RepStore: A Self-Managing and Self-Tuning Storage Backend with Smart Bricks

Zheng Zhang, Shiding Lin, Qiao Lian and Chao Jin
Microsoft Research Asia
{zzhang, i-slin, i-qlian, t-chjin}@microsoft.com

Abstract *– With the continuously improving price-performance ratio, building large, smart-brick based distributed storage system becomes increasingly attractive. The challenges, however, include not only reliability, adequate cost-performance ratio, online upgrades and so on, but also the system's ability to achieve these goals in as self-managing and self-adaptive a manner as possible. In this paper, we describe RepStore, a system that fulfills these goals. RepStore unites the self-organizing capability of P2P DHT and the completely autonomous, per-brick tuning mechanism to derive a scalable and cost-effective architecture.*

RepStore employs replication for active write-intensive data and erasure-coding for the rest, strives to achieve the best cost-performance balance automatically and transparent to application, and does so in a completely distributed manner. Our preliminary evaluations reveal that the system performs much as expected, achieving performance and reliability closer to a 3-way fully replicated system with only 60% of the cost.

1. Introduction

It is a long standing theme of the distributed system research community to replace high-end device with cluster of commodity components. To date, this design philosophy has flourished especially in NOW[1]-like clusters, powering world-wide Web engines as well as high-performance computing Grids.

The same revolution is unfolding in the field of backend storage as well. Traditionally, this is a territory of high-end disk arrays. Combined with the iSCSI (SCSI over IP) protocol, the availability of hardware-assisted TCP/IP offloading, and the increasingly improving price/performance ratio of commodity PCs, network attached storage made by the so-called *smart storage bricks* [3][8][11][12] is being actively researched and developed. These systems are disruptive technologies: they do not necessarily offer the same level of performance as the next higher ranked technologies right away, but deliver enough value to be adopted. As the component technologies continue to improve, eventually they will take an increasingly larger market share from disk array. The system described in this paper, RepStore, falls into this line of research.

Smart brick based systems such as RepStore must accomplish a set of goals in order to be successful. They must be extremely reliable, while simultaneously offering adequate cost-performance ratio. Being built with LAN-ready bricks, the architecture should be scalable and flexible such that online capacity provisioning as well as incremental upgrade can be achieved. Furthermore, changing nature of workloads implies that the system needs to be self-adaptive. The ultimate challenge, however, is to do all the above in as self-managing a manner as possible, which is of paramount importance because human-based management is the largest factor in the TCO(total cost of ownership) of such system.

Some of the above functionalities have been accomplished inside state-of-art disk array, one example is the self-adaptive capability of HP AutoRAID[24]. AutoRAID is hierarchical, using mirroring for write-intensive data and RAID5 for the rest. This is a single-box solution and uses metadata to control the placement of data among disks comprising the disk array. We believe that brick-based distributed storage should offer the same self-adaptive capability.

RepStore leverages the recent advancement of the peer-to-peer technologies to derive many of its self-managing characteristics. Layering over DHT (distributed hash table)[16][19][22][25], RepStore presents a large, abstract storage space that is populated by participating bricks. The self-organizing capability of DHT grants RepStore the potential to handle both failure and online provisioning gracefully. Just like HP AutoRAID, RepStore employs replication for active write-intensive data and erasure-coding for the rest, strives to achieve the best cost-performance tradeoff automatically and transparent to application, and does so in a completely distributed manner.

We have completed the preliminary design and our detailed evaluation verifies that RepStore performs as expected. Using trace-driven simulation, our results show that we can achieve the same level of performance and reliability as a 3-way replicated storage, while with only 60% of the cost. Furthermore, the system is robust enough to deal with changes of workload.

The remaining of this paper is organized as follows. Section-2 gives a brief introduction of P2P DHT. The main architecture of RepStore is described in Section-3. Section-4 performs workload studies to establish the foundation of the self-tuning aspect of RepStore. Performance data is presented in Section-5. We cover related works in Section-6 and conclude in Section-7.

2. Background: P2P DHT

The common terminology referring to a peer in DHT is "node", which we will adopt in this section. In the RepStore context, it will be a smart brick.

There exist many different DHT proposals [16][19][22][25], but they all share a few common invariants. In DHT, nodes join a very large (e.g. 160-bits) space with random ids and thus partition the space uniformly. The id can be, for instance, MD5 over a node's IP address. An ordered set of nodes, in turn, allows a node's responsible *zone* to be strictly defined. Let p be a node x's predecessor. One definition of a node's zone is simply the space between the ID of its immediate predecessor ID (non-inclusive) and its own ID. In other words: $zone(x) \equiv (ID(p), ID(x)]$. This is essentially how consistent hashing [22] assigns zones to DHT nodes (Figure 1).

