Design of an Exact Data Deduplication Cluster

Jürgen Kaiser, Dirk Meister, Andre Brinkmann
Johannes Gutenberg-University, Mainz, Germany
{j.kaiser, dirkmeister, brinkman}@uni-mainz.de

Sascha Effert
Christmann informationstechnik + medien, Ilsede, Germany
sascha.effert@christmann.info

***Abstract*—Data deduplication is an important component of enterprise storage environments. The throughput and capacity limitations of single node solutions have led to the development of clustered deduplication systems. Most implemented clustered inline solutions are trading deduplication ratio versus performance and are willing to miss opportunities to detect redundant data, which a single node system would detect.**

We present an inline deduplication cluster with a joint distributed chunk index, which is able to detect as much redundancy as a single node solution. The use of locality and load balancing paradigms enables the nodes to minimize information exchange. Therefore, we are able to show that, despite different claims in previous papers, it is possible to combine exact deduplication, small chunk sizes, and scalability within one environment using only a commodity GBit Ethernet interconnect. Additionally, we investigate the throughput and scalability limitations with a special focus on the intra-node communication.

I. Introduction

Tape systems have been the first choice to build backup environments for a long time, as the costs per capacity have been and still are lower than the same costs for magnetic disks. This price difference has been sufficient to outweigh the drawback of slow access and rebuild times, as the capacity of a backup system is typically much bigger than the capacity of the corresponding online storage, making cost efficiency the primary objective.

Data deduplication have turned the cost advantage from tape storage to magnetic disks. Deduplication exploits the inherent redundancy in backup environments as typically only a very small percentage of information is changed between two successive backup runs [1]. Deduplication environments are able to detect identical [2], [3] or very similar [4] blocks on a fine-granular level, so that redundant information has to be stored only once.

Deduplication technology is restricted to random-access media and cannot be efficiently used based on tape technology, as a backup is no longer a sequential data stream on the target medium. Its cost effectiveness combined with the enormous data growth and higher demands on backup window lengths and recovery times lets deduplication on magnetic disks become an important component of enterprise storage systems.

The success of data deduplication appliances can be compared with the success of network file servers some years ago. In the beginning, users experience simplified and more cost efficient backups. Afterwards, they need more than a single deduplication appliance and have to build up an infrastructure of single-node appliances. This infrastructure is difficult to manage and efficiency decreases, especially as the index is not shared among the different single-node appliances.

A promising approach is to build deduplication systems out of cooperating nodes [5]–[7]. However, most investigated solutions trade the deduplication ratio for throughput. The approaches are willing to miss opportunities to deduplicate data chunks, which a single node deduplication system would detect. Other approaches have only been evaluated in a simulation environment, with big chunk sizes, or with a very small cluster sizes.

We use the term "exact deduplication" for deduplication systems that detect all duplicate chunks. The definition helps to differentiate from similarity-based or delta-encoding-based deduplication approaches (e.g., [4]), but also from fingerprinting-based environments, which trade deduplication ratio for other properties like throughput, which are also called "approximate deduplication" systems [7].

The chunk size of an exact deduplication system is usually between 4 KB and 16 KB. These chunk sizes are set as a base line by the Venti backup system [8] or the Data Domain solution presented by Zhu et al. [2]. Very small chunk sizes (512 bytes or even smaller) mean a (slightly) higher deduplication ratio, but the overhead is unreasonably high.

It is the aim of this paper to investigate limitations and opportunities for exact deduplication systems with small chunk sizes. Deduplication systems with larger chunk sizes are not considered within this paper, because many opportunities to detect redundancy are lost.

We propose an exact inline deduplication cluster with a joint distributed index. The design avoids to send the actual deduplicated data over the network in nearly every situation. Typically, only fingerprints are exchanged between the nodes, so that we minimize information exchanged over the internal interconnect. We show that even with a commodity GBit Ethernet interconnect, the bandwidth and latency impact of the communication between the cluster nodes is very small and that the system scales at least up to 24 cluster nodes.

The **main contribution** of this paper is to show that it is possible, despite different claims in previous papers (e.g. [5]), to combine exact deduplication with small chunk sizes and scalability within one environment. This solution has not only been simulated, but all presented scalability results have been measured with a real implementation. Besides this evaluation, we present the architecture and required fault tolerance mechanisms. In addition, we provide a simulation to investigate the upper bound of the achievable throughput.

978-1-4673-1747-4/12/$31.00

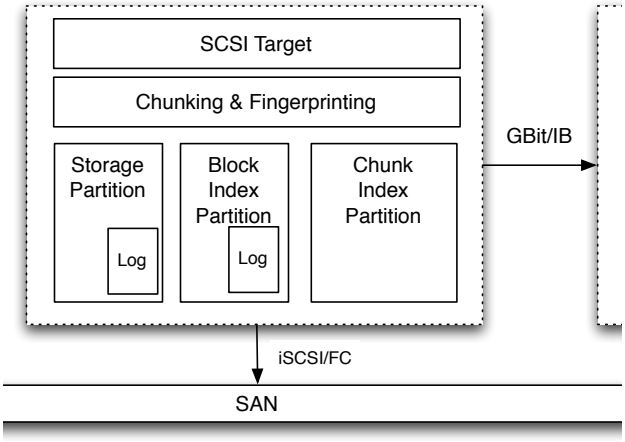

Fig. 1. Single deduplication node. Incoming SCSI requests are chunked and fingerprinted, and then chunk lookup requests are sent to the responsible nodes of the cluster.

The paper starts with an overview of the system design before describing the storage organization (Section III), the communication protocols (Section IV), and the fault tolerance mechanisms (Section V). The evaluation in Section VI investigates the throughput and scaling properties of the system. The paper concludes, after an overview of related work (Section VII), with a discussion of the results and future work (Section VIII).

II. System Design Overview

Most clustered deduplication solutions trade performance for deduplication ratio. These solutions do not detect the same amount of redundant data as a single-node solution could, which is applying the same chunking and fingerprinting algorithms. The solution presented in this paper is using a different architectural approach, which is based on **two design objectives**: We are not willing to lose part of the deduplication ratio and the deduplication appliance has to offer the standard interfaces iSCSI and FibreChannel (FC), so that clients do not have to install new drivers (this is, e.g., different from the solution presented in [6]).

The deduplication cluster is based on the "dedupv1", a single node deduplication system, which builds the foundation for a commercially available single-node deduplication appliance [9]. The dedupv1 system can be classified as an inline, fingerprinting-based deduplication system using Flash-based SSDs to overcome the disk bottleneck. The dedupv1 daemon process presents itself as iSCSI target, which can be used as a block device on most operating system. All incoming write requests are split up and are aligned to "blocks" of 256 KB. The chunking is based on content-defined chunking with Rabin's fingerprinting method [10], [11]. Each fingerprint is identified based on a 20 byte SHA-1 hash value.

An SSD-based chunk index is used to store all previously stored chunk fingerprints. The dedupv1 daemon queries this index to check whether a given chunk is new. A write-ahead log and an in-memory write-back cache are used to avoid random write operations in the critical path. This design allows us to delay persistent index updates for a very long time and also enables various optimizations to aggregate write I/Os.

The information necessary to restore the block data from the chunks is stored in the block index, which is organized as a persistent B-tree also using a write-ahead log and in-memory write-back caches. All non-redundant chunks are collected in 4 MB containers (see also [2], [3], [8], [9] for the container concept) and are stored on a disk backend. A garbage collection process counts how often a chunk is referenced and removes unused chunks from the chunk index and the backend.

