# On the Power of In-Network Caching in the Hadoop Distributed File System

Eric Newberry
The University of Arizona
Tucson, Arizona, USA
enewberry@cs.arizona.edu

Beichuan Zhang
The University of Arizona
Tucson, Arizona, USA
bzhang@cs.arizona.edu

## ABSTRACT

The Hadoop Distributed File System (HDFS) is a network file system used to support multiple widely-used big data frameworks that can scale to run on large clusters. In this paper, we evaluate the effectiveness of using in-network caching on switches in HDFS-supported clusters in order to reduce per-link bandwidth usage in the network. We discovered that some applications featured large amounts of data requested by multiple clients and that, by caching read data in the network, the average per-link bandwidth usage of read operations in these applications could be reduced by more than half. We also found that the choice of cache replacement policy could have a significant impact on caching effectiveness in this environment, with LIRS and ARC generally performing the best for larger and smaller cache sizes, respectively. Moreover, given the structure of HDFS write operations, we developed a mechanism to reduce the total per-link bandwidth usage of HDFS write operations by replacing write pipelining with multicast. In order to evaluate in-network caching potential, we developed a simulator to replay real traces through a fat tree network simulating the caching architecture used in the Named Data Networking (NDN) information-centric networking (ICN) architecture. Our results suggest that ICN-style in-network caching can provide significant benefits to HDFS-supported big data clusters, justifying future work to apply ICN architectures to cluster environments.

## CCS CONCEPTS

• **Networks** → **Network measurement**; *Network simulations.*

## KEYWORDS

Caching, Spark, HDFS, Big data, Named data networking, NDN, Information-centric networking, ICN

**ACM Reference Format:**
Eric Newberry and Beichuan Zhang. 2019. On the Power of In-Network Caching in the Hadoop Distributed File System. In *6th ACM Conference on Information-Centric Networking (ICN '19), September 24–26, 2019, Macao, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3357150.3357392


*ICN '19, September 24–26, 2019, Macao, China*

ACM ISBN 978-1-4503-6970-1/19/09…$15.00
https://doi.org/10.1145/3357150.3357392

## 1 INTRODUCTION

The Hadoop Distributed File System (HDFS) is a network file system that provides reliable and distributed storage to support multiple big data frameworks, including both Apache Hadoop and Apache Spark, along with platforms based on these frameworks such as the Hortonworks Data Platform (HDP). The Hadoop framework is utilized by many large organizations, including Adobe, Alibaba, Criteo, Facebook, Google, IBM, Spotify, Twitter, and Yahoo! [3]. Some of the Hadoop clusters used by these organizations are very large in size, including clusters of 1100 nodes at Facebook and 1650 nodes at Spotify, with the latter running over 20000 jobs daily. A 2000-node cluster at Criteo is used to run a variety of job types, including both MapReduce (standard Hadoop) and Spark. Yahoo! reports an even larger cluster with 4500 nodes, used for research applications [3].

The HDFS file system runs on standard IP networks, meaning that stored objects are mapped at the application layer. Therefore, when reading or writing data, HDFS clients must coordinate with a central server (a.k.a. the "NameNode") to determine which storage servers (a.k.a. "DataNodes") they should read/write data to/from. Moreover, HDFS uses IP unicast to perform file system operations. Therefore, multiple read requests for the same data will result in duplicate copies of this data being transmitted over the network. Additionally, stored data is replicated across multiple DataNodes for both reliability and load balancing, and writes are performed in a "pipeline", with the same data being transmitted multiple times across the network: first from the client to the first DataNode, then from the first DataNode to the second DataNode, and so on [5].

Given this potential for duplicate data being sent across the network, in this paper, we evaluate the caching potential present in the HDFS file system. We seek to determine the benefit that in-network caching can provide to Spark applications, particularly the type of in-network caching seen in Named Data Networking (NDN) [33] and other information-centric networking (ICN) architectures. ICN architectures can potentially offer excellent benefits to HDFS-based clusters. While ICN architectures have been evaluated extensively for use in WAN environments, the benefits of caching and the impact of different replacement policies on network traffic in HDFS-based big data frameworks in LAN environments are mostly unknown. As mentioned above, such frameworks dominate big data computing and subsequently the network traffic seen in many data centers. In this paper, we explore the different traffic types present in HDFS and study temporal and spatial access to data and the impact that this has on cache behavior. Numerous cache replacement policies have been developed for buffer caches in operating systems – however, their benefit for in-network caching is relatively unknown. Subsequently, we evaluate and analyze how a

range of cache replacement policies are able to reduce the network traffic seen in big data computing.

## 2 MOTIVATION

The NDN architecture uses the information-centric networking (ICN) paradigm. While traditional network architectures like IP focus on delivering data between specific endpoints, ICN networks focus on retrieving information (or content). In other words, while IP networks "push" data across the network, ICN networks "pull" data from the network to a host. This is accomplished by assigning every piece of content in the network a unique name in a hierarchical namespace. Assigning each piece of content a hierarchical name provides semantic meaning at the network layer and does not require that a mapping be established between the application and network layers' location and naming systems. This namespace design also allows for new, more data-centric approaches to security [34], which could prove useful in big data environments. However, one feature of ICN that has the potential to provide significant benefits to big data environments is "in-network caching". In-network caching is a commonality among the multiple ICN architectures, enabled through the use of named data objects ("contents" or "Data packets") [1, 30]. [1] Data retrieval from an in-network cache will be notably quicker than retrieval from the original DataNode because the data will be retrieved from memory instead of HDD- or SSD-based storage. Moreover, it provides for simpler cache management than the existing HDFS "centralized cache management" mechanism, which requires centralized coordination, knowledge of data access patterns, and that explicit requests be made to cache specific data [4]. Additionally, this still necessitates that read requests be sent all the way to the DataNode to be satisfied.

Since contents in ICN are identified by unique names, they can be cached on intermediate nodes, such as routers or switches, to satisfy future requests for the same content. If a request for content (an "Interest") passes through a network node storing a cached copy of a matching content, it can satisfy the request and decrease the load on the data producer and the network overall, as well as reduce the time the requester (the "consumer") must wait for its request to be satisfied. Additionally, mechanisms have been proposed to allow off-path caches to satisfy requests for contents in ICN [29].

HDFS is used by several big data frameworks, including Apache Hadoop and Apache Spark. The existing standard implementation of HDFS runs on the IP protocol stack, leaving it unable to provide any form of caching at the network layer. Meanwhile, as discussed above, ICN architectures like NDN provide caching natively at the network layer. Given the benefits that in-network caching could provide to HDFS, Gibbens et al. developed a prototype implementation of HDFS that ran natively over NDN [7]. However, their implementation suffered from severe performance penalties compared to the official Apache Software Foundation implementation of HDFS, which utilizes TCP over IP networks. This was largely caused by the application-layer NDN forwarder they utilized, NFD, which is designed primarily for modularity and ease of extension [26], compared to the optimized and mature implementations of TCP built into modern operating systems.

Gibbens et al. also conducted evaluations of HDFS caching potential, although their evaluations were conducted only with applications using the Hadoop MapReduce framework and did not utilize other frameworks, such as Apache Spark. MapReduce splits workloads into a "Map" and a "Reduce" phase, writing the results from the Map phase back to HDFS before reading them again for the Reduce phase. Meanwhile, Spark attempts to keep this data stored in memory between compute phases [32] – this reduces both the total amount of traffic in the network and the amount of data that must be written to and read from slower HDDs or SSDs, increasing performance.

