Cachalot: A Network-Aware, Cooperative Cache Network for Geo-Distributed, Data-Intensive Applications

Fan Jiang
Department of Computer Science
University of North Carolina at Chapel Hill
dcvan@cs.unc.edu

Claris Castillo, Stan Ahalt
Renaissance Computing Institute (RENCI)
University of North Carolina at Chapel Hill
claris@unc.edu, ahalt@renci.org

***Abstract*— Collaborative and data-intensive applications are hosted on geo-distributed infrastructures to exploit computing resources at scale. However, these applications typically incur massive data transfers over bandwidth-constrained wide-area networks (WANs) which impose significant performance overhead. Conventional distributed computing platforms (*e.g.,* Spark) leverage caching to avoid duplicate executions of common computations and thus reduce network traffic. However, these techniques were developed for data center environments and therefore lack advanced network-aware mechanisms to support high-performance, data-intensive applications over the WAN in geo-distributed environments. Hence, we develop *Cachalot* – a novel network-aware, cooperative cache network for caching datasets generated by common computations shared among geo-distributed, data-intensive applications. We perform a simulation-based deep evaluation using both synthetic and real traces. The experimental results indicate *Cachalot* speeds up data-intensive applications by over 50%, reduced network traffic by up to 60%; and, outperforms state-of-the-art baselines by over 20% in geo-distributed environments for various common user-driven performance metrics.**

I. Introduction

Scientific research is increasingly data-driven, collaborative and dependent on huge datasets that require geo-distributed computing and data sharing infrastructure. At the heart of these collaborations are scientists sharing data via public and private storage infrastructure including Cloud and on-premise resources. For data processing, scientist rely on a myriad of compute resources, from large compute facility, *e.g.,* high-performance computing (HPC) to cloud-hosted data analytic clusters [2] [8] and national cyberinfrastructure [14] [5] [4]. In order to use these resources for science research, scientists move large volumes of data over the wide-area network (WAN).

Performance overhead resulting from moving data over the WAN is considered the root cause of poor performance of geo-distributed, data-intensive applications [46] [45] [53]. Network-driven approaches have emerged as viable solutions to improve data transfer efficiency in these environments. In particularly, mechanisms based on software-defined networking (SDN) mitigate the impact of network bottlenecks by avoiding network congestion dynamically –via advanced routing techniques– or pro-actively –via optimal path assignment [35] [34] [37]. These approaches are effective in improving the utilization of network resources over the WAN. Nevertheless, given the reduced growth of network capacity [6] and the increasing growth of data volumes, data movement will continue to be a dominant factor in the performance of data-intensive applications at scale. Hence, approaches that only focus on the network may not be sufficient to overcome the aforementioned challenges.

Alternatively, *caching* strategies have the potential to alleviate the enormous bandwidth requirements associated with distributing data-intensive workloads at scale [18]. To achieve this, caching strategies trade storage capacity at the edge of the network with bandwidth at the network core. More specifically, they exploit storage to absorb traffic by maintaining replicas of data objects near the clients. For instance, *computation caching* has been recently introduced for data processing frameworks, *e.g.,* Spark [3], Nectar [32] and Tachyon [40] [1]. By avoiding redundant executions of common computations, *computation caching* reduces network traffic and enables more efficient use of compute resources within the data center. As a consequence, data-intensive applications perceive significant speedup and users observe important monetary cost savings. Nevertheless, these approaches were developed for data centers deployments and therefore lack the network awareness and distributed nature needed to perform in geo-distributed environments [26] [44] [27] [52].

Cooperative caching [29] is one caching technique that has proven effective in managing distributed cache networks in the Internet [23] [20], mobile networks [49] [41] and content distribution networks (CDNs) [51] [55]. To support WAN environments, these approaches optimize for network and disk I/O performance, but seldom consider the cost associated with generating data objects not found in the cache (cache misses). This is partly due to the fact that a typical data distribution environment is characterized by abundant storage and moderate compute resources. Therefore, as we demonstrate in this paper, state-of-the-art cooperative caching techniques fail to perform well in geo-distributed data-intensive applications that stress compute, storage and network resources.

In view of the aforementioned observations, we take a more comprehensive approach to enable cost-efficient and performant data-intensive applications. We develop *Cachalot*, a novel network-aware, cooperative cache network that amortizes the cost of data transfers by means of caching output datasets of redundantly executed computation jobs; and, makes cache placement and replication decisions based on the availability of network resources. *Cachalot* is utility-driven and builds on a combination of techniques that together performs intelligent trade-off decisions. *Cachalot*

retains in cache the data of highest overall value while simultaneously balancing the use of network and storage resources; and, improves completion time of data-intensive applications by taking into account the cost of executing jobs and generating data. To adapt to the ever-changing conditions of shared environments, *Cachalot* dynamically revises previous caching and replication decisions based on resource availability and data access history. Finally, *Cachalot* builds on dynamic logical artifacts that enable the efficient design and development of caching algorithms. We have performed a deep evaluation of *Cachalot* through simulation using real and synthetic workloads and demonstrated that it reduces network traffic by up to 60% and outperforms state-of-the-art cache algorithms by up to 20% for common performance metrics. Furthermore, *Cachalot* achieves over 50% saving of completion time for data-intensive jobs.

The rest of the paper is organized as follows. In Section II we formulate the problem formally. We describe the system design of *Cachalot* and its core network-aware cache algorithm in Section III. In Section IV we present the results of a deep evaluation on the performance of *Cachalot* against state-of-the-art baseline algorithms. We conclude the paper and refer to related works in Section V and VI, respectively.

II. Problem Formulation

We consider an environment where users, compute and data resources are geo-distributed over the WAN. Users submit jobs to compute resources. A job can be an instance of any kind of program specification, *e.g.,* MapReduce, MPI. Compute resources are hosted on premise or on public infrastructure such as Clouds and support big data processing frameworks, *e.g.,* Spark. Similarly, datasets are available via public data repositories, *e.g.,* NCBI [9], or on-premise databases. Users are located at institutions or sites with moderate storage resources for caching data products that may be reused in the future. We refer to these products datasets or data objects interchangeably in this paper. A data object is *uniquely* identified by the computation job that produces it. Similarly, a job is uniquely identified by information such as *executable binary*, *input dataset* and *execution parameters*. The *utility* associated with a data object is not only a function of its potential demand or usage but also the performance improvement that it provides to users by virtue of its placement, and the impact on the corresponding network performance, as well as its size and the cost associated with generating it. Intuitively, caching a data object that can be quickly recomputed in a cache node with poor network availability does not bring much utility to future requesters.

To process data, users transfer input and output datasets over the WAN connecting compute and storage infrastructure. Thus, the end-to-end completion time of a single job consists of the time needed for processing and transferring data. Both, the sharing of oversubscribed compute resources and lossy and high-latency network paths result in unpredictable long end-to-end execution times. By adopting an advanced caching strategy datasets can be reused and transferred from near-by cache resources instead of recomputing jobs thus effectively reducing completion time and cost on behalf of the users.

