Cache Support in a High Performance Fault-tolerant Distributed Storage System for Cloud and Big Data

Lars Lundberg, Håkan Grahn, Dragos Ilie, Christian Melander
Department of Computer Science and Engineering
Blekinge Institute of Technology
SE-37971 Karlskrona, Sweden
Lars.Lundberg@bth.se

***Abstract*—Due to the trends towards Big Data and Cloud Computing, one would like to provide large storage systems that are accessible by many servers. A shared storage can, however, become a performance bottleneck and a single-point of failure. Distributed storage systems provide a shared storage to the outside world, but internally they consist of a network of servers and disks, thus avoiding the performance bottleneck and single-point of failure problems. We introduce a cache in a distributed storage system. The cache system must be fault tolerant so that no data is lost in case of a hardware failure. This requirement excludes the use of the common write-invalidate cache consistency protocols. The cache is implemented and evaluated in two steps. The first step focuses on design decisions that improve the performance when only one server uses the same file. In the second step we extend the cache with features that focus on the case when more than one server access the same file. The cache improves the throughput significantly compared to having no cache. The two-step evaluation approach makes it possible to quantify how different design decisions affect the performance of different use cases.**

Keywords-big data; cloud; distributed storage systems; cache; performance evaluation

I. INTRODUCTION

An increasing portion data of will be stored on large servers in the cloud, thus making the information accessible from anywhere in the world. Moreover, it is in many cloud-based systems desirable to provide a storage that is uniformly accessible by all servers in the system. One reason for this is that, in order to get good utilization of the hardware resources, cloud-based systems want to allow live migration of virtual machines (VMs) from one physical server to another. If we have a unified storage in the form of a shared (virtual) disk for all servers, we do not need to copy any files when a VM is migrated from one server to another, since all files are accessible by all physical servers.

Having one physical disk (or disk array) that is shared by many servers becomes a serious performance bottleneck and a single-point of failure. Distributed storage systems make it possible to spread out the files on many storage nodes while still providing a virtual shared disk to the servers in the system. Since the files are spread out on several disks in a number of storage nodes, we avoid the problems with performance bottlenecks and single-point of failure by using distributed storage systems. In Section II we give some examples of distributed storage systems.

One example of a distributed storage system is Compuverde (www.compuverde.com), and recent performance evaluations show that the read and write performance of Compuverde is very competitive [1]. In order to provide additional performance improvements a cache system has been added. The cache system stores frequently used file blocks in fast but relatively small Solid State Disks (SSDs). Since the Compuverde storage system is targeted to the high availability market the distributed storage system, including the cache, needs to be fault tolerant. The fault tolerant requirement makes it impossible to use the normal write-invalidate cache consistency protocols, sine we always need to have multiple copies of all data in the cache.

A caching system is a common approach to improve the performance in many systems, such as web servers [25][28], storage and file systems [26][27][30][31], and databases [29]. The two main contributions in this study are: the unusual requirement that the cache needs to be fault tolerant (which excludes the most common cache consistency approaches based on so called write-invalidate), and the two-step evaluation approach that makes it possible to quantify how different design decisions affect the performance of different use cases.

II. BACKGROUND AND RELATED WORK

In distributed storage systems, the most common interfaces are Web Service APIs like Internet Small Computer System Interface (iSCSI) [2]; REpresentational State Transfer (REST)-based [3] and Simple Object Access Protocol (SOAP)-based [4]. REST is a HTTP-based architectural style to build networked applications that allows access to stored objects by an Object Identifier (OID), i.e., no file or directory structures are supported [5]. Object-based storage systems are often referred to as *unstructured storage systems*.

There are other access methods like Network File System (NFS) and Common Internet File System (CIFS). These APIs are file-based (variable-size) and use a path to identify the data; these systems are often referred to as *structured storage systems*.

978-1-4673-7684-6/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.65

TABLE I. OVERVIEW OF DISTRIBUTED STORAGE SYSTEMS

	INTERFACE				SOLUTION		REPLICATION		METADATA	
	Unstructured		Structured							
	Web Service APIs (REST, SOAP)	*Block-based APIs (iSCSI)*	*File-based APIs (CIFS, NFS)*	*Other APIs (WebDAV, FTP, Proprietary API)*	DHT	Multicast	Copying	Striping	Centralized	Distributed
AmpliStor	X	-	-	-	-	-	-	X		X
Caringo's CAStor	X	-	X	-	-	X	X	-	X	-
Ceph	X	-	-	-	X	-	-	X	-	X
Cleversafe	-	X	-	-	-	-	-	X	X	-
Compuverde	X	-	X	X	-	X	X	X	-	X
EMC Atmos	X	-	X	-	X	-	-	X	-	X
Gluster	-	-	X	X	X	-	X	-	-	-
Google File System (GFS)	X	-	X	-	-	-	-	X	X	-
Hadoop	-	-	X	-	X	-	-	X	X	-
Lustre	-	-	X	-	X	-	-	X	X	-
OpenStack's Swift	X	-	-	-	X	-	X	-	-	X
Panasas	-	-	X	-	-	-	-	X	-	X
Scality	X	-	-	-	X	-	X	-	-	X
SheepDog	-	X	-	-	X	-	X	-	-	X

Distributed storage systems use either multicasting or Distributed Hash Tables (DHTs). Data redundancy is obtained by either using multiple copies of the stored files or by so called striping using Reed-Solomon coding [24]. When using striping the files are split into stripes and a configurable number of extra stripes with redundancy information are generated. The stripes (in case of Striping) and file copies (in case of Copying) are distributed to the storage nodes in the system.

