SystemXXL: a Scalable Secondary Storage

authors omitted for blind review
both SystemXXL and AnDHT are anonymized names

Abstract

SystemXXL is a scalable, secondary storage solution aimed at the enterprise market. The system consists of a back-end architectured as a grid of storage nodes built around a distributed hash table; and a front-end consisting of a layer of access nodes scaled for performance and implementing traditional file system interface.

This paper concentrates on the back-end, which is, to our knowledge, the first commercial implementation of a scalable, high-performance content-addressable secondary storage delivering global duplicate elimination, per-block user-selectable failure resiliency, self-maintenance including automatic recovery from failures with data and network overlay rebuilding.

The back-end programming model is based on directed acyclic graphs of blocks with pointers to other blocks exposed. This model is exported with a low-level API allowing clients to implement new access protocols and to add them to the system on-line. The API has been validated with an implementation of the file system interface.

The critical factor for meeting the design targets has been the selection of proper data organization based on redundant chains of data containers. We present this organization in detail and describe how it is used to deliver required data services. Surprisingly, the most complex to deliver turned out to be on-demand data deletion, followed (not surprisingly) by the management of data consistency and integrity.

1 Introduction

The enterprise environment places strenuous demands on the secondary storage. With ever increasing amounts of data produced and fixed backup windows, there is a clear need for scaling performance and backup capacity appropriately. Varying importance of data requires corresponding reliability, availability and retention periods. Regulatory requirements (SOX, HIPPA, the Patriot Act, SEC rule 17a-4(t)) demand security, traceability and data audit. Strict data retention and deletion procedures need to be defined and followed rigorously. Retained data need to be recovered and presented on demand, and failing to do so may result not only in a serious loss to the business, but also in fines and even criminal prosecution. Last but not least, with limited IT budgets efficiency is also of primary importance, both in terms of improving storage utilization as backup and archival data consume more space, and in terms of reducing mounting data management costs.

Substantial progress has been made to address these enterprise needs, as demonstrated by advanced disk-targeted deduplicating VTLs [4, 5], disk-based back-end servers [38] and content-addressable archiving solutions [6]. However, exponential increase in the amount of data stored creates new problems not addressed by these solutions. First of all, unlike primary storage, which is usually networked and under common management (e.g. SANs), secondary storage consists of a large number of highly-specialized dedicated components, each of them being a storage island requiring customized, elaborate, and often manual, administration and management. As a result, large fraction of the total cost of ownership (TCO) can still be attributed to management of more and more of secondary storage components. Moreover, fixed capacity assignment to each storage device results in poor capacity utilization, whereas duplicate elimination limited to one device only leads to wasted space caused by duplicates stored on multiple components. Finally, since each of secondary storage devices offers fixed, limited performance, reliability and availability, the high overall requirements of enterprise secondary storage in these dimensions can be met only by implementing complex in-house solutions.

Fortunately, new technology and previous research results provide building blocks for a solution addressing these problems. Content-addressable storage

paradigm [6, 20, 38] enables cheap and safe implementation of duplicate elimination, whereas distributed hash tables [1, 18, 21, 22, 26, 37] allow for building scalable, failure-resistant systems and extending duplicate elimination to a global level. Erasure codes can add redundancy to the stored data with fine-grain control between required redundancy level and resulting storage overhead. Hardware and pricing trends are also critical for enabling SystemXXL. Large, reliable SATA disks with prices falling every year deliver vast raw yet cheap storage capacity, whereas multicore CPUs provide cheap and powerful computing resource required by such system. Other work applicable include research on self-management [13, 28], monitoring [30], and on-line reconfiguration [25] and upgrade [9]. All these elements facilitated building SystemXXL, but the task proved to be much more complex than originally envisioned, requiring a lot of original research and development effort; comparable to designing and constructing a building out of bricks and stones.

SystemXXL [2] is a commercial secondary storage solution for enterprise addressing shortcomings discussed earlier. It consists of a back-end architectured as a grid of storage nodes and a front-end consisting of a layer of access nodes scaled for performance. The front-end, implementing the file system interface, is discussed elsewhere [3]. In this paper we concentrate on the back-end. Its capacity is dynamically shared among all clients and all types of data like back-up and archiving. This sharing together with system-wide duplicate elimination allow for highly efficient use of storage capacity. The system is highly-available, as it supports on-line extensions and upgrades, tolerates multiple disk, node and network failures, rebuilds the data automatically after failures and informs users about recoverability of the deposited data. The reliability and availability of the stored data can be additionally dynamically adjusted by the clients with each write, as the back-end supports multiple data redundancy classes.

This paper makes the following contributions. First, it presents the SystemXXL as a concrete commercial implementation of scalable secondary storage system addressing today's enterprise needs. Second, it discusses in detail the SystemXXL data organization and how it is used to implement advanced data services like global duplicate elimination, on-demand deletion, and data integrity management. Third, it contains an evaluation of the SystemXXL that demonstrates effectiveness of its implementation.

The remainder of this paper is organized as follows. Section 2 describes the system's functionality including the programming interface. Section 3 contains a high-level discussion of the back-end design. It establishes context for the next section, 4, which discusses requirements on data organization and the resulting solution. Section 5 illustrates how this organization is used to deliver data services like data rebuilding and distributed data deletion. Section 6 presents evaluation of the system. Related work is discussed in Section 7, whereas conclusions and future work are given in Section 8.

2 Functionality

The back-end has been designed as a vast data repository, allowing for storing and extracting streams of data with high performance. Although internally it consists of a potentially large number of storage nodes, externally it behaves as one large system. The scalability target is at least thousands of dedicated nodes resulting in raw storage capacity on the order of hundreds of petabytes, with potentially even larger configurations. The primary deployment target is the data center.

SystemXXL back-end from the beginning was intended as a foundation for a commercial product. Therefore, one of the design targets has been to support not only tailor-made new applications, but also commercial legacy applications, as long as they use streamed data access. As a result, the system does not define one fixed access protocol, instead the system is flexible to allow support for legacy applications using standards like file system interface as well as for new applications using highly-specialized access methods. New protocols can be added on-line with new protocol drivers, without interruptions for clients using existing protocols.

One of the primary design goals has been to ensure continuous operation of the system, limiting or eliminating impact of upgrades, extensions and failures. Because of the distributed architecture of the system, it is often possible to keep system availability non-stop even during hardware or software upgrade (rolling upgrade) eliminating need for costly downtime. Moreover, the system is capable of automatic self-recovery in case of hardware failures (disk, network, power loss), and even from some of software failures. The system works correctly in the presence of up to a specific configurable number of fail-stop and intermittent hardware failures. Because of high overhead of Byzantine failure handling and low probability of their occurrence in the data center, they are not handled, except that there are several layers of data integrity checking, so random data corruption will be detected.