We represent the cache system as a network or a complete graph $G{=}(V, E)$, where V and E denote the set of cache nodes and network links between them, respectively. The capacity of each node $v \in V$ is denoted by C_v. The bandwidth and latency of each link $e \in E$ is denoted by b_e ($b_e > 0$) and l_e ($l_e \geq 0$), respectively. We use M to denote the set of data objects cached in G, in which size of each object $m \in M$ is denoted by s_m. We use $u_{m,v}$ to denote *utility* value of data object m on cache node v. The placement of data objects are represented by a binary matrix P: $P_{m,v}$ is 1 if data object m is placed in cache node v and 0 otherwise; $\sum_{v \in V} P_{m,v} = 0$ if a data object $m' \in M$ is no longer in cache. Since data objects can be transferred between cache nodes, we use binary matrix T to denote all possible data transfers: $T_{m,e}$ is 1 if data object m is transferred over link e and 0 otherwise. We aim at maximizing the total *utility* value of data objects in cache while minimizing cost for data transfers due to relocation and replication of data objects. The problem can be formulated as follows:

$$\max \quad \sum_{m \in M} \sum_{v \in V} P_{m,v} \cdot u_{m,v} - \sum_{m \in M} \sum_{e \in E} T_{m,e} \cdot \left(\frac{s_m}{b_e} + l_e \right)$$

$$\text{s.t.} \quad \sum_{m \in M} P_{m,v} \cdot s_m \leq C_v \qquad \forall v \in V \tag{1}$$

$$\sum_{v \in V} P_{m,v} \geq 1 \qquad \forall m \in M \tag{2}$$

$$P_{m,v} \in \{0, 1\} \qquad \forall m \in M, \forall v \in V \tag{3}$$

$$T_{m,e} \in \{0, 1\} \qquad \forall m \in M, \forall e \in E \tag{4}$$

Constraint (1) is the capacity constraint that ensures the total size of data objects cached in each node will not exceed the capacity of the node. Constraint (2) is the placement constraint that guarantees that every data object in cache has at least one copy available in certain cache nodes. Constraint (3) and (4) are binary constraints with P and T being binary matrices. We note that the binary constraints transform the problem into a 0-1 multiple knapsack problem, which is proven NP-complete [39]. We introduce an advance network-aware caching algorithm to address this problem.

III. System Design

Cachalot is a distributed, cooperative cache that maintains output data of executed jobs, so as to reduce redundancy of jobs execution. It adopts an adaptive network-aware cache algorithm, which adjusts to ever-changing network conditions and exploits distributed cache capacity to reduce job completion time and better utilize resources. More importantly, *Cachalot* is utility driven in that a *utility* value associated with data objects drives all cache activities including eviction, replication and placement. In *Cachalot* the utility captures the performance gain resulting from caching data objects based on their placement, usage and availability of network resources.

A. Architecture

Cachalot is a cache network. Its architecture consists of a *Cache Manager* (CM) and a number of interconnected *Cache Agents* (CAs). Each CA is equipped with storage to cache data and runs an instance of a network-aware cache algorithm to manage its local cache. Its primary role is to serve as a first-level cache for clients that are co-located with the CA. CAs work cooperatively on data placement, retrieval and replication to reduce completion time of jobs while relying on the CM to maintain a global view of the system and orchestrate caching and operational activities among them.

At a high-level *Cachalot* serves clients as follows. Upon submission of a new job, a client first contacts its local CA. If a replica of the output dataset associated with the job is hosted in the CA, it is directly sent back to the client. Otherwise, the local CA contacts the CM to find a remote CA hosting a replica. If no replica is found in the network of CAs, the client submits the job for execution to a compute resource, *e.g.,* Cloud, and cache the execution result into *Cachalot* after its execution. If the replica is found in a remote CA, the local CA fetches the remote replica and serves it to the client. Finally, the CA runs the cache algorithm to decide if to maintain a replica locally. It consults the CM about network resources and the availability of replicas in other CAs in order to act cooperatively. Following, we describe our approach in detail. We first introduce logical artifacts needed to support the efficiency of the caching algorithm followed by a description of the algorithm.

B. On datapods, dataserver and dataclients

In *Cachalot* clients retrieve replicas from the cache node that offers better network connectivity, i.e., bandwidth. Naturally, CAs can be logically organized into clusters. Each cluster serves and manages one replica and since replicas are created on demand, there may be multiple replicas for a given data object d at any point in time. We refer to a cluster as a *datapod*. There is a *datapod* for every replica. Each *datapod* consists of a *dataserver* and a group of *dataclients*. The *dataserver* hosts and serves the replica to its *dataclients*. The membership of a *datapod* consists of CAs that observe good network connectivity, *e.g.,* bandwidth availability, with the *dataserver* node – as compared with other *dataserver* nodes hosting a replica of d. The CM maintains a catalog of cached data object records, each of which is mapped to a set of *datapods*. The record of a *datapod* specifies the address of the *dataserver*, a list of its *dataclients* and their retrieval history. As explained later the CM also maintains network monitoring information to support decision-making processes and *datapods*' membership. Intuitively, a *datapod* represents an optimized cluster of consumers (*dataclients*) and a producer (*dataserver*) of a replica whose membership varies as a function of the *utility* that the replica brings to the cluster

The CM maintains monitoring information about the cache network. When a CA requests a data object on behalf of a client, the CM looks up the catalog for *datapods* with replicas of the requested data object; and, responds with the *datapod* that incurs the least data transfer time to the requesting CA together with the aggregate retrieval history of the replica. Following, the CA joins the *datapod* and fetches the replica.

When a *dataclient* joins a *datapod*, the *dataserver* updates its membership and estimates the available throughput with the new *dataclient*. The *dataserver* also maintains retrieval history of its replica. The retrieval history includes retrieval frequency and recency (time elapsed since last retrieval) of the replica for each *dataclient*. These metrics are used to run the network-aware cache algorithm, which will be introduced shortly. If a *datapod* is dismissed, i.e., replica evicted, the *dataserver* informs the CM to delete its record. Additionally, *dataclients* may migrate among *datapods* in response to changes in network conditions and replica distribution. We describe this process in detail later in this section.