The cluster version investigated in this paper inherits most of these properties. However, all storage components are split and distributed to all nodes. The design consists of the following components:

Deduplication Nodes: Deduplication nodes are extensions of the single-node variant. They export volumes over iSCSI and FC, accept incoming requests, perform chunking and fingerprinting, reply to chunk mapping requests of other nodes, and write new chunks to the container storage. Figure 1 shows the internal components of a deduplication node. All deduplication nodes are responsible for a part of the chunk index, the block index, and the container storage.

Shared Storage: Deduplication nodes have access to disks and SSDs via a storage area network (SAN). The available storage is split into many partitions. Each partition contains its own separate (standard) file system and is only accessed by a single deduplication node at a time. However, in cases of node crashes, a surviving node takes over the responsibility for a partition and can restart operating with that partition. A similar approach is used for load balancing, where the system moves the responsibility of a partition to a different node. In both cases, there is no need to move any data. The design enables us to avoid the complexity of parallel file systems.

We rely on the shared storage backend to provide a reliable access to the stored data, e.g., by using RAID-6 on HDD as well as SSD storage. As we will explain later, there is no need that a deduplication node can access the complete storage. It is sufficient if every node can only access its subset of partitions. The only requirement is that all storage partitions are accessible by enough deduplication nodes to tolerate node failures and to enable load balancing.

The system distinguishes between different types of partitions, some using HDDs, some SSD-based storage. The types of partitions and the role in the system are described in Section III.

Interconnect between deduplication nodes: The design always tries to avoid sending chunk data over the network. However, around 120,000 chunk index lookups per GB/s throughput are required in case of an average chunk size of 8 KB. We will later show that even a GBit Ethernet connection is fast enough to provide a high-throughput. We assume that the network is dedicated for internal communication. The SCSI traffic uses a separate network.

Clients: Every system that supports standard block-level protocols like iSCSI and FC can be a client and our approach

does not require any special modifications on the client.

Distributed Coordination: The open source software ZooKeeper is used for configuration as well as the master election and the assignment of nodes to storage partitions [12]. A ZooKeeper daemon runs on all or on a subset of the deduplication nodes. All deduplication nodes connect to the ZooKeeper instance for distributed coordination.

III. Storage Organization

A central design decision of the storage organization is to rely on shared storage for all data. This includes deduplicated data stored on an HDD-based storage backend as well as the performance critical index stored on SSDs. The shared storage is the foundation for our fault tolerance that relies on the ability to redistribute the tasks of a crashed node. Therefore, we move the responsibility for the system reliability from the deduplication system to the storage backend.

The design decision to use shared storage might seem counter-intuitive, because many of today's scalable storage systems apply the "Shared Nothing" approach [13]. Traditionally shared disk system use complex communication schemes so that multiple nodes can access the same storage concurrently. The distributed dedupv1 architecture avoids this and allows only a single node to access a partition. Besides few cases, the shared storage is used as if it were local. This avoids the complexity and inefficiencies of typical shared storage approaches.

Each storage partition is identical with a typical storage partition formatted with a local file system like ext4. Each partition is assigned to a node and the responsibility for a partition is moved between nodes either for load balancing, node maintenance, or crash recovery.

All persistent data structures of dedupv1, especially chunk index, block index, and the container storage are split into several parts, where each part is stored on a partition. We therefore differentiate between chunk index partitions, block index partitions, container partitions, and container metadata partitions. The partition types differ obviously by the data structures stored on them, by the type of storage system (SSD or HDD), and by the operations on the partition for fault tolerance and load balancing, which are explained in Section V.

Each **chunk index partition** contains a part of the overall chunk index. As the chunk index is a very performance-sensitive data structure, all chunk index partitions are stored on SSD storage. Chunks are assigned to partitions by hashing the chunk fingerprint. Load balancing can be performed by simply reassigning partitions to different nodes and there is no need for advanced randomized data distribution and load-balancing schemes [14] [15], [16]. A fixed number of partitions, usually much higher than the expected number of nodes, is created during initialization and the system has to ensure that all partitions must be assigned at all times. Figure 2 illustrates the shared chunk index with several chunk index partitions.

Each **block index partition** contains a part of the block index data. Each exported volume is assigned completely to one index partition. This means that only the node currently responsible for that index partition can export the volume. It is planned to implement SCSI Referral support for more fine-grained manageability [17]. A block index partition is only assigned to a node if a volume is assigned to it.

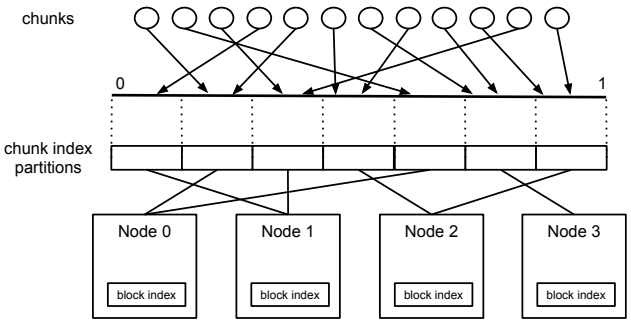

Fig. 2. Shared chunk index. The system hashes chunk fingerprints to the [0,1) interval, which is divided into equal sized partitions. When a chunk is mapped, the deduplication node that mounts the corresponding responsible partition becomes the responsible node for the chunk and adds an entry in the partition's chunk index.

A container partition contains a part of the container storage. In contrast to other partitions, this partition is stored on a disk-based RAID storage. If a node runs out of space in its assigned containers, new container partitions are mounted. If a node observes a significantly higher than average load on one of its container partitions, load balancing is performed by moving the responsibility of partitions to other nodes.

A container metadata partition stores performance-sensitive metadata about a container partition on SSDs. This includes, e.g., the current on-disk location of a container partition. There is always a one-to-one mapping between container partitions and their metadata partition.

Multiple partitions of each type can be assigned to each node. The assignment is stored in ZooKeeper and is managed by the elected master. If the master fails, a leadership election process in ZooKeeper is used to elect a new one.

It is critical to understand why we have chosen a shared storage model instead of the more common shared nothing model. One central reason is to reduce network load, as the shared nothing model requires to replicate data to multiple nodes. If data would only be stored locally on one node and this node goes down, the data would be unavailable until the node becomes online again (e.g., after a hardware replacement) or after disks are physically moved to a different node. However, replication of written data also means that the chunk data has to be copied over the internal communication network. As we will see in Section VI, we are able to scale up to 24 nodes and probably even far beyond that, while transporting data over the backend network would prohibit this. The fact that only in extreme cases (all assigned storage is full and even the load balancer is not able to provide any non-full storage) chunk data is transported over the network is an important property of our design.

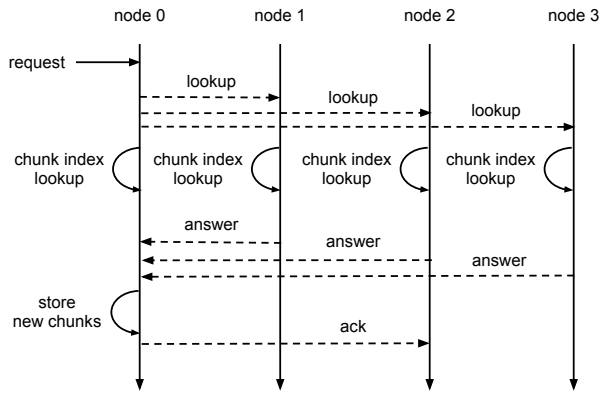

Fig. 3. Communication pattern. In this example, only node 2 has a non existing chunk index entry and thus receives an additional acknowledgement at the end.

