Datanode Optimization in Distributed Storage Systems

Xiaokang Fan, Shanshan Li, Xiangke Liao, Lei Wang, Chenlin Huang, Jun Ma
School of Computer Science and Technology
National University of Defense Technology
Changsha, Hunan, P.R.China

{fanxiaokang, shanshanli, xkliao, wanglei, huangchenlin, majun}@nudt.edu.cn

***Abstract*—Distributed storage systems designed for small files have been developing rapidly, like Facebook's hayStack, Twitter's Cassandra and so on. But, under our observation, there are still some drawbacks in these systems. For example, they do not have cache specified for files and have not taken the relationship inherent in application-specific knowledge between files into consideration. We propose a file-level cache on datanode and co-location of affinitive files based on application-specific knowledge. We use a synthetic data set and a real world trace to evaluate our optimization. The file-level cache and co-location of affinitive files together can improve system's throughput by 20%-50%.**

Keywords-distributed storage system; file-level cache; co-location.

I. INTRODUCTION

Distributed storage systems have been widely used in large datacenters. These systems are designed to provide efficient, reliable access of data using clusters of commodity hardware [22]. So far, applications specified for small files have been increasing rapidly. For example, micro blogging, facebook, twitter, and so on. These applications generate enormous amounts of small files to store in the storage systems. For example, Facebook have stored over 260 billion images and more than 20 petabytes of data so far [11]. Many systems have been developed to support these applications, like fastDFS [18], Facebook's Haystack [11], and so on.

Under our observation, these systems are not perfect. As we know, most distributed systems are deployed as userspace libraries on large clusters of commodity machines. Each node in the cluster has local operating system and file system. Local system will load popular data into its cache. Usually cache unit is block. But in systems specific for small files, block may not be an appropriate choice as for the cache unit. Since a block may contain many files and quite often only a small part of them are frequently accessed. In fact, cache space has not been made fully use of. In distributed storage systems, files are usually randomly distributed in the whole system for load balancing. Little attention has been paid on file's inner relationship when files are written into disk. In web applications, when an image file is accessed, the images that make up the same hypertext document will also be accessed. The relationship between these related files have not been made use of.

In order to avoid these two drawbacks, we have proposed two optimizations on datanode: first, we build up a file-level cache which can make full use of the cache space. Second, we propose co-location of related files that store related files close to each other on disk which can take advantage of the disk technology trend that is toward improved sequential bandwidth [28]. In our evaluation, we find that with only file-level cache, we can improve the system's throughput by maximally 40%. With only co-location of related files, we can improve the throughput by maximally 20%. With both file-level cache and co-location of related files, we can improve the throughput by maximally 50%.

The rest of this article is organized as follows: Section 2 reviews the related works. Section 3 provides detailed motivation for our optimization. Section 4 describes our file-level cache in detail and Section 5 explains our co-location strategy of related files in detail. Section 6 describes our implementation on TFS [27]. We evaluate our optimization in Section 7 and draw a conclusion in Section 8.

II. RELATED WORK

With the rapid development of data-intensive applications, traditional file systems could no longer meet the demand for mass data storage. Many distributed storage systems have been developed to support applications with enormous amounts of data. For example, Amazon has designed Dynamo [8] to power parts of Amazon Web Services. Google has developed GFS [9] for its core data storage and usage needs.

Some peer-to-peer (P2P) systems have also looked at the problem of data storage and distribution [10, 12]. But, they are generally used as file sharing systems. Distributing data for performance, availability and durability has been widely studied in the file system and database system community. Compared with P2P storage systems that only support flat namespaces [29], distributed file systems typically support hierarchical namespaces [8, 24, 25, 26].