The most well-known distributed storage systems are Amplistor [6], Caringo's CAStor [7], Ceph [8], Cleversafe[1], Compuverde, EMC Atmos [9], Gluster [10], Google File System [11], Hadoop [12], Lustre [13], OpenStack's Swift [14], Panasas [15], Scality[2] and Sheepdog[3]. Some of the distributed file systems could be used by other applications, i.e., BigTable is a distributed storage for structured data and it uses GFS to store log and data files [16].

Ceph provides an S3-compatible REST interface that allows applications to work with Amazon's S3 service. Cleversafe provides an iSCSI device interface, which enables users to transparently store and retrieve files as if they were using a local hard drive.

EMC Atmos is a structured distributed storage system that provides CIFS and NFS interfaces, as well as web standard interfaces such as SOAP and REST. Other distributed file systems such as Google File System, Hadoop Distributed File System (HDFS), Lustre and Panasas provide a standard POSIX API. Sheepdog is the only distributed storage system which is based on Linux QEMU/KVM and is used for virtual machines.

Some of the distributed file systems are also used for computing purposes, e.g., the Hadoop Distributed File System (HDFS) which distributes storage and computation across many servers. HDFS stores file system metadata and application data separately and users can reference files and directories by paths in the namespace (a HTTP browser can be used to browse the files of an HDFS instance). Lustre is an object-based file system used mainly for computing

[1] http://www.cleversafe.com/
[2] http://www.scality.com/
[3] http://www.osrg.net/sheepdog/

purposes, in particular High Performance Computing (HPC). Panasas is also used for computing purposes and similar to Lustre, it is designed for HPC.

Scality uses a ring storage system which is based on a Distributed Hashing Mechanism with transactional support and failover capability for each storage node. The Sheepdog architecture is fully symmetric and there is no central node such as a meta-data server (Sheepdog uses the Corosync cluster engine [17] to avoid metadata servers). Sheepdog provides an object (variable-sized) storage and assigned a global unique id to each object. In Sheepdog's object storage, target nodes calculated based on consistent hashing algorithm which is a schema that provides hash table functionality and each object is replicated to 3 nodes to avoid data loss [18].

The remaining distributed storage systems in Table 1 are Compuverde, Gluster and OpenStack's Swift. OpenStack's Swift is an unstructured distributed storage system that uses distributed hash tables (DHTs) and replication based on copying. Gluster is a structured (file based) distributed storage system that uses DHTs and replication based on copying. Compuverde offers a structured and an unstructured version of their system (see next section for details). The Compuverde system uses multicasting instead of DHTs and the replication can be configured for either copying or striping using Reed-Solomon coding [24].

These three systems have previously been compared in a performance evaluation [1]; Compuverde unstructured was compared to OpenStack's Swift and Compuverde structured was compared to Gluster. One major architectural difference between these systems is that Gluster and OpenStack's Swift use DHTs whereas Compuverde uses multicasting. One advantage of DHTs compared to multicasting is that we do not need to broadcast (or multicast) requests to all nodes; the hash table gives us the address of the node that stores the requested data, and we can thus avoid communication overhead. However, the disadvantage with DHTs is that we need to run a hash function to obtain the address of the data, which introduces processing overhead. This means that the architectural decision of whether to use DHTs or multicasting will introduce different kinds of overheads: processing overhead for DHTs and communication overhead for multicasting. The previous performance evaluation [1] shows that multicasting seems to result in higher performance, i.e., the communication overhead introduced by multicasting does not affect the performance as negatively as the processing overhead introduced by DHTs.

III. Distributed Storage System Architecture

The Compuverde distributed storage consists of a set of *storage nodes* providing *unstructured* data storage. This is called the Object Store. *Structured* data storage is available through a set of *gateway nodes,* which offer access to data located on storage nodes through standardized protocols such as NFS, CIFS, CDMI, OpenStack and Amazon S3.

The internal structure of the storage node is composed of four main elements (see lower part of Figure 1): frontend, cache, backend and hard drives (labeled HD in Figure 1). The cache in the storage nodes already exists and this is not the target in this project. The frontend handles RPC-like read/write requests from gateway nodes. Frequently used data are placed in the cache in order to improve the response time of the node. In case of a cache miss, the backend satisfies the request by accessing the data on the hard drives.

Structured data access is provided by a set of gateway nodes arranged in an array as shown in the upper part of Figure 1. Each gateway node implements four layers: structured data access API, gateway service, cache service (the focus of his paper), and the Object Store API.

Gateways keep information about the structure of each data item (e.g., files) in *envelope* objects. An envelope contains metadata in the form of unstructured data that are stored on the storage nodes; envelope data contain information about other envelopes and other files. An envelope can have only one *owner gateway* at each time instant. Only the owner has the authority to access the storage to read or write the envelope. A consequence of this is that when a read request arrives to a non-owner gateway node (the servicing gateway), the request is first forwarded to the owner gateway, the owner fetches the data from a storage node, forwards the data to the servicing gateway node, which then finally can respond with the data to the outside requester.

The owner can change over time, for example when the current owner fails and another gateway takes over. In the case of a directory, the directory and corresponding files have the same owner. However, a subdirectory and corresponding files can have a different owner. It is therefore possible to establish a hierarchical chain of ownership.

The purpose of the gateway service layer is to maintain and share information about envelopes. In order to improve the system response time it is desirable to add a cache layer in each gateway node, thus essentially building a distributed cache. The architecture and design of the gateway cache layer is described in this paper.

It is also possible to implement the gateway service layer directly on the storage nodes. In that case the gateway and storage functionalities are integrated in the same hardware server (see Figure 1).

The response time for a request is dominated by two components: data transfer from the storage node to the data owner gateway and transfer from the owner of the data to the gateway serving the request. Disk access times in the storage nodes are magnitudes slower than the network transfer times between gateway nodes.