However, before significant effort is spent developing an optimized and fully-featured ICN solution for HDFS, it is important to determine if the ICN paradigm can provide significant benefits to this application. As such, our evaluations in this paper seek to determine whether there is significant data reuse, as well as total network traffic, in Spark applications utilizing HDFS. In addition, the large volumes of data communicated over the network necessitate specialized replacement policies to prevent cache thrashing. Understanding the specific benefits and requirements of this application would further motivate ICN deployments in the data center to support big data computing.

Numerous cache replacement policies have been developed for buffer caches in operating systems, with the most popular being Least Recently Used (LRU). While LRU performs well when capturing temporal locality in small file systems, it is unable to handle large data systems with varying reuse distances. To address these challenges, numerous replacement policies have been developed that try to capture even larger reuse distances and quickly replace data that is only used once. Subsequently, such replacement policies will intuitively perform much better than the standard LRU policy in big data environments. We selected four additional cache replacement policies for evaluation that aim to address larger reuse distances and prevent cache thrashing: Adaptive Replacement Cache (ARC) [23], Low Inter-reference Recency Set (LIRS) [13], Multi-Queue (MQ) [35], and Two Queue (2Q) [14].

Caching can occur at numerous nodes in the network, as we are conducting evaluations on a fat tree topology [2] connecting an Apache Spark cluster with 128 compute/DataNodes nodes and 1 coordinator/NameNode. Data behavior can be significantly different at different layers of the fat tree network topology and different polices may be more suitable at different levels. Subsequently, it is critical to understand the different levels and how different caching policies can interact with one another in such a topology.

### 2.1 HDFS Write Operations

We first consider write operations, as they can be easily accomplished in ICN networks with a minimal amount of caching. In HDFS, data is stored across one or more DataNodes, with the exact number of copies depending upon the "replication factor" of the stored HDFS block (i.e., the number of identical copies of the block stored across the file system). [2] To write to a new file or append to the end of an existing one, a compute node will first contact the NameNode to retrieve a list of the DataNodes that the file will be

[1] For an overview of the similarities and differences between the various ICN architectures, we recommend a survey paper on the subject [1, 30].

[2] The default replication factor in HDFS is three; however, this factor, along with the maximum block size, can be set on a file-by-file basis [5].

written to [5]. The client will then send each chunk of the data to be written to the first DataNode in the list, which will then send the chunk to the next DataNode on the list, and so on until all DataNodes in the list have a copy of the chunk – a feature known as "replication pipelining". The last node in this pipeline will acknowledge each chunk to the client as it is received [6]. Since the data replicated across each DataNode is identical, ICN's benefits are clearly visible as all the replications after the first in a write operation (from the first DataNode to the second DataNode, if the first DataNode is the same node as the writing compute node; otherwise, from the compute node to the first DataNode) can be satisfied by in-network caches.

To use ICN for HDFS writes, the write request that pushes the data chunks across the network would be converted into multiple ICN pull requests, where DataNodes ask for each chunk of the data to be stored. These concurrent read requests convert the pipeline into multicast-like traffic. This is because requests for each chunk of the written block will intersect at one or more switches in the network before reaching the writing client and either be aggregated into a single request or, if the request has already been satisfied on that switch, be satisfied from the cache on that switch. Therefore, the requests will effectively form a multicast tree.

As in IP-based HDFS, the client will still need to receive write acknowledgements to ensure that its data was successfully written to the destination DataNodes. However, since we are not using a pipeline, each DataNode will need to individually confirm receipt of the data. Since we can rely upon these DataNodes to retry requests for failed chunks of their own accord, we do not need to acknowledge the receipt of every chunk and can simply send an Interest for a DataNode-specific prefix registered by the client to indicate transfer completion once all chunks of the block have been successfully retrieved. If a DataNode fails during the write process, the client, having never received an acknowledgement Interest from it, will time out its write and seek out alternative DataNodes through the NameNode. Since the size of Interests would be significantly smaller than the Data packets in our scenario, the overhead of chunk retrieval and completion acknowledgement would be minimal.

A priority of HDFS is to store copy of the data on the same DataNode that produced it and (for a replication factor of three) store the remaining two copies on one DataNode in the same rack as the first DataNode and one DataNode in a different rack to provide reliability [5]. [3] Therefore, the amount of data that is sent through the network amounts to approximately twice the amount of data that is generated (in the case of a replication factor of three). Subsequently, utilizing ICN in this scenario can intuitively satisfy half of the write traffic from in-network caches.

To validate our intuitions, we conducted evaluations of the network traffic in a 129-node (128 compute/DataNodes and 1 coordinator/NameNode) virtualized cluster running Apache Spark. On this cluster, we ran benchmark applications from the Intel HiBench [12] benchmark suite. Table 2 shows the amount of unique data written and total data sent across the network as a result of replication during the execution of selected benchmarks from the HiBench suite. We observe in all scenarios that the default policy of storing one replica locally and replicating twice across the network generally holds, with less than 5% of blocks following a different replication pattern (and these appear to be either Spark libraries, configuration information, or temporary files). Furthermore, we observe that Spark's fundamental principle of keeping intermediate data in memory during execution works well and only three benchmarks show a significant amount of data being generated and written across the network. The `aggregation`, `scan`, and `sort` applications write a significant amount of data because they write many temporary, intermediate files to HDFS, with many individually sized over 100 MB for the latter two traces.

To determine how much of a benefit ICN could offer to HDFS writes, we calculated how much multicast could reduce total network traffic compared to pipelining. Thanks to the structure of fat trees, all hosts in different pods are the same number of hops away from the sender, regardless of the size of the topology. Moreover, it is highly likely that a real deployment of HDFS would use rack-aware replica placement, which will (for a replica factor of 3) place one replica on the local DataNode, one replica in a remote rack, and one replica on another DataNode in the same remote rack (in the same fat tree pod, as discussed in Section 3.2) [5]. Therefore, we can calculate a reduction percentage that will hold regardless of topology size or block size. As shown in Table 1, depending upon whether the third replica is on the same edge switch as the second replica or not, traffic will be reduced by 12.5% to 20%. The values in this table only consider the HDFS data and do not consider header sizes or other packet overheads. As we can see, our intuition of ICN being able to reduce network traffic of writes even with a small amount of caching is correct, highlighting ICN's multicast-like behavior and its benefits. Since replacement policies and caching at-large are irrelevant for write traffic, we focus on read traffic in the remainder of the paper.

## 2.2 HDFS Read Operations

HDFS read operations over the network occur when a compute node needs to read data stored on a remote DataNode. The primary goal of Spark is to place computing tasks on nodes that store input data, limiting the network traffic used to fetch data from other nodes. However, in scenarios where data needs to be utilized for computations on multiple nodes, these nodes will first contact the NameNode, which will respond with the address of the DataNode that the client should send its read request to [5]. At the file system level, the read request will contain the block ID (including version), the starting byte offset, and the size in bytes to read, along with other information extraneous to our purposes. The requested data is then returned to the compute node over the network.