 ISBN: 978-1-61208-216-5

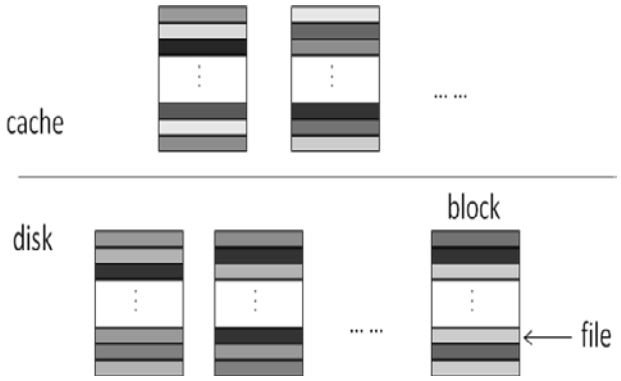

Figure 1. Cache organized in the unit of block

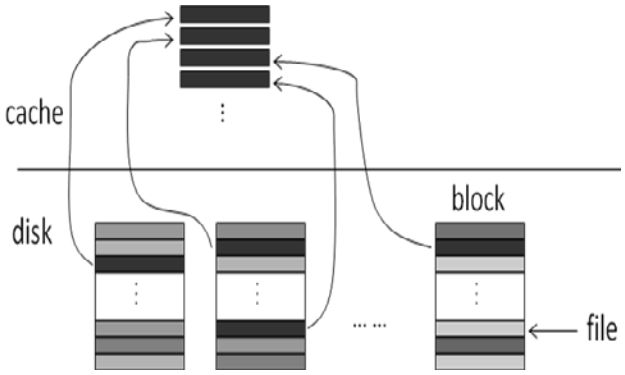

Figure 2. Cache organized in the unit of file

Studies of distributed file systems specified for small files have become a key component of storage systems research as applications specific for small files like images and micro bloggings have been increasing rapidly [21, 31, 32]. FastDFS is designed to meet the requirement of the website whose service based on files such as photo sharing site and video sharing site. Facebook has designed Haystack to serve its Photos application where data is written once, read often, never modified, and rarely deleted.

To increase the efficiency of the enormous small disk requests that characterize accesses to small files, much work have been done on the disk layout of small files. Localizing logically related objects is the choice of many file systems. Some researchers have investigated the value of breaking file system's disk storage into cylinders and moving the most popular data to the centermost cylinders in order to reduce disk seek distances [1, 2, 3]. From the same perspective of reducing disk seek distances, immediate files, an idea proposed by [4], moves inode to the first block of the file. This approach puts inodes together with their file data, which can improve the performance of read operations that read file data. Several other works [4, 5, 7] have proposed how to group related files together intelligently.

III. PRELIMINARY

Modern distributed file systems' topology can usually be divided into two kinds: master-slave structure and ring structure. Systems like GFS, HDFS [17], fastDFS usually organize machines in master-slave structure. In these systems there are two types of machines: one namenode with a large number of datanodes. Namenode handles metadata while datanode handles real data. This structure frees namenode from the data flow, so it can significantly reduce workload on namenode. Systems of the ring structure are often decentralized systems. There is no masternode, which only deals with metadata, in these systems. All machines act as the same role. All nodes can be called datanodes.

The growth and diffusion of applications specific for small files have led to systems that support efficient, secure and durable access of small files. Systems of the ring structure, like Dynamo and Cassandra [19], are intended to store relatively small objects. In systems of the master-slave structure, TFS is developed by Taobao [6] to store its enormous amounts of online commodity images.

As one part of the whole system, datanode plays a very important role. But so far there is hardly any optimization specified for datanode.

As we know, most distributed storage systems are not implemented in the kernel of operating systems, but are instead provided as userspace libraries, which means that they are all based on local file systems [20]. In systems that store small files, local file systems can hardly have any sense of single files; so they usually organize cache in the unit of block. Studies show that in most web service applications the file access pattern applies the Pareto principle, which means that only a small part of all files are frequently accessed while most files are rarely accessed [13, 14, 15, 16]. Since files are evenly distributed in the whole system, it is still true that of all the files that consist a block, only a small part are frequently accessed while others are rarely accessed. This strategy may result in low efficiency of cache and the waste of cache space.

Fig. 1 illustrates the case how cache space is wasted. Darker files are more frequently accessed while lighter files are less frequently accessed. When one block is loaded into cache as some popular files in it are accessed, files in the same block that are rarely accessed will also be loaded into cache. It stands a good chance that those unpopular files will have never been accessed before this block is replaced by other blocks. Cache space occupied by those unpopular files is wasted.