A cache shortens the response time caused by the first component, since frequently used data will be readily available at the owner gateway. Allowing the serving gateway to replicate data from the owner gateway into its own cache could also alleviate the second response time component. However, this requires the system to implement methods that ensure cache consistency across multiple gateway nodes.

The storage system provides fault tolerance, either by keeping a configurable number of copies of each file, or by using Reed-Solomon coding [24]. In order to keep the fault tolerance qualities also when using the cache, we will have a configurable number of copies of each data item in the

cache. We refer to such extra copies of cache entries as shadow copies. If the gateway node containing the main copy goes down, the content can be recreated from the shadow copies. This means that we need to keep a number of copies of each cache entry at all times. Standard write-invalidate cache consistency protocols can therefore not be used.

Designing a distributed cache module is complex and the architect is faced with many decisions, such as but not limited to cache synchronization methods, snoopy- versus directory-based protocols, local cache configurations (e.g., block size, prefetching and associativity) and consistency checking during cold start [19][20][21].

IV. First Version of the Cache Implementation

There is SSD memory in each gateway node. A small portion of the space on the SSDs is reserved for *metadata* and the remaining part is used for user data. The user data are stored in blocks of 128 kB. The metadata consist of a set of 4 kB blocks, where each block of metadata describes one block of user data (not all 4 kB metadata is actually used). Figure 2 shows contents of a cache entry (metadata are blue and actual data are purple). Each file stored on the Compuverde system receives a unique *FileID* handle. The system maintains a mapping between path and filename and the corresponding *FileID*, which is transparent to the users.

When a file is stored, its contents are divided into a set of 128 kB data blocks, where all blocks are fully occupied, with perhaps the exception of the last one (when the file size is not a multiple of 128 kB).

The *Size* entry specifies the length of the data block. The *BlockIndex* entry denotes the offset in the file where the data block belongs. The *BlockVersion* entry is incremented each time the data block is changed. This is used to ensure that all peers use the correct version of the data. The *isDirty* flag indicates that the data in the cache entry is changed but the change is not yet flushed to the storage cluster.

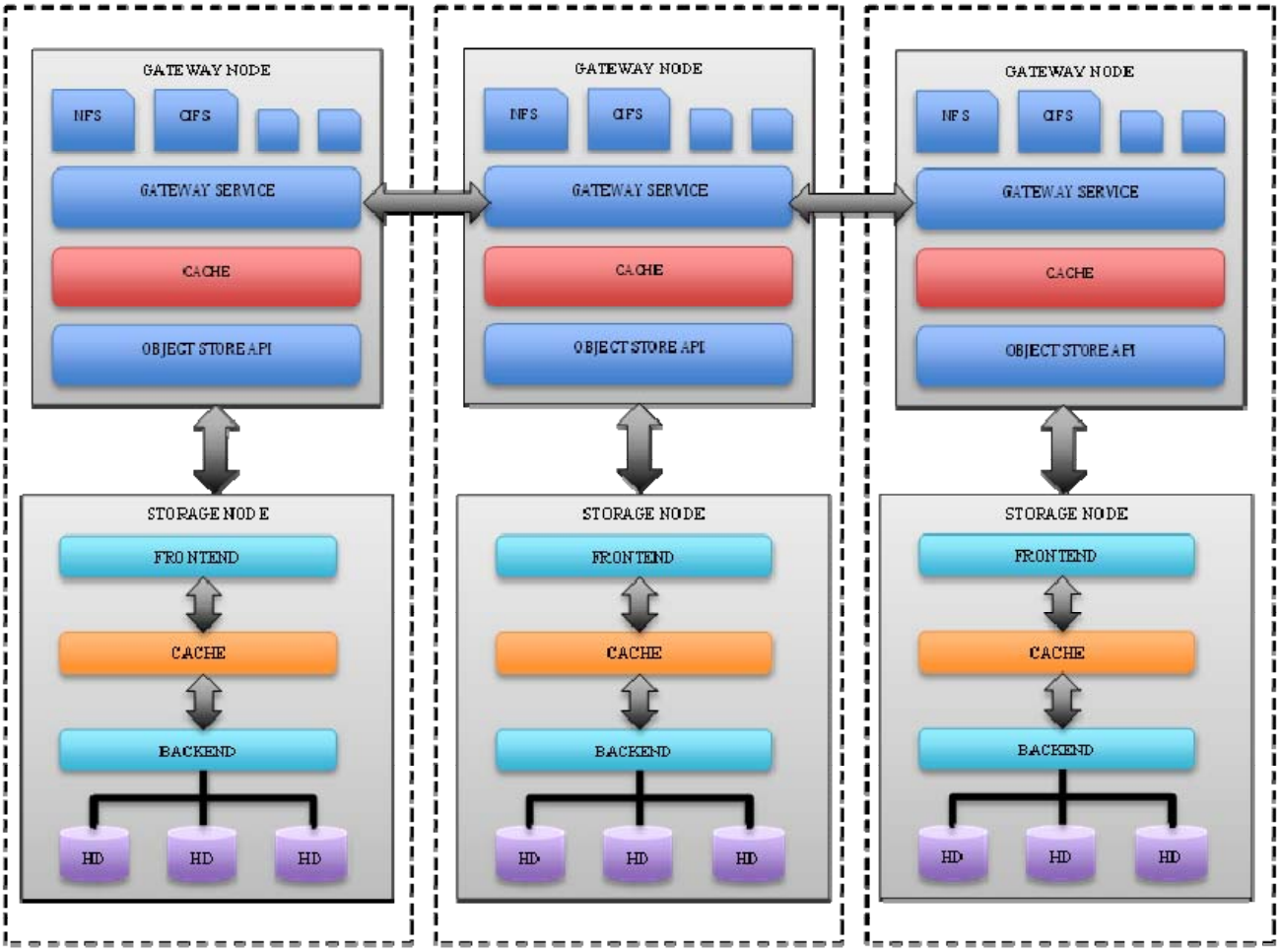

Figure 1. Overview of the Compuverde distributed storage system. The upper part shows the gateway nodes (where we implemented the cache) and the lower part shows the storage nodes. The dashed lines indicate that it is possible (but not necessary) to implement a logical storage server and gateway server on the same physical server.