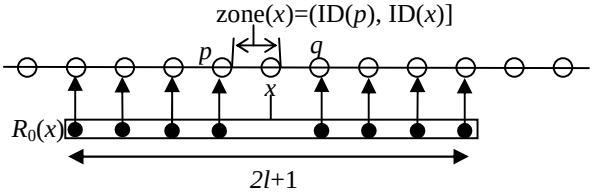

Figure 1: The simplest P2P DHT – a ring, the zone and the basic routing table that records r neighbors to each side.

If one imagines the zone being a hash bucket in an ordinary hash table, then the ring is a *distributed hash table*. Given a key in the space, one can always resolve which node is being responsible. This numerical space is what RepStore as well as other DHT-based storage systems use to present the upper-layer application as a storage space. Objects are keyed with randomized id of the same length (e.g 160 bits), and dropped onto the node that covers their keys. Therefore, storage utilization of all nodes are uniform (statistically speaking), as long as the storage capacities are about the same across nodes. When a new node arrives at the system, it will split zone with the one who covers its id. Likewise, when a node departs, its zone is taken over by its immediate neighbor. Thus, membership change in DHT only involve object redistribution amongst O(1) nodes, and is ideal to implement a scalable and flexible storage system.

To harden the ring against system dynamism, each node records l neighbors to each side in the rudimentary routing table that is commonly known as *leafset*. Neighbors exchange heartbeats to keep their leafsets up to date. A ring is the simplest P2P DHT, whose lookup performance is O(N), where N is the number of nodes in the system. Elaborate algorithms built upon the above concept achieves O(log$N$) performance with O(log$N$) states. Representative systems include Chord[22], CAN[16], Pastry[19] and Tapestry[25]. RepStore is layered over a DHT called XRing[26], which we have developed to target environments where churn rate is low and is an ideal fit for RepStore.

XRing uses O(N) state to bring 1-hop lookup performance, i.e. routing from any node to a point in the space takes one network hop to the destination. XRing achieves this by first building a O(log$N$) routing table, called *fingers*, using prefix-based routing scheme much like Pastry. A node then uses fingers to broadcast any membership change event (node addition and/or deletion) that it has observed in its leafset. These notifications propagate with O(log$N$) latency bound and reach all nodes with extremely high reliability, allowing every node to build another layer of routing table that records all other nodes. We emphasize here that O(N) state is not an issue: if each routing entry is 32bytes, a 1M-node XRing will require 32MB memory per node even if the routing table is kept completely in memory. If each node is 100GB, this amounts to a system with 100PB capacity, an extremely large system.

3. RepStore architecture

3.1. Objective and system model

The target deployment context of RepStore is data center and/or enterprise internal. The overarching objective of RepStore is to achieve the best cost-performance tradeoff, and does so with as little administration oversight as possible. Specifically, given a total storage capacity constraint (and hence total hardware cost), we would like the system to offer the best response time automatically. Management of RepStore should involve very little other than decommissioning failed components and adding new ones in response to capacity and performance need, and all such tasks should be performed online with minimum performance disturbance. To state it differently, the ambition of RepStore is to replicate the functionality of HP AutoRAID, but to do it with a farm of smart storage bricks instead of inside a disk array. The significant challenge is that RepStore has to accomplish this without any centralized control.

For the time being, RepStore exposes an object interface and is posed as a storage layer that performs between high-end disk array and tape library, offering high-availability guarantee no less than the disk array. However, if outfitted by advanced technologies (e.g. high-performance SAN, large NVRAM per drive etc), we would like RepStore to challenge disk array's performance as well. On the other hand, in terms of capacity, with the continuously improving byte per dollar ratio of commodity disk drives, the other

possibility is for RepStore to be the tape library replacement. RepStore *applications* may include distributed file system, volume manager or other new ones. These applications can be layered on one common RepStore instance if desired, since RepStore is agnostic to what gets stored inside.

We assume a failure-stop model, and that failure is rare and independent. In reality, these assumptions are true only to a certain extent, for instance individual disk drive crash.