Modern distributed systems usually distribute files randomly for load balancing [30]. But this has not taken the logical relationship of files into consideration. In most cases, files can be partitioned into small groups based on application-specific knowledge. Files in the same group are closely related with each other that if one file is accessed most probably the others will be accessed too. For example, in online business like Amazon, all images stored in the system can be partitioned based on the commodities they describe. Once a commodity is skimmed by some customer, all images that describe this commodity will be accessed.

IV. FILE-LEVEL CACHE

Since cache space in datanode has not been fully made use of and much of it has been wasted. In order to make better use of cache space, we proposed a file-level cache on datanode. Fig. 2 illustrates the basic idea of our file-level

 ISBN: 978-1-61208-216-5

cache. We organize cache in the unit of single files rather than blocks. Each time we load a single file into cache rather than a block.

Studies of cache replacement strategy have been a topic for many years. Many replacement algorithms have been proposed, such as FIFO [35], LRU [36], OPT [37] and so on. FIFO is too simple to make the most of cache while OPT is just an ideal model, which has not been put into practice. The LRU strategy discards the least recently used items when new items need to be loaded into cache. This strategy fits our applications well so we choose LRU as our replacement strategy. Since files are not in the same size, we organize files in the cache as a list. As enormous accesses to small files may generate great amounts of cache misses while all these cache misses would search the entire list, which results in great amounts of searching time. In order to avoid these unnecessary searches of the list generated by cache misses, we introduce a bloom filter.

For an incoming read request, our algorithm does the following:

- Check the bloom filter to see whether this file is in the cache. If the bloom filter indicates that it is not in the cache, load this file from disk and add a node that represents this file to the head of the list. Reply with the data loaded from disk.
- If bloom filter indicates that this file may be in the cache, search the list to see whether this file is in the cache.
- If it is in the cache, move the node that represents this file to the head of the list and reply with data in the cache.
- If it is not in the cache, load this file from disk and add a node that represents this file to the head of the list. Reply with the data loaded from disk.

Each time we load a new file into the cache, if the total size of all the data in the cache is larger than the cache size, we recursively remove the node in the tail until the total size is smaller than the cache size.

Compared with the time of loading files from disk, the time of searching file-level cache can be ignored. With bloom filter avoiding most of the unnecessary searches of cache. We believe that our file-level cache can greatly reduce the overall file access time.

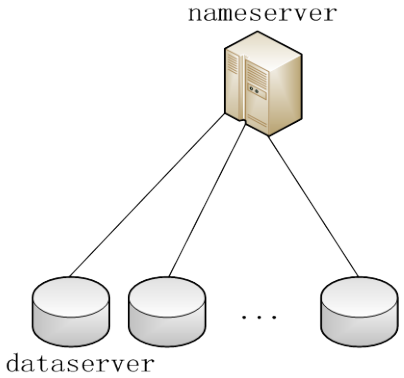

Figure 3. Stucture of TFS.

V. CO-LOCATE FILES BASED ON AFFINITY

We borrowed a concept of affinity to describe the inner relationship between files based on application-specific knowledge. We say several files are affinitive in case that if one of them is accessed, the rest are very likely to be accessed. As modern disk technology trend is toward improved sequential bandwidth. To better exploit bulk data bandwidth and avoid frequent reposition to new locations, we use co-location to place affinitive files at adjacent disk locations.

In modern distributed storage systems, a write process usually consists of the following steps:

- System randomly chooses a datanode based on the current workload on every datanode.
- Client contacts and sends data to the chosen datanode directly.
- After all data have been received, datanode commits.

Each write request goes through the entire write process. Since datanode is chosen randomly, files are distributed randomly in the whole system.

In order to co-locate affinitive files, we do not write disk for single write request, instead we collect files in a buffer and write them to disk in batches. Each time the client receives a write request, it puts the file in the buffer. Files in the buffer are divided into groups based on application-specific knowledge. When buffer is full or a time limit is met, system begins the write process.

In our approach, we have made some modifications to modern write process to achieve co-location of affinitive files. The write process consists of the following steps:

- System randomly chooses a datanode for the first group of files in the buffer based on current workload on every datanode.
- Client contacts and sends data of files in the first group to the chosen datanode directly.
- After all data have been received, datanode commits.
- Check whether the buffer is empty. If the buffer is not empty, go to the first step; otherwise ends the write process.