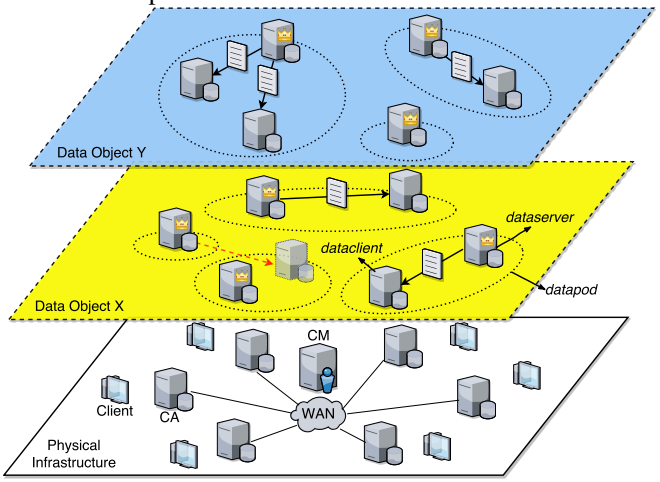

Fig. 1: *Cachalot* Architecture. The bottom layer is the physical infrastructure in which CAs are interconnected via WAN and co-located with clients. The CM coordinates cache operations among CAs. The upper layers are *datapod* topologies of different data objects. CAs with a crown are the *dataservers* and *dataclients* fetch replicas from the *dataserver* of *datapods*. Note that a *dataclient* of one *datapod* may migrate to another *datapod* (shown in the middle layer) due to changes in replica distributions and network conditions.

C. Network-Aware Cache Algorithm

Following we describe the caching algorithms behind *Cachalot* and depicted in Fig. 2 in the form of pseudo code. *Cachalot* seeks to retain in cache data objects that provide high utility to the overall system.

Recall that there is a replica per each *datapod* in the system. Therefore, the creation and dismissal of a *datapod* is equivalent to the insertion and eviction of a replica, respectively. Additionally, as explained earlier in this Section, the structure of a *datapod* is driven by the network performance observed by its *dataclients*. We leverage these features to introduce an utility function that captures the value of a data object as a function of the cost associated with generating it, i.e., computational cost in units of time, its network accessibility, i.e., achievable throughput between clients and *dataclient* and its usage, i.e., recency. This utility drives all caching and replication decisions in *Cachalot*. Following we formally introduce the utility function.

$$U(d,c) = \sum_{m \in M} \frac{f_m^d \cdot (min\left\{g^d, t^d_{\kappa_m,m}\right\} - t^d_{c,m})}{r_m^d \cdot s^d}$$

The *utility function* is built upon a conventional

cost/benefit model [21] and enhanced to consider network information. In this function, d, c and M correspond to the data object or replica, the *dataserver* of the *datapod* and the *dataclients* of the *datapod*, respectively. We use s, f and r to denote size, retrieval frequency and recency associated with the data object d. The numerator of the utility function is the product of two terms. The first term corresponds to the number of times that the replica has been served by *dataserver* c. The second term is compatible with the time cost associated with requesting the data object. More specifically, lets denote g^d the completion time of the job that produces d and would be incurred by the user if d was not cached. Alternatively, lets assume that κ_m is the *dataserver* of the nearest *datapod* to m hosting d. A user on site m would incur $t^d_{\kappa,m}$ time units to transfer d into the *dataclient* m. It then takes an additional $t^d_{c,m}$ time units to transfer d from *dataserver* c to the client. The denominator is the product of r^d_m and s^d corresponding to the time elapsed since d was last accessed and the size of d.

This utility function is at the heart of all caching activities including insertion and eviction of replicas. Upon receipt of a replica, a CA decides whether to cache the replica locally (cache insertion) and create a new *datapod*. To accommodate for a new replica, existing *datapods* (replicas) must be evicted (dismissed). To make this decision, the cache algorithm calculates the *utility* of a hypothetical new *datapod* (Line 21). If creating the hypothetical *datapod* yields a larger *utility* value as compared to dismissing the *datapods* with the least-*utility*, the hypothetical *datapod* is created (Line 20–24). Hence, the insertion of a new replica my result in the eviction of multiple replicas. The number of evicted replicas is determined by the total storage capacity needed to accommodate for the newly inserted replica.

Note that every time a remote replica is retrieved the insert function is invoked (Line 18). Notably, the algorithm is observant of the trade-off between re-executing a job and reusing a cached replica (Line 14-16). The CA raises a *dummy cache miss* forcing the submission of a job rather than enabling a *cache hit* that would further overload the system. This optimization step avoids the unnecessary caching of jobs that generate a large amount of data in a short time, thus enabling efficient cache and network utilization.

The membership of *datapods* is updated dynamically in response to two events. First, upon fetching a replica, a *dataclient* is directed to a *datapod* different to the one it belongs to. Second, a *datapod* is dismissed, i.e., a replica is evicted forcing all of its *dataclients* to *migrate* to an existing *datapod*. In the former case, member m migrates from its existing *datapod* c' to a new *datapod* c if and only if such migration results in better transfer time. In this case, the time saving yielded is $(t^d_{c',m} - t^d_{c,m})$.

Algorithmic Efficiency. *Cachalot* has been designed and developed to operate efficiently in geo-distributed environments. The storage resource of each CA (*self.pods*) is implemented using a doubly-linked list proposed in [50], in which data objects are sorted in ascending order of *utility*. In addition, we also use Fenwick trees [30] to maintain cumulative *utility* and space used for each data object in the list. This tree-based data structure allows the algorithm to efficiently identify data objects for eviction. The runtime complexity of the algorithm is linear to the number of CAs in *Cachalot* which scales well across up to hundreds of geo-distributed sites. Section III-E gives a detailed complexity analysis of the algorithm.

```
   /* Note: self is the CA that invokes the cache
      algorithm. Some implementation details are
      omitted due to space constraint.                */
   Func insert(key, obj):
1      pod ← cm.getPod(key)
2      if pod ≠ ∅ and ¬filter(obj) then return
       // Evict least-utility replicas in cache
3      self.evictForSpace(obj.size)
4      pod ← cm.createPod(obj)
5      pod.replica ← obj
6      pod.util ← calcUtil(pod)
7      self.pods ← self.pods + {pod}
   Func retrieve(key):
8      if key ∈ self.pods then
9          pod ← self.pods.getPod(key)
10         pod.util ← calcUtil(pod)
11         return pod.replica
       end
12     pod ← cm.getPod(key)
13     if pod = ∅ then return ∅
14     t ← pod.server.estimateDelay(key)
15     g ← cm.getExecTime(key)
16     if t > g then return ∅
17     obj ← pod.server.retrieve(key)
18     insert(key, obj)
19     return obj
   Func filter(obj):
20     pod ← cm.createHypoPod(obj)
21     gain ← calcUtil(pod)
22     if gain ≤ 0 then return false
23     loss ←estimateEvictionLoss(obj.size)
24     return gain > loss
   Func calcUtil(pod):
25     util ← 0, obj ← pod.replica
26     for c ∈ pod.clients do
27         g ← cm.getExecTime(obj.key)
28         sp ← cm.getSecondaryPod(obj.key)
29         t1 ← pod.server.estimateDelay(obj.key)
30         t2 ← sp.server.estimateDelay(obj.key)
31         benefit ← c.freq * (min(g, t2) − t1)
32         cost ← c.timeElapsedSinceLast * obj.size
33         util ← util + benefit/cost
       end
34     return util_m
```

Fig. 2: *Cachalot* network-aware cache algorithm

D. Computation Sharing

We consider big data applications with potentially long execution times. Therefore, concurrent request of long jobs are the norm in the scenarios we considered in our work. Concurrent requests of long jobs may result in a burst of cache misses followed by duplicated data transfers since replicas of the data are not yet available in the cache network To handle this condition, we introduce a simple yet powerful optimization mechanism. For every running job, the CM keeps track of its running time and maintains a callback waiting list. If a new request for a job currently in execution is submitted, the requester can register a *callback* for the output data on the CM and be notified once the data is available in the cache. As we demonstrate later this optimization approach has significant impact in the performance of *Cachalot*.

