Distributed Exact Deduplication for Primary Storage Infrastructures

João Paulo, José Pereira

Distributed Exact Deduplication for Primary Storage Infrastructures

João Paulo and José Pereira

High-Assurance Software Lab (HASLab)
INESC TEC and University of Minho
{jtpaulo, jop}@di.uminho.pt

Abstract. Deduplication of primary storage volumes in a cloud computing environment is increasingly desirable, as the resulting space savings contribute to the cost effectiveness of a large scale multi-tenant infrastructure. However, traditional archival and backup deduplication systems impose prohibitive overhead for latency-sensitive applications deployed at these infrastructures while, current primary deduplication systems rely on special cluster filesystems, centralized components, or restrictive workload assumptions.
We present DEDIS, a fully-distributed and dependable system that performs exact and cluster-wide background deduplication of primary storage. DEDIS does not depend on data locality and works on top of any unsophisticated storage backend, centralized or distributed, that exports a basic shared block device interface. The evaluation of an open-source prototype shows that DEDIS scales out and adds negligible overhead even when deduplication and intensive storage I/O run simultaneously.

Keywords: deduplication, storage systems, distributed systems, cloud computing

1 Introduction

Deduplication is accepted as an efficient technique for reducing storage costs at the expense of some processing overhead, being increasingly sought in primary storage systems [18, 5, 14] and cloud computing infrastructures holding Virtual Machine (VM) volumes [7, 4, 12]. As static VM images are highly redundant, many systems avoid duplicates by storing unique Copy-on-Write (CoW) golden images and using snapshot mechanisms for launching identical VM instances [6, 10]. Other systems also target the duplicates found in dynamic general purpose data from VMs volumes thus, increasing the space savings that, may range from 58% up to 80% for cluster-wide deduplication [4, 11, 18]. In fact, with the unprecedented growth of data in cloud computing services and the introduction of more expensive storage devices, as Solid State Drives (SSDs), these space savings are key to reduce the costs of cloud infrastructures [2].

Deduplication in a distributed infrastructure and across VMs primary volumes with dynamic and latency sensitive data raises, however, several challenges.

Primary volumes have strict latency requirements that are not met by traditional *in-line* deduplication systems where data is shared before being stored thus, including deduplication processing overhead in the storage writes [17]. As a matter of fact, although some attempts were made to extend traditional deduplication algorithms to support file system semantics, none of these systems were able to handle efficiently random storage workloads [21, 9, 8].

As an alternative, *off-line* deduplication decouples aliasing from disk I/O operations and performs both asynchronously thus, minimizing the impact in storage writes [7, 4]. As data is only aliased after being stored, off-line deduplication requires additional storage space when compared to the in-line approach. Also, since deduplication and storage requests are performed asynchronously, a CoW mechanism must be used to prevent updates on aliased data and possible data corruption. This mechanism increases the overhead in storage writes and the complexity of reference management and garbage collection, confining off-line deduplication to off-peak periods to avoid performance degradation [4]. Unfortunately, off-peak periods are scarce or inexistent in cloud infrastructures hosting VMs from several clients, giving deduplication a short time-window for processing the storage backlog and eliminating duplicates. Ideally, deduplication should run continuously and duplicates should be kept on disk for short periods of time to ensure a smaller storage size.

Distributed infrastructures raise additional challenges as deduplication must be performed across volumes belonging to VMs deployed on remote cluster servers [7, 4]. Space savings are maximized when duplicates are found and eliminated globally across all volumes but this requires a more complex remote indexing mechanism, accessible by all cluster servers, that tracks unique storage content and is consulted for finding duplicates. Accessing this index in the storage path has prohibitive overhead for latency-sensitive applications thus, forcing distributed in-line deduplication systems to relax deduplication's accuracy and find only a subset of the duplicates across the cluster [19, 12, 18].

For clarity purposes, we use the term *chunks* as the units of deduplication, that usually are files, variable-sized blocks, or fixed-size blocks [17, 1]. Also, most deduplication systems compare compact signatures of the chunks' content, referred to as *chunk signatures* or *digests*, instead of comparing the chunks' full content [17].