3.2. Architecture and protocol

RepStore approaches its goal by leveraging the self-organizing properties of P2P DHT and embeds within each brick self-monitoring and feedback-loop based optimization mechanisms to tune storage towards better cost-performance balance. Unlike other brick-based systems, RepStore is stringed together by using XRing as the underlying one-hop DHT routing infrastructure. As discussed earlier, using P2P DHT as the bottom-layer lends us with the desired property of self-organizing as well as ease of online provisioning.

3.2.1. Addressing space and data layout

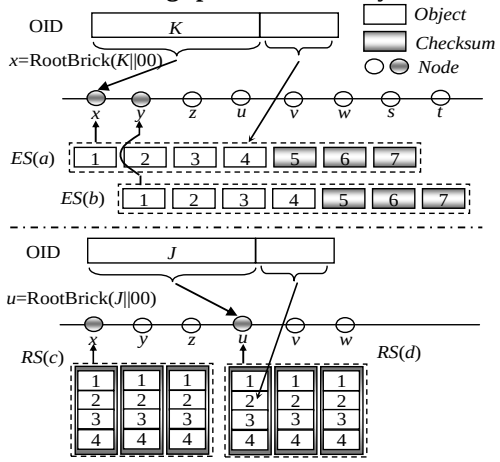

Figure 2: Data layout for RepStore. ES and RS denote Erasure-coded and Replicated set, respectively. Four sets are shown (a, b c, and d). We depict how set may be in either replication or erasure coding.

RepStore presents an abstract storage space of 160bits, and each brick joins RepStore using the XRing join protocol with a random id. As a result, the total space is (statistically speaking) uniformly partitioned, as in any such DHT-based storage system(pond[17], PAST[20]). A brick x owns a portion of the total space, called its *zone*. Specifically, $x.zone \equiv (pred(x).\text{id}, x.\text{id}]$, where $pred(x)$ is the brick whose id immediately proceeds that of x. Using XRing's routing function, any brick can lookup any key in the space – and hence reach any other brick, by one network hop.

An object stored inside RepStore is identified by an OID of 160bits. The OID can be assigned by RepStore applications such as volume manager, or is a hash over the object's content to implement an immutable archival system. This ensures that all bricks will have same storage utilization. This property may not hold if bricks have different capacities. In this case, the idea of virtual node can be borrowed: a physical brick is broken down into a number of virtual bricks of a fixed size, and then each joins XRing separately.

To achieve the best cost-performance tradeoff, a RepStore object is stored in either one of the two *coding schemas*: replication and erasure coding. For replication, r identical replicas are stored, where r is an odd integer. With erasure coding, an object is combined with m-1 other objects, and then generates n-m check objects. Therefore, the storage overhead of replication and erasure coding is r and n/m, respectively. We choose the three parameters r, m and n such that the HA guarantee for a given piece of data is about the same to tolerate the same number of concurrent brick failures. The default value that we use is 3, 4 and 7 (refer to [27] for the calculation of default parameters).

These two coding schemas provide different performance characteristics, as will be explained shortly. RepStore manages the coding schema of any object automatically. In order to achieve full transparency with minimum performance impact, RepStore adopts a data layout as depicted in Figure 2. Briefly speaking:

- m objects whose OIDs differ with only the last log(m) bits are grouped together into a set, setkey of which is the OID with last log(m) bits being zero.
- The brick that owns the setkey in the total storage space is called the *root brick* of objects in the set. The request to objects will be forwarded to root brick and the last log(m) bits of OID decides its offset in the set.
- A replicated set, or *RS* in short, is spread in r consecutive bricks, starting from the root brick.
- An erasure coded set, or *ES*, is spread in n consecutive bricks, again starting from the root brick. Normally, the i-th (i..[0..n-1]) brick away from the root brick stores the i-th object in the set, with the last n-m bricks store the checked objects. However, repair done in response to online provisioning (adding new brick) or decommissioning may change that order.

This layout is particularly helpful to store conventional disk blocks for a volume manager, though other alternatives exist as well. For example, in the case of a file system, a file can be broken down into m

fragments, and same layout arrangement as above is used for all fragments.

3.2.2. Access protocol

An access to an object always starts from its root brick. This is how changing code schema is made transparent to applications, since the root brick is determined by the setkey, which relies on OID only. The root brick also functions as the serialization point for operations that would manipulate state and/or coding of the set, and hence gives us a much simpler design.

