split large objects into sub-ranges across multiple clients (e.g. mappers), we cache objects in relatively small chunks, 4 MiB in our implementation. 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. The L_1 and L_2 caches are *unified*: L_1 requests for a chunk received at that chunk's home location result in a single cached copy of the data.

As seen in Figure 4, a further level of aggregation may be used to create an L_3 cache, and it is possible (although unlikely to be practical) to extend this to higher layers. We focus on L_1 and L_2 in this work, and leave the details of extension to L_3 to future work. With a 2-level cache the home location of a chunk is determined via consistent hashing [34]; a reasonable approach when all cache servers are within a single cluster.

Cache replacement is non-trivial in a multi-layer cache where layers compete for the same pool of resources. D3N dynamically adapts the fraction of cache devoted to L_1 vs. L_2 at each D3N 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 D3N tracks both miss overhead and the Miss Ratio Curve (MRC), using (in our implementation) 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 capacity is moved from L_1 to L_2 or vice versa, and mean response times for L_1 and L_2 misses used to estimate impact of such a change. This approach adapts the size of L_1 and L_2 based on the application working set and network bottlenecks.

On a *Read Operation* clients send requests to the closest L_1 service as identified by the lookup service. For blocks which hit in this cache, data is returned directly. For each block which misses, consistent hashing is used to locate its home cache service in higher layer, and a request is forwarded to that service.

For *Write Operations* D3N supports *write-through*, *write-back* and *write-around*, which applications can control on a per-object basis.

B. Edge Conditions and Failure Modes

Upload visibility: Objects in D3N are immutable and should only be made visible to read operations once data is fully uploaded. (For write-through this means writes have been acknowledged by the storage back-end; for write-back that they are written to their home cache location.) We defer writing the object "header" (roughly an inode) to back-end storage (for all modes) until after data writes complete, and always read it from that location, ensuring read visibility follows the same rules as in the unmodified system.

VM Migration: During a VM migration, D3N should keep active TCP sessions connected between clients and the previous L_1 until the request has completed. For this reason, we do not use anycast to address the L_1 directly, but instead

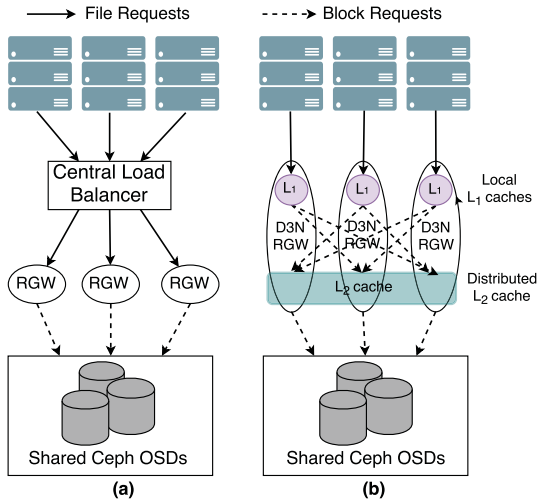


Fig. 5: Deployment of Unmodified RGW and D3N Architecture in the Datacenter. (a) Load balancing for scale-out RGW deployment. (b) D3N deployment with two-level caching.

use anycast to get to the lookup service, that provides the IP address of the L_1. When a VM is migrated it will continue to communicate to the previous L_1 until its use of the DNS mapping has expired and it opens a new connection.

Failures:: When a cache server fails, the lookup service will direct new requests to one of the remaining servers. In-process requests will be retried by the client, and thus directed to one of the remaining servers.

C. Limitations

Local cache management: Since caching decisions are performed locally within the cache pools in each layer, D3N can make globally suboptimal caching decisions. For example, a few popular blocks can be replicated across all the L_1 caches flooding the capacity and preventing caching of slightly less popular blocks.

Lack of fairness: Compute clusters participating in D3N share resources such as rack space, storage, power, and network bandwidth with D3N. Even though D3N tries to provide a common good by eliminating network bottlenecks, the individual benefits each cluster gets from D3N may be different and disproportionate to the resources they provide.

IV. Implementation within Ceph

Our prototype implements two levels of cache and consists of modifications to the Ceph RADOS Gateway [47] (RGW), allowing use by any framework which supports the S3 or Swift object interfaces (e.g. Hadoop, Spark, Storm, and Flink). The implementation added or modified 2,500 lines of code, or about 3% of the 68,000-line RGW code base. It implements a L_1 and L_2 cache, storing cached data in 4 MB blocks as individual files on an SSD-backed file system. The decision to base all caching and routing decisions on local information has allowed us to integrate D3N into RGW without changing any interfaces or clients.