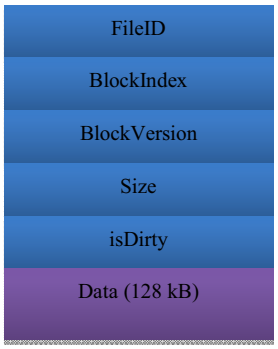

Figure 2. A cache entry.

A. Data Read and Cache Entry Creation

When data are requested from the outside, e.g., when accessing a file, there are two cases:

Case 1: The data have not been read before (or it was so long time since the last access so the cache entry has been evicted)

When data are read, the servicing gateway node must first check if any other gateway node is the owner of this data. In case 1, this is the first time data are read and the gateway node takes ownership of the data, and a cache entry is allocated in the gateway node, which now is the owner gateway of the data. Data are then fetched from a storage node. When data arrive to the owner gateway, data are put in the gateway cache, and then returned to the requester.

Case 2: The data have been read recently so there is already an owner gateway node for the data.

When data are read, the servicing gateway node must first check if any other gateway node is the owner of this data. In case 2, data are read previously and thus there exists an owner gateway. If the servicing gateway node also is the owner, it just returns the data to the requester. If the servicing gateway node is not the owner of the data, the read request is forwarded to the owner node. The owner node gets the data from its local cache and returns it to the servicing gateway. Finally, the servicing gateway can respond with the data to the outside requester.

B. Data Write and Cache Replication

In the Compuverde system, data replication in the first version of the cache is done for fault tolerance, while in most other systems data replication in the caches is done for performance reasons. If one gateway node goes down the information in the storage caches on that node should be replicated to so called shadow caches on other nodes, thus avoiding loss of data. The shadow caches/copies are spread out on the gateway nodes at random; no shadow copies are stored on the same gateway.

Upon a write (we assume that it is the owner gateway, otherwise it is forwarded similarly to case 2 for reads), the data are first written to the gateway cache. In the case of a synchronous write, data are replicated to a configurable number of shadow copies / caches to achieve redundancy. At the same time, data are also written to the storage node. When data are safely stored in the shadow copies, a write acknowledge is returned by the gateway node to the outside writer. The replication and transfer to the Object Store is scheduled to a background task.

The shadow caches are used if the ownership gateway crashes. In the case of a crash, a new owner gateway is selected among the shadow caches / gateways.

V. Final Version of the Cache Implementatione

The evaluations of the first version of the cache showed that the performance was very good for many important cases (see Section VI for details). However, the case when multiple users want to access a shared file showed insufficient performance in the first cache implementation. The case when multiple users want to read a shared file is an important case [22][23], and to obtain high performance also for this case we did the following improvements when we implemented the final version of the cache:

We now read from shadow caches instead of forwarding read requests to the owner of the cache block. Also, new shadow copies are created dynamically when a data item is read. This means that for read only files, there will be local shadow copies on all gateways that share the file. By allowing concurrent reads from these shadow copies we will increase the read performance when multiple users share the same file. This means that the shadow caches now serve two purposes: they provide fault tolerance through redundancy and they speed up read accesses.

As long as there are only reads, there is no owner of the cache block (which is a difference between the final and the first versions). In the final version, ownership is only introduced when someone wants to write to the cache block. The first gateway that writes to the block becomes the owner, and the information that there now is an owner of the block is broadcasted to all gateways. The same gateway remains the owner unless a node goes down; in that case the ownership is transferred to one of the shadow copies. There is also a long time-out; if no node has written any information to the block for a time-out period (20-30 minutes), the ownership of the block is removed (we did not have time-outs in our tests).

When there is an owner, write requests are forwarded to the owner in the same way as in the first version of the cache. However, in the final version we have introduced an optimization when reading data that has an owner. The requesting gateway node sends a read request to the owner (like in the first version), but this time the requesting gateway includes the version number (see Figure 2) of its cache block (provided that it has a local copy of the block). If the version number is up to date the owner of the block responds with an ‘OK’, and no data are transferred, otherwise the owner responds with the data and the latest version number. Besides returning the requested data to the user, the data and the latest version number are stored in the local cache on the gateway servicing the user requests. Responding with an ‘OK’ is much faster than transferring the actual data, and this optimization improves the performance of access patterns with a mix of reads and writes to shared data (see Section VI for details).

VI. PERFORMANCE MEASUREMENTS

A. Experimental Setup

We used eight gateway servers with integrated storage functionality (see Figure 1). Each server had two Intel Xeon E5-2620 (2 GHz) processors (i.e., a total of 12 cores). Each gateway server had eight 320 GB disks and six Intel SSDs with 60 GB storage each. The SSDs are configured as RAID 10 and there is a LSI 9265-8i RAID controller card with 1 GB of storage with battery backup.

We have eight load generators; each load generator runs on a separate machine, i.e., not on any of the gateway servers. Each load generator sends load in the form of read and write requests to one of the eight gateway servers over a 10 Gb network. In the tests we have used nine files, each with a size of 4 GB. One of these files is accessed by all eight gateway nodes and the other eight files are accessed exclusively by one gateway node each. There are 16 outstanding (parallel) memory requests in each load generator. We have looked at three different cases:

- All 16 requests only access the file that is exclusively accessed from the corresponding gateway. We refer to this case as *private*, since each of the eight files is only accessed from one gateway, i.e., the file that is shared between the gateways is not accessed at all.
- All 16 requests only access the file that is shared between the gateways, i.e., all gateways access the same file. We refer to this case as *shared*.
- Eight requests access the file that is only accessed from the corresponding gateway, and the other eight requests access the file that is shared between the gateways, i.e., all nine files are accessed in this case. We refer to this case as *mixed*.