E. Complexity Analysis

To analyze complexity of the cache algorithm introduced in *Cachalot*, we assume that there are n CAs and each of which can cache at most m data objects. According to [50], it takes $O(log_2 m)$ time to insert/retrieve a data object to/from the cache storage and $O(1)$ time to evict least-*utility* data objects.

1) ***Insertion****:* Every insertion operation requires a lookup for CAs with sufficient space by the CM, which takes $O(n)$ time. If such CAs exist, the CM then takes $O(n)$ time to select one that costs the least time to send the data object. Upon receipt of the data object, the selected CA calculates its *utility* and places it into cache storage with a total cost of $O(log_2 m)$ time. If the cache is full, the CA has to evict in-cache objects in order to accommodate for the new one. It costs the CA $O(log_2 m)$ to perform evictions of least-*utility* data objects, and $O(log_2 m)$ to update Fenwick trees for *utility* and space usage accordingly. In summary, the runtime complexity of an insertion operation is $O(n)$ on the CM and $O(log_2 m)$ on the CA.

2) ***Retrieval****:* A retrieval operation firstly incurs a hash table lookup on the CM to locate *datapods* with the demanded data object, which takes $\Theta(1)$ time. If multiple *datapods* own a replica of the object, it costs the CM $O(n)$ time to assign the requester to the *datapod* that requires the least time to fetch the replica. Then the requester will fetch the replica from the *dataserver* of the selected *datapod*, subsequently triggering the `calcUtil` function on the *dataserver* (Line 10). The *dataserver* takes $O(n)$ time to traverse all its *dataclients* in order to re-calculate the *utility* of its replica, and $O(log_2 m)$ time to find a new position to place the replica with new *utility*. In addition, it also takes the *dataserver* $O(log_2 m)$ time to update the Fenwick trees. Hence, the runtime complexity of a retrieval operation is $O(n)$ on the CM and $O(max(n, log_2 m))$ on the *dataserver*.

3) ***Replication****:* The replication operation is analogous to the insertion operation, except that it costs additional runtime for the specific CA to make replication decisions. On receipt of a data object, a CA requests the CM to create a hypothetical *datapod*, which takes $O(n)$ time (Line 20). Then it takes $O(n)$ time to calculate the hypothetical *utility* for the received object and $O(log_2 m)$ to estimate potential loss incurred by replicating the object. If a replication is permitted, it takes $O(max(n, log_2 m))$ time to insert the replica into the cache storage. To sum up, the runtime complexity of a replication operation is $O(n)$ on the CM and $O(max(n, log_2 m))$ on the CA.

IV. Performance Evaluation

We evaluate the performance of *Cachalot* through simulation using both synthetic and real-world datasets. We compare *Cachalot* against common decentralized cache algorithms including LFU, GreedyDual-Size and Nectar. We consider two user-driven performance metrics: *cache hit rate* and *average completion time saving*. These metrics correspond to the percentage of data accesses satisfied by the cache; and, the average time saved as a result of a cache hit as compared to executing the job that produces the desired data object, respectively. We have developed an event-based simulator using SimPy [13], which is introduced in [36] in detail. The simulations are run on a bare-metal machine on NSF Chameleon Cloud [4] with 48 Intel Xeon 2.3GHz CPUs, 128GB RAM, 128GB disk.

A. Simulator

We have developed a event-based simulation framework built in SimPy [13] to analyze the performance of *Cachalot*. The simulator is fully parameterizable with parameters configured via a JSON configuration file. Configurable parameters include workload characteristics, bandwidth allocation and data locality. Following we describe the key components of the simulator. We defer a more detailed description of the simulator software and architecture to a future publication.

1) ***Computation job****:* A computation job simulates a computational job submitted by a client and executed by the compute cluster. A job is represented by a (data item, operation) tuple. A data item is uniquely identified by an integer key and associated with a size; an operation consists of a unique integer string identifier and a set of numerical functions which are configurable and used for calculating simulation parameters such as execution time, output data size and unique identifiers.

2) ***Client****:* Currently, the topology of the cache network is fixed and it is assumed that each client is co-located with a CA. We further assume that there is one client per CA; we simulate the job submission rate so to capture the multiplicity of clients per CA (site). The job submission process follows a Poisson distribution; this is consistent with [24]. In addition, popularity and size distributions of the jobs are also configurable (defaults to normal distribution). jobs are evenly distributed across clients. The simulator supports the ingestion of synthetic data sets to drive simulation experiments. We use this feature to reproduce representative production-level environments.

3) ***Cluster****:* We assume that there is only a single centralized cluster that executes submitted jobs and its resource capacity is set infinite. This assumption is representative of common production environments wherein a *Cachalot* deployment serves a user community that relies on a single compute infrastructure for data processing. We notice that this assumption is immaterial to the performance of caching algorithms such as *Cachalot*.

4) ***Cache Network****:* The *cache network* consists of clients, the cluster, the CM and CAs interconnected using full-duplex network links, each of which is assigned bandwidth and latency values. The network links simulate data transfers. The bandwidth and latency values follow a configurable distribution that defaults to log-normal [28]. Network bandwidth is dedicated, thus enabling strict performance isolation between data transfers across links. The CM periodically measures bandwidth and latency between CAs every one hour. The network monitoring can be performed in a proactive manner using tools such as perfSONAR [11] in production environments. The CM also maintains job execution times collected from *job specifications* submitted by the clients when they attempt to retrieve data from *Cachalot*. The job execution

times can be accurately estimated in a long run as jobs are executed with varying load in the cluster.

B. Experiment Setup

*1) **System configuration and environment**:* In the synthetic dataset, we simulate a cache network with 100 CAs. The total cache capacity available in the network is 20% of the total size of unique data objects in the workload. The bandwidth of network links follows log-normal distributions [28] with mean $\mu \in \{1, 2, 3, 4, 5\}$ and standard deviation σ=1, ranging between 50 and 600Mbps.