The combined challenges of primary storage and global deduplication are addressed with DEDIS, a fully-decentralized and dependable system that performs exact and cluster-wide off-line deduplication of VMs primary volumes. Unlike previous systems, it works on top of any storage backend, centralized or distributed, that exports an unsophisticated shared block device interface. This way, DEDIS does not rely on storage backends with built-in locking, aliasing, CoW, and garbage collection operations. Instead, deduplication is decoupled from a specific storage backend, avoiding performance issues of previous systems [7, 4] and changing the system design thus favoring distinct optimizations, as discussed in Section 2. Moreover, DEDIS overhead and performance are independent from the storage workloads' spatial and temporal locality [18].

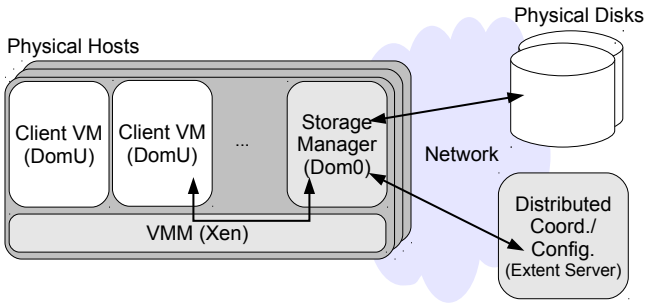


Fig. 1. Distributed storage architecture assumed by DEDIS.

As the main contribution, we present a fully-distributed off-line deduplication mechanism. VMs I/O requests are intercepted and redirected to the correct storage locations, at the fixed block granularity, by a layer that also eliminates duplicate chunks asynchronously. This design excludes costly accesses to remote metadata, hash calculations and reference management from the storage path. Deduplication is performed globally and exactly across the cluster by using a sharded and replicated fault tolerant distributed service that maintains both the index of unique chunks signatures and the metadata for reference management. This service is key for achieving a fully-decentralized and scalable design. A persistent logging mechanism stores the necessary metadata, in a shared storage pool, for recovering and reassigning volumes of failed cluster to other nodes.

As other contributions, DEDIS leverages off-line deduplication to detect and avoid I/O hotspots thus, reducing the amount of CoW operations and their cost. Latency overhead is then further reduced with batch processing and caching that, also increase deduplication throughput. Moreover, DEDIS can withstand hash collisions in specific VM volumes by performing byte comparison of chunks before aliasing them, while keeping a small impact in both deduplication and storage performance. Finally, DEDIS prototype, implemented within the middleware Xen blktap driver, is evaluated in a distributed setting where it is shown that our design scales and introduces negligible overhead in storage requests while maintaining acceptable deduplication throughput and resource consumption.

The paper is structured as follows: Section 2 discusses DEDIS components, fault-tolerance considerations, design optimizations and implementation details. Section 3 evaluates DEDIS open-source prototype and, Section 4 distinguishes DEDIS from state of the art systems. Section 5 concludes the paper.

2 The DEDIS System

Figure 1 outlines the baseline distributed primary storage architecture assumed by DEDIS. A number of physical disks are available over a network to physical hosts running multiple VMs. Together with the hypervisor, storage management services provide logical volumes to VMs by translating *logical addresses*

within each volume to *physical addresses* in arbitrary disks upon each block I/O operation. Since networked disks provide only simple block I/O primitives, a distributed coordination and configuration service is assumed to locate meta-information for logical volumes, free block extents and to ensure that a logical volume is mounted at any time by at most one VM. The main functionality is as follows:

Interceptor. A local module in each storage manager maps logical addresses of VMs to physical addresses, storing the physical location of each logical block in a persistent map structure (*logical-to-physical* map). In some LVM systems, VM snapshots are created by pointing multiple logical volumes to the same physical locations [10]. Shared blocks are then marked as CoW in the persistent map while, updates to these blocks require a free block to write the new content and updating the map to break aliasing.

Extent server. A distributed coordination mechanism allocates free blocks from a common pool when a logical volume is created, or lazily when a block is written for the first time, or when an aliased block is updated (*i.e.*, copied on write). The overhead of remote calls is reduced by allocating storage extents with a large granularity that are then, within each physical host, used for local requests [10].