Another important function of the system is to ensure high data reliability, availability and integrity. Each block of data is written with a user-selected redundancy level, allowing this block to survive up to the requested number of disk and node failures. This is achieved with erasure coding each block into fragments; as shown in [31] erasure codes increase mean time to failure by

many orders of magnitude over simple replication for the same amount of space overhead. After a failure, if a block remains readable, the system automatically schedules data rebuilding to bring the redundancy back to the level requested by the user. The system ensures that no permanent data loss remains hidden for long. Global state of the system indicates whether all stored blocks are readable, and if so, how many disk and node failures must happen before data loss occurs.

2.1 Programming Model

The back-end programming model is based on an abstraction of a sea of variable-sized, content-addressed, highly-resilient blocks. Block address is derived from SHA-1 hash of its content. A block consists of data and, optionally, an array of pointers, pointing to already written blocks. Blocks are variable-sized to allow for better duplicate elimination ratio; and pointers are exposed to facilitate data deletion implemented as garbage collection. The back-end exports a low-level block interface used by protocol drivers to implement new and legacy protocols. We have decided to provide such block interface instead of a high-level one like file system to simplify the implementation and allow for clean separation of the back-end from the front-end. Moreover, such interface allows for efficient implementation of a wide range of high-level protocols, not just one.

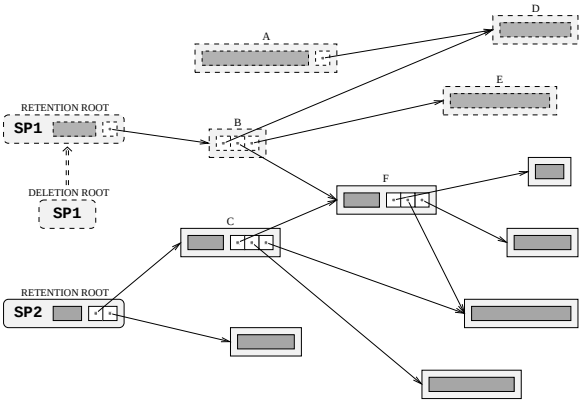

Figure 1: Blocks organized in a directed acyclic graph. Data part of each block is shaded, pointers are not.

Blocks in the back-end form a DAG (directed acyclic graph), as illustrated by Fig. 1. Drivers write trees of blocks, but because of deduplication, these trees overlap at deduplicated blocks and form directed graphs. Additionally, no cycle is possible in these structures, as long as the hash used in block address is secure. A source vertex in a DAG is usually a block of a special block type called *searchable retention root*. Besides regular data and an array of addresses, a retention root contains a user-defined *search key* used to locate the block. Such a key can be arbitrary data; a user retrieves a searchable block by providing its search key instead of a cryptic block content address. For example, multiple snapshots of the same file system can have each root organized as a searchable retention root with search key containing file system name and a counter incremented with each snapshot. Searchable blocks do not have user-visible addresses and cannot be pointed to, so they cannot be used to create cycles in block structures.

Fig. 1 shows a set of blocks organized into a DAG with 3 source vertices, 2 of them are retention roots; the 3rd source vertex is a regular block, which indicates that this part of the DAG is still under construction.

The API operations include writing and reading regular blocks, writing searchable retention roots, searching for a retention root based on its search key; and marking a retention root with a specified key to be deleted by writing an associated deletion root, as discussed below. Note that currently cutting data stream into blocks is beyond this interface and is left to the drivers, although we plan to re-evaluate this decision soon.

On writing a block, a user assigns it to one of a few available *redundancy classes*. Each class represents a different tradeoff between data redundancy and storage overhead: from low redundancy data class — a block in this class can survive only one disk failure, but storage overhead over block size is minimal — to critical data class, in which a block can be replicated multiple times on different disks and physical nodes.

The system does not provide a way to delete a single block immediately, because such block may be referenced by other blocks, so in fact it should not be deleted. Instead, the API allows to mark which parts of DAG(s) should be deleted. To mark a retention root not alive, a user writes a special block called *searchable deletion root* with the search key identical to this retention root's search key. In Fig. 1, there is a deletion root associated with the retention root SP1. The deletion algorithm marks for deletion all blocks not reachable from the alive retention roots, for example in Fig. 1 all blocks with dotted lines will be marked. Note that the block named A will also be deleted because there is no retention root pointing to it, whereas the block named F will be retained, as it is reachable from the retention root SP2, which is alive, because it does not have a matching deletion root.

During data deletion, there is a short read-only period, in which the system identifies blocks to be deleted. Actual space reclamation happens in the background during regular read-write operation. Note that before entering a read-only phase, all blocks to be retained should be pointed by alive retention roots.

3 System Architecture

SystemXXL back-end nodes are built of highly reliable server-grade components. No customized hardware is needed. Detailed description of available hardware configurations is given in Section 6. The number of storage nodes determines total raw capacity of the system as well as its maximal level of performance. Front-end access nodes can be added to realize this performance up to the limit determined by the current back-end configuration.

Software components of the back-end include *storage server*, *proxy server* and *protocol drivers*. Each storage node hosts one or more storage server processes. The number of storage servers run on a storage node depends on its resources. The bigger the node, the more servers we run, with each server responsible exclusively for a specific number of this node disks. With the advent of multicore CPUs, the critical decision has been made how to harness all this computing power. An obvious approach to parallelize each storage server more with each increase in the number of cores available suffers from programming complexity; it is much easier to keep parallelism constant per storage server and put multiple servers on one storage node.

Proxy servers run on access nodes and export the same block API as the storage servers. A proxy provides services like locating the back-end nodes, optimized message routing and caching.

Protocol drivers use the API exported by the back-end to implement access protocols. These drivers can be loaded in the runtime on both storage and proxy servers. The decision on which node to load a given driver depends on available resources and driver resource needs. Usually, resource-hungry drivers like the file system driver are loaded on proxy servers.

3.1 Storage Server

Storage server is the main software component of the back-end. The design goals for storage server architecture were: targeting it to multicore CPUs, and to distributed environment, support for parallel development by multiple teams of programmers, high maintainability and testability, and, last but not least, high reliability of the resulting system.

To satisfy these goals we have designed and implemented an asynchronous pipelined message passing framework consisting of stations called *units*. Each unit in this pipeline is single-threaded and does not write-share any data structures with other units (a unit may also have some internal worker threads). The only communication among pipelined units happens with message passing, so these units can be co-located on the same physical node, as well as distributed to multiple nodes. When communicating units are co-located on the same node, read-only sharing can be used as an optimization. Synchronization and concurrency issues have been limited to one unit only. Additionally, each unit can be tested in separation by providing stubs of other units.