A storage partition has not to be accessible from all deduplication nodes, as long as every partition is accessible from enough nodes to tolerate node failures and to ensure an even load distribution. This is similar to the "views" described in consistent hashing [14], which makes it easier and also more cost efficient to extend the backend HDD and SSD storage.

IV. Inter-node Communication

Our main approach to optimize the amount of messages and exchanged information is to never sent new chunk data over the internal network. In detail, a write request is processed as follows:

A client sends a write request with a block of data to a deduplication node. Internally the data is processed in blocks of 256 KB, which is usually also the largest size of write requests over iSCSI. All SCSI requests are processed concurrently facilitating current multi-core CPUs.

After chunking and fingerprinting, it is checked for each chunk if it is already known or not (chunk lookup). For this, the node first hashes the fingerprint of each chunk to a chunk index partition. Next, the node determines the nodes that currently responsible for the chunk index partitions. The partition-to-node mapping is stored in a distributed coordination service ZooKeeper. Since this information only changes rarely, it is easily cachable. Therefore the partition-to-node resolution usually doesn't need any communication.

After resolving the nodes that are responsible for the current chunks, chunk lookup request messages are sent to the nodes. Each message consists of all fingerprints requests on that node as well as a request id for reference.

When receiving the chunk lookup request, the requested node performs a chunk index lookup for all fingerprints in the request and answers in a single message. If there is an entry for a chunk, the node answers with the id of the partition and the id of the container in that partition that holds the chunk data. Since this is only a read operation and other requests can not modify it, there is no need for a locking mechanism. If the chunk is not found in the chunk index, the chunk has appeared for the first time and the requested node answers with an is-new flag. In addition, it locks the index entry, delays any further request for that chunk, and waits for an answer of the requesting node.

When the requesting node receives an answer, it stores new chunks and sends an acknowledgement message back, which is filled with the ids of the partition and container now holding the chunk data. The requested node then updates the chunk index and answers all delayed requests based on the new information. If no acknowledgement message arrives within a given time bound, the requested node unlocks the entry and takes no further actions. Otherwise, if there are delayed requests, the requested node answers the first delayed lookup request as if it were the first and proceeds as described.

After all chunk lookup requests are answered and possible acknowledgement messages are sent, the node updates its block index. Note that a future read operation does not need a chunk index lookup as the block index in combination with the partition mapping holds all information about the chunk location. Figure 3 illustrates the communication pattern.

After all chunk lookup requests and possible acknowledgement messages are answered, the node updates its block index. Note that a future read operation does not need a chunk index lookup as the block index in combination with the partition mapping holds all information about the chunk location.

The deduplication nodes apply further optimizations: All nodes have a cache containing least recently used chunk fingerprints it has send chunk lookup requests for. Also, all nodes use a page cache before the chunk index to avoid going to the SSD storage if the index page has been requested recently. Furthermore, the zero-chunk (a chunk of maximal length containing only zeros) is known to occur often in realistic workloads [7], [18]. This chunk is always treated locally as a special case. In addition, storing new chunks locally creates a data locality that is used to optimize the read performance. The data of all chunks that have been written to that volume first are locally available and does not have to be fetched from other nodes.

The system may query another node for each chunk that is written to the system. With our default average chunk size of 8 KB, this means 120,000 chunk lookup requests per second are necessary to reach a system throughput of 1 GB/s. We are interested in the number of messages these lookups and the rest of the communication scheme results in. For this, we compute the expected number of messages m exchanges for a write request originated on node i. It is the sum of lookup messages (phase 1), response messages (phase 2), and address notification messages exchanged about new chunks (phase 3). The cluster size is n and the expected number of chunks per write is $c = \frac{\text{request size}}{\text{average chunk size}}$. The probability of a chunk to be new is p_n. The set of chunks C of size c is assigned to nodes using a randomly selected hash function. The subset of new chunks is C_n. The expected size of C_n is $p_n \cdot c$. The set of chunks assigned to a node j is denoted with $H(C, j)$. $P[i \rightarrow_p j]$ denotes the probability that node i sends a message

to node j in phase p.

$$
\begin{aligned}
m &= \sum_{j\neq i} P[i \rightarrow_1 j] + \sum_{j\neq i} P[j \rightarrow_2 i] + \sum_{j\neq i} P[i \rightarrow_3 j] \\
&= 2 \cdot \sum_{j\neq i} P[i \rightarrow_1 j] + \sum_{j\neq i} P[i \rightarrow_3 j] \\
&= 2 \cdot \sum_{j\neq i} P[|H(C,j)| > 0] + \sum_{j\neq i} P[|H(C_n,j)| > 0] \\
&= 2 \cdot \sum_{j\neq i} (1 - P[|H(C,j)| = 0]) + \\
&\quad \sum_{j\neq i} (1 - P[|H(C_n,j)| = 0]) \\
&= 2 \cdot (n-1)(1 - \text{binom}(k=0, n=c, p=\frac{1}{n})) + \\
&\quad (n-1)(1 - \text{binom}(k=0, n=c\cdot p_{new}, p=\frac{1}{n})) \\
&= 2 \cdot (n-1)\left(1 - \left(\frac{n-1}{n}\right)^{c}\right) + \\
&\quad (n-1)\left(1 - \left(\frac{n-1}{n}\right)^{c\cdot p_n}\right) \qquad (1)
\end{aligned}
$$

We do not model the chunk caches and node failures. Messages sizes are not relevant here as all messages fit into a single TCP/IP segment.

Figure 4 illustrates the equation (1) for different average chunk sizes and cluster sizes for a 256 KB write request and probability of 2% of new chunks, which is a realistic probability for backup runs. In general, the number of necessary messages increases faster than linear when the cluster is smaller than the number of chunks c. We call this region "small cluster region". If the cluster is larger, the number of messages per node stays largely constant regardless of the cluster size.

In a single-node deduplication system doubling the chunk size usually increases the throughput by a factor of two when the chunk index are the bottleneck. It should also be noted that doubling the chunk size is not reducing the number of messages exchanged when the cluster size is still in the small cluster region. If the communication instead of the actual chunk index lookup is the bottleneck, this implies that increasing the chunk size is not increasing the throughput as it does in a chunk lookup bottlenecked system.

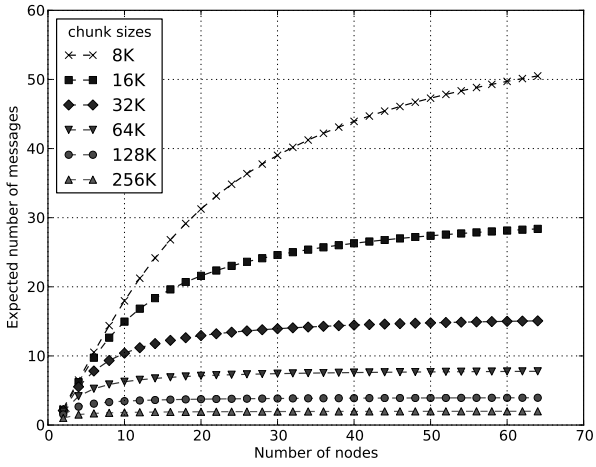

Fig. 4. Expected number of messages per SCSI write request of size 256 KB.