We believe that co-locating files based on the relationship inherent in application-specific knowledge can be exploited to successfully realize the performance potential of modern disks' bandwidth.

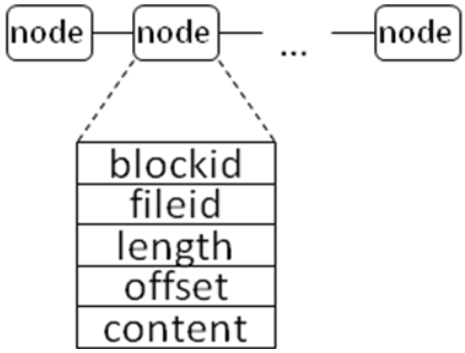

Figure 4. Structure of cache

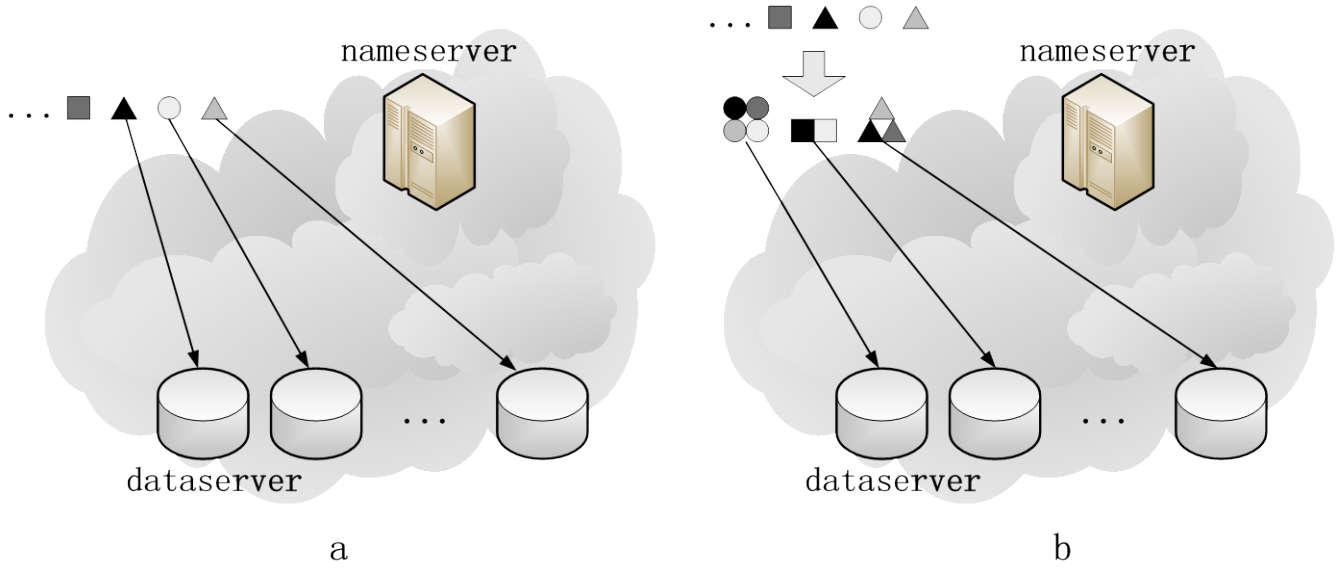

Figure 5. Write process.

VI. IMPLEMENTATION

This section describes our detailed implementation on TFS. TFS is the abbreviation of Taobao File System. It is a distributed file system and is designed to store great amounts of small images. The average file size is 16.7 kilobytes. Fig. 3 depicts the basic structure of TFS. Like GFS, a TFS cluster consists of multiple nodes which can be divided into two types: one nameserver and a large number of dataservers. Files are organized as blocks which are usually 64 megabytes in size. Dataserver stores blocks while nameserver maintains the logical mappings of blocks to dataservers. TFS also makes some optimizations on the filename to reduce metadata stored on nameserver. For each file, TFS encodes the id of the block that contains the file to the file's filename. So when a client gets the filename, it can get the id of the block that contains the file by decoding the filename. Each block is replicated several times throughout the network.