3.2 Network Overlay

Since one of our design goals has been scalability, the use of distributed hash tables has been a natural choice. However, because for a distributed storage system both storage utilization and data redundancy are extremely important, we have had additional requirements on a DHT: assurances about storage utilization and an ease of integration of the selected overlay network with data redundancy scheme we have planned to use, i.e. erasure coding. Since none of the existing DHTs allowed for that, we have decided to use a modified version of the *name removed for blind review* (AnDHT) [1] distributed hash table. AnDHT makes it possible to maintain very short routing paths for a wide range of the number of nodes and guarantees a minimal level of storage utilization.

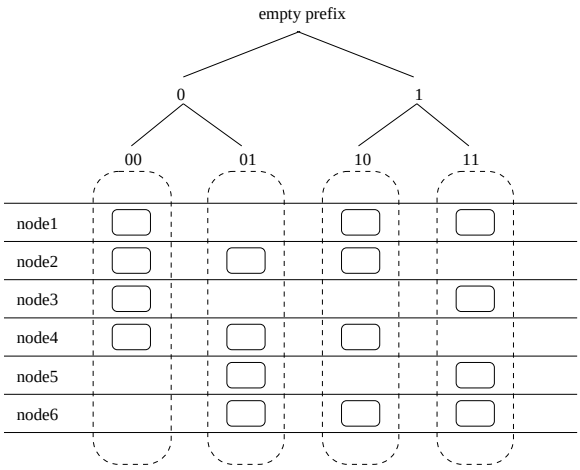

Figure 2: Supernodes and components. 4 supernodes spanned over 6 physical nodes. Each supernode has 4 components, i.e. supernode cardinality is 4.

In AnDHT, each overlay node is assigned exactly one hashkey prefix, which is also an identifier of this virtual node. All prefixes together cover entire hashkey space, and the overlay network strives to keep them disjoint. An AnDHT node is responsible for hashkeys with prefix equal to this node identifier. Upper part of Fig. 2 shows a prefix tree having as leafs four AnDHT nodes dividing the prefix space into four disjoint subspaces.

To meet our DHT requirements, we have extended the original AnDHT with *supernodes*. A supernode rep-

resents one AnDHT node (and as such, it is identified with a hashkey prefix), but is spanned over several physical nodes to increase resiliency to node failures. Each supernode consists of a fixed number (called *supernode cardinality*) of *supernode components*. Components of the same supernode are called *peers* and are usually placed on separate physical nodes, as show on Fig. 2. Usually, supernode cardinality is in range of 4-32, and in the commercial SystemXXL it is set to 12. For a given SystemXXL incarnation, its supernode cardinality is the same for all supernodes and is constant throughout entire system lifetime.

Supernode peers use distributed consensus algorithm to decide what change should be applied to the supernode — for example, after node failure, they decide on which physical nodes new incarnations of lost components should be re-created.

3.3 Read and Write Handling

On write, a block of data is routed to one of the peers of the supernode responsible for the hashkey space to which this block hash belongs. Next, this write-handling peer checks if a suitable duplicate is already stored; this process is described in detail in Section 5.2. If a duplicate is found, its address is returned; otherwise the new block is compressed (if requested by a user), fragmented, and its fragments are distributed to remaining peers.

A read request is also routed to one of the peers of a supernode responsible for this block hashkey. Such peer first locates the block metadata (usually it is found locally), and next sends fragment read requests to some of other peers in order to read the minimal sufficient number of fragments required for this block reconstruction. If any of these requests times out, all remaining fragments are read. After sufficient number of fragments have been found, the block is reconstructed, decompressed (if it was compressed), verified and, in case of successful verification, returned to the user.

In general, reading is very efficient for streamed access, as all fragments are sequentially pre-fetched from disk to a local cache. However, fragment location by a peer can be a quite elaborate process. Usually, it is enough to check the local node index and the local cache (their organization is beyond the scope of this paper), but in some cases (for example, during component transfers or after intermittent failures), the requested fragment may be present only in one of the previous locations of this component. In such case, the component directs distributed search for missing data. In particular, the trail of previous component locations can be searched in the reverse order.

3.4 Load Balancing

In a distributed storage system like the SystemXXL back-end, the distribution of components among physical nodes is critical for system survivability, data resiliency and availability, storage utilization, and system performance. For example, placing too many peer components on one machine may have catastrophic consequences if this node is lost. The affected supernode may not recover, because too many components have been lost; and even in case it is recoverable, some or even all of the data handled by this supernode may not be readable, due to loss of too many fragments. Also, performance of the system is maximized, when components are assigned to nodes proportionally to available node resources, as the load on each node is proportional to the hashkey prefix space covered by the components assigned to this node.

Our system continuously attempts to balance component distribution over all physical machines to reach a state where failure resiliency, performance and storage utilization are maximized. The quality of a given distribution is measured by a multi-dimensional function prioritizing these objectives, called *system entropy*. Such balancing is carried out by each machine, which periodically considers set of all possible transfers of locally hosted components to neighboring nodes. If the machine finds a transfer that would improve the distribution, such component transfer is executed (with safeguards preventing multiple conflicting transfers happening at the same time). After a component arrives at a new location, its data is also moved from old location(s) to the new one; but this data transfer happens in the background and may take a long time.

Load balancing is also used to manage adding and removing machines to/from the system, with the same entropy function applied to measure quality of resulting component distribution after these changes.

3.5 Impact of Supernode Cardinality

Selection of supernode cardinality has profound impact on properties of SystemXXL. First of all, it determines the maximal number of tolerated node failures. The network survives node failures as long as each supernode remains alive, and for that at least half of each supernode peers plus one should remain alive to reach a consensus. As a result, the system survives at most half of supernode cardinality minus 1 permanent simultaneous node failures among physical nodes hosting peers of each supernode.

Supernode cardinality influences also scalability, at least in theory. For a given cardinality, the probability that each supernode survives is fixed; the higher cardi-

nality the higher probability. When a system size grows, its number of supernodes also grows, and, as a result, the system reliability decreases, as for the system to be operational we require all supernodes to be alive. However, since permanent loss of a physical node is very unlikely, the practical impact of this limitation is negligible in the target range of the system size.

Finally, supernode cardinality defines the number of data redundancy classes available. Erasure coding is parametrized with the maximal number of fragments that can be lost while a block remains still reconstructible. Since in SystemXXL the erasure coding always produces supernode cardinality fragments, the tolerated number of lost fragments can vary from one to supernode cardinality minus one (in the latter case we keep supernode cardinality copies of such block). Each such choice of tolerated number of lost fragments defines one data redundancy class. Each class represents different tradeoff between storage overhead (due to erasure coding) and failure resilience. Such overhead is given by the ratio of the tolerated number of lost fragments to the difference between supernode cardinality and the tolerated number of lost fragments. For example, if supernode cardinality is 12 and a block can lose no more than 3 fragments, then the storage overhead for this class is given by the ratio of 3 to (12-3), i.e. 33%.