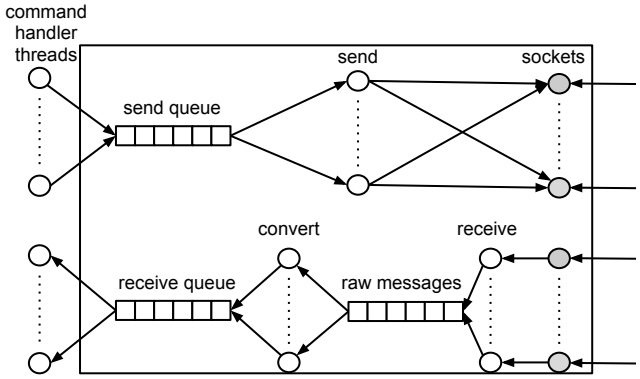

Fig. 5. Architecture of the communication system.

From this discussion follows that the cluster needs a well-designed inter-node communication system that optimally minimizes the amount of information exchanges and is optimized for very high messages per second rates. Other properties like latency and data throughput are only of secondary importance in our setting.

All messages are received and sent through a highly parallelized central communication system (Figure 5) within each node, which keeps TCP/IP connections to all other cluster nodes open. All incoming messages are first delegated to a series of converter threads that convert the raw messages to higher-level objects. The converter also check additional message checksums. The objects are then delegated to a thread pool using a concurrent queue. Also, outgoing messages are placed in a queue, where the messages are popped from a series of sender threads that serialize the message and send them over the outgoing sockets.

In Section VI we not only present an evaluation of the deduplication prototype, and also investigate the limits on the message rates the communication system is able to process. This limits which determines the achievable throughput and scalability.

V. Fault Tolerance and Load Balancing

Fault tolerance is the most important property of storage systems. For backup storage systems, it is critical to be able to continue operations within a short timeframe after a failure because of the limited size of backup windows.

Fault tolerance can be implemented on different levels. We assume that the backend storage (HDD/SSD) is an enterprise-class storage system uses appropriate RAID schemes, disk scrubbing, or even intra-disk redundancy [19]–[21]. In the following we assume that do not have to deal with the reliability of the shared storage system.

However, we have to deal with any kind of node crashes. As explained before we are using the shared storage for fault tolerance instead of explicit replication. When the current master node is informed about a node crash, the master node reassigns all partitions to other nodes that also see the storage of the partitions. After all partitions are mounted on the new assignees and the necessary recovery operations are complete, the operations can continue. When the current master node fails, a new master node is elected and the same operations are performed. The recovery operations for the partition types differ:

The **chunk index partition** is especially important because the complete system halts if it is not available. Therefore the operations must resume after a few seconds. A new node immediately mounts the partition and starts processing messages. However, we use write-back caches for the chunk index and while all operations are logged in the write-ahead log, our design avoids replaying the complete write-ahead log because this might take too long. The system can not wait until the log is replayed. Therefore, we do not recover the exact state. This leads to a situation where chunks are stored in containers, but the chunk index has not entry for these chunks. Chunks that have been stored previously in the write-back cache, but whose entires have not yet been persisted to disk are classified as new chunks. They are *false negatives* as these chunks have actually already been stored. This leads to unwanted duplicates. However, when the write-ahead log is replayed in the background these false negatives are detected and removed. As these false negatives only happening after crashes, the overhead is negligible.

As the system is used for backups, a downtime of up to a minute is acceptable for the **block index partition.** However, delivering out-dated results is not. Therefore, another node takes over the block partition, reads all un-replayed log entries and rebuilds the exact state of the block index. All operations on other volumes can continue while the partition is being rebuild. As the usually store only small amounts of blocks in the write-back cache of the block index, a rebuilding process completes very fast.

The recovery of a **container partition** is even easier. It consists only of mounting the partition and the corresponding management partition at another node and recovering the last containers that have not been closed and finally written to disk. We write all uncommitted container data to a temporary storage location. We recover these containers from this temporary location and commit them to the container storage.

Placing partitions on that shared storage allows a fast recovery from node failures while avoiding explicit replication in the communication protocol. Therefore, we optimize for the common case and can free network for e.g. chunk lookup requests.

A related topic is load balancing. Again, the load balancing differs between the partition types:

The system always tries to distribute the **chunk index partitions** uniformly among the nodes. Given a fixed amount of nodes, rebalancing is not necessary because the direct hashing ensures a uniform distribution of chunks (and thus chunk lookups) to the equal-sized chunk index partitions. This follows basic theorem of balls-into-bins games stating with a high probability that if m items are distributed randomly to n nodes with $m > n \cdot \log n$, no nodes stores more than

$$\frac{m}{n} + \Theta\left(\sqrt{\frac{m \ln(n)}{n}}\right)$$

items [22]. Here our static scheme with equal-sized partitions ensures that the data distribution is not introducing a imbalance. A more dynamic scheme would require methods like consistent hashing, in which the partitions with a high probability differ by a factor of $O(\log n)$ in size. In the basic form, this would introduce a imbalance [14].

Rebalancing a **block index partition** becomes necessary if a deduplication nodes exports SCSI volumes which are heavily used. This step involves a reconfiguration of the client because otherwise the node would have to redirect every SCSI to the overtaking node and thus flood the network and reduce overall performance. The system first evicts all dirty pages from the block index write back cache, so that a new node can immediately provide the SCSI volumes after the remount.

The **container partition** are the easiest to balance. The load is measured in terms of the fill ratio and the write and read operations. The load balancing ensures that all nodes have a container partition mounted with enough free space if this is possible.

All rebalancing invalidates the partition-to-node caches mentioned in Section IV. If a node receives a request that requires a partition it no longer mounts, it answers with an error code. The requesting node then updates its cache and resends the request to the correct node.

It should be noted that the prototype evaluated in Section VI writes all write-ahead logs necessary for the recovery process here, but the recovery and the load balancing are currently not fully implemented.

VI. Evaluation

We have evaluated the throughput and scalability of our exact deduplication cluster by building a prototype based on a single node variant and running performance tests using a real cluster of machines. This differs from other evaluations of clustered deduplication systems that are mostly based on simulations or theoretical analysis.

A. Evaluation Methodology

There is no agreed upon public benchmark for deduplication systems (see, e.g., [23]). New deduplication systems are often not evaluated using a prototype or a real implementation. Furthermore evaluations often use non-public data sets, which are either gathered ad-hoc or which are based on customer data [2].

We evaluate our prototype by generating a fully artificial data stream that resembles a block based backup. The data generation model is based on data deduplication traces from a previous study of the authors [1]. The source code and all other

necessary data of the data generator are available on GitHub[1]. We hope that this will help other researchers to perform similar tests and to provide comparable evaluation results.