A. Ceph RGW background

Ceph is a replicated, mutable object store, accessible via the RADOS protocol, and a suite of services implemented on top. The RADOS gateway (RGW) is one such service, providing S3/Swift compatible object storage interfaces, using multiple RADOS objects—much like a file system uses disk blocks—to store a single RGW object and its header. RGW is highly scalable, since it is in effect a stateless translator from S3/Swift to RADOS requests, and due to its S3-compatible interface it may be accessed directly by most big-data applications.

A typical Ceph/RGW deployment is shown in Figure 5(a). 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 MB by default) which are stored as individual RADOS objects; in the typical configuration each block is replicated on three independent hosts, chosen by the CRUSH algorithm [48]. RGW has a window of 4 outstanding requests per connection, limiting single-stream throughput to no more than four times that of a single device (e.g. disk, in capacity-optimized deployments), but with sufficient connections will spread load across all devices in the cluster.

B. D-RGW: our D3N prototype

Our prototype, D-RGW, implements two levels of cache within RGW (see Figure 5(b)). All client requests are routed to a rack-local D-RGW L_1 cache, 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. Cache hits are read from files; misses are redirected to another D-RGW for L_2 lookup via S3/Swift range requests. The L_1 and L_2 caches are *unified*: one copy of a block is stored, although layer membership is tracked for eviction purposes. Several alternatives for local file I/O were evaluated—memory mapping, GNU POSIX `aio`, and native kernel asynchronous I/O (libaio)—and POSIX aio was used due to its performance in our tested configuration. Native file system I/O was fast enough to saturate our 40 Gbit network; thus SPDK [46] etc. were not considered.

Reads: Clients direct read operations to their rack-local D-RGW, which divides each request into (typically 4 MB) blocks. As described in the previous section, each block in local cache is returned immediately; other blocks are requested from their "home location" computed via consistent hashing [34]. The request is forwarded via a range request, or if the "home location" is local (i.e. a L_2 miss) a block request is sent to Ceph. As responses are received (from L_2 or Ceph) data is written to cache and returned to the client in parallel. All steps are asynchronous except for the client HTTP response, where blocks must be ordered sequentially.

Writes: Since RGW objects are *immutable*, and cannot be modified or appended to once created, write handling is greatly simplified. Upon receiving a write object request, RGW (and thus D-RGW) generates an object ID and divides the object into blocks.

Write-around mode is identical to unmodified RGW: blocks are written asynchronously to the Ceph back-end, and an additional header write updates the mapping from RGW object to blocks stored as RADOS objects; no blocks are cached. Write-through caches data at the points it would be cached by reading: L_1 writes blocks locally, to the Ceph back-end, and to their L_2 home location. The header is updated when all writes to the back-end complete, and the write is then acknowledged to the client. Write-back caches blocks in L_2 (last layer), and as soon as all data blocks are cached in L_2, updated header is written to the Ceph back-end and the write is acknowledged to the client. Dirty blocks in cache are flushed periodically, or under certain circumstances such as eviction of a block or a user flush command. In both write-through and write-back, failure during write may result in blocks prior to the point of failure being cached; however, since header has not been committed, these stale blocks will be inaccessible and eventually evicted.

C. Dynamic Cache Size Management

D3N dynamically partitions the cache space into layers to minimize mean request latency. If L_1 is too small, its miss rate will be high and there will be lots of remote access to some L_2 and if L_1 is too large, L_2 will be too small and its miss rate will be large causing even longer latencies due to fetches to the data lake. Algorithms 1 and 2 depicts our algorithm where we based the predictions of miss rates as a function of layer size on observed access patterns and measured miss latencies in the near past.

To approximate the miss rates, a shadow LRU cache *SL*, is maintained for both layers. Shadow caches have been explored in other works [15], [33], [37]. Each shadow cache is of full size, S_t and only stores the keys but not the data. There is a hit counter, HC_l, associated with each shadow cache. For an access to block b to some layer, the associated shadow cache is accessed. If b is found in location i, then the hit counter for location $\mathrm{HC}_l[i]$, is increased and the blocks rearranged to maintain the LRU ordering. If the layer has size s, then the sum of all HC[i] for $i > s$ is the miss rate for that layer.