A read process in TFS goes like this: client sends a read request to nameserver. Nameserver replys with the cooresponding location (i.e., the dataserver). The client contacts the corresponding dataserver. Dataserver replies with the file's content.

A write process in TFS goes like this: client sends a write request to nameserver. Nameserver chooses a dataserver based on the current workload on every dataserver and replys with the dataserver. Client sends data to dataserver. After all data have been received, dataserver commits to nameserver. Dataserver replys to client that write process completes.

In our file-level cache implemented on datanode, files are organized as a list. As Fig. 4 shows, each node in the list represents a file which records the file's blockid, fileid, file length, offset in the block and the file's content. We use a standard bloom filter in our implementation. Each bit in the array is set to 0 when service starts. In order to improve our bloom filter's accuracy, i.e., to make the false positive [33] rate as low as possible, we use three hash functions [34].

Fig. 5a shows the writing mechanism of TFS: a random dataserver is chosen to store the first file in the write request queue. Fig. 5b shows the writing mechanism in our approach: files are stored in the buffer and grouped according to which hypertext document they make up. Then a dataserver will be allocated for each group of files. Files in the same group will be written into the same dataserver and the storage mechanism in dataserver will guarantee that these files will be placed in adjacent locations on disk. In our implementation, since file size is relatively small, we believe that 4 megabytes is enough for the buffer size and our experiments have proved that this is an appropriate choice.

VII. EVALUATION

This section reports measurements of our implementation on TFS, which shows that it can dramatically improve the system's performance.

Our TFS cluster contains 22 nodes, with 2 of them act as nameservers and the rest act as dataservers. Each node runs 64-bit ubuntu10.04 and uses an Intel Core 2 Duo E6850 3GHz CPU, 4GB RAM, and a 500GB 7200rpm hard disk.

We use two main data sets for testing. Our synthetic data set simulates the trace in applications in which files show a certain clustering effect, which means that each file may have a strong relationship with several other files. So we can use this characteristic to co-locate files that have strong relationships with each other. Files' average size is 20 kilobytes and total size is 2 terabytes. Our second data set is a real world data set which comes from our college's online teaching system. Teachers use this system to share slides with students, while students use this system to submit homework. This system is also used as a communication platform for students and teachers to discuss with each other. So a great number of small files are stored in this system.

 ISBN: 978-1-61208-216-5

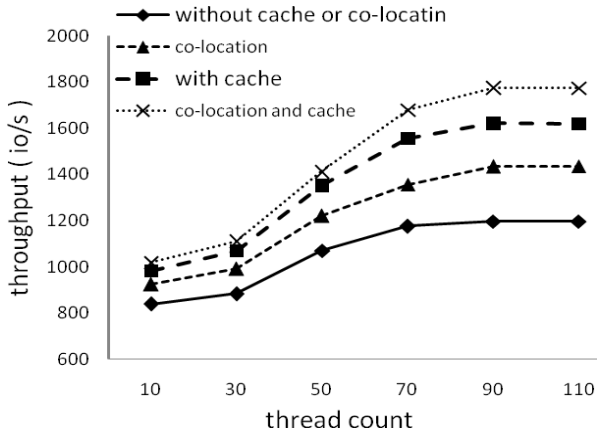

Figure 6. Read throughput under different numbers of thread.

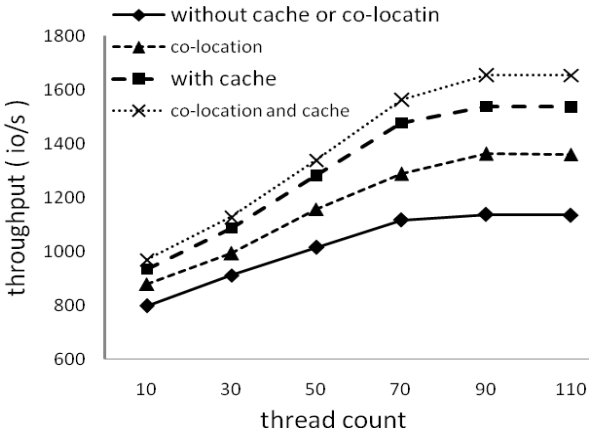

Figure 8. Throughput under different numbers of thread.

