A CACHING EVALUATION OF THE HADOOP DISTRIBUTED FILE SYSTEM

By

ERIC EVAN MICHAEL NEWBERRY

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

MAY 2018

Approved by:

Dr. Beichuan Zhang
Department of Computer Science

Abstract

The Hadoop Distributed File System (HDFS) is a distributed file system used to support multiple widely-used big data frameworks, including Apache Hadoop and Apache Spark. Since these frameworks are often run across many compute nodes, it is possible that multiple nodes will read the same data. In addition, since data is replicated across multiple nodes for storage, the same data will be written multiple times across the network. In this paper, we conduct an evaluation of the caching potential present in HDFS in order to determine if in-network caching, particularly of the type seen in Named Data Networking (NDN), would reduce the amount of traffic seen in a Spark cluster network, as well as the average load on each data storage node. Our results show that for most benchmarks running on Apache Spark, a majority of the large read operations were done to transfer the Spark and application dependency libraries to each compute node. In addition, there was not a significant amount of read traffic in the network for most of the applications we evaluated, making the benefits of in-network caching for HDFS questionable.

1 Introduction

The Hadoop Distributed File System (HDFS) is a network file system that provides reliable and distributed storage for multiple big data frameworks, including both Apache Hadoop and Apache Spark. When writing or retrieving data over HDFS, clients must coordinate with a central server (or "NameNode") to determine which storage server (or "DataNode") they should read/write data to/from. However, because HDFS uses IP unicast to perform file system operations, multiple read requests for the same data will result in duplicate copies of this data being transmitted over the network. In addition, during writes, data is replicated to multiple DataNodes for reliability and load balancing purposes.

In this paper, we evaluate the caching potential of the HDFS file system. In particular, we seek to determine the benefit that could be gained from utilizing in-network caching to cache named pieces of HDFS data, as seen in the information-centric networking (ICN) paradigm. This paradigm is exemplified by the Named Data Networking (NDN) [1] project. We believe that NDN may be well suited for this purpose.

To investigate how in-network caching could function on a real HDFS system, we gathered HDFS traces by running the HiBench benchmark suite on the Apache Spark framework. These benchmarks were run on a 129-node cluster on the Amazon Web Services (AWS) Elastic Map Reduce (EMR) platform and succinct traces were generated from the combined HDFS DataNode logs. Then, the caching potential of these traces was evaluated using two custom-developed caching simulators. In our evaluations of caching potential, we only concerned ourselves with read traffic, since write traffic can simply use multicast to write data to multiple DataNodes simultaneously.

In Section 2, we discuss existing work and motivate our evaluation, indicating why NDN-like in-network caching may be of benefit in HDFS environments. Section 3 discusses how we conducted our evaluation, including the tools and environment utilized. Section 4 contains the results of our evaluation and discusses our observations from these results. Section 5 discusses future work that could be conducted based upon our results, while Section 6 concludes the paper.

2 Background and Motivation

The existing HDFS implementation runs on IP networks, the de facto standard network-layer protocol for computer communications. In IP networks, traffic is routed from a particular source to a particular destination, whether unicast, multicast, or broadcast. This architecture was likely influenced by telephony networks, which route phone traffic between two endpoints. However, more recently, new network paradigms have appeared, including information-centric networking (ICN). While traditional networks like IP focus on delivering data between specific endpoints, information-centric networks instead 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 particular host. In our evaluations, we utilize the Named Data Networking (NDN) implementation of the ICN paradigm to model network-layer caching. In NDN networks, each piece of content has a unique name in a hierarchical namespace made up of "components" separated by slashes ('/'). For example, a piece of content representing the index page of the website of the University of Arizona Computer Science department could have the name "`/edu/arizona/cs/www/index.html`". Content is requested with an "Interest" packet from the client ("consumer"), which, if the content exists, will eventually reach a cache or a server (the latter known as a "producer") containing a piece of content with a name matching the Interest's name - this content will then be returned to the consumer in a "Data" packet. To accomplish this, Interests are routed based upon the name they contain and returning Data packets follow the reverse path of the Interest(s) they are satisfying. NDN names can also include specialized components to indicate segmented portions of a piece of content, as well as the version of that content, among other information [2].

However, while naming would be important for identifying blocks in an HDFS file system running on NDN, the availability of in-network caching at the network layer is more significant to our caching evaluation. Since content in NDN is identified by a hierarchical name, we can cache requested content on intermediate nodes to satisfy future requests if such a cache exists on the route taken by the Interest. This can allow requests to be satisfied before they reach the producer, likely decreasing the load on the producer and overall network, as well as the time it takes for an Interest to be satisfied for the consumer. Additionally, off-path caching mechanisms have been proposed for NDN, including one for wired Ethernet networks [3].

There has been previous work to adapt HDFS to run over an NDN network by Gibbens, et al [4]. They developed a prototype implementation that ran natively over NDN. However, their

implementation suffered from severe performance penalties compared to the official implementation of HDFS, which utilizes TCP over IP networks. However, this may be largely caused by the application-layer NDN forwarder they utilized, NFD [5], which is designed primarily for modularity and ease of extension [6], compared to the optimized and refined implementations of TCP built into modern operating systems.

3 Methodology

The traces we used in our evaluations were gathered in November 2017 on the Amazon Web Services (AWS) Elastic Map Reduce (EMR) system. Our trace generation clusters each consisted of 129 `m3.xlarge` nodes, with 1 NameNode and 128 compute nodes, running in the "US West (Oregon)" AWS region. Each of these nodes featured 4 virtual CPUs, 15 GiB of RAM, and 2 x 40 GB of storage. Spark ran with 5120 MiB of memory allocated to each executor and the Spark driver. Each executor was also allocated 4 CPU cores. Spark was set to partition jobs into 256 partitions. Additionally, dynamic allocation was disabled. The nodes in the cluster ran Amazon EMR release 5.10.0, which utilizes Apache Hadoop version 2.7.3 and Apache Spark version 2.2.0. Our cluster did not run Spark standalone, but instead utilized Hadoop YARN as the master.

On each node, HDFS was set to log in debug mode to allow us to obtain block-by-block traces. In addition, HDFS was set to use the legacy `RemoteBlockReader`.