Periodically we use the re-use distance histogram and mean miss latency measurements $\overrightarrow{L} = (L_1, L_2)$ to adapt the cache capacity allocation. We first considered a simple additive increase/decrease mechanism; however due to the wide range of possible allocations ($S_t \approx 10^6$ in our prototype) the response time of such an algorithm is very slow. Instead we search a fairly wide range[3] of possible allocations ($\pm q$, where q called adaptation limit, which has been set empirically to $0.05S_t$ in our prototype) and select the best from this range.

More specifically, Algorithm 2 shows how starting at an allocation $\overrightarrow{s} = (s_1, s_2)$, we find a new assignment $\overrightarrow{s}'$ with a predicted miss rate $\overrightarrow{m} = (m_1, m_2)$ which minimizes expected latency $m_1L_1 + m_2L_2$. We first (lines 6, 7) calculate the miss ratio curve MR_ℓ for each layer, allowing us to predict the miss rate at that layer for varying cache sizes. We then search a range of possible cache allocations $\overrightarrow{s}'$, centered at $\overrightarrow{s}$, selecting the allocation $\overrightarrow{s}'$ which minimizes the expected request latency. After adapting the cache sizes (if $\overrightarrow{s}' \neq \overrightarrow{s}$), we scale the distance histogram HC_ℓ so that it represents a moving average[4] of re-use distance frequency, balancing accuracy (from accumulating data over multiple periods) with rapid adaptation (due to the rapid decay constant).

Algorithm 1 Re-use distance measurement

```
1: b: requested block
2: ℓ: layer (1 or 2)
3: S_t: total cache size (in blocks)
4: SL_ℓ: shadow LRU list (length S_t)
5: HC_ℓ: re-use distance histogram
6: s⃗ = (s_1, s_2): cache distribution, s_1 + s_2 = S_t

   ▷ Measure re-use distance for access to block b, layer ℓ
7: procedure MEASURE(b, ℓ)
8:     if b ∈ SL_ℓ then
9:         find i s.t. SL_ℓ(i) = b                    ▷ LRU position
10:        HC_ℓ(i)++
11:    reorder SL_ℓ LRU due to access to b
```

Algorithm 2 Cache distribution adaptation

```
1: b, ℓ, S_t, s⃗, HC_ℓ: As in Algorithm 1
2: MR_ℓ: miss rate (i.e. miss ratio curve)
3: L_ℓ: measured miss latency
4: q: adaptation limit (maximum assignment change in blocks)
5: i: cache server location

   ▷ Calculate updated L_1L_2 cache distribution s⃗_new
6: procedure ADAPT
7:     for ℓ in 1, 2 do
8:         MR_ℓ(i) = Σ_{k=i}^{S_t} HC_ℓ(k)            ▷ Calculate miss ratio curve
9:     C_min = inf
10:    s_new = ∅
11:    for s⃗' in (s_1 − q, s_2 + q) ... (s_1 + q, s_2 − q) do
12:        m⃗ = (MR_1(s_1), MR_2(s_2))                 ▷ Predicted miss rate
13:        C = m_1L_1 + m_2L_2                        ▷ Expected latency
14:        if C < C_min then
15:            C_min = C
16:            s_new = s'
```

Memory Overhead and Algorithm Complexities: The overhead of the D3N adaptation algorithm includes (1) memory used for the shadow LRU lists SL_ℓ and re-use

[3]We limit the space searched, and thus the absolute magnitude of any correction, in order to bound the eviction overhead after a large change in allocation.

[4] Since the incoming counts are not scaled, the expectation of HC_ℓ is actually $2\times$ the mean for a single collection period, a constant factor which does not affect the location of the optimal point.

TABLE I: Configurations of nodes in the cluster.

	Compute Node	Storage Node	Cache Node
`CPU`	1x Intel E5-2650	2x Intel E5-2650v2	2x Intel E5-2699v3
`Ram`	64 GB	128 GB	128 GB
`Disk`	2x 1.8 TB HDDs 5400 RPM	9x 3.6 TB HDDs 7200 RPM	2x Intel P3600 1.6 TB NVMe SSDs (RAID0)
`Network`	10Gb/s	10Gb/s	2x40Gb/s

distance histograms HC_ℓ, (2) computation to track re-use distance statistics (Algorithm 1) and (3) computation to find an optimal capacity allocation (Algorithm 2).