Similar to our prior discussion of writes, our intuition tells us that there should be very little reads across the network if the application only performs computations on a single piece of data at a time and the Spark framework schedules compute tasks on the nodes containing the relevant input data. To validate our assumption, we executed the same benchmarks shown in Table 2 and focused on data read requests across the network. We eliminate read requests destined to the local node from our evaluations, as they do not transmit HDFS data over the network and therefore do not impact our caching evaluation. Table 3 shows the amount of data read

[3] An exception to this case is if there is no space remaining on the producing node, in which case this node will be replicated in the replication pipeline.

**Table 1: Reduction in write traffic for a 128 MB HDFS block on a fat tree from using multicast instead of pipelining**

| Scenario | Pipelining (MB) | Multicast (MB) | Reduction |
|---|---|---|---|
| Third replica on same edge switch | 1024 | 896 | 12.5% |
| Third replica on different edge switch | 1280 | 1024 | 20.0% |

**Table 2: Unique data written and network data transfer for writes in the selected benchmarks**

| Trace | Written Data (MB) | Network Transfer (MB) |
|---|---|---|
| aggregation | 3519 | 7038 |
| als | 24 | 48 |
| gbt | 99 | 198 |
| join | 101 | 202 |
| kmeans | 6 | 12 |
| linear | 9 | 19 |
| lr | 19 | 38 |
| rf | 11 | 22 |
| scan | 19169 | 38338 |
| sort | 27784 | 55568 |
| wordcount | 5 | 11 |

**Table 3: Unique and total read data size**

| Trace | Unique (MB) | Total (MB) |
|---|---|---|
| aggregation | 221 | 230 |
| als | 1147 | 1156 |
| gbt | 33 | 43 |
| join | 4 | 13 |
| kmeans | 395 | 573 |
| linear | 20952 | 111195 |
| lr | 8893 | 19966 |
| rf | 1084 | 1094 |
| scan | ~0 | 9 |
| sort | 740 | 749 |
| wordcount | 847 | 1499 |

during benchmark execution. There are two types of read activity we can consider: the "unique data size", which only counts each read unique byte a single time, no matter how many times it is read, and the "total data size", which counts every byte once for each time it is read. One thing that should be noted about HDFS read operations is that, although read offsets and lengths can be specified at a byte granularity, data is actually transferred from the DataNode to the client in fixed-sized, aligned chunks with a default size of 512 bytes, excepting the last chunk in a block, which will be of its actual size. The client will simply discard excess data outside its requested read boundaries. Therefore, when calculating the unique and total read data sizes, we aligned reads to these boundaries to produce the values shown in Table 3. We did not have knowledge of the exact sizes of each block, so we were unable to exactly match the HDFS algorithm in this calculation, given the exception to the fixed chunk size for the last chunk in a block. However, we note that this has an insignificant impact on our results, as the total and unique data sizes with exact byte boundaries and with aligned chunks differed by less than 1 MB for all applications.

As seen in these results, our intuition is validated and there is little read network traffic in most benchmarks, demonstrating the ability of Spark framework to place computation close to input data. However, the exceptions are `linear` and `lr`, which feature total data sizes of approximately 109 GB and 19 GB transferred across the network, respectively. These applications train machine learning models, namely using linear and logistic regression, respectively [12]. Based upon an analysis of the file names used for data stored on HDFS by these applications, we believe they show high potential for caching because they have large amounts of intermediate data that must be shared between many partitions of the application.

From the above observations, ICN deployment for big data services can potentially help five out of eleven applications, three for writes and two for reads, with no overlap between the two lists. The applications that experienced the greatest benefit from multicast-like write behavior were `aggregation`, `scan`, and `sort`; however, due to close temporal locality, replacement policies have little effect on the write traffic and we instead focus on read traffic and the impact that cache replacement policies have on read performance. Two applications (`linear` and `lr`) generated significant read traffic and can potentially benefit from caching when combined with a proper replacement policy. Subsequently, in the rest of the paper, we will focus on `linear` and `lr` and analyze the caching behavior for these two applications. `Linear` and `lr` are machine learning applications, a very important class of applications considering current trends of applying machine learning in big data computing.

## 3 DESIGN

HDFS blocks can be of very large sizes (up to 128 MB in the default configuration), while compute nodes may only request a small portion of a block. Therefore, our system splits HDFS blocks into small, sequential "cache blocks" to make it easier for replacement policies to operate more precisely on cached data, i.e., to discover "hot" portions of an HDFS block, if they exist. In our system, each of these cache blocks would be an ICN Data packet, necessitating that an Interest be sent individually for each cache block that a host wishes to read. Therefore, while we wish to optimize caching granularity as much as possible, there is a simultaneous motivation to use larger ICN Data packets to reduce the number of Interests that need to be sent, improving performance. This is opposed to the existing IP-based HDFS read mechanism, which uses very small chunks (by default, 512 bytes) within larger IP packets, which would result in high retrieval overhead if used in our system.

To determine the best cache block size for the traces we collected, we evaluated the number of network requests that would result from the use of cache blocks of a particular size – we chose to evaluate cache block sizes starting at 8 KB, doubling beyond that

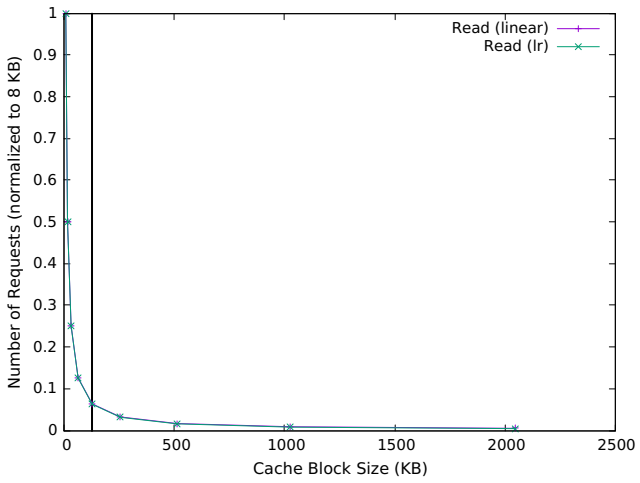


**Figure 1: Normalized comparison (between applications) of the number of read requests occurring from the use of cache blocks of various sizes**

point, and ending at 2 MB. 8 KB was chosen as the starting value because it is close to, but less than the current default maximum packet size (8800 bytes) in the NFD forwarder for NDN. Despite the use of 8 KB in the current implementation of NDN, we believe that larger blocks, and therefore larger Data packet sizes, may be more efficient for our system.

To determine the best cache block size, we must consider the side effects of having fewer requests, but larger cache blocks, versus having more requests, but smaller cache blocks. When requesting cache blocks using ICN, each node on the path will keep track of all pending Interests in its "Pending Interest Table" (PIT) until the requests are satisfied or time out [31]. Depending upon the number of cache blocks requested simultaneously, this data structure may become very large and require lots of memory to store on each node. Additionally, with smaller cache blocks, the caches on each device will need to keep track of many smaller packets, also leading to significant storage overhead for the relevant cache data structure (the "Content Store").

Meanwhile, the use of larger cache blocks would require fewer requests to obtain the same amount data and therefore lead to smaller data structures. However, this may lead to decreased caching efficiency, as replacement policies will have less ability to identify smaller regions of "hot" data, meaning that caches will not be able to store as many of these regions. Additionally, more extraneous data (outside of the single byte-granularity bounds of the HDFS read request) will be returned in start and end cache blocks. This network bandwidth will be wasted because the end host reading an HDFS block would simply discard portions of the received cache blocks falling outside of the byte range they requested.