On the clusters, we ran the HiBench [7] benchmark suite, version 7.0. Each benchmark in this suite can be split into a "prepare" and a "run" stage - the prepare stage generates the data to be processed across all compute nodes, while the run stage actually processes the data. In our evaluation, we only considered the run stage because a real system would presumably already have data to process, instead of needing to generate random data to process. We ran each benchmark on the "gigantic" data size.

The HiBench benchmarks that we were able to successfully run and gather a trace from included `aggregation`, `als`, `bayes`, `gbt`, `join`, `kmeans`, `linear`, `lr`, `rf`, `scan`, `sort`, and `wordcount`.

After running the above benchmarks, a trace was generated for each cluster by parsing the HDFS reads and writes in the DataNode log files. Each operation in the generated traces contains: (i) the time of the operation, (ii) the type of the operation (`READ` or `WRITE`), (iii) the ID of the block, (iv) the starting byte offset of the operation, (v) the size (in bytes) of the operation, (vi) the source node, and (vii) the destination node. These traces do not include the HDFS "generation stamp" of a block because we verified that every block in the trace was only associated with one generation stamp value. This value would change if the block had been modified. Therefore, a cached version of a block would still be applicable to future reads of that block in our evaluations.

We sorted our combined traces by time and used this to break the trace up into the prepare and run stages of each benchmark. Then, we filtered out all write operations, as well as all

local read operations (where the source and destination nodes were the same) from each run stage.

Caching evaluations were performed using two caching simulators. For the ARC [8], LIRS [9], and MQ [10] cache replacement policies, we utilized a simulator written in Java. For the LRU replacement policy, we utilized two simulators: the previously-mentioned Java simulator, as well as a simulator written in C++ for more intensive workloads (due to runtime considerations).

4 Evaluation

4.1 Unique Data Size

The unique data size is the number of unique bytes read, excluding overlapping operations on a byte-level granularity. For each benchmark, we calculated the unique data size of reads done over the network. "Over the network" in this case refers to reads where the data in question was not stored by the DataNode instance running on the same machine as the requesting compute client.

Benchmark	Unique Data Size (MiB)	Total Data Size (MiB)	Ratio (Uniq/Total)
aggregation	419	24918	0.01682306351
als	1345	25913	0.05189665802
bayes	367	25023	0.01464920543
gbt	523	25379	0.02061643429
join	202	24770	0.008156921264
kmeans	593	25330	0.02339635177
linear	21150	135952	0.155568045
lr	9091	44723	0.2032722353
rf	1282	25851	0.04960448447
scan	198	24766	0.008000000116
sort	938	25506	0.03677945751
wordcount	1045	26256	0.03980127935

Table 1: Optimal unique data size vs optimal total data size for reads done across then network

For each benchmark, we calculated both the unique data size and the total data size of network reads. The latter value is the combined number of bytes read by each read operation. We also calculated the unique data size divided by the total data size (the "ratio"). These results are presented in Table 1. We refer to the unique data size at a granularity of 1 byte as the "optimal unique data size" because HDFS operations are performed at a byte-level granularity.

However, NDN caching functions on a packet-level granularity, instead of a byte-level granularity. Therefore, we also calculated the unique data size of reads if HDFS blocks were segmented into cache blocks of various sizes. In a real implementation of HDFS over NDN, data would be fetched in units of cache blocks, with the requested data being extracted from

the retrieved blocks and combined before being sent to the requesting client application. Meanwhile, data in a retrieved cache block falling outside the specific boundaries requested by the client would be discarded. We calculated the starting cache block of each read by flooring the read starting offset divided by the cache block size. We calculated the ending cache block of each read by flooring the read ending offset (offset + size of the read) divided by the cache block size. These equations are represented below:

$$\text{Start}_{\text{CacheBlock}} = \lfloor \text{Offset}_{\text{ReadOperation}} / \text{Size}_{\text{CacheBlock}} \rfloor$$

$$\text{End}_{\text{CacheBlock}} = \lfloor (\text{Offset}_{\text{ReadOperation}} + \text{Size}_{\text{ReadOperation}}) / \text{Size}_{\text{CacheBlock}} \rfloor$$

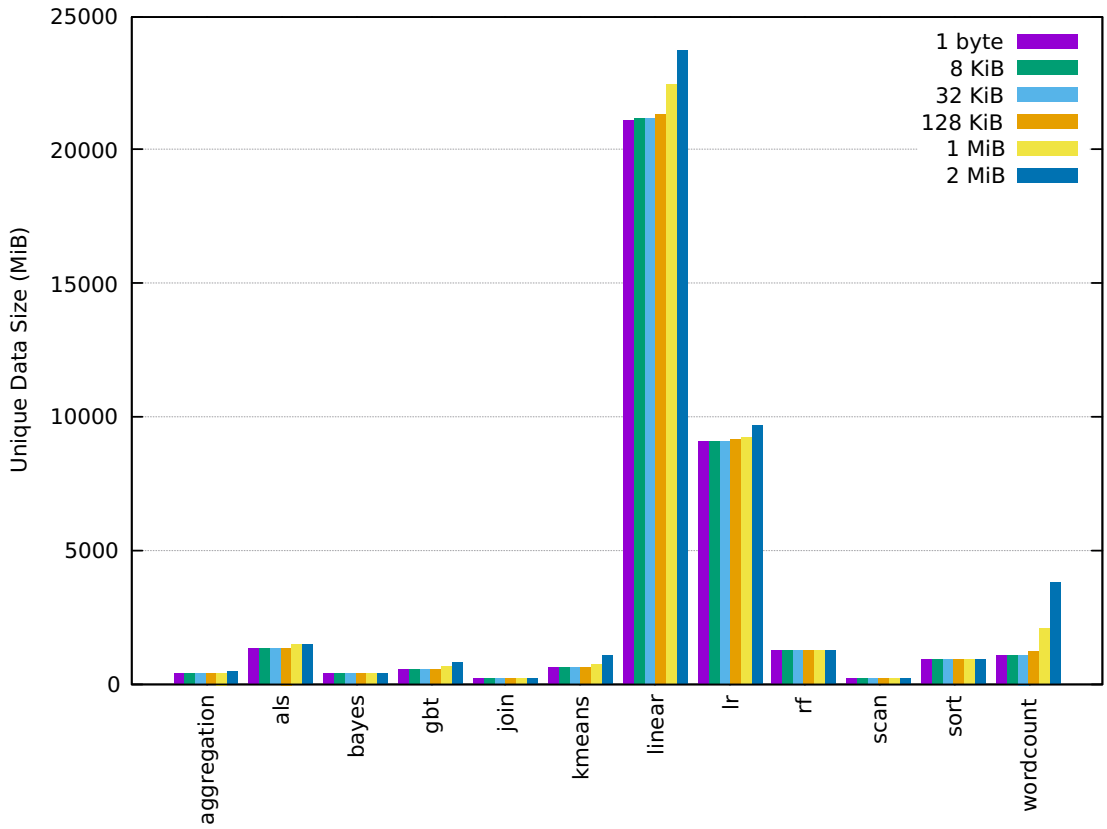

Figure 1: Unique data sizes for cache blocks of varying size (including Spark libraries)