The shadow LRU list must store S_t different block identifiers each of which is an RGW object ID (up to 128 bytes) plus an offset (4 bytes), or $132 \cdot S_t$ bytes each for the L_1 and L_2 shadow lists; with a 4 TB cache and a block size of 4 MB, this is a total of up to 264 MB for the two layers. We use a skip list [38] to calculate LRU position in $O(logN)$ time, for an asymptotic complexity of $O(logN)$ for Algorithm 1. Locating $\overrightarrow{s'}$ in Algorithm 2 searches $2 \cdot q$ cases, where $q = O(S_t)$, for an asymptotic complexity of $O(S_t)$, although this may be reduced by using a more sophisticated minimization algorithm.

Extension to 3 layers and more: In this case Algorithm 1 is unmodified, updating e.g. SL_3 and HC_3 in the same way as for lower layers. The exhaustive search in Algorithm 2, however, is clearly infeasible for 3 or more layers, and must be replaced by a more efficient minimization algorithm such as hill climbing [15], [16].

V. Evaluation

Section V-A describes our experimental setup. Section V-B and V-C evaluates our prototype using micro and macro benchmarks. Finally, at larger scale than our experimental infrastructure, Section V-D examines the dynamic cache-size management capability of D3N and the runtime improvements it offers for real workloads.

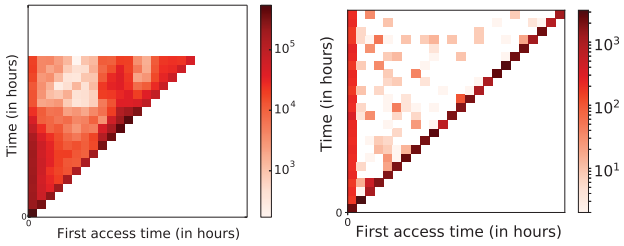
A. Experimental Setup

Environment: Experiments are executed on a private datacenter where racks interconnected in a partial-mesh topology, with the equivalent of 4×40GbE inter-rack links on each top-of-rack switch and an average distance of 3 hops between racks. The storage cluster is comprised of 10 storage nodes, running Ceph 10.2.2. Table I lists the details of the nodes used in the experiments.

Compute and cache nodes are located on 2 racks; on each rack we allocate 1 cache and 8 compute nodes for our experiments. All systems (storage, compute, and cache nodes) run RHEL (Red Hat Enterprise Linux) 7.2, with Linux kernel 3.10 and backported patches.

Workload Characterization; We use the 2010 publicly available Facebook trace [13] and a 2017 trace from an industry partner.[5] Figure 6a and Figure 6b are heat maps showing access counts to each file over time, respectively for each trace, with files identified by their time of first access.Both figures show that 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.

[5]We are working with our partner to make the trace publicly available.



(a) Industrial Partner **(b)** Facebook

Fig. 6: Re-use patterns for Industrial Partner and Facebook traces:. Files are identified by time of first access (X axis), actual access time is on Y axis; color intensity indicates access count.

B. Micro-Benchmarks

We craft and run a series of micro-benchmark on both D3N and unmodified (Vanilla) RGW to measure the imposed overhead and the maximum performance gains achievable by D3N. We submit read and write requests[6] to the Ceph data lake via D3N and the Vanilla RGW.

Figure 7 compares the D3N hit and miss throughput performances with that of the Vanilla RGW. In these experiments, for Vanilla RGW, the Ceph OSD buffer caches are flushed before each test run to ensure that data is fetched from the 90 OSD disks rather than the RAM of the OSD servers. For presenting D3N hit performance, the datasets are first prefetched into the L_1 cache of D3N before each run. The black dotted line (5.0 GB/s) represents the maximum read rate of the cache server SSDs as measured with "`dd`"; essentially the "speed of light" for our experiments. It is also coincidentally the throughput of the cache server's 40GbE NIC.

As seen in Figure 7, our implementation is efficient enough to saturate the dual NVMe SSDs and the 40 Gbit NIC; meaning almost 5× of improvement over Vanilla RGW. Also, misses on D3N (which incurs the overhead of storing missed data to SSDs in both L_1 and L_2) has ∼26% impact on the read throughput. This overhead seems easily justified if it results in subsequent hits.

Figure 8 compares D3N write performance using write-through and write-back to vanilla RGW. Write-back mode improves the write throughput 3×. Our implementation is efficient enough to saturate the capacity of the dual NVMe SSDs. Write-through, which blocks until the write has completed to Ceph, incures additional overhead over vanilla RGW of traversing multiple D3N caches and writing the data to the SSDs. As we see this overhead

[6]Requests are submitted using `curl`, a high-performance HTTP client, with eight nodes each issuing eight concurrent 4 GB read or write requests based on the experiment. The largest object that can be uploaded in a single PUT is 5 GB in S3 API.