Request arrives at the root brick by using the setkey as the lookup key to route through XRing. This shall always succeed (since the DHT space does not have hole) and with high probability be done with one network hop. What happens next depends on the operation as well as the coding schema of the object:

- Replicated object. Read will retrieve the object from the set and return right away. Write employs a 2-phase commit to update the other replicas.
- Erasure-coded object. The root brick keeps a map which tells what brick among the *n* bricks is responsible for which objects in the set. The map is built and maintained on-demand: the first time the set is formed, and subsequently updated when changes occur in the brick membership (we will describe this shortly). Thus, a read request is forwarded to the brick that keeps the requested object. A write operation will retrieve all the check objects as well as the object being updated, and then update all of them, again using a 2-phase protocol. This amounts to *n-m*+1 reads and *n-m*+1 writes.

What we just described is failure-free cases. Failures are handled by operating over the redundancy afforded by the coding schemas, which we will not discuss further here and will refer to our full report[27].

3.2.3. Self-tuning

Having described the operation, it is now easy to see how the two coding schemas afford us with the flexibility to play with the cost-performance tradeoff. We have already described that we have chosen parameters such that the two offer the same level of HA. The capacity overhead of replication versus erasure code is *r* versus *n/m*, for instance 3:(7/4). On the other hand, the two differ dramatically on performance. Once a read request arrives at the root brick, either coding schema entails one disk access, but erasure coded object may require one more network hop if the object is not in the root brick. As in RAID, erasure coded objects are more expensive serving writes: 2(*n-m*+1) read and writes. Whereas a replicated object only requires *r* writes. More disk accesses will build longer queues at the disk drives, further degrading performance.

Therefore, the main thrust of RepStore is self-optimizing to constantly tune towards the best cost-performance tradeoff. This is accomplished by tracking the workingset and using replication to code the hot and write-intensive data and erasure coding for the rest. How necessary statistics are gathered and when the tuning is triggered is the focus of Section-4. Regardless, self-tuning employs one primitive that first creates a new set with the target coding schema, and then initiates a transaction that allow the old set to be garbage collected later. The root brick is responsible for coordinating all the above processes. Our current implementation admits read access while these changes are ongoing but defer write requests. A more advanced implementation will allow write to proceed concurrently as well, similar to online volume migration.

3.2.4. Failure-handling and online provisioning

The brick membership in RepStore may change as a result of taking out dead (or old) brick and/or adding new ones. The ability to do both online allows the system to grow gradually with new breed of bricks. Therefore, this part of design is integral to RepStore's self-managing capability.

The basic principle is to ensure that the replication set and erasure-coded set adhere to their invariants. Due to space limitation, we refer readers to our technical report for more details [27] and will only give a brief outline here.

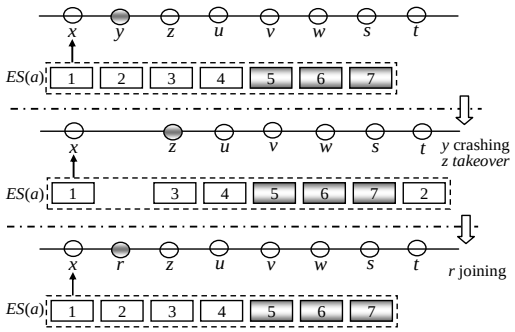

Figure 3: Handling failure and addition in RepStore.

The HA invariants include two conditions:

1. A set is stored starting from its root brick (i.e. the brick that owns the setkey).
2. For a replication set and erasure-coded set, they are spread in consecutive *r* and *n* bricks (refer to Figure 3), respectively.

Membership change is a basic service provided by XRing, since XRing is a one-hop-anywhere DHT. Whether the repair is done right away is a policy issue.

Enforcing the first invariant is straightforward. In case of new brick join, sets that should be rooted at it are copied over. If it is for a brick crash, then the neighbor brick on the right now becomes the new root brick for any sets that were rooted at the departed brick. Note that each brick can calculate what sets are rooted at it completely independently.

When this is done, the second invariant is reinstituted. This will involve r and n bricks for replicated sets and erasure-coded sets, respectively. Whatever the course, there is always one brick who should now host some data of the set affected. This is performed through the root brick, which instructs the copy (for brick addition) or reconstruction of a piece of data (for brick deletion) to that new brick. For brick addition, there is also one brick gets excluded from storing some data of affected sets, and the root brick instructs that brick to garbage collect these data as well. Figure 3 shows the end results of an exemplary brick crash and addition. Since the repair is serialized through the root brick, the all brick-to-object map is updated for the set being repaired.