We conducted evaluations of unique data size using cache block sizes of 8 KiB, 32 KiB, 128 KiB, 1 MiB, and 2 MiB. The unique data size of each benchmark for each of the above cache block sizes is presented in Figure 1. We also calculated the overhead caused by using cache blocks of each of the above sizes - this is presented in Figure 2. In this figure, the

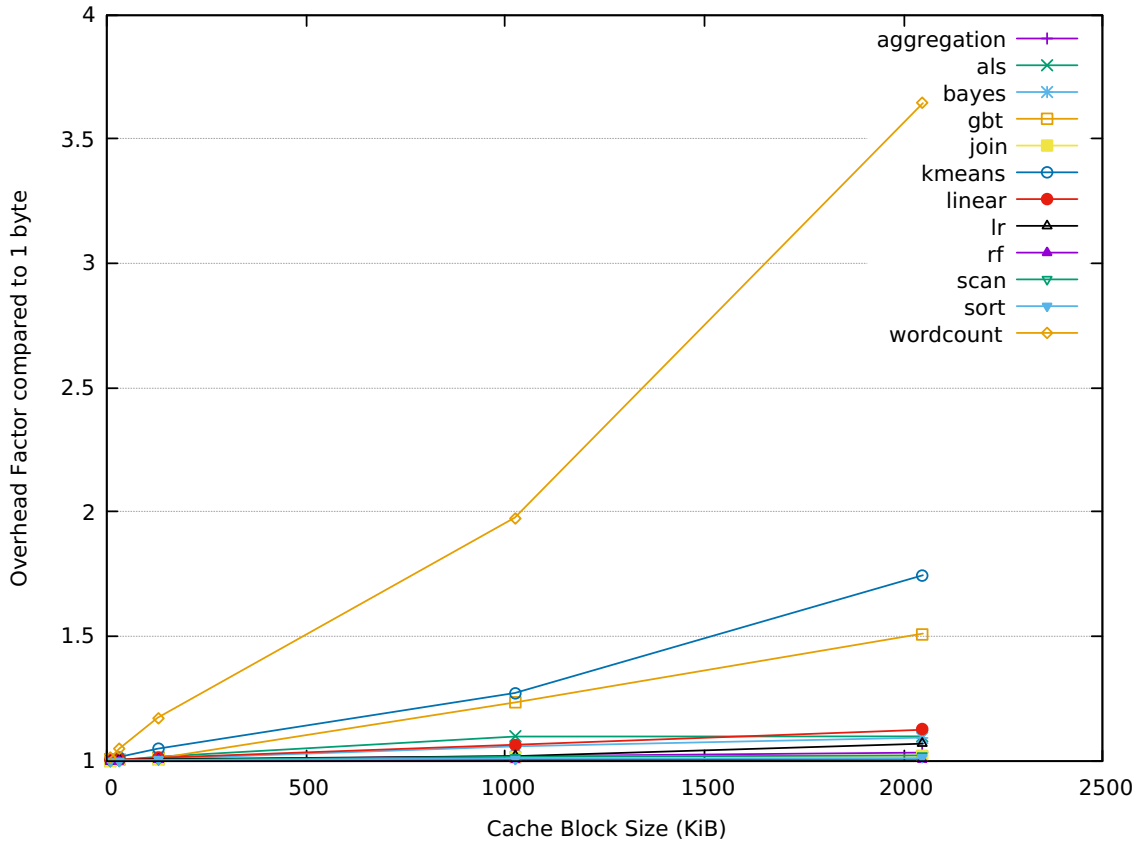

Figure 2: Overhead of each cache block size (compared to cache blocks of 1 byte)

"overhead factor" is computed as the unique data size at that cache block size divided by the optimal unique data size for that benchmark. An overhead factor greater than 1 indicates that unnecessary data ("overhead data") was read when satisfying HDFS read operations. As can be seen in the figures, increasing the cache block size increases the unique data size read across the network. However, the amount to which increasing the cache block size affected the unique data size varied by benchmark. With the largest cache block size (2 MiB), this ranged anywhere from 1.005997595 times the optimal unique data size for the `rf` benchmark to 3.643980699 times the optimal cache block size for the `wordcount` benchmark.

4.2 Characteristics of Read Data

When evaluating the caching potential of a system, it is important to determine what data is being read and how frequently. We developed a script to map HDFS block IDs to HDFS file names to allow us to determine which block(s) corresponded to a file. In all but one benchmark,

we found exactly three blocks that were read across the network by 125 of 128 compute nodes. Given a default replication factor of 3, it appears that the other three compute nodes read these blocks from their local DataNode instances. Two of these blocks correspond to a zip archive that appears to contain Spark libraries and one corresponds to a zip that appears to contain Spark configuration files. Reads of the first and second Spark library blocks are of sizes 129 MiB and approximately 69 MiB, respectively.[1] Meanwhile, the Spark configuration block reads range from approximately 76.4 KiB to approximately 76.6 KiB in size, depending upon the benchmark. It makes sense that these blocks would be read by every node, as they would be needed to run the Spark application. All nodes in all benchmarks read these blocks from the same starting and ending byte offsets. However, each benchmark used different block IDs for these blocks, making it difficult to cache these blocks across applications (provided they do contain the same data). Additionally, with the `aggregation` benchmark, two of the blocks (the second block in the Spark library zip file and the Spark configuration zip) were only read across the network by 124 compute nodes. Investigation of the full `aggregation` trace revealed that all three blocks were only requested by 127 read operations (including both network and local reads). All other traces showed 128 read operations for their equivalent blocks. It is uncertain why `aggregation` behaved differently, but it may have been caused by the failure of an application container.