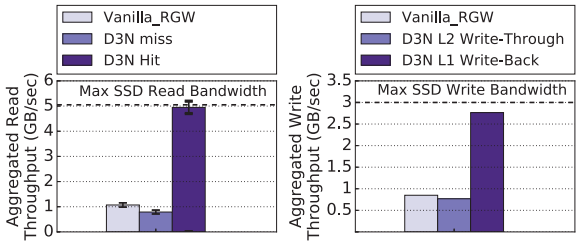


Fig. 7: D3N hits improve the read throughput 5×, saturating the dual NVMe SSDs and a 40 Gbit NIC.

Fig. 8: Write-back improves the throughput by ∼3× and write-through incurs a small (∼10%) overhead.

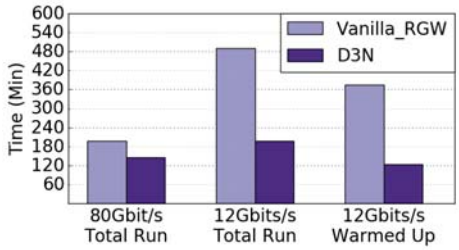


Fig. 9: Facebook workload runtime, D3N vs. vanilla RGW, full-bandwidth (80 Gbit) and throttled (12 Gbit) network to data lake, full run and after 1/3-trace warmup. D3N improves runtime by 25% with (unrealistically) high data lake bandwidth, and by 4× with a more realistic network.

is only around ∼10%. While the default policy right now is write-around (given the experimental nature of our changes), we expect users to use write-back for intermediate data sets, where resilience is less critical, and write-through in all other cases given the modest cost and the significant value for subsequent hits.

To summarize, in our environment (with a very high speed closely coupled data lake) D3N offers a 5× higher throughput on L_1 cache hits and 3× higher throughput on writes with a write-back policy (saturating the SSDs on cache servers in both cases) and incurs modest overhead on cache misses (∼26%) and write through (∼10%).

C. Macro-benchmark

We use the Facebook trace described in Section V-A to create a macro-benchmark and measure the runtime impact of D3N for more realistic workloads. To be able to test the trace over a smaller-sized cluster, we scale down the original file sizes by half[7] and then replay a part of the trace with 75% percent re-use (a low re-use ratio compared to our Industrial Partner's trace). Our workload includes 853 jobs processing 40 TB of data with a 10.1 TB footprint; 8 TB of the data is repeatedly accessed while 2.1 TB of the data is accessed only once. We preserve the original arrival order of the jobs and consider only read traffic to emulate access to a shared read-mostly multi-institutional data lake.

We create a two layer D3N configuration using two cache servers with a total 5 TB capacity. We use the production (non-adaptive) version of D3N, with a static cache of 50% to each cache layer. Aggregate bandwidth to the back-end storage cluster was 80 Gbit/s (one 40 Gbit NIC per server); additional experiments were performed with an aggregate bandwidth of 12 Gbit/s, by throttling each cache server-to-storage connection to 6 Gbit/s. Hadoop clients were emulated by a custom tool, generating HTTP requests (with a 512 MB block size) to mimic the S3 requests each mapper would have initiated, allowing all mappers to be emulated from two 36-core nodes with 40 Gbit NICs. Since the trace lacks client node information, requests were randomly assigned to each mapper with no locality, giving conservative results vs. real workloads with non-zero locality.

Figure 9 displays the trace completion time with and without D3N for the two network bandwidth configurations. The experiment was divided into two consecutive phases, warm-up and measurement. The warm-up phase consisted of the first 33% of the total trace; performance results are reported for the measurement phase alone as well as for the full workload run time (warm-up plus measurement). With an unrealistic 80 Gbit of bandwidth to the storage pool, performance improved by about 25%. With 6 Gbit from each cache server to the backend (a conservative emulation of a shared 10 Gbit connection) the full workload and measurement-only times improved by 2.4× and 3× respectively.

With only 75% re-use, we see that the cache is still highly effective, greatly increasing storage system throughput and thus application performance. Throughput also compares favorably with the traditional alternative of manually copying hot datasets into a disk-based cluster-local HDFS system. Due to the huge speed disparity between NVMe and disk, it would take 60 to 80 disk spindles across this cluster to equal the throughput of the two dual-SSD cache servers.

Another benefit is the reduction in inter-cluster traffic. Figure 10 we see cumulative data transferred from the back-end storage (sampled using `pbench`) for the 12 Gbit/s experimental configuration. With the vanilla RGW, the Ceph link is nearly always saturated, with 23 Tb of observed data transferred after the warmup phase, while D3N transfers about 5 Tb, more than a 4× improvement (e.g. allowing 4 times as many analysis jobs to share the storage backend or bottleneck links).