The more complex part is to allow concurrent access to data when repairs are being made. This is not a problem for read access because the read protocol is fault-tolerant to begin with assuming enough redundancy exists to reconstruct the data. Care must be taken to handle write. Our approach is to trigger the repair of set being written right away.

4. Workload analysis

As discussed earlier, RepStore tries to achieve the best cost-performance tradeoff by devoting higher storage overhead to write-intensive workingset. For this to work, the necessary conditions must be that A) the workingset is generally small and B) it does change but changes relatively infrequently. Without meeting the first condition, mixing coding schemas with different cost/performance tradeoffs can not possibly work. Likewise, without the second, tuning is either not necessary or brings too much overhead and wipes out the benefits.

Past research work on storage systems have verified that the working set size is usually small [21]. Thus, condition A is generally met. Intuitively, condition B should hold as well. However, empirical evident are lacking. Here, we present the methodology as well as the results that support both conditions.

Let a trace of application I/O access be divided with a fixed time window. Further, let vector $V(w_i)$ record unique objects accessed in the i-th window w_i. $V(w_i)$, called the *workingset vector*, is thus a representation of workingset in the i-th window. The size of the vector is the total number of objects ever accessed in the entire trace. $V(w_i)[j]$ is 1 if the j-th object is accessed in window i, and 0 otherwise. The *weighted workingset vector* $V^*(w_i)$ differs from $V(w_i)$ only in that each element records the number of accesses to the associated object, instead of a binary flag. The method with which we understand the workingset change is through looking at correlations of these vectors. For instance, to understand the change between window i and j, we can compute the normalized dot product between $V(w_i)$ and $V(w_j)$, or between $V^*(w_i)$ and $V^*(w_j)$ for a different perspective.

We processed two traces: the cello disk trace from HP[21] and RES file system trace from Berkeley[18]. Cello trace is gathered at server side of a timesharing system at disk block level for a one month period. Berkeley RES trace is collected from client side of 13 HP 7000 workstations on file level for half a year. We have investigated different time window; the results presented here uses the window size of a day.

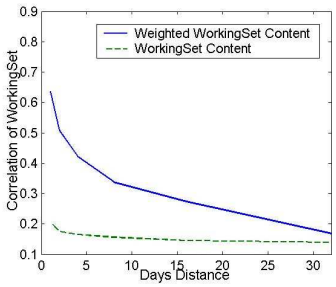

Figure 4: Workingset correlation curves for the Berkeley trace on a daily granularity. Solid line is for active workingset.

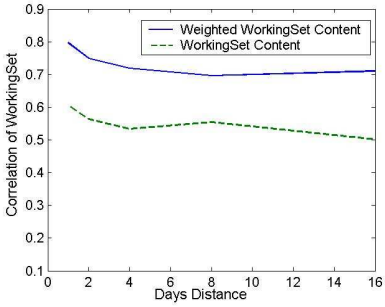

Figure 5: Workingset correlation curves for the cello trace on a daily granularity. Solid line is for active workingset.

For a given trace, we compute the correlation between $V(w_i)$ and $V(w_{i+d})$, where d is 1, 2, 4, 8… till K. K is 32 and 16, for the Berkeley and cello trace, respectively. We average through all possible i for each value of d and present a K points curve, called the *correlation curve*. We then repeat the same but use the weighted workingset curve $V^*(w_i)$ instead. The results are presented in Figure 4 and Figure 5, for Berkeley and cello respectively.

Focus first on the Berkeley case. The correlation curve for the workingset vector is flat and low, indicating that the daily footprint is rather random across time. The active workingset, represented by the correlation curve of the weighted workingset vector, is rather different. It is very high for small *d* and then decreases when *d* gets larger. Recall that *d* is the distance between two vectors. This clearly suggests that the active workingset does change over time, and changes slowly (i.e. hourly tuning will be sufficient). The cello trace shares with the Berkley trace to the extent that correlation of the active workingset is stronger and that it also decreases with longer time span, but the curve is flatter and that the correlation of unweighted workingset is pretty high (>0.5) as well.

Confirming with prior findings, the size of a daily workingset is about 10% and 15%, for Berkeley and cello respectively. Furthermore, 90% of the accesses are to the 8% and 1% of the daily workingset. These, combined with the analysis over the correlation curves, indicate that the RepStore approach is likely to be effective.