We investigated three different block sizes for read and write requests from the load generators:

- Each read or write request is 0.5 kB
- Each read or write request is 4 kB
- Each read or write request is 128 kB (i.e. the same size as the block size used in the cache system)

We looked at three different mixes of read and write requests:

- 100% read requests
- 100% write requests
- 95% read requests and 5% write requests

Finally, we considered the case when all accesses from each of the load generators are done sequentially, and the case when the accesses from each of the load generators are done to random places in the file.

We used the IOmeter tool (iometer.org) on the load generators to generate the data accesses.

The measurements have been done:

- without the cache
- with the first version of the cache
- with the final version of the cache

B. Results

All values in this section are for one gateway, i.e., in order to see the capacity of the entire system, consisting of eight gateways, the values should be multiplied by a factor of eight.

1) Read Accesses

Figure 3 shows the results for 100% read accesses. The figure shows that the first version of the cache (#1 cache) improved the mixed and private cases significantly, for random accesses (right side of Figure 3). For sequential accesses (left part of Figure 3) the mixed and private cases were also improved for large blocks (128 kb). For small blocks (0.5 and 4 kb) and sequential accesses, the first version of the cache only showed limited improvement.

Figure 3 shows that the first version of the cache does, however, not improve the shared case (at least not in any significant way) for any block size.

Compared to the first version of the cache, the final version of the cache (#2 cache) provides limited improvement for the private and mixed cases. It is, however, very clear that compared to the first version of the cache, the final version of the cache results in very significant improvements for the shared case.

Compared to the case with no cache, the final version of the cache shows very significant improvements for all Read cases considered here. When looking at the performance figures, one should remember that the cache system as well as the storage nodes support fault tolerance through replication.

2) Write Accesses

Figure 4 shows the results for 100% write accesses. The figure shows that the first version of the cache (#1 cache) improved the mixed and private cases significantly, for both sequential and random accesses, and for all block sizes. The first version of the cache does, however, not improve the shared case (at least not in any significant way) for any block size.

Compared to the first version of the cache, the final version of the cache (#2 cache) provides very limited improvement for the mixed and private cases. It is, however, very clear that the final version of the cache results in very significant improvements for the shared case.

Compared to the case with no cache, the final version of the cache shows very significant improvements for all Write cases considered here.