The benchmarks varied in how many other blocks they read, as well as how many times these other blocks were read. We did not examine blocks read across the network only once, as these blocks do not have any caching potential (for obvious reasons). However, some benchmarks show interesting results. The `bayes` benchmark had a block from another Spark library zip archive, which was read across the network 119 times, with reads starting from various offsets and of sizes ranging from 645 KiB to approximately 69 MiB. `gbt` also read from additional Spark library zip files, but from 4 blocks instead of 1. These blocks were read across the network 76, 75, 7, and 6 times, with varying starting offsets and with sizes ranging from 645 KiB to approximately 115 MiB. All other blocks read more than once by benchmarks were located under directories named "`_temporary`" or featured "`_tmp`" at the end of their file names, indicating that they were temporary files. However, these temporary blocks were sometimes read from as many as 59 times in a single benchmark.

The scarcity of network reads of non-library and non-configuration blocks is likely caused by the framework attempting to place computation as close to input data as possible. This may be why the `scan` benchmark did not feature any reads for blocks located on remote DataNodes (apart from the three above-mentioned library and configuration blocks).

However, Spark contains options[2] to load Spark library JARs from a local source on the compute node, instead of over HDFS. Therefore, we also evaluated the unique data size for each benchmark with Spark library reads excluded from the trace. These values[3] are presented in Table 2. In addition, the ratios by which the unique and total data sizes were compared to

[1] In `aggregation`, the size of reads of the first block is 1 byte larger.

[2] `spark.yarn.jars` and `spark.yarn.archive`

[3] The unique data size of `scan` is actually 78285 bytes, but appears as 0 MiB due to rounding.

Benchmark	Unique Data Size (MiB)	Total Data Size (MiB)	Ratio (Uniq/Total)
aggregation	221	230	0.9601282823
als	1147	1156	0.991979303
bayes	85	94	0.9016089059
gbt	33	43	0.7818859845
join	4	13	0.3013469235
kmeans	395	573	0.6888746343
linear	20952	111195	0.1884235353
lr	8893	19966	0.4454015303
rf	1084	1094	0.9915230851
scan	0	9	0.008000102192
sort	740	749	0.9876453487
wordcount	847	1499	0.5651766778

Table 2: Optimal unique data size vs optimal total data size for reads done across the network (excluding Spark libraries)

the full network read traces presented in Table 1 are presented in Table 3.

As can be seen in Table 3, for most benchmarks, the optimal unique data sizes decreased by less than half from the full network read trace (ratios less than 1 indicate a reduction in unique or total data size[4]). However, some optimal unique data sizes decreased significantly, as exemplified by `scan`, which decreased to approximately 0.0004 times the original unique data size when the Spark library blocks were excluded. Additionally, total data sizes decreased by greater factors than unique data sizes for every benchmark. Some significant factors to which the total data size decreased are approximately 0.0005 for `join` and approximately 0.0004 for `scan`. For `join`, the only remaining network reads after the Spark library blocks were excluded were those for the Spark configuration archive and two blocks containing temporary files (with both of the latter reads having sizes around 2 MiB). Meanwhile, for `scan`, the only remaining network reads were those for the Spark configuration archive. However, some benchmarks show only an insignificant reduction in unique and total data sizes, namely `linear` and `lr`.

The unique data size calculations excluding Spark library blocks for cache blocks of various sizes are presented in Figure 3. As can be seen in the figure, with the removal of Spark library blocks decreased the unique data sizes of most benchmarks by approximately 198 to 200 MiB. The exceptions were `bayes` and `gbt`, where decreases were approximately 282 to 302 MiB and approximately 490 to 510 MiB, respectively. These additional decreases were caused by the removal of the additional Spark library blocks present in the traces of these two benchmarks.

[4]There is no way the unique or total data sizes of a benchmark could increase when blocks are excluded from its trace. Therefore, we have no values greater than or equal to 1.

Benchmark	Unique Data Size	Total Data Size
aggregation	0.5275400582	0.00924338973
als	0.8527244439	0.04461136308
bayes	0.2317416502	0.003765303356
gbt	0.06351730818	0.00167479714
join	0.01976243881	0.0005349338082
kmeans	0.6657983569	0.02261260873
linear	0.990635505	0.81789798
lr	0.9782139263	0.4464370189
rf	0.8455464602	0.04230148232
scan	0.000376811658	0.0003768068501
sort	0.7888772374	0.02937742467
wordcount	0.8104735787	0.0570757545

Table 3: Ratios of the optimal unique and total data sizes of all reads done over the network compared to network reads with Spark library blocks excluded

4.3 Caching Potential

For each of the benchmarks above, we evaluated the caching potential using the two caching simulators described previously. Simulations were run for every combination of replacement policy (ARC, LIRS, LRU, and MQ) and cache block size (8 KiB, 32 KiB, 128 KiB, 1 MiB, and 2 MiB). A cache size of 1 GiB was chosen, as a cache of this size would be reasonable on modern computer systems, including routers.

We found that, for most benchmarks, the cache replacement policy did not make any difference in the hit ratio seen with a given cache block size, even with Spark library blocks included. The exceptions to this were `linear`, `lr`, and `wordcount`. The hit ratios for all of the benchmarks where the replacement policy did not change the hit ratio, including and excluding Spark library blocks, can be found in Figures 4 and 5, respectively. Meanwhile, the hit ratios for `linear` can be found in Figures 6 and 7, `lr` in Figures 8 and 9, and `wordcount` in Figures 10 and 11.

As can be seen in the Figures 4 and 5, for all but three benchmarks (when Spark libraries were both included and excluded), the choice of cache replacement policy did not change the observed hit ratio. Except for `als`, `kmeans`, and `rf`, this was because the unique data size was less than 1 GiB, meaning that no blocks needed to be evicted from the cache, making the replacement policy in use irrelevant.