The architecture presented in Figure 1 is a logical architecture, as physical disks and even the *Extent* service itself can be contained within the same physical hosts. For simplicity, we assume that the Xen hypervisor is being used and we label payload VMs as DomU and the storage management VM as Dom0. However, the architecture is generic and can be implemented within other hypervisors, while using networked storage protocols distinct from ISCSI, which is the one used in DEDIS evaluation. Since we focus on the added functionality needed for deduplication, we do not target a specific map structure of logical to physical addresses. Also, DEDIS does not require built-in CoW functionalities, as we introduce our own operation. Finally, DEDIS uses fixed-size blocks because the *interceptor* module already processes fixed-size blocks and, generating variable-sized chunks would impose unwanted computation overhead [7, 4].

2.1 Architecture

DEDIS architecture is depicted in Figure 2 and requires, in addition to the baseline architecture, a *distributed* module and two *local* modules, highlighted in the figure by the dashed rectangle.

Distributed Duplicates Index (DDI). Indexes unique content signatures of blocks belonging to the primary storage. Each entry maps a signature to the physical address of the corresponding storage block and to the number of logical addresses pointing to (sharing) that block, which allows aliasing duplicate blocks and performing garbage collection of unreferenced blocks. Index entries are persistent and are not assumed to be fully-loaded on RAM. Also, entries are sharded and replicated across several DDI nodes for scalability and fault tolerance purposes. The size of each entry is small (few bytes) so, each node can index many blocks, thus allowing a small number of DDI nodes even in large infrastructures.

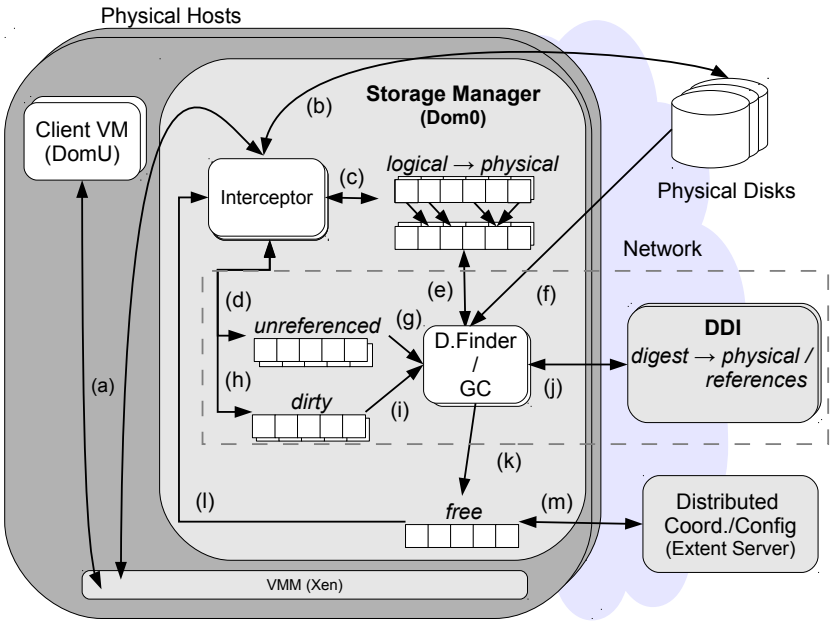


Fig. 2. Overview of the DEDIS storage manager.

Duplicate Finder (D. Finder). Asynchronously collects addresses written by local *interceptors*, which are stored at the *dirty* queue, and matches the correspondent blocks with others registered at the DDI. Blocks processed by this module are preemptively marked as CoW in order to avoid concurrent updates and possible data corruption. This module is thus the main difference from a storage manager that does not support deduplication.
Garbage Collector (GC). Processes aliased blocks that were updated (copied on write) and are no longer being referenced by a specific logical address. Physical addresses of copied blocks are kept at the *unreferenced* queue, and the number of references to a specific block can be consulted and decremented at the DDI. Blocks are unused when the number of references reaches zero. Both *D. Finder* and *GC* modules free unused blocks by inserting their physical addresses in a local *free* blocks pool that provides blocks addresses for CoW operations.

2.2 I/O and DEDIS Operations

The operations executed by each DEDIS module are depicted in Figure 2. Bidirectional arrows mean that information is both retrieved and updated at the target resource. *GC* and *D. Finder* modules are included in the same process box because both run in distinct threads of the same process, within the Xen Dom0, and perform concurrent operations for each VM. The latency of I/O requests is reduced by finding and managing duplicates in a background thread. Each VM volume has an independent process running its own *interceptor*.