Therefore, our goal is to reduce the number of requests as much as possible, while still allowing for reasonable caching granularity. In order to determine a reasonable cache block size, we conducted evaluations of the number of Interests that would need to be issued to retrieve all cache blocks at various cache block sizes for both the `linear` and `lr` traces – for each trace, these values were normalized to the number of requests for that trace with 8 KB cache blocks (which is shown with a value of 1 in the figure). The results of this evaluation are shown in Figure 1. As expected, this approximately follows an inversely proportional distribution ($\frac{1}{x}$), as each doubling of cache block size results in approximately half as many requests. In this figure, the rate at which the number of network requests changes significantly decreases when cache block sizes are approximately 128 KB – 128 KB is indicated with a vertical bar in this figure. Further increases in cache block size beyond this point will therefore provide only minimal improvements to router overhead, while harming caching granularity. Therefore, we determined that 128 KB was the best cache block size to conduct our evaluations with, given its balance between generating fewer network requests per read operation and being able to evaluate content popularity on a fine granularity.

## 3.1 Network and Caching Behavior

Now, we will address how a read request in an HDFS system running on ICN would operate. First, the client would need to, based upon the cache block size in use (128 KB in our case), find the range of cache blocks that must be read to satisfy the request. To determine which cache blocks a request covers, we use the following formulas to find the start and end cache blocks of a read request:

$$Start_{CB} = \lfloor Offset_{OP}/Size_{CB} \rfloor$$
$$End_{CB} = \lfloor (Offset_{OP} + Size_{OP} - 1)/Size_{CB} \rfloor$$

In these formulas, the *CB* subscript refers to the cache blocks used for in-network caching, while the *OP* subscript refers to the original HDFS blocks. Since cache blocks are of a constant size and HDFS operations are specified on a byte-level granularity (featuring a starting byte offset and a total size), we find the starting cache block by calculating which cache block contains the offset. Similarly, we find the ending cache block by finding the ending offset of the original HDFS operation, which is the starting byte offset summed with the read size (subtracting one from this sum because the starting offset is included in the operation size). These values are floored to get integers representing the range of cache blocks necessary to complete the operation.

Once this range has been determined, the client will send Interest packets to request all the blocks in the range [start, end] (one Interest will be sent per requested block). ICN routing is based upon Interest name, rather than source and destination address. Many novel routing protocols have been proposed for ICN, including link-state [11, 20] and hyperbolic [18, 19] routing protocols. However, for the purposes of our evaluations, we relied upon heuristics about the structure and operation of fat trees to find routes between hosts. In particular, we used a simple calculation to determine which switch at the core layer an Interest packet would traverse, as shown in the following formula:

$$(src + dest) \mod nCore$$

In this formula, **src** and **dest** refer to the original IDs of the client and DataNode hosts on the trace generation cluster, respectively, while **nCore** refers to the number of core switches in the topology

(in the fat tree topology, the number of core switches varies with the parameter $k$ [2]). The original host IDs were the concatenated string representations of the last two bytes of a node's IP address. [4]

When an Interest reaches a switch or router in the network, it checks the name of the Interest (containing the HDFS block ID and cache block ID) against the names of Data packets in its "Content Store" [31]. If it finds a match, it returns the cached Data packet and considers the Interest to be "satisfied". Otherwise, it will send the Interest on one or more path(s) determined by the forwarding system. While NDN uses longest-prefix matching of names [5] (with names structured as components separated by forward slashes) to match Interest names against Data packets in the Content Store, we envision that a real implementation of this system would use specific enough Interest names to ensure that only the correct version and cache block ID are matched. Returning Data packets will be cached in the Content Store, which is managed by a cache replacement policy, as discussed in Section 3.2. Data packets simply follow the reverse path of Interest(s) they satisfy, so we leave the optimization of HDFS routing in an ICN network to future work.

While one might presume that we could optimize storage utilization by reducing the number of copies of a piece of data stored on DataNodes based upon how frequently it is cached in the network, this may in fact be detrimental to HDFS's critical goal of data integrity in the face of DataNode failure. This is because in-network caching is simply an optimization, with no guarantee that cached data will remain so. This makes in-network caches too volatile to be treated as a replica of the original data by HDFS.

## 3.2 Multi-Layer Caching Hierarchy

The "fat tree" network topology that we utilize in this paper was proposed by Al-Fares et al. in 2008 [2], building upon an earlier topology proposed by Leiserson in 1985 [21] (also called a "fat tree"), but making use of a Clos network containing links of equal bandwidth, instead of containing "fatter" (i.e., greater bandwidth) links when approaching the root of the tree. This topology was chosen because it is applicable to the data center environments utilized by large big data systems, particularly because of its redundant links and convergence onto fewer switches as one approaches the core of the network. An example of this topology can be seen in Figure 2.

The fat trees developed by Al-Fares et al. feature a parameter $k$ that controls the size of the topology, including the number of end hosts. Fat trees are split up into three layers of switches and one layer of end hosts. The layers of switches are the "core" layer, the "aggregation" layer, and the "edge" layer – edge switches connect directly to end hosts and aggregation switches, aggregation switches connect to edge switches and core switches, and core switches connect only to aggregation switches. The topology is divided into pods containing an equal number of end hosts, with the topology containing $k$ pods. Each pod also contains $k/2$ edge switches and $k/2$ aggregation switches; each edge switch connects to every aggregation switch in the pod. Meanwhile, core switches are outside the pods (there are $(k/2)^2$ core switches in the topology) – each aggregation switch connects to $k/2$ core switches and each core switch connects to one aggregation switch in each pod.

We sought to match, as closely as possible, the number of DataNodes in the trace to the number of end hosts in the fat tree. Therefore, given traces containing 128 DataNodes and 1 NameNode, we set $k = 8$, which gave us a topology with 128 DataNode end hosts [6], 32 edge switches, 32 aggregation switches, and 16 core switches.

In our generated evaluation topology, caches exist on every switch in the network and can store the same amount of data. On these caches, we simulated five cache replacement policies: 2Q, ARC, LIRS, MQ, and LRU, comparing the performance of the first four against the LRU policy that is currently used by NDN. We chose these five policies because they are widely used and known in various areas of computer systems. We will now describe them.

***Two Queue (2Q).*** utilizes three queues, *Am*, *A1in*, and *A1out*. The first two queues contain blocks that are currently stored in the cache, while the latter contains the metadata of recently-evicted blocks. When a block is first accessed, it will be added to *A1in*, which is structured as a FIFO queue. When it is evicted from *A1in*, the block will be evicted from the cache, but its metadata will be moved to *A1out*, which is also structured as a FIFO queue. If the block is accessed again before its metadata is evicted from *A1out*, it will be moved to *Am*, which contains only hot blocks and is structured as an LRU queue [14]. In our evaluations, *A1in* held 25% of the cache, while *Am* held 75% of the cache, as recommended by the authors [14]. The goal of 2Q is to evict data that is used only once and store in the LRU cache data that is accessed more than once. Subsequently, we expect to see better performance from 2Q in big data environments compared to LRU, as large amounts of data may be used only once, polluting the standard LRU cache.