4 Data Organization

Proper representation of stored data is critical for meeting reliability, availability and performance targets of SystemXXL. The system should be able to easily identify the availability of stored data, and in case of a failure, rebuild the data only to the requested redundancy level (as opposed to RAID, which rebuilds entire disk even if it contains no valid user data). Since components move between nodes followed by the data transfer, it should be possible to locate and retrieve data from old component locations. When such data is available, it should be transferred instead of being rebuilt, as transfer is much cheaper operation. Data written in one stream should be placed close to each other to maximize write and read performance. Last but not least, the data organization should support on-demand distributed data deletion, in which data blocks not reachable from any alive retention root are deleted and the space occupied by them is reclaimed.

4.1 Synchruns and Synchrun Components

As discussed earlier, we use erasure coding for data redundancy. Resulting fragments of one block are distributed to peer components of the supernode responsible for this block. The basic logical unit of data management in SystemXXL is *synchrun*, containing a limited number of consecutive blocks written by one write-handling peer component and belonging to a given supernode. Since writing a block really means writing supernode cardinality of its fragments, each synchrun is represented by supernode cardinality of *synchrun components*, one for each peer. For the i-th peer of a supernode, the corresponding synchrun component contains all i-th fragments of the synchrun blocks. A synchrun is a logical structure only, but synchrun components actually exist on corresponding peers.

4.2 Chains of Containers

For a given write-handling peer, only one synchrun is open at any given time. As a result, all such synchruns can be logically ordered in a chain, with the order determined by the write-handling peer. Synchrun components are placed in a data structure called *synchrun component container (SCC)*. Each SCC can contain one or more chain-adjacent synchrun components, and as a result, SCCs form also chains similar to synchrun component chains.

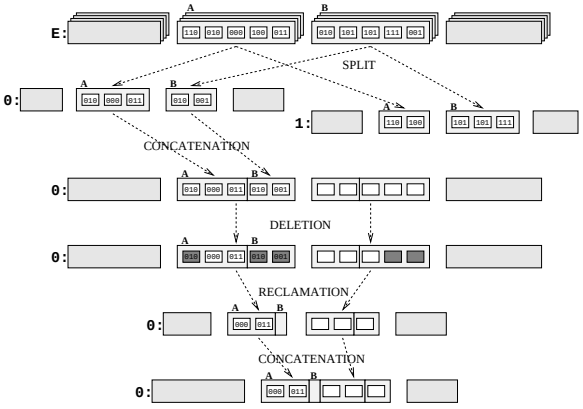

Figure 3: Data organization with synchruns and synchrun containers.

Upper row in Fig. 3 shows two synchruns A and B both belonging to the empty prefix supernode (covering entire hashkey space). Each synchrun component is placed here in one SCC, with individual fragments also shown. SCCs with synchrun components of these synchruns are shown as rectangles placed one behind the other. A chain of synchruns is represented by the supernode cardinality of SCC chains, we call them *peer SCC chains*. In the remainder of the Fig. 3 we show only one such peer SCC chain.

Peer SCC chains, in general, are identical with respect to synchrun components metadata and the number of fragments in each of them, but occasionally there may

be differences caused, for example, by node failures resulting in chain holes. This chain organization allows for relatively simple and efficient implementation of required features. For example, data is available (i.e. all blocks are reconstructible), if sufficient number of peer chains (equal to the number of fragments needed to reconstruct each block) do not have any holes. Note that in such way determination of data availability can easily be made for each redundancy class.

Each supernode will eventually be split due to loading more data or adding more physical nodes (this is a regular AnDHT split and results in two new supernodes with prefixes extended from the ancestor prefix with, respectively, 0 and 1). After the supernode split, each synchrun in this supernode is also split in half, with fragments distributed between them based on their hash prefixes. Second row of Fig. 3 shows two such chains, one for the supernode with the prefix 0, and the other for the supernode with the prefix 1. Note that, as a result of the split, fragments of synchruns A and B are distributed to these two chains; and we end up with 4 synchruns now, but each of them approximately half the size of the original one.

The system strives to maintain limited number of local SCCs, and merges adjacent synchrun components into one SCC (as shown on the third row of Fig. 3) until maximum size of SCC is reached. Limiting the number of local SCCs let us keep their metadata in RAM which in turn enables fast determination of actions necessary to provide required data services. The target size of an SCC is a configuration constant (set usually well below 100 MB), so multiple SCCs can be read in the main memory. These SCC concatenations are loosely synchronized on all peers, so peer chains look the same. Similar operation is needed after deletion, shown in the remaining rows of this figure and discussed later in Section 5.3

This data organization is relatively simple in a static system, but it becomes quite complex due to the dynamic nature of the SystemXXL back-end. For example, when a peer is transferred to another physical node because of load balancing, its chains are transferred in the background to a new location, one SCC at a time. Similarly, after a supernode split, not all SCCs of the supernode are split immediately; instead we run background operations adjusting chains to the current supernode locations and shape. As a result, in any given moment, we may have chains partially-split, partially present in previous locations of this peer, or both. After failure, we may have serious holes in some of the chains. Fortunately, since peer chains describe the same data, we have the supernode cardinality chain redundancy in the system, so usually there is a sufficient number of complete chains. This chain redundancy allows for reasoning about the data in the system even in the presence of transfers/failures. Additionally, more refined algorithms are used in some cases constructing chain coverage from chain parts present on different peers.

5 Data Services

Based on the data organization described above, SystemXXL can efficiently deliver data services like identification of recoverability of data, automatic data rebuilding, load balancing, deletion and space reclamation, data location in network, deduplication and others. Detailed description of all of them is beyond the scope of this paper, but we sketch below how we deliver efficient data rebuilding, deletion and duplicate elimination.

5.1 Data Rebuilding

On node or disk failure, the SCCs residing there are lost. As a result, the redundancy of the data blocks with fragments belonging to these SCCs is at best reduced below the level requested by users when writing these blocks. In the worst case, a given block may be lost completely if not enough fragments survive. To keep the block redundancy at the desired levels, the system scans SCC chains looking for holes and schedules data rebuilding as background jobs for each missing SCC.

Multiple peer SCCs can be rebuilt in one rebuilding session. Based on SCC metadata, minimal number of peer SCCs needed for data rebuilding is read by a peer performing the rebuilding, and then erasure coding and decoding are applied to them in bulk to obtain fragments which should belong to rebuilt SCC(s). Next, the rebuilt SCCs are sent to the current target locations. Before SCCs are rebuilt, all input SCCs are made to look the same, i.e. required splits and concatenations are performed first. This requirement allows for fast bulk rebuilding as measured in Section 6.