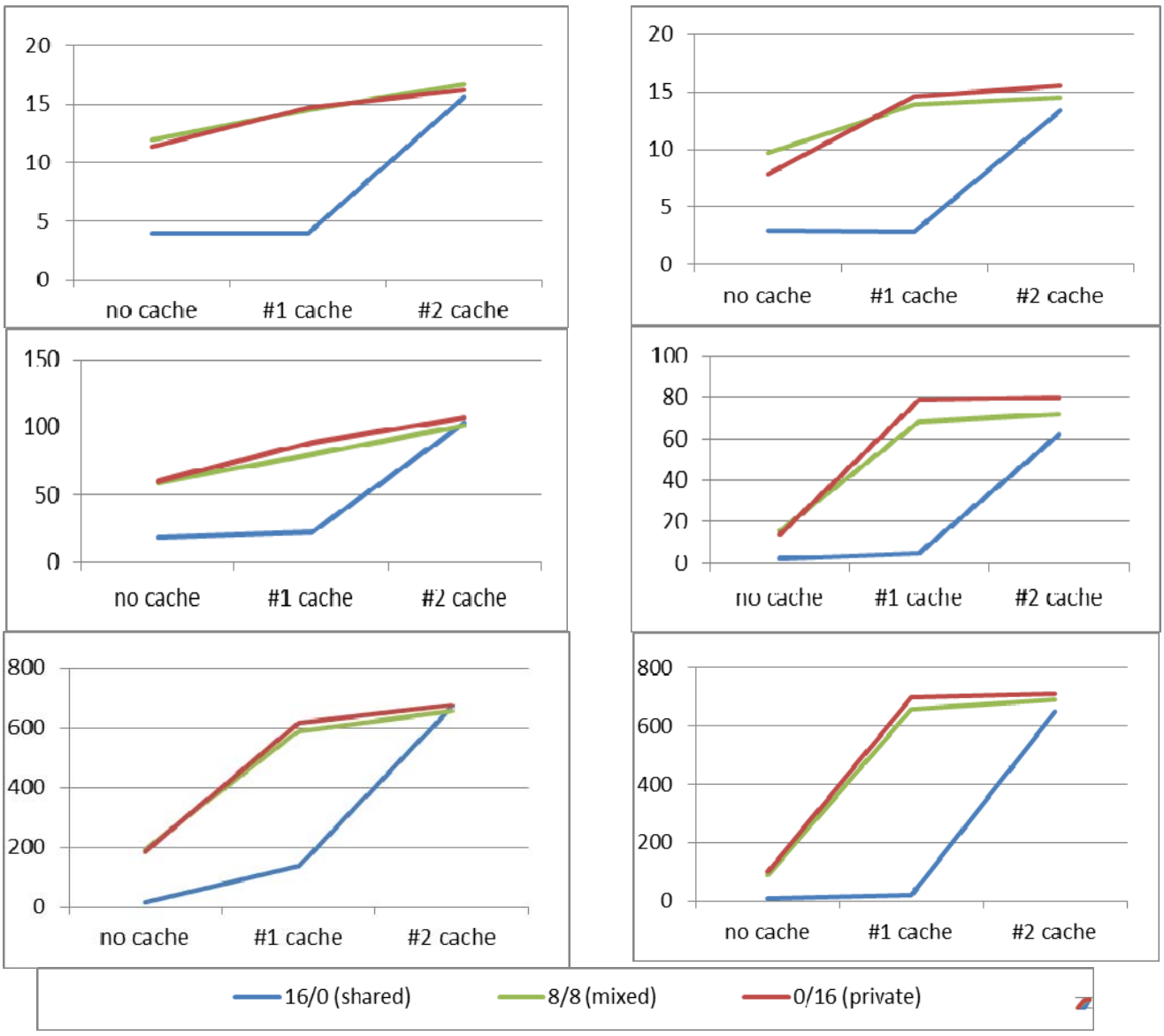

Figure 3. Throughput in MB/s for Read (upper: 0.5 kb blocks, middle: 4 kb blocks, lower: 128 kb blocks, left: sequential, right: random).

3) Read and Write Accesses

Figure 5 shows the results for 95% read and 5% write accesses. The figure shows that the first version of the cache (#1 cache) improved the mixed and private cases significantly, for both sequential and random accesses, and for all block sizes. The first version of the cache does, however, not improve the shared case (at least not in any significant way) for any block size.

Compared to the first version of the cache, the final version of the cache (#2 cache) provides rather limited improvement for the mixed and private cases. It is, however, very clear that the final version of the cache results in very significant improvements for the shared case.

Compared to the case with no cache, the final version of the cache shows very significant improvements for all the cases with 95% Read and 5% Write.

VII. CONCLUSIONS

The measurements show that the first version of the cache gives significant performance increases for the mixed and private cases. However, for the shared case the performance increase for the first version cache was very limited. When using the final version of the cache we get significant performance improvements for all cases. In some cases the throughput is improved with as much as a factor of 50-75 compared to the case with no cache.

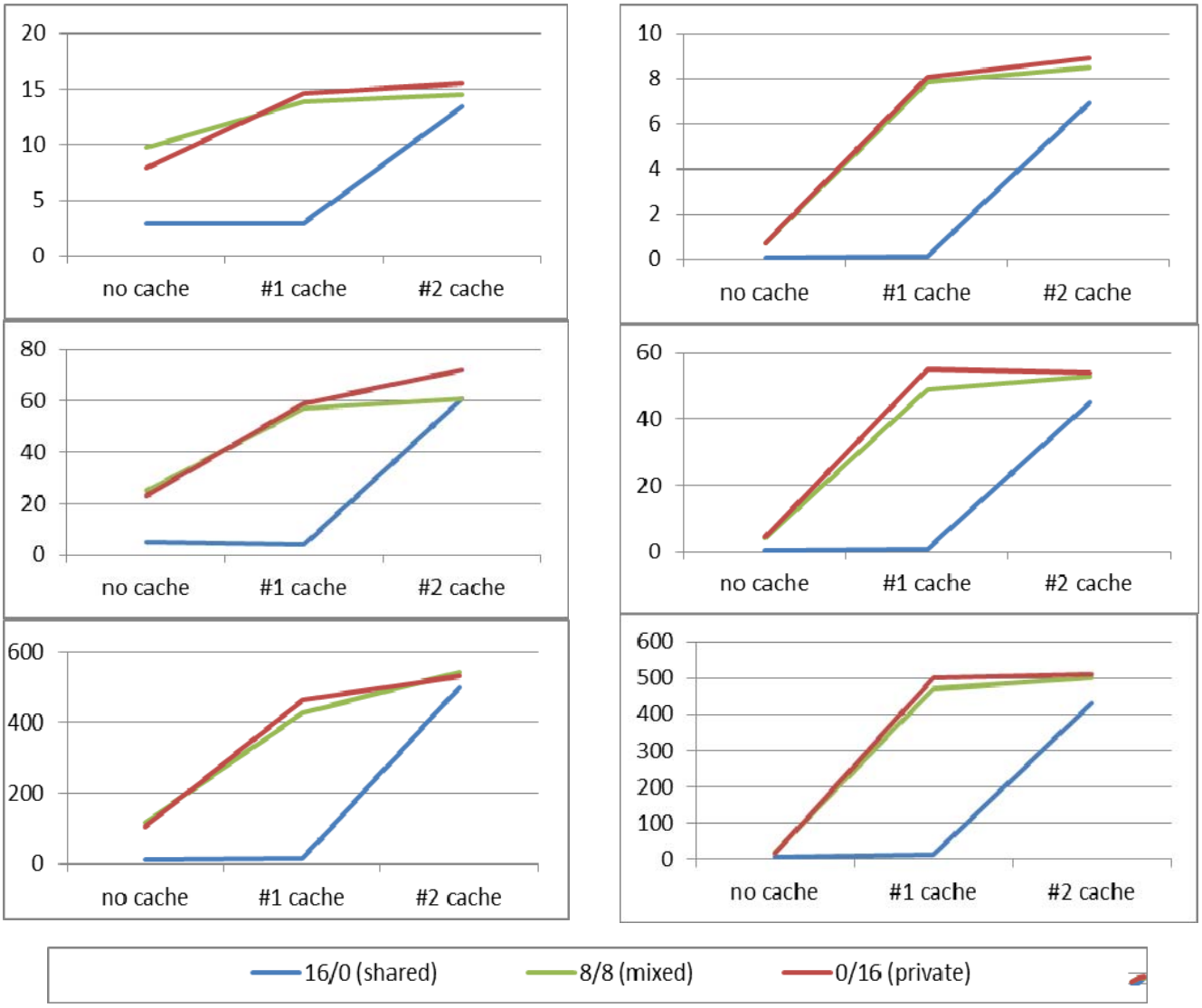

Figure 4. Throughput in MB/s for Write (upper: 0.5 kb, middle: 4 kb, lower: 128 kb, left: sequential, right: random).