***Adaptive Replacement Policy (ARC).*** is similar to 2Q and uses two lists to filter single use blocks and keep popular block in the cache. While 2Q uses fixed-sized queues, ARC dynamically adjusts queue sizes based on their usefulness in capturing temporal locality in the moving data [23]. Dynamic distribution between two queues may be more beneficial for big data traffic than 2Q, as 2Q may discard data earlier due to it's fixed size *A1in* queue, which may be overflowed if large amounts of data are seen by the cache.

***Multi-Queue (MQ).*** extends the idea of 2Q by providing more queues and allocating the data to a given queue based on its access frequency. The idea is to keep blocks with higher access frequencies in the cache for a longer amount of time. Similar to 2Q, MQ keeps track of recently-evicted block addresses and access frequency in a buffer called $Q_{out}$. If a cache miss occurs for a block listed in $Q_{out}$, the block will be restored to the queue matching its access frequency [35]. MQ has the potential to detect and sort more popular data, benefiting big data applications. However, if the data reuse frequency is similar between blocks, it will perform similarly to 2Q.

***Low Interreference Recency Set (LIRS).*** enhances the idea of frequency by keeping track of reuse distance (how soon the data is used again), keeping data in the cache according to this metric. Similar to 2Q, LIRS filters single use blocks and does not pollute the main

[4] In each cluster, we verified that there were no collisions between the IDs of any hosts caused by this mechanism.

[5] Other ICN architectures may only use exact matching of names in their forwarding systems.

[6] We solved the issue of placing the NameNode by mapping it to first end host in the topology, unless these would be the same host, in which case we mapped it to the second end host in the topology.

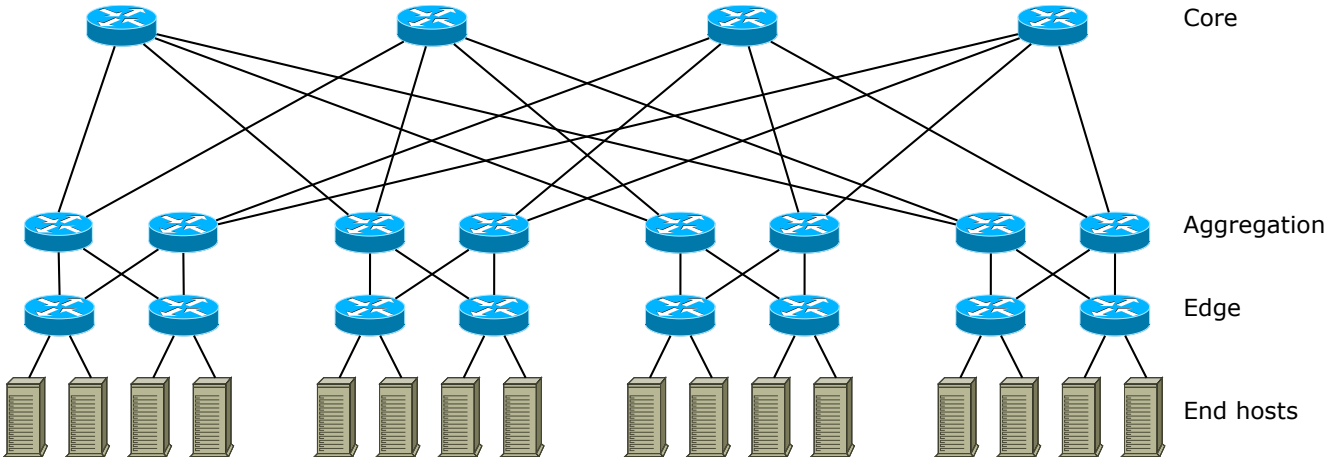


**Figure 2: Fat tree network ($k = 4$)**

cache [13]. LIRS is among the best performing policies for cache management and we also expect it to effectively capture temporal locality in big data applications and offer good performance.

### 3.3 File System Mechanics

When adapting a distributed file system to the ICN architecture, one must ensure the continued correct functionality of various file system mechanisms. Therefore, we considered how two essential HDFS mechanisms, data integrity validation and version discovery, would operate in an ICN environment.

***Data Integrity.*** HDFS relies upon checksums to ensure that retrieved data has not been corrupted in storage or transit [5]. When a DataNode transmits a read block to a client, each chunk in the block will be accompanied by a separately-stored checksum, with the client having the option of validating the retrieved block against these checksums to ensure their integrity. If a chunk in the block fails this validation, the client can reattempt the read from a different DataNode. However, since ICN operates based upon data names instead of host names, clients would not be able to explicitly request the data from a different DataNode. Moreover, the incorrect version of the block would likely still be cached within the network. Instead, in our system, the client could report this incident to the NameNode, which would then instruct the DataNode to delete the corrupted version of the block. After receiving confirmation of this, the client could override potentially incorrect versions of the block cached in the network by retrieving the blocks again while specifying the "MustBeFresh" element in their Interests. When combined with a "FreshnessPeriod" of zero in all transmitted Data packets, this mechanism would prevent the Interest from being satisfied by cached Data packets, forcing it to be forwarded all the way to a producer DataNode [24, 25]. However, this is not an ideal solution – in an ideal ICN distributed file system, data blocks would be stored as Data packets at rest (and therefore be combined with their checksum). This would avoid this issue, because corrupted Data packets would be dropped in the network after failing signature validation.

***Version Discovery.*** Given that blocks in HDFS can be updated while keeping the same block ID (while changing the version number/"generation stamp" value), there may be concerns about the consistency of caches in our system. However, in ICN, the problem changes from one of cache consistency to one of version discovery, allowing us to work around the traditional caching issue of consistency. In traditional HDFS systems, the latest version information will be retrieved directly from the NameNode. In ICN, the default caching mechanisms may prevent information retrieved from the NameNode from being the absolute latest. To solve this issue, we rely upon work by Mastorakis et al., who proposed a mechanism to discover the latest version of a given content by first retrieving an up-to-date "metadata" Data packet that describes how to obtain the latest version of a content [22]. Since version numbers can be included as a component of an Interest's name, after retrieving a "metadata" Data packet from the NameNode, the version number can be included to ensure that the latest version of a block will be retrieved. As such, it will not matter whether out-of-date blocks are cached in the network, as they will be ignored due to the differing version number included in the Interest name.

## 4 EVALUATION ENVIRONMENT

The benchmark application traces used in our evaluations were collected on the Amazon Web Services (AWS) Elastic Map Reduce (EMR) system. The clusters consisted of 129 `m3.xlarge` nodes, with 1 coordinator/NameNode and 128 compute/DataNodes. The nodes in the cluster ran Amazon EMR release 5.10.0, which features Apache Hadoop version 2.7.3 and Apache Spark version 2.2.0. We used the default configuration provided by Amazon, with a few changes that we will now discuss. In order to obtain log entries from which to generate a trace of HDFS read and write operations, we set HDFS daemons to log at the "DEBUG" log level. We disabled dynamic resource allocation in Spark and manually set the number of executors to 128 to ensure that the workload would be truly spread across the entire cluster, fully demonstrating the caching potential available in Spark. We set Spark to partition jobs into 256 partitions to reduce the workload on each Spark executor.

In the benchmark suite we used for our evaluations, HiBench, each application's runtime is split into a "prepare" and a "run" stage, which distribute the input data across the cluster and perform the big data computations, respectively. We are only focusing on the "run" stage as we want to explore benefits of ICN for big data computing. Initial data allocation and replication would benefit from the same multicast-like features offered by ICN, as we discussed in the case of writes and, subsequently, we do not consider it when studying cache replacement behavior during execution time.