The real-world dataset consists of network bandwidth traces obtained from ExoGENI [17], a production network testbed funded by the National Science Foundation (NSF) with more than 14 Cloud sites distributed worldwide and connected via more than 10 network providers. To simulate a representative WAN environment, we collected the bandwidth statistics using `iperf3` [7] during the week of July 22−29, 2017.

In both datasets, we assume network latency and packet loss are negligible and computing resources are infinite.

*2) **Workloads**:* To drive our evaluation we use a synthetic and a real-workload dataset.

Our *synthesized workloads* consist of 10^6 computational jobs with 10^5 unique jobs. Both job popularity and input data size distribution follow a Zipf distribution with $\alpha \in \{0.1, 0.3, 0.5, 0.8, 1.2\}$ which is consistent with characteristics of data-intensive applications in production environments [19] [31] [47] [25]. Input dataset sizes range between 1.25MB and 125GB which is large enough to cover a broad range of requirements. We assume that the execution time and output data size of jobs are proportional to their input data size which is representative of the vast majority of use cases driving our work.

To gain insight into the performance of *Cachalot* against real workloads we use the OpenCloud [10] dataset. This dataset consists of a 31-month log of Hadoop jobs running on a research cluster in the Carnegie Mellon University and contains 19,198 unique jobs. We have created a workload with 200,000 jobs following the job distribution in this dataset.

*3) **Baseline algorithms**:* We compare the network-aware algorithm in *Cachalot* against three cache algorithms adopted in state-of-the-art distributed computing platforms as below:

- **Least-Frequent-Used (LFU)** favors frequently accessed data objects in cache
- **GreedyDual-Size (GD)** [21] extends LFU by favoring small-sized data objects that are frequently accessed
- **Nectar** [32] adopts a *cost/benefit* model and favors data objects with high *benefit* but low *cost*. The *benefit* is defined as a product of access frequency and completion time savings, while the *cost* is a product of data size and time elapsed since the data object was last accessed (recency). Notice that Nectar does not take into account network factors

However, since both LFU and GD are frequency-based algorithms and GD outperforms LFU, we only show results of GD. We also consider two common replication strategies:

- **Single-copy**: there can be only a *single copy* for each data object existing in the system at any point of time.
- **Replication-based**: data objects can be replicated across CAs with or without constraints.

C. Synthetic Dataset

We start our evaluation with the synthetic dataset.

Network-awareness and performance breakdown. We investigate the impact that each mechanism introduced in *Cachalot* has on the two performance metrics. We also consider the performance of *Cachalot* under three different replication-based strategies: relaxed, filtered and random. In the *relaxed replication* strategy a CA replicates every data object it retrieves. This strategy is common in production environments. *Filtered replication* relies on the *filter function* introduced in Section III to make replication decisions. In the *random replication* a CA makes replication decisions following a uniform random function. Fig. 3 shows the results of this experiment.

It is observed that when *Cachalot* operates without taking into consideration network monitoring information (static network in Figure) the completion time saving obtained is less than 20% which is significantly lower as compared to all the other cases. Notably, incorporating network information for the scenario with the single-copy replication strategy results in an improvement of almost 100% and 50% in completion time saving and hit rate, respectively. Intuitively, without network information *Cachalot* is unable to adjust its replication and caching strategies to network congestion conditions resulting from transferring data across CAs. Our experimental data shows that in the absence of network information *Cachalot* inadvertently exacerbates network congestion by sending data over congested network links. Thus negatively impacting the completion time of jobs.

We then investigate the performance of *Cachalot* under the filtered, relaxed and random replication-based strategies. These strategies present interesting trade-offs between data availability, *i.e.,* larger number of replicas and cache capacity efficiency. It can be observed that both the relaxed and random replication strategies exhibit comparable performance for both metrics with hit rate under the random strategy being slightly better (25%) as compared to the relaxed strategy. This can be explained by the reduction in effective cache capacity resulting from aggressive replication. We then focus on the filtered replication strategy. The filtered replication yields additional 6% time saving in average as compared to the single-copy strategy. Recall from Section III that the *filter function* in *Cachalot* only replicates data when there is a significant gain in *utility*, hence it only trades off an acceptable amount of cache capacity for improved data locality and network performance. We conclude that the *filter function* introduced in *Cachalot* offers the best performance overall as compared to the other strategies.

Finally, we evaluate the impact that the optimization mechanism *computation sharing* has over both user-driven performance metrics. Recall that this optimization effectively delays concurrent requests for data objects soon to be available in the cache network. This technique yields roughly

65% and 18% improvement on job completion time saving and hit rate, respectively. Note that without this optimization, *Cachalot* would *blindly* replicate and transfer replicas of the same data object, thus hindering the efficient use of network resources in the system.

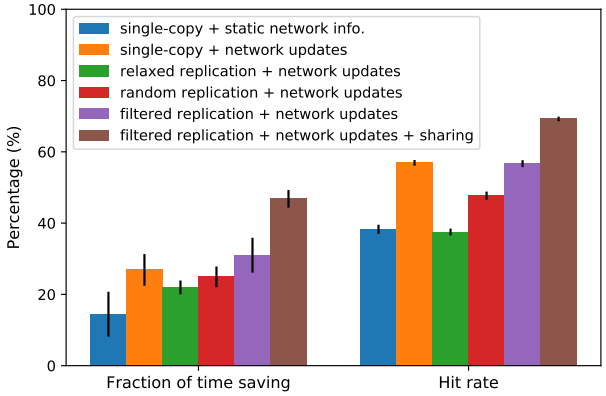

Fig. 3: Performance breakdown of *Cachalot* cache algorithm

Impact of network bandwidth. In the following experiment we investigate the effect that network bandwidth has over *Cachalot* as compared to the baseline techniques. Fig. 4 depicts the results of this investigation. Note that [s] and [r] refer to single-copy and relaxed replication strategies, respectively.

Particularly, the time savings achieved under the single-copy replication strategy increases as available bandwidth increases. This is because single-copy algorithms exhibit lower data locality and take advantage of increased network bandwidth to transfer remote replicas. In contrast, replication-based algorithms achieve higher local cache hits. Nevertheless, they experience worse time savings due to the reduced effective cache capacity.

As observed, all the other strategies are more or less insensitive to variations in network bandwidth except under extremely constrained conditions (50 Mbps). *Cachalot* achieves up to 50% completion time saving as compared to up to 30% and 25% for Nectar[s] and GD[s], respectively, thus demonstrating its ability to efficiently leverage network information.

Similarly, the hit rate for *Cachalot* increases with increasing available bandwidth. This is mainly because *Cachalot* adapts its replication strategy to bandwidth variations while the other strategies are network-agnostic. More importantly, *Cachalot* achieves up to 75% hit rate followed by GD[s], GD[r], Nectar[s] and Nectar[r] with 62%, 51%, 50% and 40%, respectively. In-depth analysis of the data shows that *Cachalot* replicates more aggressively under constrained conditions and more conservatively otherwise. When bandwidth is constrained, it trades cache capacity for data locality by keeping popular data objects locally and avoiding data transfers.