However, because data is cached in units of cache blocks, we must also consider the effect that cache block size has on hit ratio. When Spark library blocks were included in the trace, two benchmarks experienced a hit ratio increase as cache block size increased (`rf` and `sort`), four experienced a hit ratio decrease (`aggregation`, `bayes`, `gbt`, and `kmeans`), two experienced a mixture of hit ratio increases and decreases (`als` and `join`), and one had hit ratios remain

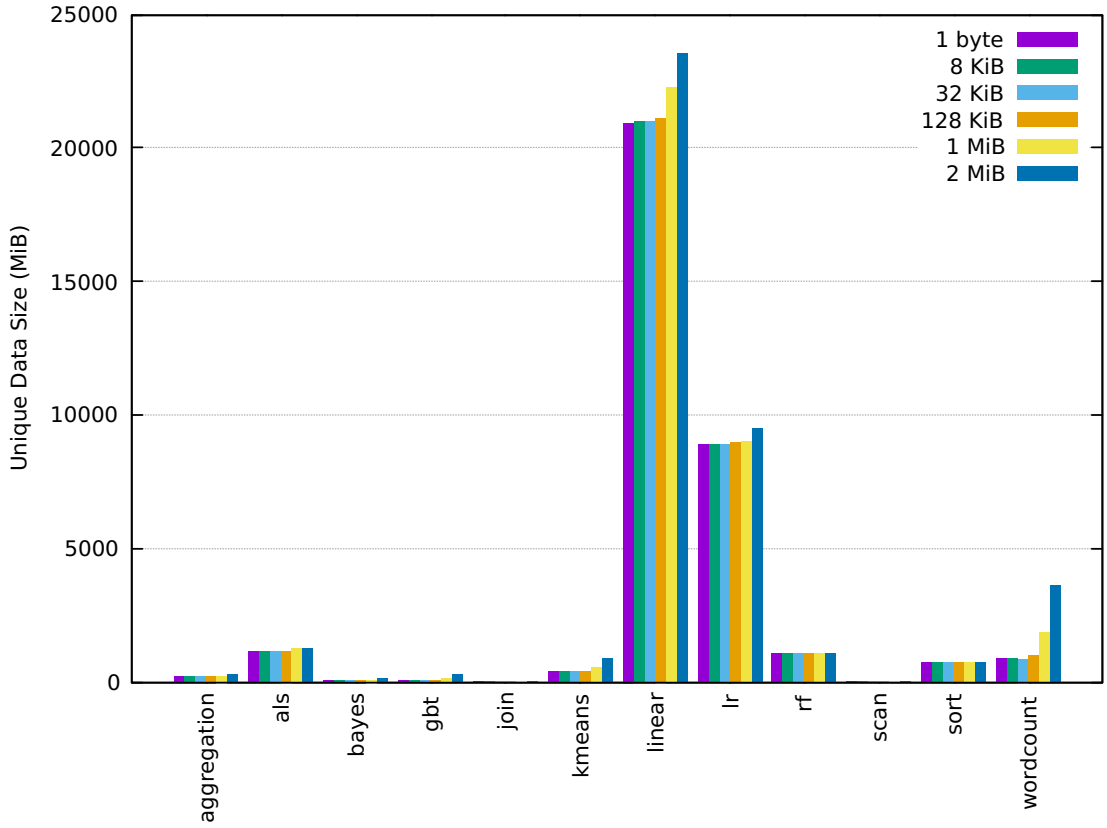

Figure 3: Unique data sizes for cache blocks of varying size (excluding Spark libraries)

constant (`scan`). However, none of the cache block sizes were affected significantly by changing the cache block size, with the benchmark with the greatest range of hit ratios, `kmeans`, having a range of approximately 0.015.

When Spark library blocks were excluded from the traces, all of the above nine benchmarks except `scan` and `gbt` experienced an increase in observed hit ratio as block size increased. Instead, the hit ratios for `scan` remained constant and the hit ratios for `gbt` increased on all intervals except from 1 MiB to 2 MiB, where the hit ratio remained constant. The reason that hit ratios increased on almost all intervals for almost all benchmarks is because, as cache block sizes increase, reads with larger sizes have a lesser impact on the hit ratio compared to reads with sizes smaller than the cache block size, as reads smaller than the cache block size are "rounded up" in size so that they are considered to have the same size as the cache block size. This gives the Spark configuration block, which generates very high hit ratios, a greater impact on the hit ratio. For example, in the `join` benchmark, the only reads for non-configuration blocks are less than 2 MiB. However, they have the same effect on the hit ratio

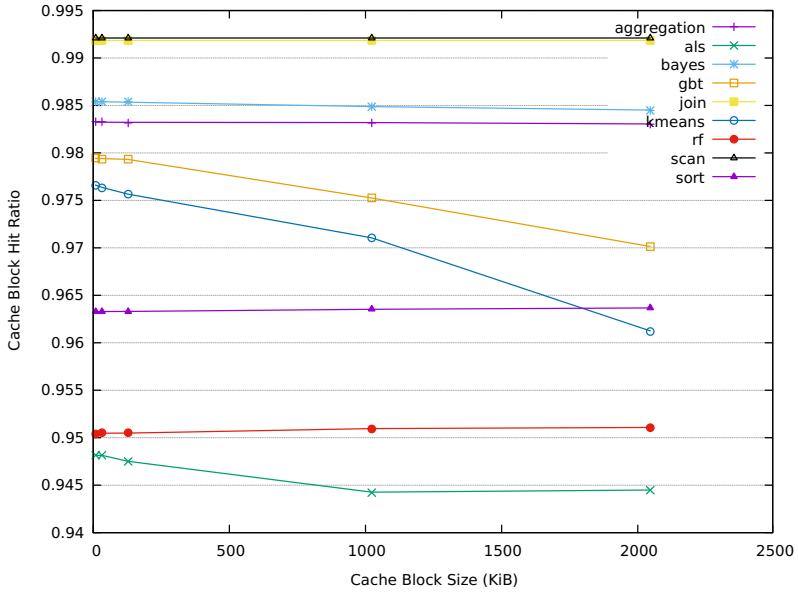

Figure 4: Hit ratios (including Spark libraries) for benchmarks where replacement policy did not make a difference in hit ratio