5. Experiments

In what follows, we will collectively call the total capacity dedicated to replicated data and erasure-coded data as *hot space* and *cold space*, respectively. Intuitively, the performance will continue to rise as hot space increases, until a "knee" after which there will be no more significant improvement. The larger the hot space, on the other hand, means higher storage cost; whereas when the hot space is below some critical value, changing coding schema may be triggered too often such that we will see a negative performance hit – to the point that we might as well freeze the system into a static configuration. Thus, adequate tuning policy is necessary.

We have studied a variety of different policies. Due to the space limitation, we only report the policy LRU-W (Least-Recent-Used-Write) which is used in HP AutoRAID. In RepStore, this tuning algorithm is employed in each brick for all the sets that root on it based on its local statistics. Specifically:

- Nothing is done upon a read request.
- Upon a write request, if the set is already replicated, update its recency; otherwise it is re-coded into replication set. In the latter case, tuning is triggered.
- At the tuning time, if the size of hot space exceeds a given quota (e.g. not more than 5% of total unique blocks can be in replication form), the oldest written set is re-coded back into erasure.

Obviously, this policy tilts heavily towards the active and write-intensive workingset. This makes sense because write to erasure-coded set is more expensive in terms of number of I/O, minimizing which, in turn, will reduce queue length buildup at disk drives. However, as we shall see, it pays a penalty for read requests.

All performance results are based on replaying the cello trace against an event-driven simulator that models RepStore's protocols in sufficient details. Since the total footprint of the trace is small (10GB), total number of bricks is only 32. The parameters for the coding schema are: r=3, m/n=4/7. Thus, a set consists of 4 blocks. Also, sets are randomly distributed to bricks. We assume network latency is negligible, and the main cost is disk latency which we set as *6ms*. Requests to a disk are served in sequence. This is rather simplistic, however we believe the results should be accurate enough to draw the conclusion.

5.1. Overall performance

The first group of experiments is to measure the number of disk accesses, average latency of requests with different settings of hot space size. We use the trace file in 920504 of cello, in which the workingset of that day amounts to more than 25% of the total footprint.

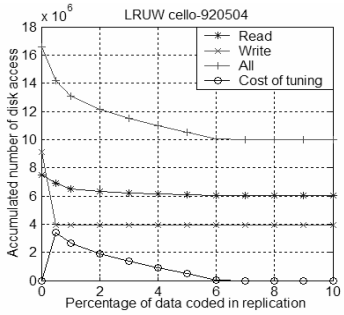

Figure 6: Number of disk access versus percentage of data coded in replication.

Figure 6 shows how the number of disk IO changes with the hot space size. As expected, the LRU-W affects mostly the accesses corresponding to writes and tuning (i.e. the cost of tuning; shown as "tuning" in the figure), both decreases as hot space increases and the knee is 6% (i.e. 6% of data is coded in replication), at which point the algorithm has captured the write-intensive blocks in the active workingset. The number of accesses corresponding to read request, on the other hand, hardly changes. Disk accesses due to tuning also decrease. The exception is when the hot space is 0%, at which point there is no tuning because all objects are erasure coded.

Figure 7 takes a different perspective by counting number of IO per request type. When hot space is 0%, all data are erasure coded. So, each write request involves 7 times of disk IO. The algorithm effectively

codes write-intensive blocks in active workingset, as the same number drops rapidly down to 3 (the size of replicated set) while adding space for replicated data. In cello, a read request typically accesses a number of consecutive blocks. Since read-intensive data are mostly erasure-coded and are coded with replication only when it is written, the number of IO per read decreases very slowly (from 3.92 to 3.16), which roughly indicates that, at that point, 26% of read requests hits in hot space, i.e. 74% of read data are never written (for that day at least). As for tuning cost, the total access is composed of one access reads the object, and 7 for writing the new erasure-coded set. The original set will be garbage collected later at idle time so their effective cost is zero.

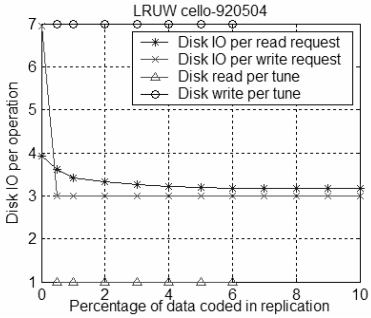

Figure 7. Disk IO per operation versus percentage of data coded in replication.

Obviously, this policy heavily tunes towards minimizing total number of IO, especially for writes. The question is whether it pays off for the final performance measure, namely the latency.