Moreover, *Cachalot* saves up to 60% network traffic which is in contrast with 35% and 39% by GD and Nectar, respectively. This experiment is included in Fig 4. In addition, it is also observed that the network traffic savings are indirectly proportional to the available bandwidth. This is because *Cachalot* is network-aware and keeps replicas of large data objects local to clients, therefore reducing traffic into the network.

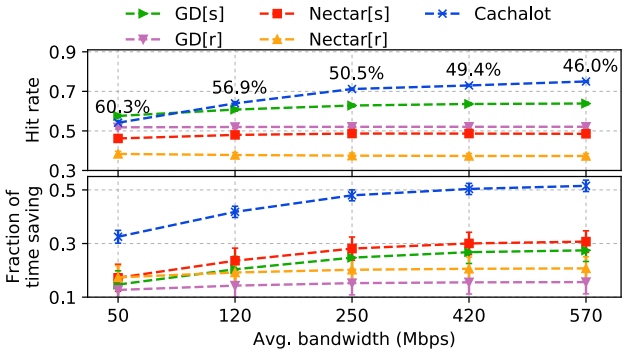

Fig. 4: Performance with varying bandwidth. Percentages annotated in the upper figure are the maximum network traffic saving achieved by *Cachalot*, as compared to 34.9% and 39.1% by GD and Nectar, respectively

Stability of *datapods* and replicas. To confirm our previous observation and assess the stability of *datapods* (replicas) in Fig. 5(a) and Fig. 5(b) we plot the cumulative distributed function (CDF) of the lifetime of *datapods* and their cardinality, for various values of network bandwidth. As observed earlier, under constrained bandwidth conditions *Cachalot* replicates more aggressively resulting in larger number of *datapods* (See Fig. 5(a)) with shorter lifetime (See Fig. 5(b)).

Fig 5 also includes graphs for different replication-based algorithms. Single-copy algorithms maintain one long lived replica (*datapod* in *Cachalot*) for each data object as compared to replication-based algorithm. Furthermore, this strategy can lead to the creation of network *hot spots* which have detrimental impact in the overall performance of geo-distributed, data-intensive environments. We do not include graphs for single-copy algorithms due to the lack of space.

As shown, GD[r] creates the largest number of replicas; 40% data objects have more than 5 replicas on average. However, most replicas have a very short lifetime. Likewise, although Nectar[r] creates much fewer replicas than GD[r], their average lifetime is also short. In comparison, *Cachalot* effectively trades replica availability by intelligently controlling the creation of *datapods* based on the overall utility that they bring into the system. This observation in particularly important for reducing the management overhead in production environments.

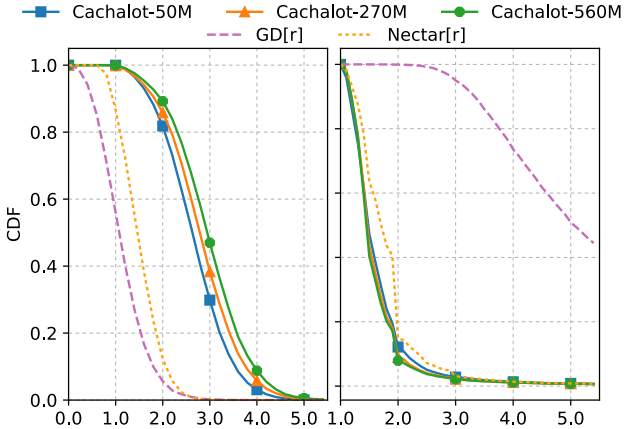

Fig. 5: Distribution of *datapod* number and lifetime

In the next experiment we evaluate the impact that the data size distribution has on the performance of *Cachalot*. Fig. 6 depicts hit rate and average time saving as a function of α for GD[s], GD[r], Nectar[s], Nectar[r] and *Cachalot*. The larger α the larger the fraction of small data objects. The graph shows that the hit rate is less sensitive to the data size distribution under *Cachalot* as compared to the other techniques. This follows intuition since the data size factor is outweighed by the network factor in the utility function of *Cachalot* (See Section III-C).

We also observe that the job completion time saving decreases when α=1.2 for all algorithms including *Cachalot*. This is due to the fact that the majority of data objects are small and hence their corresponding jobs have short execution times. Hence, these jobs observe marginal gain from caching since the time for fetching data from remote CAs may be comparable to the time it takes to generate the data. For instance, GD and Nectar cache aggressively when data objects are small (reflected on the increasing hit rate in the Figure), but fail to produce significant time saving. To make things worse, they generate more network traffic and fail to efficiently use network bandwidth. In contrast, *Cachalot* outperforms all baseline algorithms achieving up to 50% completion time saving for all values of α due to its unique ability to take into account network monitoring information and demand.

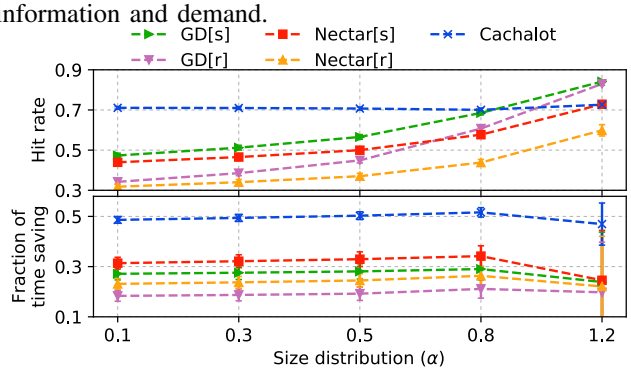

Fig. 6: Performance with varying size distributions

Naturally, as some data objects become very popular it becomes more difficult to balance the use of cache and network resources in the system. To investigate this observation we evaluate the performance of the replication algorithms as a function of data popularity. More specifically, we vary α to skew the popularity of data objects. Fig. 7 depicts the results of this evaluation. As expected, since caching top-ranked data objects can fulfill most data requests, hit rate increases as increasing function of α. Particularly, algorithms with replication-based strategy outperform algorithms with single-copy replication strategy for large values of α (α=1.2). This follows intuition since aggressive replication effectively improves data locality of top-ranked data objects and thus reduces the need for data transfers. This reasoning is confirmed in the plot depicting the *fraction of local hits* in Fig. 7 which shows that both GD[r] and Nectar[r] have a higher fraction of local cache hits as compared to *Cachalot*, Nectar[s] and GD[s]. This result is important also because it demonstrates that *Cachalot* outperforms the baseline algorithms in hit rate and average job completion time by virtue of its network awareness as demonstrated by the modest fraction of local cache hits (less than 20%). We conclude that *Cachalot* is able to more effectively utilize cache and network resources while observing user-driven metrics by deciding when to replicate (via the utility function) and which replica to select in a network-aware fashion.