5.2 Duplicate Elimination

Duplicate elimination can be classified in many dimensions: (1) the level at which duplicates are detected: an entire file, a subset of a file, fixed-size block or variable-size block; (2) time when the deduplication happens: inline when a duplicate is detected before it is stored, or in the background after it hits the disk; (3) how accurate it is: reliable in which if a duplicate of an object being written is present, it will be detected, or approximate, in which some duplicates may go undetected at a gain of better performance; (4) how equality of two objects is verified: by comparing secure hashes of two object contents; or by comparing data of these objects; and, last, but not least, (5) scope of detection: it can be local, re-

stricted only to data present on a given node; or global, i.e. using all data from all nodes.

Today SystemXXL implements variable-sized block, in-line, hash-verified global duplicate elimination implemented on storage nodes. For regular blocks, we use fast approximate deduplication, whereas for retention roots, we do reliable duplicate elimination to ensure that two or more blocks with the same search prefix point to the same blocks. In both cases, for successful deduplication, we require that the potential duplicate of a block being written has redundancy class not weaker than the class requested by this write; and that the potential old duplicate is reconstructible.

On a regular block write, the peer handling this write is selected based on the hash of this block; so two identical blocks written when this peer is alive will be handled by it and the second block will be found a duplicate of the first one. More complicated case arises when the write-handling peer has been recently created because of transfer or component recovery, and it does not have yet all the data it should have, i.e. its local SCC chain is not complete. In such case, we go to the longest-alive peer in the current supernode to check for possible duplicates. This is just a heuristics, as this peer may also not have the proper SCC chain complete, so a duplicate may not be detected. However, such miss occurs only in corner cases, after massive failures when most likely all chains are broken. Moreover, for a particular block, we miss only one opportunity to eliminate a duplicate; next identical block will be deduplicated unless more failures or this peer transfers happen, and only before chain rebuilding on the oldest peer is completed.

For retention roots, we need to ensure that two blocks with the same search prefix point to the same blocks (otherwise retention roots will not be useful to identify snapshots). As a result, we need an accurate duplicate elimination for retention roots. When a local full SCC chain does not exist at the peer handling this write, the peer sends duplicate elimination queries to all other peers in this supernode. Each of these peers checks locally for a duplicate. A negative answer includes also a summary description of parts of the SCC chain on which this answer is based. The write handling peer collects all replies. If there is at least one positive, a duplicate is found; otherwise, when all are negative, this peer tries to build full chain coverage using chain information attached to negative replies. If the entire SCC chain can be covered, the new block is not a duplicate; otherwise the write of the retention root is rejected with special error status indicating that data rebuilding is in progress (this may happen after massive failures); in such case this write should be submitted later. Needless to say, such situations so far happened only in special tests, and never in practice.

5.3 Deletion and Space Reclamation

Implementing data deletion in a system like SystemXXL turned out to be surprisingly difficult because of many challenges which stem from the nature of the system: content-addressability, distribution, failure tolerance, and duplicate elimination. While deletion in our content-addressable system is somehow similar to distributed garbage collection [24], which is well understood, overcoming remaining challenges, discussed below, required new research.

When deciding if a block is to be duplicate-eliminated against another old copy of this block, we must be sure that this old block is not scheduled for deletion. A decision which block to keep and which to delete must be consistent in the distributed setting and in the presence of failures. For example, a deletion decision made should not be temporarily lost due to intermittent failures, as otherwise we may eliminate duplicates using blocks which are really scheduled for deletion. Moreover, robustness of the data deletion algorithm should be higher than data robustness. We need this property because, even if some blocks are lost, data deletion should be able to proceed to logically remove the lost data and heal the system (obviously, only when such action is explicitly requested by a user).

To simplify the design and make the implementation manageable, we have implemented deletion split in two phases: read-only, during which blocks are marked for deletion and users cannot write data; and read-write phase, during which blocks marked for deletion are reclaimed and users can issue both reads and writes. Having read-only phase simplified the deletion implementation, because it lets us eliminate the impact of writes on the process of marking blocks for removal.

Deletion is implemented with per-block reference counter that counts the number of pointers in blocks in the system pointing to this block. Reference counters are not updated immediately on write. Instead, they are updated later in the read-only phase processing all pointers written since the previous read-only phase (so counter update is incremental). For each such pointer, the reference counter of the pointed block is incremented. After all such incrementation is completed, all blocks with reference counter equal to zero are marked for deletion (dark-shaded fragments in Fig. 3). Moreover, reference counters of blocks pointed by blocks already marked for deletion (including roots with associated deletion roots) are decremented. Next, the whole decrementation process (i.e. marking for removal blocks with reference counters equal to zero and decrementing reference counters of blocks pointed by pointers included in these blocks) is repeated, until no more new blocks can be marked for deletion. At this point, read-only phase

ends, and blocks marked for deletion can be removed in the background.

Deletion algorithm as described above requires metadata of all blocks as well as all pointers to be present to be able to proceed. The pointers and block metadata are replicated on all peers, so the deletion can proceed even if some blocks are no longer reconstructible, as long as at least one block fragment exists.

Since blocks are really kept as fragments, a copy of the block reference counter is kept per-fragment, and each fragment of a given block should have the same value of this counter. Reference counters are computed independently on peers participating in the read-only phase. Before deletion is started, each such peer must have its SCC chain complete with respect to fragment metadata and pointers. Not all peers in a supernode have to participate, but some minimal number of peers is required to complete the read-only phase. Computed counters are later propagated in the background to remaining peers.

The redundancy in counter computation allows deletion decision to survive node failures. However, the intermediate results of deletion computations are not persistent. Any failure before the decision is made wipes out these results on the affected nodes, and the whole computation needs to be repeated if too many peers cannot participate in this phase anymore; or deletion can still continue, if sufficient number of peers in each supernode were not affected by such failure. Upon conclusion of read-only phase, the new counter values are made failure-tolerant. All dead blocks i.e. blocks with counters equal to zero are then swept out from physical storage in background (reclamation in Fig. 3).

6 Evaluation

In our evaluations, we have used the current SystemXXL hardware. Each storage node (SN) runs one back-end server, and has six 500 GB SATA disks, 6GB RAM, two dual-core 3 GHz CPUs and two GigE cards. Some experiments have been done also with the experimental next generation hardware (denoted SN2), in which each storage node runs two back-end servers and has twelve 1 TB SATA disks, 20 GB of RAM, two quad-core 3GHz CPUs and four GigE cards. In all experiments, we have used the current access node (AN) with 6 GB RAM, two dual-core 3 GHz CPUs, two GigE cards and only a small local storage. All nodes run Linux version Red Hat EL 5.1.