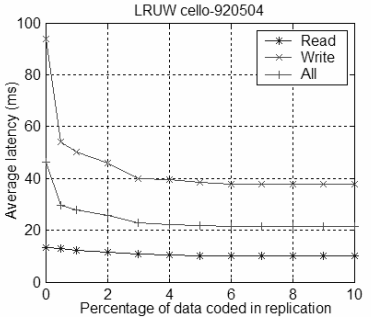

Figure 8. Average latency per request versus different size of hot space percentage of data coded in replication.

Figure 8 shows that the overall performance in terms of latency is rather adequate. With 3% data coded in replication, the latency for write and overall requests are reduced to 43% and 46% comparing with all erasure-coding, respectively.

From Figure 6, Figure 7 and Figure 8, we find that, when we code 6% of data in replication, which corresponds to a capacity saving of 40% comparing to a 3-way replication, the overall performance in terms of latency is very close to optimal. We will now take this configuration and examine how the system performs along the temporal dimension.

5.2. Robustness and responsiveness

Next group of experiments takes a one-month cello trace (May 1992) as input to test the efficacy of the tuning algorithm over a long run.

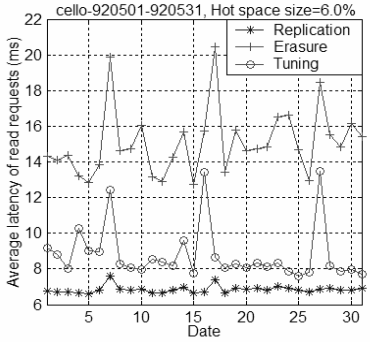

Figure 9: Average latency of read requests.

Figure 9 shows that the read performance after tuning is much better than that of fully erasure mode, though about 10% worse than optimal. This is because the tuning algorithm cares about write requests much more than read ones. It would be interesting to perform some optimization for read requests, e.g. cache the data in root brick or use some more sophisticated tuning policy.

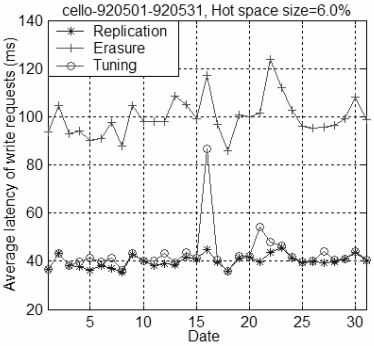

Figure 10. Average latency of write requests.

Figure 10 shows that the write performance after tuning is very close to that of fully replicated mode. This, in turn, means that the system is rather responsive. However, there are a few days that the algorithm does not work that well, we are in the process of understanding the cause.

6. Related work

One of the main P2P DHT applications has been wide-area distributed storage service, with pioneer works including OceanStore[15], Pond[17], CFS[7], PAST[20] and Pastiche[6]. Though recent work of [4] argues that, due to unpredictable nature of WAN connectivity, some fundamental compromises have to

be made. RepStore defines its architecture for enterprise-internal, and aims at pushing the state of art for P2P DHT-based storage backend by dramatically reducing the management overhead while retaining the best cost-performance tradeoff. Being layered over XRing without assuming the availability of IP-level multicast, however, there is nothing preventing RepStore from being deployed in a wide-area context. In fact, we believe that the lessons gained by designing a robust and adaptive storage in a controlled, LAN-based environment will be invaluable before taking that ambitious step into wide-area.

The philosophy of designing a storage system with a sparsely populated storage space starts from Petal[14] and Frangipani[23]. In the brick-based system, NASD[11], FAB[8], IceCube[13], and Google File System[10] all aim at building extremely scalable brick cluster with high throughput. However, reducing management overhead in a self-optimized and self-tuned manner is not their particular focus.

The idea that storage should be self-adaptive starts from the exemplary work of HP AutoRAID and IStore[5]. These are single-box solutions. RepStore replicates the functionality of HP AutoRAID while expanding it to a brick-based distributed storage without any centralized metadata. The objective of self-managing storage farm has been well articulated in Self-*[9], WiND[3] and Hippodrome[2]. We believe that the approach of RepStore where we leverage the self-scaling and self-managing property of P2P DHT is an interesting alternative.

7. Conclusion and future work

While building smart-brick based storage backend is potentially cost-effective, it is of paramount importance to have an architecture that not only is reliable, scalable and flexible, but also delivers these goals in a self-managing and self-tuning manner. We believe that leveraging the self-organizing strength of P2P DHT is an interesting and important alternative. RepStore is our attempt to utilize the advent of P2P technology to derive a more practical system.