After obtaining the logs from each DataNode, a trace was generated for each cluster by parsing each HDFS log file. Our traces contained the following information for each HDFS read and write operation: (i) the operation timestamp, (ii) operation type (`READ` or `WRITE`), (iii) block ID, (iv) starting byte offset, (v) size (in bytes), (vi) source node, and (vii) destination node. Our traces do not include the "generation stamps" (i.e., versions) of blocks because we verified that every block in the traces we evaluated was only associated with a single generation stamp value.

After extracting the traces, we split the read and write operations into separate logs for each application, also splitting the "prepare" and "run" stages, although only the read traces from the "run" stage were used for our evaluations in Section 5.

The generated traces were analyzed on a fat tree [2] topology, as discussed in Section 3.2. DataNodes (and the NameNode) were assigned to end hosts in the topology as they were referenced by operations during runtime. To avoid spatially clustering DataNodes based upon the temporal order in which operations occurred, positions in the topology were assigned using a seeded pseudo-random number generation (PRNG). This PRNG assigned end hosts to a random position in the cluster; if the generated position was already assigned, it would generate another position and repeat this process until it found an one that was empty. Due to the fixed order of operations, evaluations with the same input parameters, including PRNG seed, would result in the same end host positions. To avoid any irregularities caused by the use of a particular PRNG seed, ten random seeds were chosen and each scenario was repeated with each input seed. This set of seeds was used across all scenarios.

HDFS instances can be configured to be "aware" of which rack a DataNode is placed on, allowing the NameNode to place replicas of blocks and direct read requests in a manner that improves performance and reliability [5]. However, the clusters we obtained traces from placed all DataNodes in the same rack. Therefore, our random host placement did not interfere with this feature.

## 5 EVALUATION

In IP networks, a DataNode will be contacted every time a compute node wants to read data, even if that data was recently read by another host in the network. This is because IP lacks a network-layer caching feature to deduplicate redundant or overlapping requests. Meanwhile, the ICN architecture natively provides a caching feature at the network layer, allowing it to deduplicate these requests - "in-network caching". In ICN networks, each node in the network, whether an end host or intermediate network device, contains a cache of recent data packets that traversed the forwarder running on that host (the "ContentStore"). However, our evaluations only consider caches within the network itself and exclude those on end hosts, since a host will likely not request the same HDFS data again within a short time frame. Therefore, we calculated the reduction in total network traffic that resulted from using in-network caching on a fat tree topology. For each benchmark application showing promise for read applications in Section 2 (`linear` and `lr`), we conducted an evaluation of the total network traffic across all nodes while varying the cache size (powers of two from $2^0$ to $2^{10}$ MB) and replacement policy on each host. As justified in Section 3, we divided each HDFS block into cache blocks sized at 128 KB (including the last in an HDFS block, regardless of its actual size). We also conducted evaluations of the total network traffic with no caching, as well as with unlimited cache space to find the maximum and minimum bounds, respectively, in a given scenario – this allowed us to determine the improvement brought by caching.

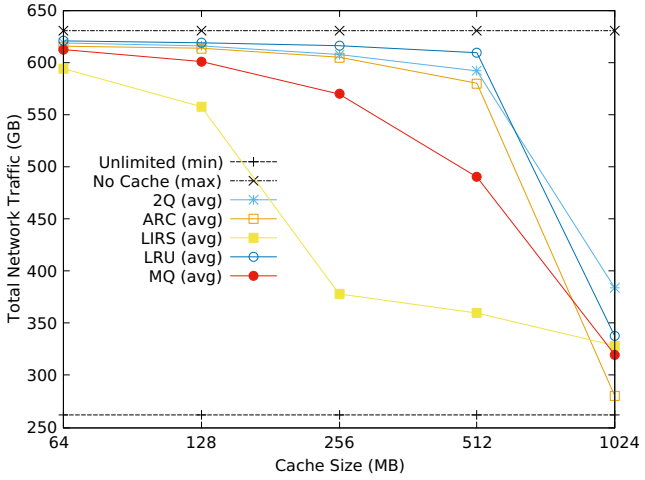


**Figure 3: Read traffic of `linear` with 128 KB cache blocks**

To evaluate the effect of caching, we compared the averaged total traffic from our 10 seeded PRNG trials (for each combination of application, cache block size, cache replacement policy, and cache size at each node) with the minimum observed traffic with unlimited cache and the maximum observed traffic with no cache for each application. The specific results for the benchmark applications `linear` and `lr` are presented in Sections 5.1 and 5.2, respectively.

### 5.1 linear

With `linear`, which had a significantly larger total data transfer size than `lr`, we discovered that in-network caching could significantly reduce network traffic. However, the level to which traffic was reduced varied with the replacement policy. For comparison, we discovered that, if unlimited-sized caches were used in the scenarios we evaluated for `linear`, total traffic in our scenarios could be reduced to only be approx. 45% of the total traffic with no in-network caches. Notably, this is not the "perfect" reduction of approx. 81% promised by the unique and total read data sizes in Section 2.2. This is because of the large number of paths that traffic can take in the fat tree topology, meaning that Interests will not necessarily cross paths with a previously cached matching Data packet, even with infinite cache capacity. The total traffic seen in `linear` with each

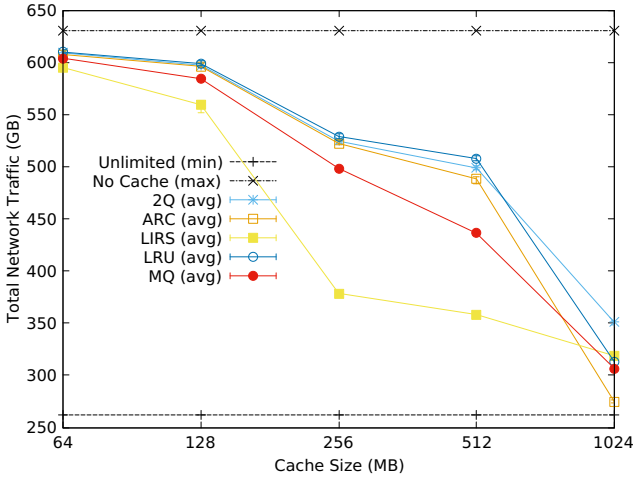


**Figure 4: Read traffic of `linear` with 128 KB cache blocks using 2Q at the edge layer and LIRS at the aggregation layer**

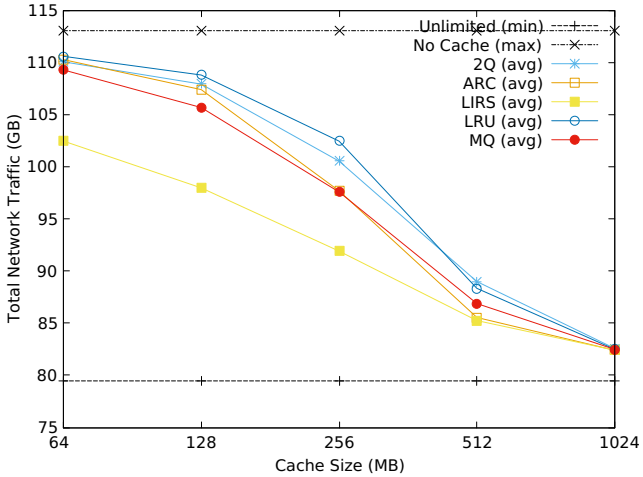


**Figure 5: Read traffic of `lr` with 128 KB cache blocks**

combination of parameters, including with unlimited cache and with no cache, can be found in Figure 3.