The data written to the deduplication system is generated as follows. As long as the test runs, we pick the next from two states (U for unique data or R for redundant data) and choose independently and randomly how much data is continuously generated from the chosen state. The length is chosen based on the empirical distribution extracted from the raw traces of a previous study [1]. In the study the central file server of the University of Paderborn, Germany, was scanned every weekend and all fingerprints were traced. The trace data was used to calculate the possible deduplication ratio on the file server and between different backup runs. It was not possible to fit the empirical distributions good enough to well-known probability distributions like the exponential distribution. The probability distributions of U and R are highly skewed and appear to be heavy-tailed. It is an important insight that unique and redundant chunks are not distributed uniformly in the data stream, but they come in "runs". If a chunk is unique, it is probably that the next chunk is unique, too. We found that besides the basic property of the deduplication ratio, this distribution of chunk lengths is the most important property to generate artificial data for the evaluation of deduplication systems. Park and Lilja confirmed these runs [24]. They observed that the average run length varies little between the different data sets they have analyzed. However, the maximal run length and the skewness varies, which has an impact on the throughput observed in their simulations. The run length statistics of redundant data observed in the trace file used for the data generation have a mean of 143 KB, a median of 16 KB, and a maximum of 3498 MB. The statistics for the run lengths of unique data have a mean of 337 KB, a median of 19 KB, and a maximum of 6582 MB.

The data for the unique state U is generated using a pseudo random number generator with a sufficient sequence length. To generate redundant data (R), we pre-generate a number of redundant blocks and then pick one of these blocks independent at random via an exponential distribution, as we observed that an exponential distribution is a good fit for the reuse pattern of redundant chunks.

Similar to previous studies, we distinguish the first backup generation and further backup generations. The system is empty before the first generation, so that no inter-backup redundancy can be used. In the second (and later) generations, the deduplication system can use chunk fingerprints already stored in the index.

The second data generation is generated similarly to the first one. Here we use the empirical distributions for new unique data (U), intra-backup redundant data (R), and inter-backup redundant data (BR). The probability distributions are also extracted from the same real-world traces. The redundancy between consecutive backup runs is determined by comparing the changes between weekly trace runs. In addition, we

[1]https://github.com/dmeister/dedupbenchmark/tree/online

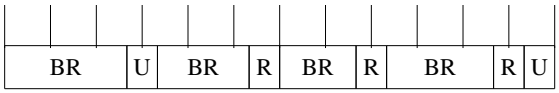

Fig. 6. Illustration of the data pattern of the second backup generation. BR = inter-backup redundant data, R = intra-backup redundant data, U = unique data. The length of the blocks are randomly chosen according to a probability distribution extracted from real world data. The marker in upper half denote the chunks.

extracted the probability distribution of state switches between the states (U, R, BR). The most common state switch is to the R state, as up to 98% of the data is redundant data. We assume that the Markov condition holds for state switches, which means that the probability for a state switch does not depend on previous state switches. The generation for U and R is the same as in the first generation. Intra-backup redundant data is generated by generating the same data as in the first run. Figure 6 shows an illustration of the data generation for the second generation.

We have chosen to simulate a block based backup with 4 KB blocks. All generated run lengths are rounded up to the block size. We only benchmark the first and the second backup generations and are unable to include long-term effects, which is not the focus of this study. As we have no trace data about data sharing between clients, we have not been able to model cross-client sharing. We generate data with different random generator seeds for each client and therefore do not have any data sharing, which is a worst-case assumption for the environment.

The approach of the workload generation allows an evaluation of a fingerprinting-based deduplication system without using proprietary ad-hoc datasets. It allows to reasonably reassemble backup runs of arbitrary size. It generates backup data with similar deduplication ratios and similar data layout of redundant and non-redundant parts of the data stream.

We validated the workload generation comparing a real data set from work-group computers with a similar deduplication ratio and the data generation on a single-node dedupv1 system. The throughput differs within a margin of 5%. The data generation seems to capture the essence of the data pattern and data localities needed for the throughput evaluation of our deduplication prototype.

An early version of the scheme has been described in a previous work of the authors [9]. The two most important differences are: 1) We are now able to generate the data during the test run instead of only pre-generating data, which is read from disk/SSD during the tests. The benchmark environment therefore does not have to include large and fast disk storage on the workload generating clients and allows the generation of arbitrary large data sets. 2) The generation of redundant data is now modeled more accurate. It has been observed (e.g., [1]) that some redundant chunks are referenced more often than other redundant chunks. We now incorporate the skew in the usage of redundant chunks into the data generation algorithm.

We have had exclusive access to a 60-node cluster for the scalability tests. All nodes are identical 1U Fujitsu

RX200S6 servers with two 6-core Intel X4650 CPUs running at 2.67 GHz and having 36 GB RAM. All nodes have GBit Ethernet and InfiniBand 4xSDR HCAs. Both networks are fully switched. We used a CentOS 5.4 with a custom 2.6.18 Linux kernel. The distributed dedupv1 system is actually designed to work with deduplication nodes built out of low-end hardware and to scale horizontally. Most CPU power and the RAM available in the evaluation cluster has been mainly unused, originating from the fact that we had only the nodes of an HPC cluster available for our evaluation.

The 60-node cluster has been partitioned as follows. Up to 24 nodes built the deduplication cluster. Up to 24 nodes are used as clients for workload generation. Up to 12 nodes are used as iSCSI targets to provide the shared storage emulation (6 SSD, 6 HDD). There is always a 1:1 correspondence between deduplication nodes and clients and a 4:1 ratio between deduplication nodes and storage emulation nodes if more than 4 deduplication nodes are used.

The internal communication of the deduplication cluster can be switched between GBit Ethernet and IP over InfiniBand (IPoIB), which is a protocol to tunnel IP packets over InfiniBand. Unless otherwise noted, the GBit interface is used. The system is configured to use 8 KB content-defined chunking using Rabin Fingerprints and to store chunks in 4 MB containers. We did not compress each chunk using an additional compression as usually done in deduplication systems, e.g., in [2]. Given enough CPU power, a fast compression additionally helps to increase the throughput by reducing the amount of data written to the backend storage. The distributed chunk lookup cache is set to 2 MB per deduplication node.

The client nodes generate backup data using the data generation scheme described above and write the data to volumes exported by the deduplication system via iSCSI. Each client writes 256 GB data in each generation. Therefore, the evaluation scaled up to 12 TB of written data in the largest test configuration. The iSCSI traffic uses IP over InfiniBand (IPoIB) to ensure that the GBit Ethernet network is free for the internal communication. A real deduplication system would instead have two or more separate Ethernet connections. Additionally, this ensures that the iSCSI transfer throughput is not the bottleneck.

The HPC-cluster had no shared storage available that would have been appropriate for our use case. As hardware changes were not possible, we had to simulate SSD storage using an SSD performance model proposed by El Maghraoui et al. [25]. We patched an iSCSI target framework, so that a RAM-disk-based volume has been exported over iSCSI that behaved according to the SSD performance model. The extra latency for a 4 KB read was 245 μs, the write latency of it 791 μs. The general processing overhead, e.g., of iSCSI added additional latency. We have emulated the HDD-RAID-like storage in a similar way. As only sequential write operations (4 MB container commit) are performed in the tests, we adjusted the latency accordingly to 1,700 μs. The data itself has not been stored, as we did not wanted to include the influence of backend HDD storage, which can be nearly arbitrarily scaled today. We validated the storage setup with our single-node version and have seen a similar performance with real hardware. The differences can be explained, e.g., as we have used a different SSD model than it was used by El Maghraoui et al. to generate the SSD performance model.