D. Performance of Dynamic Caching

In Section V-C, the cache space was allocated statically across co-located L_1 and L_2. To be able to evaluate the performance of the D3N partitioning algorithm at larger scale, we run a series of trace-based simulations.

In our simulation, we assume a datacenter with 10 racks, each rack containing one D3N cache server and 20 client

[7]Prior work has shown this has little or no impact on performance [6].

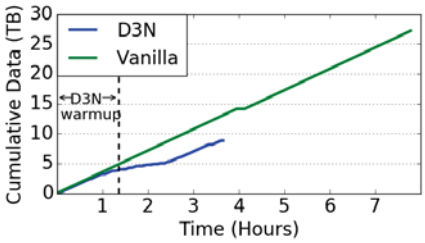


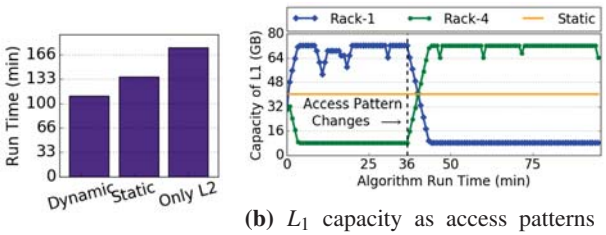
Fig. 10: Cumulative (sampled) Ceph transfers for Facebook workload, D3N vs Vanilla , 12 Gb/s. For D3N the link is saturated during the warm-up phase due to cache misses, but demand is reduced during the measurement phase due to cache hits; Vanilla takes 3× as long, saturating the link throughput.

machines. The aggregate bandwidth between clients and D3N servers is 50 GB/s (i.e. one 40 Gbit NIC per server) and the bandwidth between L_1 and L_2 is 15 GB/s, and with a rack-to-rack over-subscription of 3.3:1. Each client issues concurrent 150 requests for 4MB objects. Both synthetic traces and Facebook traces were used. The Facebook trace does not contain mapper information needed to determine request locality; therefore we generated synthetic locality information, assuming that a repeat access to a file occurred on the same rack with $p = 0.7$, and from another rack (chosen randomly) with $p = 0.3$. At simulation start, each cache was divided equally between L_1 and L_2; every 1.5 minutes the cache allocation was adjusted by up to 5% of capacity in either direction. (Sensitivity to these parameters was tested, and any combination able to adapt by at least 2% every minute was found to give equivalent performance.) Each experiment has a warm-up phase and run phase. In this section, we only report the results, which are collected during the run phase.

Adaptability to different access patterns:

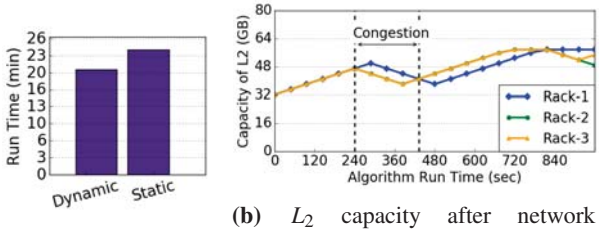
To analyze D3N's reaction to workload pattern changes over in time, we split the Facebook trace and assume that for the first 36 minutes of the trace all requests arrive from Rack-1 and after the 36 minute mark all requests arrive from Rack-4, mimicking behavior when different parts of the datacenter have high workloads at different points in time. In Figure 11(b) we see L_1 capacity for Rack-1 and Rack-4; after the access pattern changes at the 36th minute mark, Rack-4's L_1 capacity starts to increase while Rack-1's L_1 capacity decreases, indicating rapid reaction to workload pattern changes. Figure 11(a) compares the overall runtime of the workload under dynamic and static partitioning with a 50/50 L_1/L_2 assignment, and when only a single layer distributed L_2 cache is deployed. Runtime for dynamic partitioning is improved by 19% over static allocation and 36% over a single L_2 cache, respectively.

Adaptability to network load changes: Next we evaluate D3N's adaptability to changes in the network loads. We use synthetic trace, where all racks request the same set of files and the size of the requested files are bigger than the total size of the cache service. Starting



(a) Trace runtime. **(b)** L_1 capacity as access patterns change.

Fig. 11: Dynamic cache allocation: (a) runtime, static / dynamic / L_2-only; (b) L_1 capacity changes as access patterns change.