With the evaluated cache sizes up to, but not including, 256 MB, no replacement policy appeared to significantly reduce the amount of network traffic. However, with 256 MB of cache, the LIRS replacement policy reduced total traffic to approx. 60% of the total without caching. At this cache size, the other cache policies did not significantly differ from their previous pattern of gradual reduction, although MQ performed notably better than 2Q, ARC, and LRU. When cache sizes were 512 MB, this pattern continued.

However, by the time cache sizes reached 1 GB, ARC performed the best of the five policies, reducing total traffic to approx. 44.5% of the total traffic without caching. LIRS, LRU, and MQ performed reasonably well at this cache size, reducing traffic to approx. 52.1%, 53.6%, and 50.7% of the total traffic without caching, respectively. Interestingly, 2Q performed the worst of the five evaluated policies at 1 GB, performing significantly worse than LRU and only reducing traffic to approx. 60.9% of the total traffic without caches.

Therefore, the best caching policy for `linear` at larger cache sizes appears to be ARC. However, for smaller cache sizes, LIRS appears to be the best option.

***Multi-policy topologies.*** We also evaluated the effect of using different replacement policies at each layer of the fat tree, given that using a combination of replacement policies may produce lower overall traffic levels than the uniform use of a single replacement policy. Therefore, we evaluated the `linear` trace using a combination of the 2Q, ARC, LIRS, LRU, and MQ, resulting in a total of 125 series (with 5 cache sizes evaluated for each combination of policies). The results for these evaluations were split into separate figures by combination of edge and aggregation policy, with the core policy being represented by the series within the figure. We found that the results for each combination of edge and aggregation policy followed four general patterns: those that were not significantly different from the base result in Figure 3; those that reduced total traffic from the base by a small, but significant margin; those that reduced total traffic by a very significant amount; and those whose effect fell somewhat between the latter two.

All of the configurations that reduced traffic the most featured LIRS at the aggregation layer – we show the results with 2Q at the edge layer and LIRS at the aggregation layer in Figure 4 as a generalization of this class of results, as the rest of the results with LIRS at the aggregation layer are very similar to this. Moreover, the best results were seen with 1 GB cache sizes using ARC at the edge and core layers and LIRS at the aggregation layer. ARC is intended to act as a second-layer cache, so this explains its effectiveness at the core layer [23]. Therefore, it appears that the best topology for this type of workload would feature LIRS at the aggregation layer.

## 5.2 lr

The `lr` application also showed significant potential for reductions in network traffic through the use of in-network caches for read operations. However, likely due to the smaller data size of this application compared to `linear`, all cache replacement policies showed similar traffic reduction patterns to each other – this is shown in Figure 5. In fact, the total network traffic seen with all replacement policies almost converged with cache sizes of 1 GB, with the range between policies being approx. 195 MB with 1 GB caches. However, for all evaluated cache sizes less than 1 GB, LIRS performed the best. LRU performed the worst at evaluated cache sizes up to and including 256 MB. Meanwhile, 2Q performed the worst when caches could store 512 MB and 1 GB.

With `lr`, caches of unlimited size were able to reduce traffic to approx. 70.3% of the total traffic with no caching. However, at the largest evaluated cache size (1 GB), all the replacement policies were able to reduce traffic to approx. 3 GB above the total traffic with unlimited cache sizes (approx. 82 GB vs. 79 GB), out of a total observed traffic size without caching of approx. 113 GB. This indicates that caching is very effective for `lr`, even with only 1 GB of cache on each switch in the network.

However, while increases in cache size produced increasing reductions in total network traffic up to cache sizes of 512 MB, from

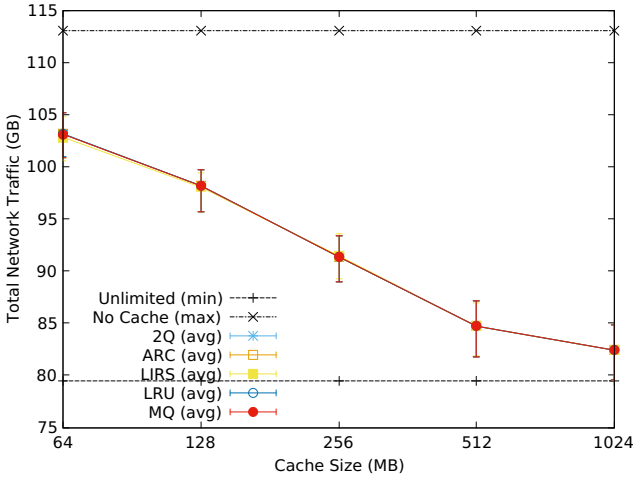


**Figure 6: Read traffic of `lr` with 128 KB cache blocks using LIRS at the edge layer and MQ at the aggregation layer**

512 MB to 1 GB, total traffic does not reduce as significantly. Therefore, 512 MB appears to be the best of the evaluated cache sizes for use with `lr`, especially when LIRS or ARC are utilized.

***Multi-policy topologies.*** As with `linear`, we evaluated the effect of using different replacement policies at each layer of the topology with the `lr` trace. Of these, the best performing scenarios were those that used LIRS at the edge layer, with the choice of the replacement policies at the other layers having little impact on the results. To represent this class of results, Figure 6 shows the results when LIRS was used at the edge layer and MQ was used at the aggregation layer. However, it is notable that the error bars (representing the min and max values observed for that datapoint) are significant, indicating that some configurations of each scenario (based upon the chosen end host placement PRNG seed) varied significantly from the average. However, the total range was less than 10 GB. Therefore, it appears that the best choice of policy for `lr` is LIRS at all layers of the fat tree.

## 6 CONCLUSION

We have demonstrated the benefits that in-network caching can provide to big data applications running on top of the Hadoop Distributed File System by reducing the total amount of traffic sent over cluster networks. We have used different replacement policies to evaluate the caching potential of ICN in big data applications.

Our results suggest that, overall, LIRS and ARC provide the best performance for in-network caching in HDFS deployments in fat trees, a common data center topology. Generally, we found that LIRS performed the best for smaller cache sizes and ARC performed the best for larger cache sizes. In addition, by combining LIRS and ARC at different layers of the fat tree topology, we were able to provide better performance in the larger of our two application traces.

While the in-network caching architecture we used for our evaluations was based upon that used in ICN, the key feature that enables the benefits we observed is the mapping of data name to content at the network layer, a central feature of ICN, and the knowledge of this mapping on forwarders in the network. However, ICN architectures can provide many additional benefits to distributed file systems beyond caching. This includes the ability to implicitly retrieve data from multiple sources and over multiple paths, as well as a stateful forwarding plane that can detect failures and discover alternative paths. These features would allow distributed file systems to be more resilient to failure, as well as increase the efficiency of data retrieval.