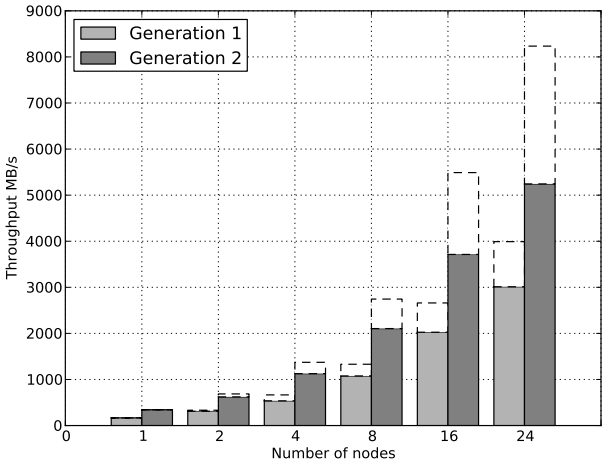

Fig. 7. Throughput using Gigabit Ethernet interconnect for different cluster sizes. The dashed bars show a linear scaling.

The deduplication nodes are stopped and restarted between the first and second data generation as well as all operating system caches are cleared after the system initialization and the simulated backup runs.

B. Scalability and Throughput

The throughput of a deduplication system is the most important performance property besides the storage savings, because the size of a backup window is typically very small. As our deduplication scheme ensures the same deduplication ratio as a single-node deduplication system with the same chunk size, we focus here on the overall cluster throughput. For the evaluation, we scaled the cluster from 2 to 24 nodes as described above. The throughput and scalability for the GBit interconnect is shown in Figure 7 for up to 24 deduplication nodes. We achieved 3.01 GB/s for the first generation and 5.2 GB/s for the second generation. We can observe an exponential scaling with a factor of over 0.86. This means that if we double the size of the scenario and the system under evaluation, then the throughput increases by at least 86% (at least up to 24 nodes).

Figure 8 shows a detailed breakdown of the timings for an average write for the first and second generation in the 24 nodes case. In the first generation, the chunk writing and the chunk lookups dominate with a total share of 91.9%. The high chunk write share of 66.9% is caused by the high percentage of new data. The next larger component is the chunk lookup with 24.5%, which represents the time to collect necessary chunk information from all nodes. The remaining components (block index 4.4%, chunking 2.7%, fingerprinting 1.3%) play a smaller role.

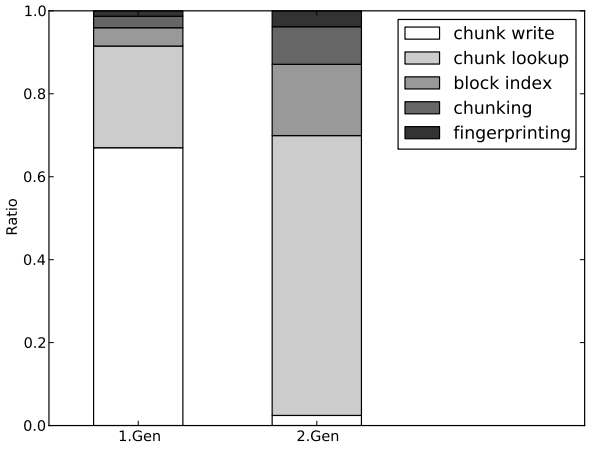

Fig. 8. Relative latency of different components for a write operation in a 24 node cluster.

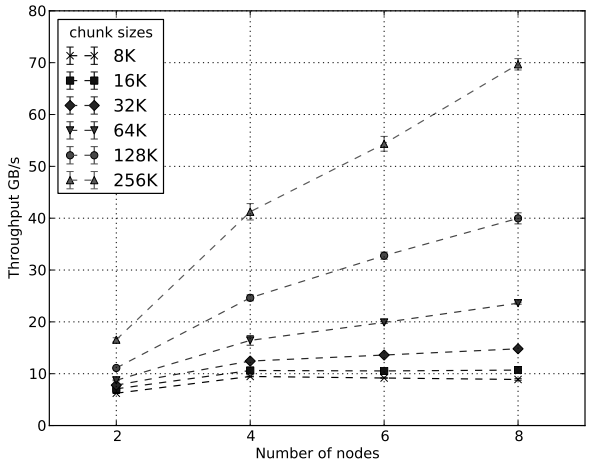

Fig. 9. Theoretical limits on cluster throughput based on network communication.

In the second generation, the chunk write share (2.4%) nearly vanishes, because most data is now known and does not need to be stored. As a consequence, the fractions of chunk lookup (67.4%), block index (17.1%), chunking (9.0%) and fingerprinting (3.8%) rise.

Additionally, we used InfiniBand hardware as a drop-in replacement using IPoIB for intra-node communication, which is tunneling IP traffic over the InfiniBand network. The throughput results are shown together with the GBit results in Table I. Surprisingly, the throughput is not significantly different. Even lower throughput has been measured. This is another indication that the internal processing capabilities are the bottleneck in our evaluation, and not the network itself. The additional evaluations focused on the network, which we present in the next section, support this assumption.

C. Communication-induced Limits on Throughput

An interesting property we focus our further analysis on is the interconnect. In the evaluation with 4 nodes, each node sends on average 6,573 messages per second with an average message size of 118 bytes. Each chunk lookup request message contains on average requests for 3.6 chunks (see Section IV for the underlying mechanisms).

The aggregation of multiple chunks into a single chunk lookup request message significantly reduces the load in smaller clusters, but the effect vanishes later. In the case of a 16 node deduplication cluster, the average message size decreases to 58 bytes and each node has to send 19,528 messages per second on average. Each message includes on average requests for 2.3 chunks. This drops even to 1.5 chunks on average for clusters with 32 nodes.

The important observation is that the number of generated messages per node based on incoming SCSI writes cannot increase to a number higher than the number of chunks per request. For each incoming 256 KB data block, a node generates at most as much messages as there are chunks, regardless of the number of nodes in the network.

This explains the sub-linear scalability observed in the evaluation measurements for our cluster with up to 24 nodes. In that node range, each increase in the node count increases the number of messages a node has to process in two ways: 1) More nodes (in our experiment) means more write requests, which create more chunk lookup requests and 2) the average number of chunks per chunk lookup request is reduced, which means more messages per write request. However, with each increase in the number of nodes the ability to process messages increases only linearly.

This opens up a new question. Assuming that we can scale the actual chunk index lookups on the SSD storage using faster and/or more SSDs: what is the performance limit caused by the communication overhead if we also ignore all other components of the deduplication system. The focus on the impact of the communication and networking aspects allows us to derive a limit using the current setup and architecture.

For the estimation, we used the same hardware as before (see Section VI-A), but this time we only had up to 8 nodes available. Each node generated one million requests that represent writes of size 256 KB. For each request, a node randomly maps chunks to nodes and sends the fingerprints to the chosen ones. All chunks to the same node are aggregated together

TABLE I
AVERAGE PER-NODE/TOTAL THROUGHPUT IN MB/S USING DIFFERENT INTRA-NODE INTERCONNECTS.

	Gigabit				InfiniBand			
	1.Gen		2.Gen		1.Gen		2.Gen	
1	166	166	343	343	166	166	342	342
2	155	310	311	622	156	312	304	608
4	134	536	281	1124	134	536	278	1112
8	134	1072	263	2104	133	1064	255	2040
16	126	2016	232	3712	127	2032	229	3664
24	125	3000	218	5232	124	2976	209	5016

and sent as a single chunk lookup message. A receiving node declares each chunk as new with a probability of 2% to reflect the rate of new data in consecutive backup runs. This models the communication pattern of the chunk index lookups in our distributed deduplication system, while eliminating all other factors.