(a) Trace runtime. **(b)** L_2 capacity after network congestion.

Fig. 12: Impact of dynamic cache allocation of D3N when a network congestion occurs. (a) Runtime of static and dynamic allocation. (b) L_2 capacity with changing network congestion.

from 240th second until the 440th second, we congest the link that ties Rack-4 to other racks and the bandwidth drops from 15Gbits/s to 1Gbits/s. As a result, it becomes costly for other cache-servers to fetch data from L_2 of Rack-4. Therefore, all racks increased the capacity of their L_1 caches during the congested window.

Figure 12(b) shows the L_2 capacity of Rack-1, Rack-2, and Rack-3 before, during and after the congestion. We plot only 3 racks to make the figure more readable. We note that remaining 7 racks follow the same pattern. When the congestion is over, the dynamic algorithm slowly increases the capacity of L_2. As seen in the figure, D3N adjusts the cache capacities as expected. Figure 12(a) compares the overall job completion time of the workload under dynamic and static allocation((50/50 L_1/L_2) mechanisms. As shown in the figure, dynamic allocation completes 14% faster than static allocation. Both figures indicate that D3N quickly reacts to network congestion and adjusts cache sizes towards ideal settings.

Performance under different locality levels: In Figure 13 we see job completion time for dynamic and static (50/50 L_1/L_2) allocation for the Facebook trace benchmark with different locality levels, from 100% (files always accessed at the same L_1) to 0%(access locations are random). Dynamic allocation improves job completion time for high-locality cases, e.g. by 42% and 26% for localities of 100% and 80%, respectively.

VI. Related Work

A number of projects have explored caching (mostly in-memory) for big-data analytics [4], [6], [26], [35],

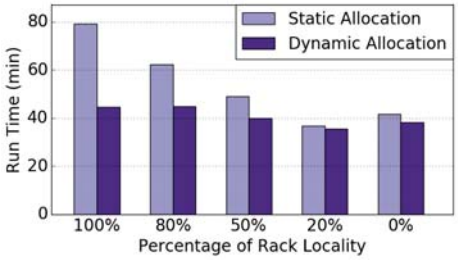


Fig. 13: Runtime comparison of static and dynamic cache allocation for the Facebook trace under different locality levels.

[36], [50]. The most related approach to D3N, Alluxio, aims to alleviate the cluster to data lake bottleneck by implementing a distributed cache layer that can interface with a variety of data lakes. A key difference in goals, that impacts many elements of the architecture, is that Alluxio implements a separate caching layer rather than modifying the data lake itself. This results, in Alluxio adopting a centralized metadata server for locating cached blocks while D3N uses consistent hashing. More fundamentally, Alluxio's current caching policy on an object is controlled by the client accessing the object. For example, in its default policy Alluxio caches data local to the client and uses other caches randomly only if a requested file is larger than the worker's entire cache; this is more similar to D3N's pure L1 schema rather than a distributed cache.

D3N borrows many ideas from the rich cooperating caching research that ranges from CDNs [7], [22], [32], [49], to network file systems [17], [23], [43], and multiprocessors architectures [12], [19], [20]. D3N differs from previous work in focusing on throughput and network contention rather than on latency [17], [43] and in introducing a dynamic cache management algorithm to find the optimal cache size allocation for dynamic caching demands of local/global accesses. Similar to [17], [32], and different from most other work, D3N manages cache space purely based on local decisions with explicit cooperation and avoid a central coordinator.

Cache replacement policies such as Pacman [6] increases cache efficiency by guaranteeing that all data needed to start a new job is read into its caches at the same time. Cliffhanger [15] adaptively sizes per-workload cache sizes to increase hit rates in a manner similar to the way D3N adapts per-layer cache sizes. D3N complements these other caching solutions by focusing on a different problem: network imbalances that limit bandwidth to cached data. As such, D3N's multi-layer caching approach could be use to improve the performance afforded by these existing caching systems.

Many in-network protocols address congestion or load imbalances within data centers by identifying congested paths and routing packets or flows on different ones [3], [21], [27]. In contrast, D3N addresses the case where all paths to a specific location have limited bandwidth due to over-subscription or organic growth.

D3N shares many architectural similarities with CDNs [2], [10], [25], [30]. For example, D3N anycast-based lookup service, which identifies the nearest cache service is similar to Akamai's DNS black-magic method for routing users to the closest CDN clusters. Unlike CDNs that are designed for WAN to minimize read only access latency, D3N supports both read and write-caching and it's multi-layer architecture aims to improve data lake throughput and mitigate network bottlenecks within datacenters.