An I/O operation in the Interceptor. The interceptor (a) gets read and write requests from local VMs, (c) queries the *logical-to-physical* map for the corresponding physical addresses and, (b) redirects them to a physical disk over the network. As potentially aliased blocks are marked in the map as CoW by *D. Finder*, writes to such blocks must first (l) collect a free address from the *free* pool, (b) redirect the write to the free block and update the map accordingly (c). Then, the physical address of the copied block is added to the *unreferenced* queue (d) and, processed later by *GC*. For both regular and CoW write operations, (h) the logical address of the written block is inserted in a *dirty* queue. I/O requests are acknowledged as completed to the VMs (a) after completing all these steps.

Sharing an updated block in D. Finder. This background module aliases duplicate blocks. Therefore, each logical address that was updated and inserted in the *dirty* queue is eventually picked up by the *D. Finder* module (i), that preemptively marks the address as CoW (e), reads its content from the storage (f), computes a signature and queries the DDI in search of an existing duplicate (j). This is done using a test-and-increment remote operation, that stores the block's information as a new entry at the DDI if a match is not found. If a match is found, the counter of logical addresses (references) pointing to the DDI entry is incremented while, locally (e), the *logical-to-physical* map is updated with the new physical address found at the DDI entry and (k) the physical address of the duplicate block is added to the *free* pool. Blocks are marked as CoW before reading their content because deduplication runs in parallel with I/O requests and updates to these blocks could originate data corruption.

Freeing an unused block in GC. This background module examines if a copied block, at the *unreferenced* queue (g), has become unreferenced. The block's content is read from the storage (f), its signature is calculated and then the DDI is queried (j) using a remote test-and-decrement operation that decrements the number of logical addresses pointing to the corresponding DDI entry. If the block is unused (zero references), its entry is removed from the DDI and, locally, the block address is added to the *free* pool (k). This pool keeps only the necessary addresses for local CoW operations, while the remainder is returned to the *extent* server (m). If the queue is empty, unused block addresses are requested from the *extent* server pool (m).

The latency-critical *interceptor* does not invoke any remote service and, only blocks if the local *free* pool becomes empty, which can be avoided by tuning the frequency of the *GC* execution. The *test-and-increment* and *test-and-decrement* operations and metadata stored in each DDI entry allow performing the lookup of unique block signatures and incrementing or decrementing the entry's logical references in a single round-trip to the DDI. Unlike in DDe and DeDe, this design combines aliasing and reference management in a single remote invocation, avoiding a higher throughput penalty and reducing metadata size [7, 4]. VM volumes have an independent *D. Finder* and *GC* thread, as well as, a distinct *logical-to-physical* map, *dirty* queue and *unreferenced* queue. Only the *free* pool is shared across VMs in the same server thus, requiring mutual exclusion for concurrent acesses. Multiple updates to the same block between two *D. Finder*

iterations count as a single one because only the latest written content is shared. Finally, the *interceptor* is able to collect writes from applications and from the operating system so, deduplication can be applied to both types of dynamic data.

2.3 Concurrent Optimistic Deduplication and Fault Tolerance

DEDIS removes deduplication processing overhead, including chunk signatures calculation, from the storage write path. However, contention still exists when the *D. Finder* and *interceptor* modules access shared metadata. To reduce contention and its penalty in storage latency, DEDIS uses an optimistic deduplication approach that only performs fine grained locking when *D. Finder* and *interceptor* operations access common metadata (e.g. *logical-to-physical* map), while avoiding remote invocations to the DDI and other time consuming operations in the mutual exclusion space. This decision leads to race conditions that are detected and processed accordingly, as explained in previous work that validates our algorithm with a model checker [15]. Also, since signatures are calculated asynchronously by *Share* and *GC* modules, an additional read to the storage backend is required for each processed block, whose overhead is accounted in our evaluation.

DEDIS is resilient to cluster nodes crashes and to lost and repeated requests by writing meta-information persistently. Transactional logs track changes to metadata structures and allow logical volumes of a crashed physical node to be recovered by another freshly booted node. To reduce the impact on storage latency, logging operations are performed outside the storage path with only two exceptions, namely, when a block is copied at the *interceptor* and when a block is preemptively marked as CoW by *D. Finder*. Logs and persistent metadata structures may be stored in a shared storage pool and the recovery of failed nodes is then ensured by the distributed coordination and configuration service that provides the *extent* server functionalities. DDI nodes have on-disk persistent entries and are fully replicated using the primary-backup approach with a virtually-synchronous group communication protocol. This way, DDI entries can be stored in the shared storage pool or in cluster nodes local disks. The overhead of all logging operations is contemplated in our evaluation.