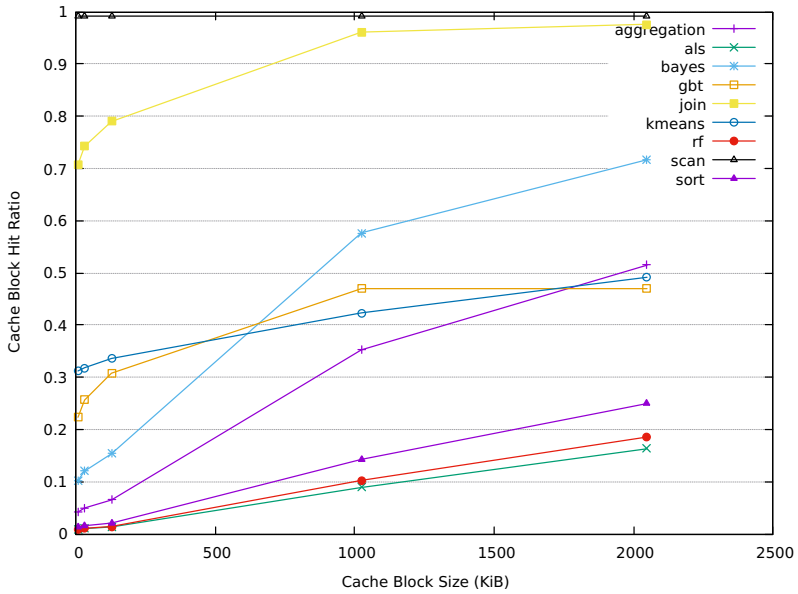

Figure 5: Hit ratios (excluding Spark libraries) for benchmarks where replacement policy did not make a difference in hit ratio