Multiprocessor cache systems normally invalidate all copies of a cache block when a processor writes to the cache block. This approach could not be directly applied in this case, since we, for fault tolerance reasons, need to keep a configurable number of shadow copies of each cache block. Since we are not invalidating copies of the cache block on writes, some cache copies may have an old value. In order to handle this, we have included a version number of each cache block (see Figure 2). This version number serves as a "lazy" invalidation when a cache block is updated, i.e., the cache system compares the version number with the owner's version number in order to decide if the local copy is valid of not.

The workload is very transparent, i.e., we know the exact mix of Read/Write, the block size, and if accesses are private/shared and sequential/random. This transparency and our two-step evaluation approach make it possible to quantify how different design decisions affect the performance of different workload cases. This gives a more detailed understanding than just comparing the final version of the cache with the case with no cache, running some unknown workload mix. Having this kind of detailed understanding is valuable for designers of distributed cache systems in general, and of course for designers of cache systems for fault-tolerant distributed storage systems in particular.

ACKNOWLEDGMENT

This work is part of the research project "Scalable resource-efficient systems for big data analytics" funded by the Knowledge Foundation (grant: 20140032) in Sweden..

REFERENCES

[1] S. Shirinbab, L. Lundberg, and D. Erman, *"Performance Evaluation of Distributed Storage Systems for Cloud Computing"*, International Journal of Computer and Their Applications, Vol. 20, No. 4, pp. 195-207, Dec. 2013.

[2] P. Wang, "IP SAN- from iSCSI to IP-addressable Ethernet disks", Mass Storage Systems and Technologies. Proceedings, 20th IEEE/11th NASA Goddard Conference, 2003.

[3] Roy T. Fielding, Richard N. Taylor, "Principles design of the modern Web architecture", ICSE'00 Proceedings of the 22nd international conference on software engineering, ACM New York, NY, USA, 2000.

[4] F. Curbera, M. Duftler, R. Khalaf, N. Mukhi, W. Nagy, S. Weerawarana, "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI", Internet Computing, IEEE, NY, USA, 2002.

[5] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, Julian Satran, "Object storage: The future building block for storage systems", 2nd International IEEE Symposium on Mass Storage Systems and Technologies, Sardinia,2005.

[6] Amplidata,"Amplistor: Unbreakable Object Storage for Petabyte-Scale Unstructured Data" Whitepaper, April, 2011.

[7] Caringo CAStor (2011, September 15). "Castor the Market Leading Object Storage Engine" [Online]. Available: http://www.caringo.com/downloads/datasheets/Caringo-CAStor-Object-Storage.pdf

[8] Scott A. Brandt, Darrell D. E. Long, Carlos Maltzahn, Ethan L. Miller, Sage A. Weil, "Ceph: A Scalable, High-Performance Distributed File System", Proceeding of the 7th Conference on Operating Systems Design and Implementation (OSDI'06), November, 2006.

[9] EMC Atmos, "EMC Atmos Cloud Optimize Storage for Web Services" Whitepaper, April, 2010.

[10] Gluster Inc., "An Introduction to Gluster Architecture", Whitepaper, 2011.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System", 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2006.

[12] Shunsuke Mikami, Kazuki Ohta, Osamu Tatebe, "Using the Gfarm File System as a POSIX Compatible Storage Platform for Hadoop MapReduce Applications", GRID'11 Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing, IEEE Computer Society Washington, DC, USA, 2011.

[13] Sarp Oral, Galen Shipman, Feiyi Wang, "Understanding Lustre FileSystem Internals", OAK RIDGE, 2009.

[14] OpenStack, LLC, "Welcome to Swift's documentation!", Swift v1.4.8-dev documentation, 2011.

[15] Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Marc Unangst, Brent Welch, Jim Zelenka, Bin Zhou. "Scalable Performance of The Panasas Parallel File System", FAST'08 proceedings of the 6th USENIX Conference on File and Storage Technologies, USENIX Assosiation Berkeley, CA, USA, 2008.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, "BigTable: A Distributed Storage System for Structured Data", ACM Transactions on Computer Systems (TOCS), New York, USA, June, 2008.

[17] Andrew Beekhof. Christine Caulfield, Steven C. Dake, "The Corosync Cluster Engine", Proceedings of the Linux Symposium, Ottawa, Ontrio, Canada, July, 2008.

[18] George Parisis, "DHTbd: A Reliable Block-based storage system for High Performance clusters", Proceedings of the IEEE/ACM CCGRID, UK, 2011.

[19] T. Cortes, S. Girona, and J. Labarta. *"Avoiding the cache-coherence problem in a parallel/distributed file system,"* High-Performance Computing and Networking, Lecture Notes in Computer Science 1225, pp. 860-869, 1997.

[20] P. Triantafillou and C. Neilson, *"Achieving Strong Consistency in a Distributed File System,"* IEEE Transactions on Software Engineering, vol. 23, no. 1, pp. 35-55, Jan. 1997.

[21] R.C. Burns, R.M. Rees, D.D.E. Long, *"Safe caching in a distributed file system for network attached storage,"* 14th International Parallel and Distributed Processing Symposium, pp. 155-162, 2000.

[22] N. Agrawal, W.J. Bolosky, J.R. Douceur, and J.R. Lorch, *"A five-year study of file-system metadata,"* ACM Trans. Storage, Vol. 3, No. 3, Article 9, October 2007.

[23] A.W. Leung, S. Pasupathy, G. Goodson, and E.L. Miller, *"Measurement and analysis of large-scale network file system workloads,"* USENIX 2008 Annual Technical Conference (ATC'08), pp. 213-226, 2008.