All experiments were performed using 64KB block size, which were 33% compressible to 48 KB (except as noted).

6.1 Read/Write Bandwidth

This experiment shows write throughput as a function of fraction of blocks detected as duplicates for two different compression ratios. We have used 4 *SN2* machines, and 4 *AN* machines. Each AN runs one testing driver able to generate a stream of blocks with a specified percentage of duplicates and compression ratio. Duplicates are evenly distributed in the stream. Duplicated data is written in the same order as the base data, re-creating the original data stream. For the read experiment, testing driver attempts to read data in the same order as it was written.

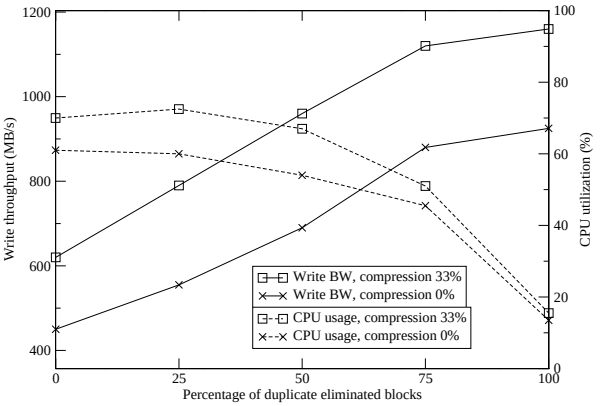

Figure 4: Write throughput as a function of duplicate ratio.

As shown in Fig. 4, very high bandwidth is achieved, which is a consequence of a carefully chosen data organization utilizing bulk transfer to disk. Duplicates are processed much more effectively than non-duplicated data, because they do not require fragmentation, fragment distribution and storage. Moreover, SCC-based organization allows the write-handling peer to perform fast local duplicate elimination by checking block reconstructibility with SCC reports submitted in the background from the remaining peers. However, when all writes are duplicates, the network bandwidth between AN and SNs becomes a bottle-neck, and the overall performance does not increase as much as expected (both curves flatten a bit at 100% duplicates). For high deduplication ratios, the CPU utilization decreases dramatically and network bandwidth between storage nodes remains available, so background tasks like data reconstruction and data scrubbing can be run without impact on user-visible performance.

Read bandwidth highly depends on factors like sequentiality of read data, and a number of drivers reading simultaneously, but it is not directly determined by the percentage of duplicates when the read data was

written (more precisely, for a given duplicate elimination ratio, read performance highly depends on granularity of distribution of duplicates). Detailed discussion of impact of these factors on read performance is beyond the scope of this paper. Instead, we only give read throughput achieved when drivers read data written sequentially. With four drivers reading, the total combined read bandwidth was 700MB/s for 33% compressible data and 500MB/s for 0% compressible data.

The time to fill the 4 SN2 node system highly depends on the ratio of duplicates in the data written: from 1 day for writing data with no duplicates, to up to 10 days of continuous writing with 95% of duplicates. In general, for configurations, in which high performance is not a priority, fewer ANs can be used, resulting in extended time-to-fill.

6.2 System Scaling

This experiment, with up to 12 *SN*s and the number of *AN*s set to half of the number of SNs, shows how performance is scaled when numbers of storage nodes and access nodes increase. Two set of measurements are done — a dynamic one, in which nodes are added while user is writing, and a static one, in which number of nodes is stable. In the latter case, each measurement was performed independently initializing system from scratch and loading the same amount of random, non-duplicated data. Time on X axis refers to the dynamic case only.

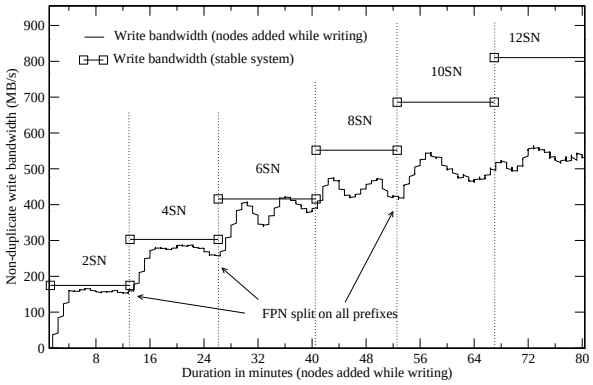

Figure 5: Dynamic vs. static scalability test.

The results indicate that in the range of nodes tested the system performance scales linearly with the system growth in the stable case. The system attempts to balance components so hash space is divided equally across storage nodes. Such balancing guarantees that every machine is equally loaded and does not become a bottleneck. In the dynamic case, there is a cost of system growth, so resulting user bandwidth is lower, because most of data is on the oldest nodes, and to check for duplicate elimination we need to query these nodes on every write. However, after all data transfers are completed, the performance in the dynamic case will be the same as in the stable case.

6.3 Node Failure and Data Rebuilding

This experiment shows the system behavior and its performance just after node failure, during resulting data reconstruction, and after the failed node is recovered. We have used 4 *SN2* machines and 4 *AN* machines.

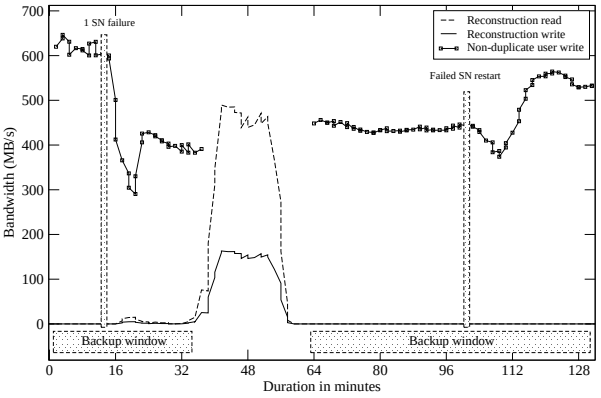

Figure 6: Node failure during backup.

We started writing to the healthy system with four storage nodes, achieving write throughput over 600 MB/s. After about 14 minutes one storage node failed (both back-end servers crashed). Write performance just after the node failure dropped to 300 MB/s, then stabilized at about 400 MB/s. The initial drop was caused by timed-out messages to the failed node and overhead for system rebalancing. Data rebuilding (reconstruction) tasks were ordered, however they were suppressed because of the ongoing user backup. Reconstruction started to work with full bandwidth just after all user writes had been finished. Every block reconstruction required read of 9 fragments to rebuild 3 lost fragments. The reconstruction read bandwidth reached 480 MB/s on 3 surviving machines, whereas reconstruction write bandwidth reached 160 MB/s. The rebuilding was finished in the 58th minute and the system became healthy once again, but containing only 3 storage nodes.