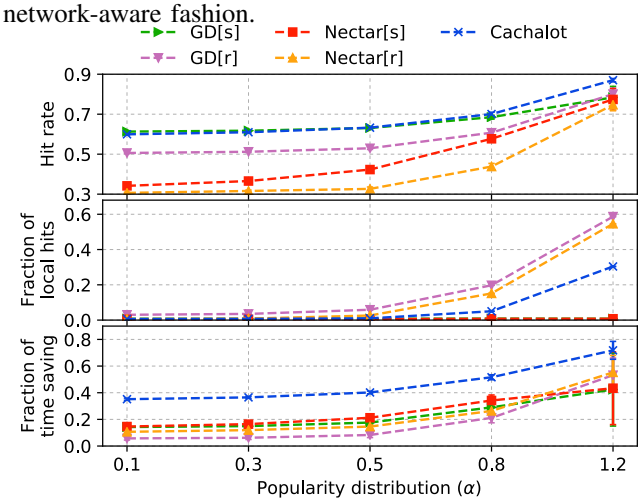

Fig. 7: Performance with varying popularity distribution

D. Real-World Dataset

Finally, we simulate a real-world environment by merging the workload characteristics of the OpenCloud dataset – to simulate workload – with the network bandwidth collection from ExoGENI – to simulate a realistic network as explained in Section IV-B. Fig. 8 depicts four dimensions of the dataset: output data size, popularity distribution of data, network bandwidth and job execution time. We observe that the majority of jobs in the workload have relatively short execution times and generate small data output. The size and popularity of data objects follow Zipf-like distributions. In contrast with the assumption made in our earlier experiments, the job execution time is independent of the output data size. The network consists of 14 CAs interconnected with network links with an average bandwidth of 215Mbps.

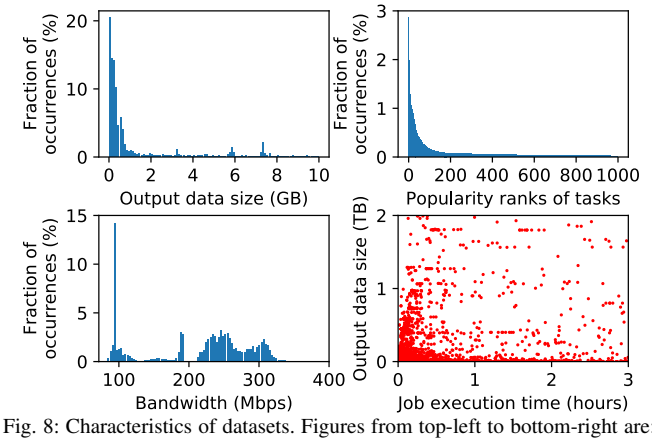

Fig. 8: Characteristics of datasets. Figures from top-left to bottom-right are: 1) Distribution of output data sizes; 2) Distribution of job popularity; 3) Bandwidth distribution in the WAN; 4) Relationship between job execution time and their output data size

Impact of Cache Capacity on Performance. In a production federated deployment there is no control over the

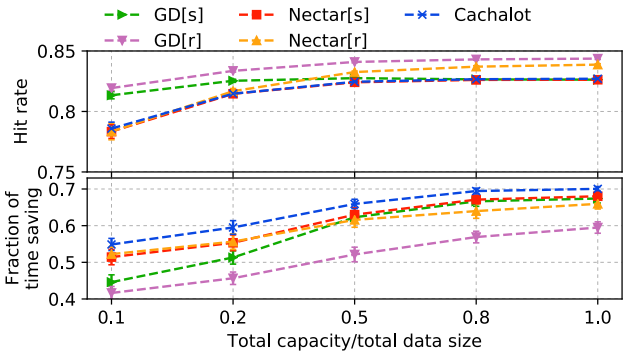

Fig. 9: Performance with OpenCloud and ExoGENI datasets

amount of cache resources that each site (CA) contributes with to *Cachalot*. In this experiment we study the impact that total cache capacity has on performance for all the algorithms. Our results are plotted in Fig. 9. We make three observations. First, all algorithms perform well in terms of hit rate for a sufficiently provisioned cache (hit rate varies between 80% and 85%). This is a due to the fact that most popular data objects are small in size hence the algorithms take advantage of statistical multiplexing of cache resources even when the cache capacity is limited. In particularly, replication-based strategy can efficiently utilize storage cache capacity by packing replicas of small data objects. Second, *Cachalot* exhibits the lowest hit rate. A deeper look into the results shows that *Cachalot* tends to evict jobs that although small in duration generate large output (See Fig. 8(4)). Notice that these jobs can't take advantage of caching due to their prohibitive transfer cost which in turn exacerbates congestion in the network. Third, as a consequence of our previous observation, *Cachalot* achieves the largest completion job saving as compared to the other algorithms. GD performs the worst in terms of completion time saving since it favors caching small data objects regardless of their overall completion time.

V. Conclusion and Future Works

We have presented *Cachalot*, a WAN-based, cooperative cache network for caching results of repetitive computation jobs in geo-distributed environments. We have also introduced an efficient distributed, network-aware cache algorithm, which adapts its caching strategy to varying network conditions and effectively reduces completion time of computation jobs. We have investigated a novel caching problem, in which cache misses will incur extra cost for regenerating data objects, in addition to disk and network I/O costs that are considered in traditional caching problems. Our simulation-based evaluation demonstrates that *Cachalot* is able to save over 50% completion time for computation jobs, and our network-aware cache algorithm outperforms our baselines by up to 20% in completion time saving.

In the immediate future, we will implement *Cachalot* as part of SciDAS [12], a national cyberinfrastructure to enable large-scale scientific data analysis at scale. To improve cache efficiency in the presence of very large data objects we will extend *Cachalot* to support chunk-based data management.

VI. Related Work

Recent studies [15] [43] [33] have shown that duplicate executions of computations cause severe wastage of computing and network resources and degrade performance of data-intensive applications. An array of works [22] [42] [32] [54] [40] propose to cache results of frequently executed computations for reusing, in order to eliminate duplicate executions and conserve resources. Spark [54] adopts *lineage* to allow efficient, explicit caching of computations within applications. Nectar [32] and Tachyon [40] break the limits to support cross-application computation caching within data centers. However, it is found in [26] [27] [52] that these works inherently lack network awareness and are inefficient in tapping network resources for accelerating data transfers in data-intensive applications. This problem exacerbates in geo-distributed environments since application performance is deeply influenced by data transfers over WANs as identified in [46] [45] [53]. Our work also recognizes the importance of network factors and proposed a network-aware system and algorithm design to accelerate data-intensive computations in geo-distributed environments.