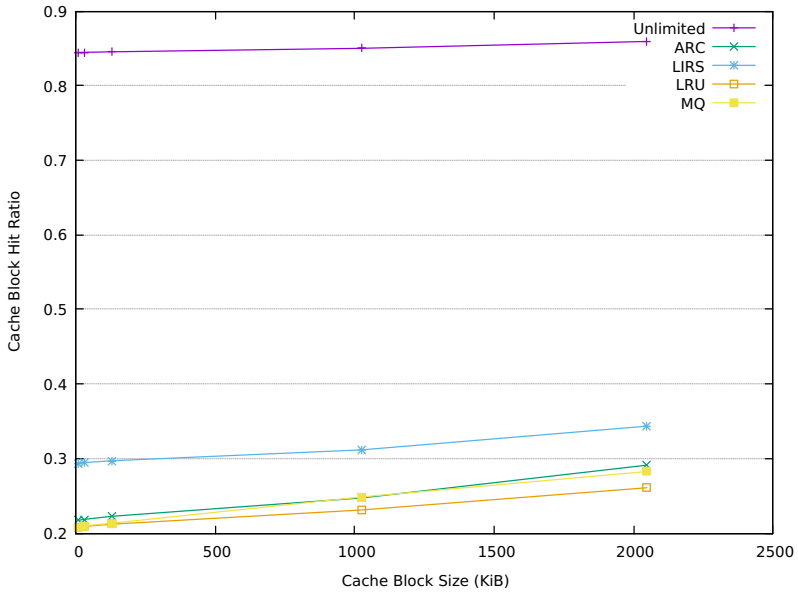

Figure 6: Hit ratios (including Spark libraries) for the `linear` benchmark

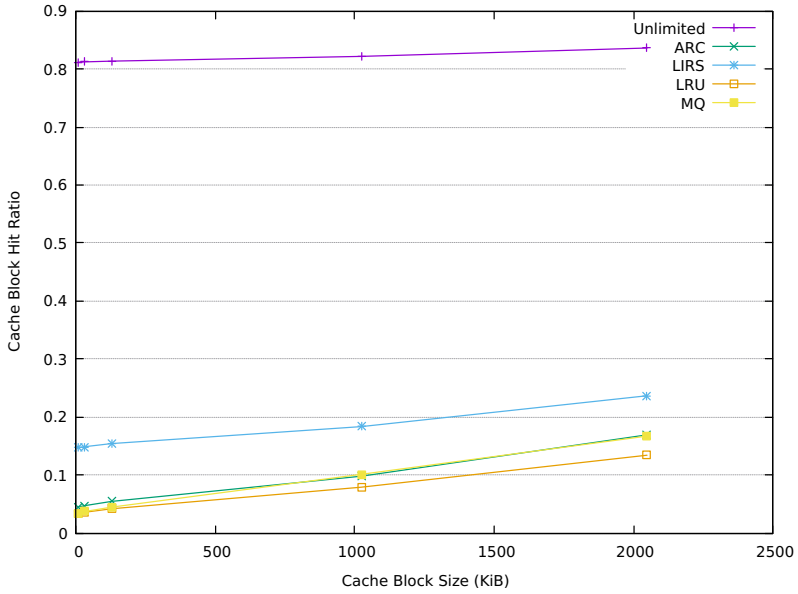

Figure 7: Hit ratios (excluding Spark libraries) for the `linear` benchmark

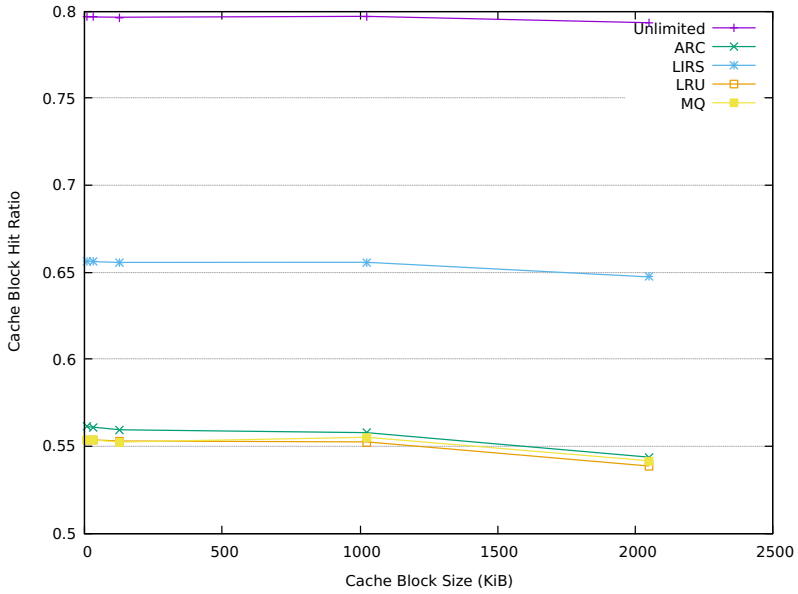

Figure 8: Hit ratios (including Spark libraries) for the `lr` benchmark

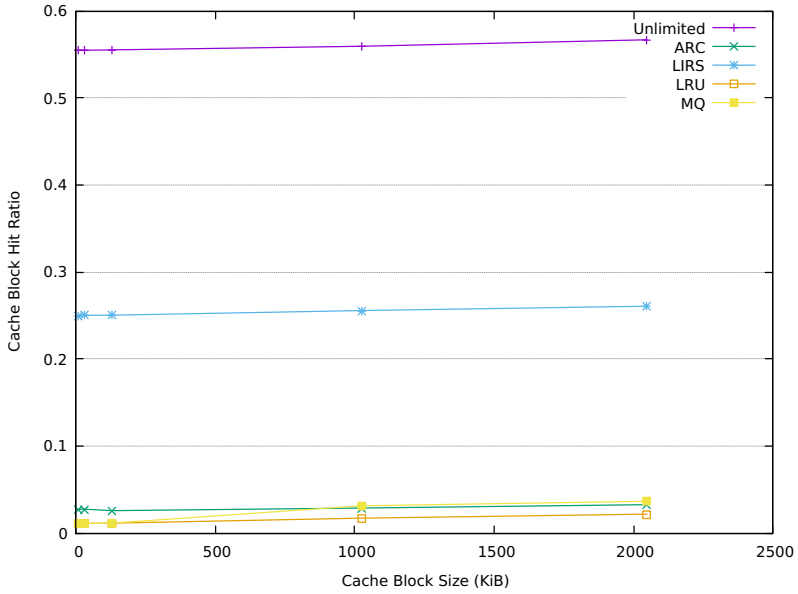

Figure 9: Hit ratios (excluding Spark libraries) for the `lr` benchmark

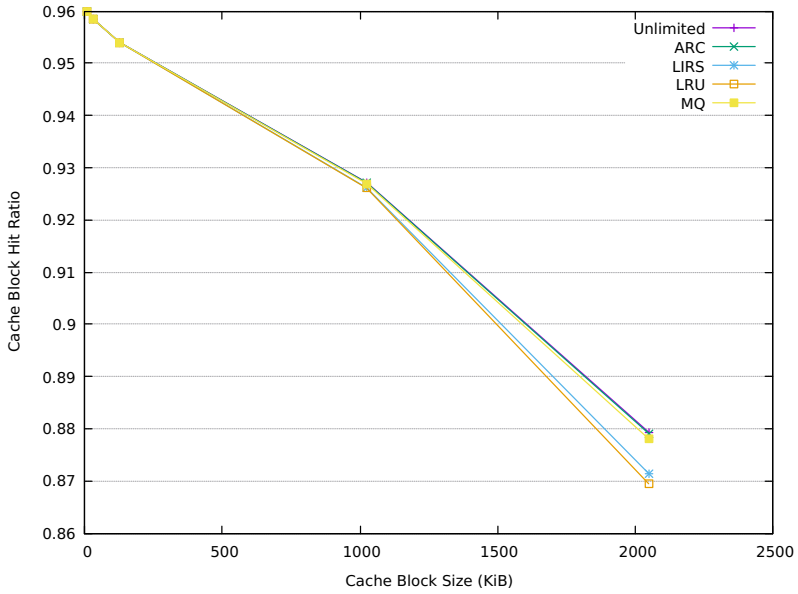

Figure 10: Hit ratios (including Spark libraries) for the `wordcount` benchmark

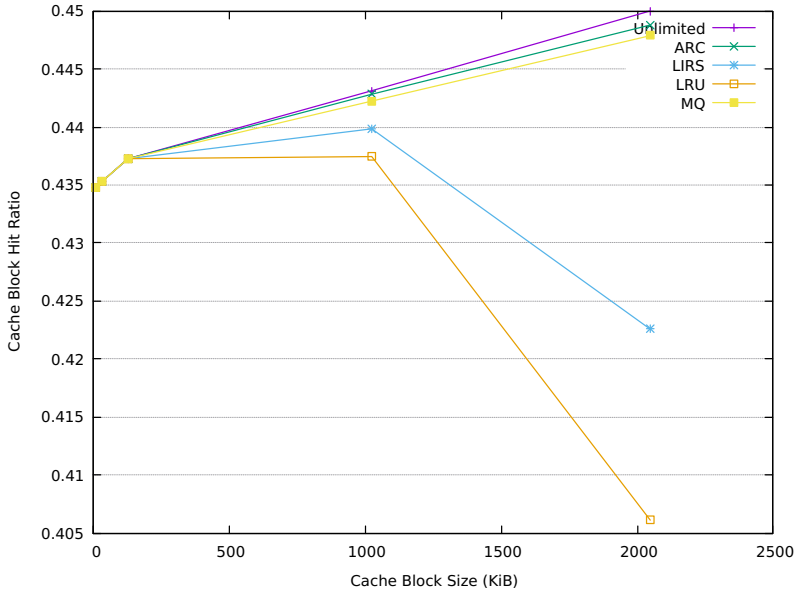

Figure 11: Hit ratios (excluding Spark libraries) for the `wordcount` benchmark

as the approximately 76 KiB Spark configuration block.

Hit ratios were significantly higher when Spark library blocks were included because almost all compute nodes request the same regions of these blocks over the network, which has a positive effect on the hit ratio. The only exception to this was `scan`, which stayed the same because the remaining Spark configuration block has the same caching properties as the Spark library blocks.

For `linear` and `lr`, LIRS performed notably better than any other cache replacement policy (Figures 6, 8, 7, and 9). Meanwhile, the remaining three replacement policies achieved lower hit ratios than LIRS, but did not differ significantly from each other. For `linear`, generally ARC performed better than MQ, which always performed better than LRU (the exception to the former occurred with cache blocks of 1 MiB, where MQ performed better than ARC). For `lr` including Spark library blocks, whether LRU or MQ performed better depended upon the cache block size (both policies performed the same with 8 KiB cache blocks), while ARC always performed the best of the three. For `lr` excluding Spark library blocks, LRU and MQ similarly change which performed better based upon the cache block size, also performing the same with 8 KiB cache blocks. However, while ARC performed the best of the three with cache blocks of sizes 8 KiB, 32 KiB, and 128 KiB, MQ performed better than ARC with cache blocks of sizes 1 MiB and 2 MiB. Interestingly, when Spark libraries were excluded from the `lr` trace, LIRS was the only policy that achieved hit ratios higher than 0.04.