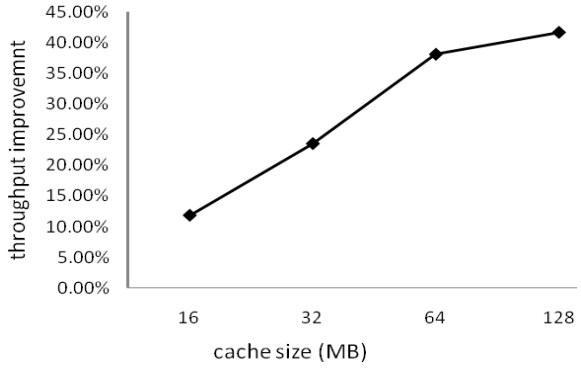

Figure 7. Throughput improvement of different cache size.

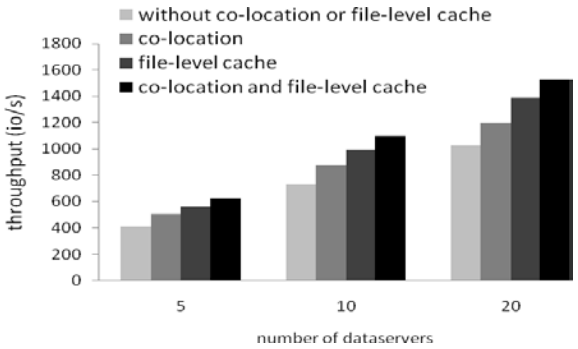

Figure 9. Throughput of different numbers of datanode.

Since the disk throughput is hard to evaluate and is not suitable for the evaluation of the whole system, we use the I/O throughput of the whole system, i.e., the read/write requests complete per second, as our criterion.

Results with synthetic data

Fig. 6 shows the throughput of only read requests under different thread counts. Co-location improves performance by 10%-20%. File-level cache improves performance by 17%-35%. Co-location and file-level cache together improve performance by 20%-49%.

Fig. 8 shows the throughput of read and write requests together under different thread counts. Read write ratio is 20:1. The performance improvement is nearly the same as the performance improvement in conditions with only read requests.

Results with real world data

First, we conduct an experiment with our real world data set in 2 scenarios: without co-location or file-level cache, with only file-level cache. Fig. 7 shows the throughput improvement with varying cache size. When cache size is smaller than 64 megabytes, throughput increases almost linearly as cache size increases. But, it increases much slower after cache size reaches 64 megabytes. Since the file list gets longer as cache size increases, the searching time becomes larger.

Fig. 9 shows the throughput of the 4 scenarios with varying numbers of datanode. We can see that averagely co-location can improve throughput by 20%, file-level cache can improve throughput by 35%, co-location and file-level cache together can improve throughput by 50%.

VIII. CONCLUSION AND FUTURE WORK

Distributed storage systems designed for small files are widely used in large datacenters to power today's popular applications specific for small files. Aiming at drawbacks that exist in these systems, we proposed two optimizations on datanode in distributed storage systems. By co-locating related files, we can increase the system's throughput by maximally 20%. By implementing a file-level cache on datanode, we can increase the system's throughput by maximally 40%. By implementing the two optimizations together on datanode, we can increase the system's throughput by maximally 50%.

We believe that, for most applications, page information provides useful information about relationships between files that can be exploited by grouping. So, in this paper, we investigate the approach of grouping files that make up a single hypertext document. Other approaches based on application-specific knowledge are worth investigating. In

 ISBN: 978-1-61208-216-5

our implementation, we have provided interfaces to support other application-specific knowledge.

ACKNOWLEDGMENT

We would like to thank our program committee shepherd Petre and the anonymous reviewers for their insightful comments and constructive suggestions. This research is supported by NSFC 61133005.

REFERENCES

[1] P. Vongsathorn and S. Carson, "A System for Adaptive Disk Rearrangement", *Software – Practice and Experience*, Vol. 20, No. 3, March 1990, pp. 225–242.