[24] T. Karlsson and L. Lundberg, "*Performance Evaluation of Cauchy Reed-Solomon Coding on Multicore Systems*", 2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip, 26-28 September, 2013, Tokyo, Japan, pp. 165-170.

[25] A. Waleed, S.M. Shamsuddin, and A.S. Ismail, "A survey of Web caching and prefetching", Int. J. Advance. Soft Comput. Appl 3, no. 1 (2011), pp. 18-44.

[26] S. Byna, Y. Chen, X-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and IO signatures", ACM/IEEE conference on Supercomputing (SC'08), 2008, pp. 1-12.

[27] W-K. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman, "Collective Caching: Application-aware Client-side File Caching", Symposium on High Performance Distrbuted Computing 2005 (HPDC'05).

[28] S.V. Nagaraj, "Web Caching and Its Applications", Springer 2004. ISBN 978-1-4020-8050-0.

[29] X. Wang, T. Malik, A. Burns, S. Papadomanolakis, and A. Ailamaki, "A workload-driven unit of cache replacement for mid-tier database caching", 12th International Conference on Database Systems for Advanced Applications, 2007 (DASFAA'07), pp. 374-385.

[30] S-D. Yoon, I-Y. Jung, K-H. Kim, and C-S. Jeong,"Improving HDFS performance using local caching system", 2nd International Conference on Future Generation Communication Technology 2013 (FGCT'13), pp. 153-156.

[31] J. Zhang, G. Wu, X. Hu, and X. Wu, "A Distributed cache for HADOOP Distributed File System in Real-Time Cloud Services", ACM/IEEE 13th International Conference on Grid Computing, 2012 (GRID'12), pp. 12-21.

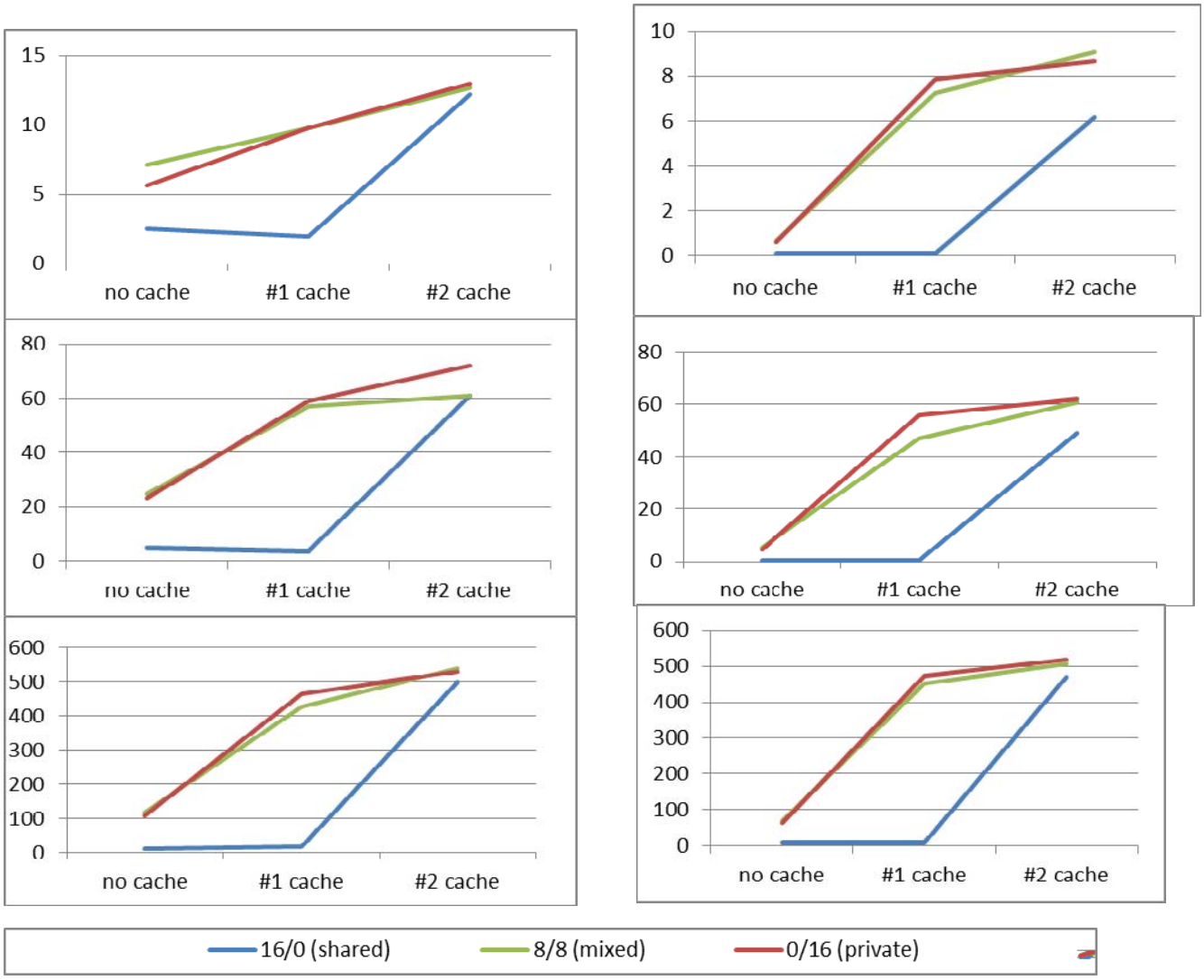

Figure 5. Throughput in MB/s for 95% Read and 5% Write, sequential access (upper: 0.5 kb, middle: 4 kb, lower: 128 kb, left: sequential, right: random).