In the 64th minute the next writing session started achieving write bandwidth of 430 MB/s. The failed node was recovered and connected once again in the 100th minute. Just after the re-connection, system write bandwidth dropped to 380 MB/s, but when components rebalancing was finished it increased to about 550 MB/s. At the end of the experiment the system had 4 storage nodes, however it was not healthy, as not all data (SCCs) were

in the correct places. Write performance will increase to the initial (600 MB/s) after all pending transfers are finished and the system becomes healthy again.

The results show that the system maximizes user bandwidth during backup even if background tasks are pending. Such approach allows user to minimize costly backup windows regardless of internal system state, but carries the risk of starvation of critical data rebuilding tasks. However, this may happen only if the system is fully loaded by a user all the time and only when user writes non-duplicated data. If the user load decreases or some duplicates are written, reconstruction is executed in the background. Finally, this experiment also shows how quickly the system adjusts to changed environment. It is only a matter of minutes for the system to fully utilize all available resources.

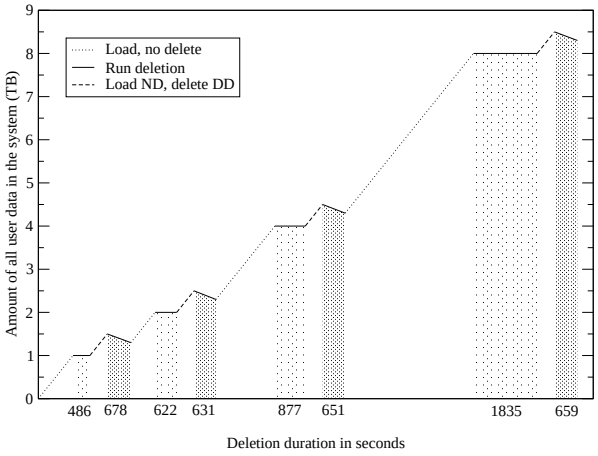

Figure 7: Read-only phase duration.

6.4 Data Deletion

The purpose of this experiment is to evaluate the duration of read-only phase as a function of data loaded. We focus here on the scenarios that resemble real usage of the system, therefore we use the file system interface to write to and remove data from the back-end periodically, with the overall amount of the data in the system increasing. Deletion experiments were performed using 4 *SN2* machines and 1 *AN* machine.

We repeat four incarnations of the following scenario consisting of four steps, as illustrated in Fig. 7. In the 1^{st} step (shown with dotted lines), the data is loaded doubling the total amount each time from 1 to 2 to 4 and to 8 TB. In the 2^{nd} step, we run read-only phase recomputing the counters for recently loaded data (no data is marked for deletion in this step). The duration of each 2^{nd} step is shown with light-gray bars in Fig. 7. After that, in the 3^{rd} step (shown with the dashed lines), additional half TB of new data (ND) is loaded and a user asks also for deletion of a 0.2 TB of older data (DD). In the last step, one more read-only deletion phase is run to recompute counters reflecting recently loaded data and mark the blocks to be deleted. The duration of each such read-only phase is shown with dark-gray bars. In all cases, the new data is not compressible and does not contain any duplicates, but with duplicates present the results will be similar, except that all phases will be shorter.

Although the X axis in Fig. 7 shows duration of each read-only phase, the data-loading steps are not shown in proportion there, because they are too big (we load terabytes of data and it takes several hours). We note that all read-only phases are relatively short, the longest one, after loading 3.7 TB of data (which took about 4 hours) is about 30 minutes, resulting in deletion time of under 13% of writing time. For writing with two ANs, this fraction can go up to 20% in case of not-duplicated streams, but for highly-duplicated data (which is the common case for backups), deletion takes significantly less, on the order of 5% of writing time, because less data need to be read to access all pointers, and filling in the capacity takes so much more time. Moreover, the duration of the first read-only phase (shown with the light-gray bars) in each incarnation is proportional to the new data loaded in the first step of the scenario. Finally, the duration of the second read-only phase (shown with dark bars) is more less constant for all 4 incarnations and is about 11 minutes. This shows also the power of the incremental reference-counting based deletion in SystemXXL, as clearly the duration of the read-only phase depends only on the amount of data added and deleted since the previous run of this phase, but not the total amount of data in the system.

7 Related Work

Significant number of distributed storage systems [10, 11, 12] targeted very large scale, distribution over wide-area network and use of untrusted peers leading to frequent configuration changes. For example, the goal of OceanStore [10] was to provide reliable storage for all data ever created. As a result, these systems concentrated on scalability (e.g. OceanStore, PAST [12]) and tolerance of wide class of failures, including Byzantine and large-scale correlated failures (Glacier [16]), at expense of performance.

Another group of distributed storage systems targeted the data center and, in this, are more like SystemXXL. These systems include distributed virtual disk like Petal [17], distributed file systems like CEPH [32] and Farsite [8], clustered file systems like Sorrento [29], Panasas [34], and GoogleFS [15], clustered storage in-

cluding Ursa Minor [7], RADOS [33], and FAB [23]. Compared to SystemXXL these systems have different target applications and are not advertised as secondary storage. As a result, they do not provide duplicate elimination (except Farsite, which does it on file level); these systems are not CAS-based, but need to deal with issues of consistency in the presence of write-sharing, which do not occur in our system. Ursa Minor does support user-selected choices of data redundancy, similar to our data redundancy classes. DISP [14] is a flexible system that can be specialized to both WAN and data center. Like SystemXXL, DISP uses erasure codes, but it does not provide duplicate elimination.

Venti [20], EMC Centera [6], Pergamum [27] and DataDomain [38] are secondary storage systems. Venti, Pergamum and Centera target archiving, whereas DataDomain is designed to store backup data. Pergamum does not support duplicate elimination, Venti prototype and Centera do it, respectively, on fixed block size and entire file level. These approaches result in lower deduplication than a variable-block size approach used by SystemXXL and DataDomain. However, DataDomain is a centralized system and does not do global deduplication in distributed environment. SystemXXL provides global deduplication using also variable block chunking with comparable write performance. RepStore [36], a smart-brick storage system built around DHT, uses erasure codes and content-based addressing, but does not provide deduplication. Deep Store [35], an archiving system, employs multitude of techniques for reducing stored data size, including delta compression and variable-block-size deduplication. However, this system does not target backup data.

Blocks in our system have some resemblance to objects in the object-based storage [19], as they have attributes (for example redundancy class) and simple interface to access its components like pointers.