### 6.1 Future Work

While fat-trees are an effective data center topology, other data center topologies have been developed and utilized in real environments. These include the more traditional two- and three-tier architectures [16], BCube [9], DCell [10], PortLand [27], SEATTLE [15], and VL2 [8]. Alternative topologies may produce different results than fat-trees. However, we believe the fat-tree topology to contain many features generally present in data center environments, including path redundancy and "tapering" of links as one approaches the core layer of the network. Therefore, we believe our results are generally applicable to data center environments and not simply to those that use the fat tree topology.

We conducted our evaluations using the 2Q, ARC, LIRS, LRU, and MQ cache replacement policies. However, this is only a subset of the many replacement policies in existence. Additional replacement policies that we would like to evaluate include LRFU [17] and LRU-K [28]. This would increase the breadth of our evaluations, as other replacement policies may display significantly better or worse hit ratios than the five we evaluated in this paper.

Additionally, while we chose our target cache block size by weighing the costs and benefits of greater caching granularity versus the overhead of a large number of network requests, it would be good to conduct a thorough evaluation to determine the optimal cache block size for this environment.

Moreover, further work should be conducted to evaluate the overall benefits that ICN networks can provide to distributed file systems. This includes implicit multisource and multipath retrieval and the stateful forwarding plane, as mentioned previously, among other benefits – these mechanisms can lead to enhanced file system resiliency and efficiency. Additionally, while our proposed HDFS design retains the NameNode from the standard design to coordinate operations, it would be good to evaluate whether some or all of its functionalities could be decentralized through ICN mechanisms.

## ACKNOWLEDGMENTS

The authors would like to thank Ali Anwar and Ali R. Butt of Virginia Tech for contributing to our understanding of Hadoop and Spark system behavior. Additionally, we would like to thank our shepherd, Börje Ohlman, as well as Hamed Yousefi, Teng Liang, and the anonymous reviewers for their helpful feedback on the paper. Moreover, we thank Mathias Gibbens for his technical assistance and Navdeep Singh for his contributions to trace collection and tool development.

This work was supported by the National Science Foundation under award CNS-1629009.

## REFERENCES


[1] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. 2012. A survey of information-centric networking. *IEEE Communications Magazine* 50, 7 (July 2012), 26–36. https://doi.org/10.1109/MCOM.2012.6231276

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review* 38, 4 (2008), 63–74.

[3] Apache Software Foundation. [n. d.]. PoweredBy - Hadoop Wiki. ([n. d.]). https://cwiki.apache.org/confluence/display/HADOOP2/PoweredBy

[4] Apache Software Foundation. 2018. Centralized Cache Management in HDFS. (2018). https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html

[5] Apache Software Foundation. 2018. HDFS Architecture. (2018). https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[6] Robert Chansler, Hairong Kuang, Sanjay Radia, Konstantin Shvachko, and Suresh Srinivas. 2012. The Hadoop Distributed File System. In *The Architecture of Open Source Applications*, Amy Brown and Greg Wilson (Eds.). Vol. 1. lulu.com, Chapter 8.

[7] Mathias Gibbens, Chris Gniady, Lei Ye, and Beichuan Zhang. 2017. Hadoop on named data networking: experience and results. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 1 (June 2017), 2.

[8] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1592568.1592576

[9] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. 2009. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1592568.1592577

[10] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. 2008. Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/1402958.1402968

[11] A K M Mahmudul Hoque, Syed Obaid Amin, Adam Alyyan, Beichuan Zhang, Lixia Zhang, and Lan Wang. 2013. NLSR: Named-data Link State Routing Protocol. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-centric Networking (ICN '13)*. ACM, New York, NY, USA, 15–20. https://doi.org/10.1145/2491224.2491231

[12] Intel Corporation. [n. d.]. HiBench. ([n. d.]). https://github.com/intel-hadoop/HiBench

[13] Song Jiang and Xiaodong Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 31–42.

[14] Theodore Johnson, Dennis Shasha, et al. 1994. 2Q: a low overhead high performance bu er management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*. 439–450.

[15] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. 2008. Floodless in Seattle: A Scalable Ethernet Architecture for Large Enterprises. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1402958.1402961

[16] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. 2012. GreenCloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing* 62, 3 (01 Dec 2012), 1263–1283. https://doi.org/10.1007/s11227-010-0504-1

[17] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers* 50, 12 (2001), 1352–1361.

[18] Vince Lehman, Ashlesh Gawande, Rodrigo Aldecoa, Dmitri Kiroukov, Beichuan Zhang, Lixia Zhang, and Lan Wang. 2016. *An Experimental Investigation of Hyperbolic Routing with a Smart Forwarding Plane in NDN.* Technical Report NDN-0042. NDN Project.

[19] Vince Lehman, Ashlesh Gawande, Beichuan Zhang, Lixia Zhang, Rodrigo Aldecoa, Dmitri Krioukov, and Lan Wang. 2016. An experimental investigation of hyperbolic routing with a smart forwarding plane in NDN. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*. 1–10. https://doi.org/10.1109/IWQoS.2016.7590394

[20] Vince Lehman, A K M Mahmadul Hoque, Yingdi Yu, Lan Wang, Beichuan Zhang, and Lixia Zhang. 2016. *A Secure Link State Routing Protocol for NDN.* Technical Report NDN-0037. NDN Project.

[21] C. E. Leiserson. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* C-34, 10 (Oct 1985), 892–901. https://doi.org/10.1109/TC.1985.6312192

[22] S. Mastorakis, P. Gusev, A. Afanasyev, and L. Zhang. 2018. Real-Time Data Retrieval in Named Data Networking. In *2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN)*. 61–66. https://doi.org/10.1109/HOTICN.2018.8605992

[23] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache.. In *FAST*, Vol. 3. 115–130.

[24] Named Data Networking Project Team. [n. d.]. Data Packet. ([n. d.]). https://named-data.net/doc/NDN-packet-spec/current/data.html

[25] Named Data Networking Project Team. [n. d.]. Interest Packet. ([n. d.]). https://named-data.net/doc/NDN-packet-spec/current/interest.html

[26] Named Data Networking Project Team. [n. d.]. NFD Readme. ([n. d.]). https://github.com/named-data/NFD/blob/master/README.md

[27] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/1592568.1592575

[28] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. ACM, New York, NY, USA, 297–306. https://doi.org/10.1145/170035.170081

[29] Junxiao Shi, Eric Newberry, and Beichuan Zhang. 2017. On broadcast-based self-learning in named data networking. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*. 1–9. https://doi.org/10.23919/IFIPNetworking.2017.8264832

[30] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos. 2014. A Survey of Information-Centric Networking Research. *IEEE Communications Surveys Tutorials* 16, 2 (Second 2014), 1024–1049. https://doi.org/10.1109/SURV.2013.070813.00063

[31] Cheng Yi, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. 2013. A case for stateful forwarding plane. *Computer Communications* 36, 7 (2013), 779 – 791. https://doi.org/10.1016/j.comcom.2013.01.005

[32] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. 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*. USENIX Association, 2–2.

[33] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, Patrick Crowley, Christos Papadopoulos, Lan Wang, Beichuan Zhang, et al. 2014. Named data networking. *ACM SIGCOMM Computer Communication Review* 44, 3 (July 2014), 66–73.

[34] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. 2018. An Overview of Security Support in Named Data Networking. *IEEE Communications Magazine* 56, 11 (November 2018), 62–68. https://doi.org/10.1109/MCOM.2018.1701147

[35] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches.. In *USENIX Annual Technical Conference, General Track*. 91–104.