2.4 Optimizations

The *D. Finder* module uses a hotspot detection mechanism for identifying storage blocks that are write hotspots. By not sharing blocks that are frequently rewritten, the amount of costly CoW operations is reduced. Namely, logical addresses in the *dirty* queue are only processed in the next *D. Finder* iteration if they were not updated during a certain period of time. For instance, in our evaluation, only the logical addresses at the *dirty* queue that were not updated between two consecutive *D. Finder* iterations (approximately 5 minutes) are shared. The time period can be tuned independently for each VM. This mechanism is essential for keeping low storage overhead because DEDIS preemptively

marks written blocks as CoW so, without this mechanism, every re-write would generate a copy operation. In previous work, CoW is reduced by only marking a block when a duplicate is actually found at the storage, however, DEDIS does not assume a storage backend with locking capabilities so, implementing such strategy would require costly cross-host communication [4].

As other optimizations, a resilient in-memory cache of unused storage blocks addresses, from the persistent *free* pool, allows serving free blocks to CoW operations more efficiently. The throughputs of *D. Finder* and *GC* operations are improved by performing batch accesses to persistent logs, the DDI, the *extent* server and to the *free* pool, which allows using efficiently both disk and network resources. Moreover, the DDI nodes can serve batch requests efficiently without requiring the full index on RAM. Finally, although DEDIS implementation uses the SHA-1 hash function which has a negligible probability of collisions [17], the comparison of chunks bytes before aliasing them can be enabled independently for VM volumes persisting data from critical applications. This comparison requires reading the content of an extra block (referenced in the DDI entry) from the storage but it is performed outside the storage write path.

2.5 Implementation

DEDIS prototype is implemented within Xen and uses the Blktap mechanism for building the *interceptor* module. Blktap exports an user-level disk I/O interface that replaces the commonly used loopback drivers while providing better scalability, fault-tolerance and performance [3]. Each VM volume has an independent process intercepting VM disk requests with a fixed block size of 4KB, which is also the block size used in DEDIS.

The goal of the current implementation is to highlight the impact of deduplication, and not to re-invent a LVM system or the DDI. Simplistic implementations have thus been used for metadata and log structures. Namely, both the *logical-to-physical* map, *dirty* queue and *free* blocks queue cache are implemented as arrays fully loaded in memory, accessible by both *interceptor* and *D. Finder* modules. The *unreferenced* and *free* queues are implemented as persistent queues. The DDI is a slightly modified version of the Accord replicated, transactional and fully-distributed key-value store that supports atomic test-and-increment and test-and-decrement operations [20]. The *extent* server is implemented as a remote service with a persistent queue of unused storage blocks.

Despite being simplistic, all these structures are usable in a real implementation, so the resource utilization (*i.e.*, CPU, RAM, disk and network) in our evaluation is realistic. In fact, this implementation presents a worst-case scenario for the storage and RAM space occupied by metadata and persistent logs as more space-efficient structures could have been used instead.

3 Evaluation

This section validates the following properties of DEDIS. First, that an acceptable deduplication throughput is achievable and, the size needed for VM volumes

Table 1. Deduplication gain and throughput (Thr).

	DEDIS hash	DEDIS byte
Space shared (MB)	696	684
% VM data volume size reduced	17%	17%
Average Thr (MB/s)	4.78	4.55
Required continuous Thr (MB/s)	0,76	0,75

is reduced. Then, that deduplication does not overly impact write I/O latency even with deduplication and I/O intensive workloads running simultaneously. Finally, that DEDIS scales out for several cluster machines.