Many systems introduce structures similar to SCCs for aggregating a number of blocks. Venti uses arenas to serve as a unit of data maintenance; however, they do not take advantage of the sequential nature of incoming data streams and achieve very low performance. DataDomain introduces containers to group sequential writes from each stream of data to increase effectiveness of read-ahead caching. SystemXXL achieves a similar result by sorting incoming blocks by their stream id and flushing them out to disk in batches. Using separate containers for every stream in SystemXXL is not feasible, as the number of containers written concurrently may be very large for big systems. SystemXXL data organization is unique in use of replicated chains of containers which allow for reasoning about state of the data in the system.

Deletion in a distributed storage system is relatively simple if there is no duplicate elimination. It can be done with leases like in Glacier [16], or with simple reclamation of previous obsolete versions like in Ursa Minor. However, with deduplication, deletion becomes difficult for reasons explained earlier. For example, Venti and Deep Store have not implemented deletion. As far as we know SystemXXL back-end approach to deletion is unique. Introduction of blocks with pointers, retention and deletion roots and redundant chains of containers enable us to implement easy-to-use fault-tolerant distributed deletion at the lowest part of the system.

8 Conclusions and Future Work

SystemXXL is a decentralized, scalable secondary storage system commercially available today. It can be used as on-line repository for all enterprise backup and archiving data, dynamically and efficiently sharing available capacity. Critical features like high availability and reliability, ease of management, capacity and performance scalability, and storage efficiency make the system unique in addressing today's enterprise needs caused by the explosive growth of data. The system is externally visible as one storage pool and can be accessed by legacy applications using traditional file system interface.

The core architecture is built around a DHT extended with virtual supernodes, each spanning multiple physical nodes. Data redundancy is provided with erasure codes, with fragments of erasure-coded blocks distributed among supernode components. Redundancy in the network and data allows for on-line upgrades and extensions increasing availability of the system. High storage efficiency is facilitated by variable block size global deduplication. The back-end exports low-level API providing operations on content-addressed blocks, with pointers to other blocks exposed. In this way blocks form a directed acyclic graph. A novel data organization based on redundant chains of data containers is used to deliver reliably multitude of data services, including failure-tolerant deletion and fast verification of data health.

Although the system is fully functional today, there is important work left to further improve its value proposition delivered to the end user. Read-only phase of deletion will be eliminated to make the system fully usable all the time. Deduplication can be improved with multi-size block techniques and moved to a proxy server at least in the common case, saving bandwidth and improving write performance of highly-duplicated streams. Additionally, since potentially multiple types of clients can access the back-end, there is a need for a stream interface in which cutting data into blocks is standardized by the back-end. This will ensure higher deduplication among data written by different types of clients.

References

[1] Reference omitted for blind review.

[2] Reference omitted for blind review.

[3] Reference omitted for blind review.

[4] Vtf open, 2005. http://www.diligent.com/products:VTF-Open-2.

[5] Overland storage unveils reo 9500d all-in-one deduplicating vtl appliance, 2007. http://www.overlandstorage.com.

[6] EMC Corp. EMC Centera: content addressed storage system, 2008. http://www.emc.com/products/family/emc-centera-family.htm?-openfolder=platform.

[7] ABD-EL-MALEK, M., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M. P., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa minor: Versatile cluster-based storage. In *FAST* (2005).

[8] ADYA, A., BOLOSKY, W., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. Farsite: Federated, available, and reliable storage for an incompletely trusted environment, 2002.

[9] AJMANI, S., LISKOV, B., AND SHRIRA, L. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)* (July 2006).

[10] BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., ZHAO, B., AND KUBIATOWICZ, J. Oceanstore: An extremely wide-area storage system. Tech. rep., Berkeley, CA, USA, 1999.

[11] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with cfs. *SIGOPS Oper. Syst. Rev. 35*, 5 (2001), 202–215.

[12] DRUSCHEL, P., AND ROWSTRON, A. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII* (Schloss Elmau, Germany, May 2001), pp. 75–80.

[13] EL MALEK, M. A., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, O., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Early experiences on the journey towards self-* storage. *IEEE Data Eng. Bulletin* (2006).

[14] ELLARD, D., AND MEGQUIER, J. Disp: Practical, efficient, secure and fault-tolerant distributed data storage. *Trans. Storage 1*, 1 (2005), 71–94.

[15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[16] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 143–158.

[17] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1996), pp. 84–92.

[18] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *In Proceedings of IPTPS02* (Cambridge, USA, March 2002).

[19] MESNIER, M., GANGER, G. R., AND RIEDEL, E. Object-based storage. *IEEE Communications Magazine 41* (2003), 84–90.

[20] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies* (Monterey,CA, 2002).

[21] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2001), ACM, pp. 161–172.

[22] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science 2218* (2001), 329+.

[23] SAITO, Y., FROLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev. 38*, 5 (2004), 48–58.

[24] SHAPIRO, M. A survey of distributed garbage collection techniques. In *In Proceedings of the 1995 International Workshop on Memory Management* (1995), Springer-Verlag, pp. 211–249.

[25] SOULES, C. A. N., APPAVOO, J., HUI, K., WISNIEWSKI, R. W., SILVA, D. D., GANGER, G. R., KRIEGER, O., STUMM, M., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., AND XENIDIS, J. System support for online reconfiguration, June 2003.

[26] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications* (2001), ACM Press, pp. 149–160.

[27] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–16.

[28] STRUNK, J. D., AND GANGER, G. R. A human organization analogy for self-* systems. In *In First Workshop on Algorithms and Architectures for Self-Managing Systems, in conjunction with Federated Computing Research Conference* (2003), pp. 1–6.

[29] TANG, H., GULBEDEN, A., ZHOU, J., STRATHEARN, W., YANG, T., AND CHU, L. A self-organizing storage cluster for parallel data-intensive applications. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 52.

[30] THERESKA, E., SALMON, O., STRUNK, J., WACHS, M., EL MALEK, M. A., LOPEZ, J., AND GANGER, G. R. Stardust: Tracking activity in a distributed storage system. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Saint-Malo* (2006), ACM Press, pp. 3–14.

[31] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), Springer-Verlag, pp. 328–338.

[32] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *OSDI'06: 7th USENIX Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 307–320.

[33] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *PDSW* (2007), G. A. Gibson, Ed., ACM Press, pp. 35–44.

[34] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., AND MUELLER, B. Scalable performance of the panasas parallel file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 17–33.

[35] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep store: An archival storage system architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 804–8015.

[36] ZHANG, Z., LIN, S., LIAN, Q., AND JIN, C. Repstore: A self-managing and self-tuning storage backend with smart bricks. In *ICAC* (2004), pp. 122–129.

[37] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22* (2004), 41–53.

[38] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–14.