Additionally, a number of distributed cache algorithms have been developed to *cooperatively* improve cache performance [19] [31] [16] [49] [41] [51] [55]. PeerOLAP [38] builds a peer-to-peer (P2P) cache system equipped with a network-aware cache algorithm, but it relies on a static, coarse-grained metric to capture network factors with low accuracy. In [48] the authors assume uniform data transfer rate in the proposed algorithm, which is unrealistic in real WAN deployments. There are also optimization approaches that address the problem as a distributed data placement problem, which is proven NP-complete in [39], and a number of heuristic algorithms are proposed to approximate the optimal placement [51] [18]. However, these algorithms assume a finite set of data objects in the workloads and the system has prior knowledge about them, which are not assumed in our work.

Acknowledgment

This work is supported by the NSF RADII (ACI #1440715) and NSF SciDAS project (ACI #1659300). We also thank ExoGENI for providing WAN resources for collecting network traces.

References

[1] Alluxio (formerly Tachyon). http://www.alluxio.org/.
[2] Amazon Web Service. http://aws.amazon.com.
[3] Apache Spark. https://spark.apache.org/.
[4] Chameleon Cloud. https://www.chameleoncloud.org/.
[5] Extreme Science Engineering Discovery Environment (XSEDE). https://www.xsede.org/home.
[6] Global Internet Geography. http://www. telegeography.com/research-services/global-internet-geography/.
[7] iperf3. http://software.es.net/iperf/.
[8] Microsoft Azure. https://azure.microsoft.com.
[9] National Center for Biotechnology Information (NCBI). https://www.ncbi.nlm.nih.gov/.
[10] OpenCloud Hadoop cluster trace. http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html.
[11] perfSONAR. https://www.perfsonar.net/.
[12] SciDAS. http://scidas.org.
[13] SimPy 3.0.10. https://simpy.readthedocs.io/en/latest/.

[14] Texas Advanced Computing Center (TACC). https://www.tacc.utexas.edu/.

[15] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling shared scans of large data files. *Proceedings of the VLDB Endowment*, 1(1):958–969, 2008.

[16] David Applegate, Aaron Archer, Vijay Gopalakrishnan, Seungjoon Lee, and Kadangode K. Ramakrishnan. Optimal content placement for a large-scale VoD system. In *Proceedings of the 6th International COnference*, page 4. ACM, 2010.

[17] Ilia Baldine, Yufeng Xin, Anirban Mandal, Paul Ruth, Chris Heerman, and Jeff Chase. ExoGeni: A Multi-Domain Infrastructure-as-a-Service Testbed. *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 97–113, 2012.

[18] S. Borst, V. Gupta, and A. Walid. Distributed Caching Algorithms for Content Distribution Networks. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, March 2010.

[19] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings*, volume 1, pages 126–134 vol.1, March 1999.

[20] Jiannong Cao, Yang Zhang, Guohong Cao, and Li Xie. Data consistency for cooperative caching in mobile environments. *Computer*, 40(4), 2007.

[21] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[22] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. FlumeJava: Easy, Efficient Data-Parallel Pipelines. 2010.

[23] Narottam Chand, Ramesh C. Joshi, and Manoj Misra. Cooperative caching strategy in mobile ad hoc networks based on clusters. *Wireless Personal Communications*, 43(1):41–63, 2007.

[24] Balakrishnan Chandrasekaran. Survey of network traffic models. *Waschington University in St. Louis CSE*, 567, 2009.

[25] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[26] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 98–109. ACM, 2011.

[27] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 443–454. ACM, 2014.

[28] Crow. *Lognormal Distributions: Theory and Applications*. CRC Press, New York, 1 edition edition, December 1987.

[29] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David 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.

[30] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

[31] Christine Fricker, Philippe Robert, James Roberts, and Nada Sbihi. Impact of traffic mix on caching performance in a content-centric network. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, pages 310–315. IEEE, 2012.

[32] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association.

[33] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 63–74, New York, NY, USA, 2010. ACM.

[34] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, and Mohan Nanduri. Achieving High Utilization with Software-Driven WAN. *Microsoft Research*, August 2013.

[35] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, and others. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[36] Fan Jiang, Claris Castillo, and Stan Ahalt. Cachalot: A Cooperative Cache Network for Geo-Distributed, Data-Intensive Computations (Technical Report). http://renci.org/technical-reports/tr-17-01, 2017.

[37] Fan Jiang, Claris Castillo, and Charles Schmitt. RADII: Bridging the divide between data and infrastructure management to support data-driven collaborations. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 370–377. IEEE, 2016.

[38] Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, and Kian-Lee Tan. An Adaptive Peer-to-peer Network for Distributed Caching of OLAP Results. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 25–36, New York, NY, USA, 2002. ACM.

[39] Jussi Kangasharju, James Roberts, and Keith W. Ross. Object Replication Strategies in Content Distribution Networks. *Comput. Commun.*, 25(4):376–383, March 2002.

[40] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[41] Zhongxing Ming, Mingwei Xu, and Dan Wang. Age-based cooperative caching in information-centric networking. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–8. IEEE, 2014.

[42] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[43] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. DryadInc: Reusing Work in Large-scale Computations. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[44] Pawan Prakash, Advait Dixit, Y. Charlie Hu, and Ramana Kompella. The TCP outcast problem: Exposing unfairness in data center networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 30–30. USENIX Association, 2012.

[45] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low Latency Geo-distributed Data Analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 421–434, New York, NY, USA, 2015. ACM.

[46] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 275–288, Berkeley, CA, USA, 2014. USENIX Association.

[47] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[48] Luigi Rizzo and Lorenzo Vicisano. Replacement Policies for a Proxy Cache. *IEEE/ACM Trans. Netw.*, 8(2):158–170, April 2000.

[49] Muhammad Zubair Shafiq, Alex X. Liu, and Amir R. Khakpour. Revisiting Caching in Content Delivery Networks. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 567–568, New York, NY, USA, 2014. ACM.

[50] Ketan Shah, Anirban Mitra, and Dhruv Matani. An O (1) algorithm for implementing the LFU cache eviction scheme. *dhruvbird. com/lfu. pdf*, pages 1–8, 2010.

[51] Vasilis Sourlas, Lazaros Gkatzikis, Paris Flegkas, and Leandros Tassiulas. Distributed cache management in information-centric networks. *IEEE Transactions on Network and Service Management*, 10(3):286–299, 2013.

[52] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. On the [ir] relevance of network performance for data processing. *Network*, 40:60, 2016.

[53] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementa-*

tion, NSDI'15, pages 323–336, Berkeley, CA, USA, 2015. USENIX Association.

[54] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[55] Guoqiang Zhang, Yang Li, and Tao Lin. Caching in information centric networking: A survey. *Computer Networks*, 57(16):3128–3141, 2013.