Figure 9 shows the theoretical achievable maximum throughput with different cluster and chunk sizes when we only focus on the chunk lookup network traffic (with 95% confidence intervals). The cluster throughput for 8 KB chunks starts at 6.2 GB/s and rises to 9.4 GB/s for 4 nodes after which it slowly degrades to 8.8 GB/s for 8 nodes. This shows that the message processing rate of the nodes is the bottleneck rather than the network itself, because the total number of messages sent during the experiment has no impact on the performance. In a 4 node cluster with an average chunk size of 8 KB each request results in expected 32 chunks such that each node has to answer expected 8 chunks. Hence, a request generates at least 3 messages w.h.p. not counting extra communication for new chunks. In an 8 nodes cluster each node has to answer expected 4 chunks, which results in at least 7 messages per request. The slight performance decrease comes from the increased waiting time, as the nodes must process more messages.

The results support our thesis introduced earlier that the nodes' ability to process messages is the bottleneck in these experiments. However, as explained before, the number of generated messages only grows linearly after a certain cluster size is reached. We see this for the scalability in latter experiments for the large chunk sizes, even for the small cluster of only up to 8 nodes. The turning point where the communication system would stop to be a scalability problem is approximately 32 nodes for 8 KB chunks and 16 nodes for 16 KB chunks.

Another source of scalability problems could be the network switch. It would be possible to reach a point where the switch is not able to deliver the messages per second. However, the results show no sign of a saturation in the GBit Ethernet network switch. As long as all nodes are connected to a single HPC-class switch, we do not expect this to be an issue. However this is here the case, because only fingerprints and not the actual chunk data is exchanged over the network.

The overall throughput improves if the message processing load per node decreases. This can be achieved by either adding more nodes while fixing the number of requests or by increasing the average chunk size. The latter can be seen in Figure 9 and reduces the number of chunks per request and thus, the maximum number of messages for collecting the chunk information. The best performance is reached for 256 KB chunks, where each request only contains one chunk.

As we ignored the actual chunk index lookup on the SSD storage in this setting, these limits are also valid for deduplication systems which hold the complete chunk index in RAM instead on an SSD. It does not hold for other types of deduplication system like approximate deduplication systems or systems, where it is not necessary to lookup most chunks over the network which use, e.g., a locality-preserving caching scheme [2].

Another way to increase the upper-bound is to further optimize the communication system. For example decreasing lock contentions would actually have an impact on the upper bound on the throughput. However, we claim that while optimizations are certainly possible, we already put much effort into the communication system, so that this is not trivial.

The major conclusion of this section is that the network will be able to scale further. Factors like chunking and fingerprinting can obviously scale linearly with the number of nodes. If also the storage components are scaled with the number of cluster nodes, the overall system is scalable.

VII. Related Work

Data deduplication systems have been subject to intensive research for the last few years. The approaches can be classified in fingerprinting-based approaches and delta-encoding approaches. All fingerprinting-based approaches share that after a chunking step the chunks are hashed using a cryptographic fingerprinting method like SHA-1. The chunking splits the data stream either into fixed-size chunks [8], [26] or uses a content-defined chunking methods [2], [3], [9], [27]. Delta-encoding based systems do not search for identical chunks, but for similar chunks and then delta-compress chunks based on similar chunks. An example for a delta-encoding system is IBM's Diligent system, whose design has been described by Aronovich et al. [4].

Recently, it became clear that a single-node deduplication system cannot fulfill the throughput and scalability needed by today's enterprises. Most realistically investigated solutions trade-off throughput versus deduplication ratios and are willing to miss opportunities to deduplicate data chunks, which a single node deduplication systems would detect.

Extreme Binning by Bhagwat et al. chooses a representative chunk id per file [5]. The authors do not analyze the throughput of their system. Redundancies between files are only found when the chosen chunk id is the same for both files. The approach is based on Broder's theorem that essentially states that if two files share the same chunk id, both files are likely very similar. Dong et al.'s extension of the Zhu et al. work on Data Domain file system is based on similar ideas, but use so called "super chunks" for the data routing instead using a chunk id per file [2], [6].

Dubnicki et al. presented an deduplication cluster system called HYDRAstore using large chunks (64 KB) and a Shared Nothing architecture. The chunk fingerprint data is distributed using a distributed hash table (DHT). The data is distributed to multiple nodes using erasure coding for resiliency. The bypass the disk bottleneck usually seen in data deduplication systems by using 64 KB chunks and holding the chunk metadata in memory al the time. While the design is that of an exact deduplication system, the choice of the chunk size favors throughput over deduplication ratio [28], [29]. They report scalability results up to 12 nodes with a throughput of 800 MB/s for non-deduplicating data.

MAD2 by Wei et al. is another approach for a distributed deduplication system applying exact deduplication [7]. The chunk data is distributed using a DHT using a bloom filter scheme and a locality preserving caching scheme very similar to Zhu et al.'s [2]. The system using local storage, but doesn't replicate chunk fingerprints to multiple nodes so that the system cannot tolerate node crashes. Additionally, the prototype has been demonstrated only using two nodes. The authors do not try to evaluate further scalability nor do they have a look on the impact of the underlaying network.

DEBAR, an adjacent work by partly the same authors as MAD2, proposed a post-processing deduplication system cluster [30]. In a post processing system, all data is immediately written to disk and only in a post-processing stage

There are some commercial clustered storage systems with deduplication or similar techniques available: Septon provide a clustered deduplicating backup system using a content metadata aware deduplication scheme with a byte-level comparison. Based on the information available it is hard to classify it, but it is certainly not fingerprinting-based deduplication [31]. This is different from the systems by Permabit which are inline, fingerprint-based deduplication system using a DHT-based approach using local storage at the nodes [32]. According to [33], Permabit may use a scheme storing metadata about the fingerprints in RAM using a scheme that allows membership-testing with a certain false-positive rate. The data structure uses the property that the keys are cryptographic fingerprints for a scheme more efficient than Bloom filters. Little is known about its approaches for data distribution, load balancing, and fault tolerance.

An alternative to distributed deduplication systems is the use multiple separate deduplication appliances that are not cooperating. However, this reduces the possible savings because overlap between backup data stored on different machines is stored multiple times. Additionally, moving backups between machines, e.g. after adding new appliances, causes to store already deduplicated data another time. Douglis et al. present a load balancing/backup assignment scheme so that backups are assigned to deduplication instances so that the overlap and affinity between backups runs is increased [34]. However, a clustered deduplication system like ours where the data deduplicated between all deduplication nodes makes the management easier, enabled an easier load balancing, and increases the storage savings of data deduplication.

The discussion between shared disk and stored nothing storage systems is not a new one. There are several examples for both concepts. IBM's GPFS and IBM Storage Tank are classic examples for a shared disk storage systems [35], [36]. On the other hand, Ceph, Lustre, and PanFS use local disks, to which other nodes never have block-level access [37]–[39].