As the cache block size increased for `linear`, the hit ratio increased (both with and without Spark library blocks). Meanwhile, `lr` experienced a mostly decreasing hit ratio (with some increases) when Spark library blocks were included, but experienced increasing hit ratios when Spark library blocks were excluded from the trace (with the notable exception of the interval between 32 KiB and 128 KiB, where the hit ratio decreased). This is likely due to similar reasons as the nine benchmarks above.

The `wordcount` benchmark behaved differently (Figures 10 and 11). For cache blocks up to and including 128 KiB, we observed that the replacement policy in use did not make a difference in the observed hit ratio and presented the same hit ratios as the unlimited cache. For both 1 MiB and 2 MiB cache blocks, the replacement policies performing best to worst for this benchmark were ARC, MQ, LIRS, and LRU, although the hit ratios did not differ by more than 0.05. This was the case regardless of whether Spark library blocks were included in the trace. When Spark library blocks were included in the trace, `wordcount` experienced a decrease in hit ratio as cache block size increased for all replacement policies. Meanwhile, when Spark library blocks were excluded, almost all replacement policies experienced an increase in hit ratio as cache block size increased (with the exception of a decrease on the interval from 1 MiB to 2 MiB for LRU and LIRS). The reasons for this are unclear at the moment and are left to future work.

5 Future Work

In this paper, our evaluations were conducted on traces obtained from benchmark applications. Since the input data to these applications is synthetic and is purposefully pre-distributed across DataNodes, it may not accurately represent real workloads, which may have input data spread across DataNodes in a much less balanced manner. Therefore, in the future, we hope to conduct caching evaluations based upon traces taken from HDFS clusters used for research or commercial applications.

Additionally, we would like to perform evaluations of caching hit ratios using more realistic topologies with multiple possible caching locations (in NDN networks, each NDN-capable router). In this paper, our evaluations were only performed on a theoretical topology with one router that directly connected to every end host. One topology we would like to perform evaluations on is the fat-tree topology [11].

We only conducted our evaluations using the ARC, LIRS, LRU, and MQ cache replacement policies. However, this is only a subnet of the many replacement policies in existence, including LRFU [12], LRU-K [13], and 2Q [14]. We would like to increase the breadth of our evaluations in case there are some replacement policies that display better hit ratios than the ones we evaluated in this paper.

6 Conclusion

In this paper, we evaluated the potential benefits of caching in the Hadoop Distributed File System (HDFS). This file system is used to support various big data platforms, including Apache Hadoop and Apache Spark. We utilized virtualized Apache Spark clusters running the Intel HiBench benchmark suite to conduct our evaluations. Through our evaluations, we obtained hit ratios for read operations utilizing four cache replacement policies (ARC, LIRS, LRU, and MQ) as we varied the cache block size. This was also compared with the hit ratios seen with an unlimited cache. The caching model we utilized was intended to be analogous to that utilized in the Named Data Networking ICN architecture.

Through our evaluations, we discovered that, for most of the applications we utilized, Spark does not send very many read requests over the network. In fact, for most benchmarks, we found that most of the data read from the network consisted of the Spark libraries and configuration files. This is likely because the Spark framework is effective at scheduling jobs close to the location of input data. We did not evaluate HDFS writes because these could easily be made more efficient through the use of multicast, since the multiple duplicate data streams composing an HDFS write operation occur within a very short time frame.

However, since Spark libraries can be pre-loaded on compute nodes, instead of being read from the network at the start of each application's run, and Spark libraries made up most of the network read traffic in most of the applications we evaluated, caching appears to be of limited effectiveness in most of these applications. However, two applications (`linear` and `lr`)

generated enough traffic to justify caching. For these applications, we found that the most effective cache replacement policy was LIRS, since it produced significantly greater hit ratios than any other replacement policy we evaluated. For these applications, we conducted an evaluation of the effect of cache block size on hit ratio. We found that cache block size made some difference in the hit ratio, increasing it in some cases, but decreasing it in others. However, overall its effect on observed hit ratios was not very significant, except in `wordcount` including Spark library blocks, where the hit ratio decreased by around 0.09 between 8 KiB and 2 MiB cache blocks. The maximum packet size in existing NDN implementations is 8800 bytes [15], so we foresee 8 KiB as the cache block size most likely to be used in a real implementation of HDFS over NDN.

References

[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, July 2014.

[2] Y. Yu, A. Afanasyev, Z. Zhu, and L. Zhang, "Ndn technical memo: Naming conventions," NDN, Tech. Rep. NDN-0022, July 2014.

[3] J. Shi, E. Newberry, and B. Zhang, "On broadcast-based self-learning in named data networking," in *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, June 2017, pp. 1–9.

[4] M. Gibbens, C. Gniady, L. Ye, and B. Zhang, "Hadoop on named data networking: experience and results," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 1, p. 2, June 2017.

[5] Named Data Networking Project Team, "Named data networking forwarding daemon." [Online]. Available: https://github.com/named-data/NFD

[6] ——, "Nfd readme." [Online]. Available: https://github.com/named-data/NFD/blob/master/README.md

[7] Intel Corporation, "Hibench." [Online]. Available: https://github.com/intel-hadoop/HiBench

[8] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, no. 2003, 2003, pp. 115–130.

[9] S. Jiang and X. Zhang, "Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 31–42, 2002.

[10] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 91–104.

[11] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[12] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[13] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '93. New York, NY, USA: ACM, 1993, pp. 297–306. [Online]. Available: http://doi.acm.org/10.1145/170035.170081

[14] T. Johnson and D. E. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, 1994, pp. 439–450. [Online]. Available: http://www.vldb.org/conf/1994/P439.PDF

[15] Named Data Networking Project Team, "Ndn tlv packet constants." [Online]. Available: https://github.com/named-data/ndn-cxx/blob/master/src/encoding/tlv.hpp