[2] C. Ruemmler and J. Wilkes, "Disk Shuffling", Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, October 3, 1991.

[3] "Smart Filesystems", *Winter USENIX Conference*, 1991, pp. 45–51.

[4] S. J. Mullender and A. S. Tanenbaum, "Immediate Files", *Software–Practice and Experience*, 14 (4), April 1984, pp. 365–368.

[5] G. R. Ganger and M. F. Kaashoek, Embedded inodes and explicit grouping: exploiting disk bandwidth for small files, In ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[6] http://www.taobao.com: June, 2012

[7] Z. Zhang and K. Ghose, hfs: a hybrid file system prototype for improving small file and metadata performance, SIGOPS Oper. Syst. Rev., 41(3):175–187, 2007.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205-220, 2007.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29-43, 2003.

[10] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The Bittorrent P2P File-Sharing System: Measurements and Analysis" Peer-to-Peer Systems IV. vol. 3640, M. Castro and R. van Renesse, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 205-216.

[11] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: facebook's photo storage," presented at the Proceedings of the 9th USENIX conference on Operating systems design and implementation, Vancouver, BC, Canada, 2010.

[12] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in P2P systems," *Commun. ACM*, vol. 46, pp. 43-48, 2003.

[13] M. E. J. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary Physics*, vol. 46, pp. 323-351, 2005/09/01 2005.

[14] A. Charnes, W. W. Cooper, B. Golany, L. Seiford, and J. Stutz, "Foundations of data envelopment analysis for Pareto-Koopmans efficient empirical production functions," *Journal of Econometrics*, vol. 30, pp. 91-107, 1985.

[15] S. Joseph E, "Self-selection and Pareto efficient taxation," *Journal of Public Economics*, vol. 17, pp. 213-240, 1982.

[16] T. M. Tripp and H. Sondak, "An evaluation of dependent variables in experimental negotiation studies: Impasse rates and pareto efficiency," *Organizational Behavior and Human Decision Processes*, vol. 51, pp. 273-295, 1992.

[17] http://hadoop.apache.org/hdfs/: May, 2012

[18] http://code.google.com/fastdfs/: April, 2012

[19] http://cassandra.apache.org/: June, 2012

[20] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," presented at the Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, Stevenson, Washington, USA, 2007.

[21] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, pp. 1277-1288, 2008.

[22] J. Dean, "Evolution and future directions of large-scale storage and computation systems at Google," presented at the Proceedings of the 1st ACM symposium on Cloud computing, Indianapolis, Indiana, USA, 2010.

[23] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic Metadata Management for Petabyte-Scale File Systems," presented at the Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004.

[24] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," presented at the Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, San Francisco, California, 1994.

[25] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: fast, scalable metadata search for large-scale storage systems," presented at the Proccedings of the 7th conference on File and storage technologies, San Francisco, California, 2009.

[26] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," presented at the Proceedings of the 7th international conference on Autonomic computing, Washington, DC, USA, 2010.

[27] http://code.taobao.org/p/tfs/wiki/index/: May, 2012

[28] L.W. McVoy and S.R. Kleiman, "Extent-like Performance from a UNIX File System", in Proc. USENIX Winter, 1991, pp.33-44.

[29] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," presented at the Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, 2001.

[30] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," presented at the Proceedings of the 7th symposium on Operating systems design and implementation, Seattle, Washington, 2006.

[31] Z. Zhang and K. Ghose, "hFS: a hybrid file system prototype for improving small file and metadata performance," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 175-187, 2007.

[32] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," SIGOPS Oper. Syst. Rev., vol. 25, pp. 198-212, 1991

[33] http://en.wikipedia.org/wiki/Type_I_and_type_II_errors: June, 2012

[34] http://en.wikipedia.org/wiki/Bloom_filter: June, 2012

[35] http://en.wikipedia.org/wiki/FIFO: June, 2012

[36] http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used : June, 2012

[37] http://en.wikipedia.org/wiki/Page_replacement_algorithms#The_theoretically_optimal_page_replacement_algorithm: June, 2012

 ISBN: 978-1-61208-216-5