Our hybrid approach to use the shared disk only for crash recovery is similar to Devarakonda et al.'s approach presented in the Calypso file system [40]. They also use shared storage, called multi-ported disk there, to recover the storage system state after crashes, while only a single server accesses a disk at any time. They also relied on RAID or mirroring approaches to provide disk fault tolerance. In addition, they used client-state to recover from node failures, a possibility we don't have. A contrary position is a Sprite position statement that proposes using replicate state in main memory for recovery. As explained before, we avoid this mainly to free the intra-node network from the replication traffic [41]. Our approach to allow that nodes are allowed to only see a subset of the available storage is related to the storage pool concept of V:Drive [42].

The communication and the network interconnect between storage peers has been analyzed before by Brinkmann et al. [43]. They concluded that the interconnect can become the bottleneck, but the authors concentrate on bandwidth limited traffic. We bypass this trap by avoiding that the actual data is sent over the internal communication network. Similar observations have been done presented in [39].

VIII. Conclusion

This paper presents an exact inline-deduplication cluster, which is able to detect as much redundancy as a single-node solution by using a joint distributed chunk index and small chunk sizes. We show that the solution has good scaling properties by presenting a prototype evaluation. One key element of the ability to scale well is the communication protocol that avoids sending the full data over the network. Instead only fingerprints are exchanged between nodes. The use of shared storage furthermore enable an efficient support for fault tolerance. Therefore, we have been able to show that, despite different claims in previous papers, it is possible to combine exact deduplication, small chunk sizes, and scalability within one environment using only a commodity GBit Ethernet interconnect. Additionally, we investigated the throughput and scalability limitations with a special focus on the intra-node communication.

IX. Acknowledgments

The authors would like to thank the German Federal Ministry of Economics and Technology (BMWi) for partly funding this work in the project *SIMBA*.

References

[1] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario," in *Proceedings of the the 2^{nd}Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[2] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the Data Domain deduplication file system," in *Proceedings of the 6^{th} USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[3] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proceedings of the 7^{th} USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[4] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *Proceedings of the the 2^{nd} Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[5] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proceedings of the 17^{th} IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2009.

[6] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," in *Proceedings of the* 9^{th} *USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[7] J. Wei, H. Jiang, K. Zhou, and D. Feng, "MAD2: A scalable high-throughput exact deduplication approach for network backup services," in *Proceedings of the* 26^{th} *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[8] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in *Proceedings of the* 1^{st} *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[9] D. Meister and A. Brinkmann, "dedupv1: Improving deduplication throughput using solid state drives," in *Proceedings of the* 26^{th} *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[10] M. O. Rabin, "Fingerprinting by random polynomials," TR-15-81, Center for Research in Computing Technology, Harvard University, Tech. Rep., 1981.

[11] U. Manber, "Finding similar files in a large file system," in *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the USENIX annual technical conference (ATC)*, 2010.

[13] M. Stonebraker, "The case for shared nothing," *Database Engineering*, vol. 9, 1986.

[14] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the* 29^{th} *ACM Symposium on Theory of Computing (STOC)*, 1997.

[15] R. J. Honicky and E. L. Miller, "Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution," in *Proceedings of the* 18^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[16] A. Miranda, S. Effert, Y. Kang, E. Miller, A. Brinkmann, and T. Cortes, "Reliable and randomized data distribution strategies for large scale storage systems," in *Proceedings of the* 18^{th} *International Conference on High Performance Computing (HiPC)*, 2011.

[17] M. Evans, "SCSI block command 3 (SBC-3) revision 24," T10 Committee of the INCITS, 2010.

[18] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of the the* 2^{nd} *Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[19] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (raid)," *SIGMOD Rec.*, vol. 17, 1988.

[20] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, and S. Ng, "Disk scrubbing in large archival storage systems," in *Proceedings of the* 12^{th} *IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2004.

[21] B. Schroeder, S. Damouras, and P. Gill, "Understanding latent sector errors and how to protect against them," *Trans. Storage*, vol. 6, 2010.

[22] M. Raab and A. Steger, ""balls into bins" - a simple and tight analysis," in *Proceedings of the* 2^{nd} *International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, 1998.

[23] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer, "Benchmarking file system benchmarking: It *is* rocket science," in *Proceedings of the* 13^{th} *USENIX Workshop on Hot Topics in Operating Systems*, 2011.

[24] N. Park and D. J. Lilja, "Characterizing datasets for data deduplication in backup applications," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010.

[25] K. El Maghraoui, G. Kandiraju, J. Jann, and P. Pattnaik, "Modeling and simulating flash based solid-state disks for operating systems," in *Proceedings of the* 1^{st} *joint WOSP/SIPEW international conference on Performance engineering*, 2010.

[26] S. Rhea, R. Cox, and A. Pesterev, "Fast, inexpensive content-addressed storage in foundation," in *Proceedings of the USENIX 2008 Annual Technical Conference (ATC)*, 2008.

[27] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, 2001.

[28] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "HYDRAstor: A scalable secondary storage," in *Proceedings of the* 7^{th} *USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[29] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra, "HydraFS: a high-throughput file system for the hydrastor content-addressable storage system," in *Proceedings of the* 8^{th} *USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[30] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan, "DEBAR: A scalable high-performance de-duplication storage system for backup and archiving," in *Proceedings of the* 24^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[31] Septon, "DeltaStor Software Datasheet," http://go.sepaton.com/rs/sepaton/images/2011%20DeltaStor%20Datasheet.pdf, 2010.

[32] Permabit, "Permabit deduplication 2.0, powered by scalable data reduction," White paper, http://www.permabit.com/resources/pdf-wp/WP-Dedupe2.pdf?DocID=19, 2009.

[33] N. H. Margolus, E. Olson, M. Sclafani, C. J. Coburn, and M-Fortson, "A storage system for randomly named blocks of data," Patent WO 2006/042019A2, 2005.

[34] F. Douglis, D. Bhardwa, H. Qian, and P. Shilane, "Content-aware load balancing for distributed backup," in *Proceedings of the* 25^{th} *Large Installation System Administration Conference (LISA)*, 2011.

[35] F. Schmuck and R. Haskin, "GPFS: a shared-disk file system for large computing clusters," in *Proceedings of the* 1^{st} *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[36] J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM storage tank– a heterogeneous scalable san file system," *IBM Syst. J.*, vol. 42, 2003.

[37] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the* 7^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[38] P. Braam, "Lustre: A scalable, high performance file system." Available at http://www.lustre.org/docs/whitepaper.pdf, 2002.

[39] D. Nagle, D. Serenyi, and A. Matthews, "The panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC)*, 2004.

[40] M. Devarakonda, B. Kish, and A. Mohindra, "Recovery in the calypso file system," *ACM Trans. Comput. Syst.*, vol. 14, 1996.

[41] B. Welch, M. Baker, F. Douglis, J. Hartman, M. Rosenblum, and J. Ousterhout, "Sprite position statement: Use distributed state for failure recovery," in *Proceedings of the* 2^{nd} *Workshop on Workstation Operating Systems WWOS-II*, 1989.

[42] A. Brinkmann, M. Heidebuer, F. Meyer auf der Heide, U. Rückert, K. Salzwedel, and M. Vodisek, "V:Drive - costs and benefits of an out-of-band storage virtualization system," in *Proceedings of the* 12^{th} *NASA Goddard,* 21^{st} *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2004.

[43] A. Brinkmann and S. Effert, "Inter-node communication in Peer-to-Peer storage clusters," in *Proceedings of the* 24^{th} *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2007.