Acknowledgement

The authors wish to thank the anonymous reviewers and their insightful feedback, and all members of the System Research Group of MSR-Asia for their support.

8. Reference

[1] T.E. Anderson, D.E. Culler, D. A. Patterson, et al, “A case for networks of workstations: NOW”, IEEE Micro, Feb. 1995.

[2] E. Anderson, M. Hobbs, K. Keeton, et al, “Hippodrome: running circles around storage administration”, USENIX FAST'02, Monterey, CA.

[3] A. Arpaci-Dusseau, R. Arpaci-Dusseau, et al, “Manageable Storage via Adaptation in WiND”, 2001 IEEE CCGrid'01, Brisbane, Australia.

[4] C. Blake, R. Rodrigues, “High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two”, HotOS 2003.

[5] A. Brown, D. Oppenheimer, K. Keeton, et al, "ISTORE: Introspective Storage for Data-Intensive Network Services", HotOS-VII, Rio Rico, Arizona, March 1999.

[6] L.P. Cox, C.D. Murray, and B.D. Noble, “Pastiche: Making Backup Cheap and Easy”, OSDI '02, Boston, MA, Sep., 2002.

[7] F. Dabek, M.F. Kaashoek, D. Karger, et al, “Wide-area cooperative storage with CFS”, SOSP 2001.

[8] S. Frolund, A. Merchant, Y. Saito, et al, “FAB: enterprise storage systems on a shoestring”, HOTOS, May 2003, Kauai, Hawaii.

[9] G.R. Ganger, J.D. Strunk, A.J. Klosterman, “Self-* Storage: Brick-based Storage with Automated Administration”, Published as Carnegie Mellon University Technical Report, CMU-CS-03-178, August 2003.

[10] S. Ghemawat, H. Gobioff, S.T. Leung, “The Google File System”, 19th ACM SOSP, Bolton Landing, NY, USA, 2003.

[11] G.A.Gibson, Nagle, D.F., Amiri, K., Butler, et al. “A Cost-Effective, High-Bandwidth Storage Architecture”, ASPLOS, October, 1998.

[12] J. Gray, “Storage Bricks Have Arrived”, invited talk FAST 2002.

[13] IBM. IceCube: storage server for the Internet age. http://www.almaden.ibm.com/StorageSystems/autonomic_storage/CIB/

[14] E.K. Lee and C.A. Thekkath, “Petal: Distributed Virtual Disks”, ASPLOS 1996.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, et al, “OceanStore: An Architecture for Global-Scale Persistent Storage”, ASPLOS 2000.

[16] S. Ratnasamy, P. Francis, M. Handley, et al, “A Scalable Content-Addressable Network“, ACM SIGCOMM 2001.

[17] S. Rhea, P. Eaton, D. Geels, et al, “Pond: the OceanStore Prototype”. Proceedings of the 2nd USENIX FAST '03, March 2003.

[18] D. Roselli, J.R. Lorch and T.E. Anderson, “A comparison of file system workloads”, USENIX Annual Technical Conference, San Diego, CA, June 2000.

[19] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", IFIP/ACM Middleware, Heidelberg, Germany, November, 2001.

[20] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", 18th ACM SOSP'01, Lake Louise, Alberta, Canada.

[21] C. Ruemmler and J. Wilkes, “A trace-driven analysis of disk working set sizes”, HP Laboratories Technical Report HPL-OSR-93-23 (April 1993).

[22] I. Stoica, R. Morris, D. Karger, et al, “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”, ACM SIGCOMM 2001, San Deigo, CA.

[23] C.A. Thekkath, T. Mann, and E.K. Lee, “Frangipani: A Scalable Distributed File System”, ACM SOSP 1997.

[24] J. Wilkes, R. Golding, C. Staelin, et al, “The HP AutoRAID hierarchical storage system”, ACM TOCS Volume 14 , Issue 1, 1996.

[25] B.Y. Zhao, J. Kubiatowicz, and A.D. Josep, “Tapestry: An infrastructure for fault-tolerant wide-area location and routing”, Tech. Rep. UCB/CSD-01-1141, UC Berkeley, EECS, 2001.

[26] Z. Zhang, Q. Lian, Y. Chen, et al, “XRing: Achieving High-Performance Routing Adaptively in Structured P2P”, Submitted for publication.

[27] Z. Zhang, S. Lin, Q. Lian, et al, “RepStore: A Self-Managing and Self-Tuning Storage Backend with Smart Bricks”, Microsoft Research, technical report, MSR-TR-2004-21.