Usage of shadow pages to estimate hit ratio statistics have been explored in other areas. For example Dynacache [14] and Cliffhanger [15] optimize memory usage in Web caches by estimating the hit ratio statistics and resizing the queues allocated for different sized objects in Memcached [24] and Redis [41]. Similarly, Memshare [16] offers efficient memory management for multi-tenant key-value stores by estimating application hit ratios to automatically sharing the pooled and idle memory resources among the tenants while providing performance isolation guarantees. D3N utilizes the shadow pages in a similar fashion with these works albeit on a different problem. In addition to utilizing shadow queues for collecting reuse distance histograms, D3N also observes network latencies to dynamically resize the dedicated space to different cache layers.

Some works focus on tiering data near clients rather than caching [11], [39], [40]. We considered tiering as opposed to caching for D3N, but found that it wasn't a good design choice because moving frequently-changing hot data between tiers would incur too much overhead.

VII. Conclusion

We present D3N, a transparent caching extension to the (Ceph-based) storage system itself, providing acceleration to all existing clients. As a cooperative cache it scales naturally with the number of clients, and its cache allocation policy adapts to network imbalances in real, imperfect data center networks. We show that by adding one additional caching server to each rack, adding around 3% increase in costs to the typical new racks in our datacenter, a cluster will automatically obtain SSD-like performance for hot data sets while having the economics of inexpensive disk based storage for cold data sets. With micro-benchmarks we show that we can saturate the SSDs and NICs of our servers and obtain a 5x improvement in throughput from two SSDs versus 90 disk spindles. With a macro-benchmark based on Facebook traces we show that D3N can reduce the bi-sectional bandwidth demands by a factor of 4 and improve throughput by a factor of 3; these results are highly conservative in that we assume no locality in access and re-use is much lower than recent (not yet public) industry traces. With simulation we show that D3N's adaptive algorithm can rapidly and automatically adjust to changes in workload

and network hotspots; resulting in substantial gains over existing static single layer caching technologies like Aluxio.

VIII. Acknowledgment

We would like to thank Mass Open Cloud team for their assistance while performing the experiments and the anonymous reviewers for their valuable suggestions.

Partial support for this work was provided by the National Science Foundation awards 1414119 and 1149232, Netapp Faculty Fellowship and Red Hat Collaboratory as well as the several industry partners of the Mass Open Cloud (MOC), which include Red Hat, Two Sigma Investments LP, Intel, Brocade, Cisco, and Lenovo.

References

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

[2] Akamai web site. http://www.akamai.com.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, and N. Huang. Hedera: Dynamic Flow Scheduling for Data Center Networks. *NSDI'15*, 2010.

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

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

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

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

[8] J. Arnold. *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. "O'Reilly Media, Inc.", Oct. 2014.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. *OSDI*, 2010.

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

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

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

[14] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, 2015. USENIX Association.

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

[16] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, 2017. USENIX Association.

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

[18] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan. Integrating Parallel File Systems with Object-based Storage Devices. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 27:1–27:10. ACM, 2007.

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

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

[21] T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. *CONGA: distributed congestion-aware load balancing for datacenters*, volume 44. ACM, Feb. 2015.

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

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

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

[25] A. Flavel, P. Mani, D. A. Maltz, N. Holt, J. Liu, Y. Chen, and O. Surmachev. FastRoute: A Scalable Load-aware Anycast Routing Architecture for Modern CDNs. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 381–394, Berkeley, CA, USA, 2015. USENIX Association.

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

[27] A. Ford, C. Raiciu, M. Handly, and O. Bonaventure. TCP extensions for multipath operation with multiple addresses. RFC 6824, IETF, Jan. 2013. https://tools.ietf.org/html/rfc6824.

[28] Apache hadoop. http://hadoop.apache.org/.

[29] A. Halevy, P. Norvig, and F. Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24(2):8–12, Mar. 2009.

[30] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. *An analysis of Facebook photo caching*. ACM, Nov. 2013.

[31] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, Berkeley, CA, USA, 2010. USENIX Association.

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

[33] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, 1994.

[34] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. 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*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

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

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

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

[38] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[39] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, 2014. USENIX Association.

[40] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure data lake store: A hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 51–63, New York, NY, USA, 2017. ACM.

[41] RedisLab. Redis. available at http://redis.io.

[42] S3a filesystem client.

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

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

[45] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, Mar. 2007.

[46] Storage Performance Development Kit | 01.org. http://www.spdk.io.

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

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

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

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