Tests ran in cluster nodes equipped with a 3.1 GHz Dual-Core Intel i3 Core Processor, 4GB of RAM and a 7200 RPMs SATA disk. The VMs were configured with 2GB of RAM and two disk volumes: a 16GB volume holding the Operating System (OS) and a 4GB data volume. Two fully-replicated DDI instances ran, for all tests, in isolated cluster nodes thus, including the overhead of remote calls and replication. The *extent* server ran together with one of the DDI instances. DEDIS, DDI and *extent* server persistent metadata and logs were stored in the local disks of cluster nodes to have a distinct storage pool for the logs and exclude their overhead from the VM data volumes. Similarly, OS volumes were stored in local disks and left out of this evaluation to avoid an unknown number of duplicates originating from OS images, and ensure that duplicate chunks are introduced in a controllable way by the benchmark.

As DEDIS targets dynamic data, static traces of VM images are not suitable for its evaluation, so the open-source DEDISbench disk micro-benchmark was used, in each VM, to assess the storage performance [16]. DEDISbench simulates realistic content for written blocks by following a content distribution extracted from real data sets that mimics the percentage and distribution of duplicates per block. Our tests used an workload that simulates the content of a primary storage, with ≈1.5 TB and 25% of duplicates, which fits our storage environment. DEDISbench also supports an access pattern based on the TPC-C NURand function that simulates a random I/O workload where few blocks are hotspots and most blocks are accessed sporadically. Moreover, to clearly understand the latency introduced by deduplication in storage requests, the *fsync* primitive was enabled in the benchmark to ensure synchronous storage writes.

I/O operations were measured at the VM (DomU) while, deduplication, CPU, metadata, RAM and network utilization were measured at the host (Dom0). Measurements were taken for stable and identical periods of the workloads, excluding ramp up and cool down periods, and, include the overhead of all DEDIS modules, both local and remote, as well as, the overhead of persistent logging.

3.1 Deduplication Results

Deduplication's overhead and performance were measured in a *single node setting* with one VM deployed on a single cluster node. VM data volumes, with 4GB,

Table 2. DEDIS overhead (o/h) in VMs storage latency (Lat) and throughput (Thr) with concurrent storage writes and deduplication.

	AIO	DEDIS w/o hspot	o/h	DEDIS hash	o/h	DEDIS byte	o/h
Thr (IOps)	720.528 ±13.730	636.031 ±14.403	11.73%	688.844 ±15.383	4,40%	662.863 ±22.316	8.00%
Lat (ms)	5.575 ±0, 106	6.470 ±0, 130	16.05%	5.850 ±0, 155	4.93%	6.094 ±0, 289	9.31%
% CoWs avoided	-	0	-	73.33%	-	72.89%	-

were stored in a HP StorageWorks 4400 Enterprise Virtual Array (EVA4400). As our cluster nodes could not directly access the EVA storage, volumes were exported via iSCSI and over a gigabit link by a server equipped with an AMD Opteron(tm) Processor 6172, 24 cores and 128 GB of RAM.

Table 1 shows deduplication space savings and throughput for a 90 minutes run of DEDISbench performing hotspot random writes (with a block size of 4K) and for the subsequent 30 minutes, where deduplication ran isolated from the I/O workload. 5 minutes were chosen as the interval between *D. Finder* iterations to obtain several iterations of the module during the test (≈16).

Tests ran for DEDIS prototype with hash (DEDIS hash) and byte (DEDIS byte) comparison enabled. As expected, DEDIS byte has a slightly lower deduplication throughput and, consequently, smaller space savings due to the extra computation for comparing the content of the blocks. DEDIS hash processes in average 4.78 MB/s and, this value is identical when *D. Finder* is processing requests simultaneously or isolated from the I/O workload.

We also calculated the minimum continuous deduplication throughput needed to keep up with this workload, for an unbounded amount of time, without accumulating unprocessed duplicate storage backlog. The value is ≈0,76 MB/s and DEDIS is able to accomplish it.

Some writes are not processed by the *D. Finder*. First, multiple updates to the same block, between two iterations, originate a single share operation for the latest content written. Also, the hotspot avoidance mechanism avoids sharing frequently updated blocks. In this run, the *D. Finder* only processed 1 million blocks while the benchmark wrote ≈13G. Both DEDIS versions deduplicated approximately 17% of the original data volume size (4GB), which is smaller than the 25% of duplicates simulated by DEDISbench workload. However, as explained in the benchmark's paper, the algorithm only converges to the expected percentage of duplicates for higher volumes of written data.

3.2 Performance and Resources Consumption Results

To assess DEDIS's performance we compared it with the default Blktap driver for asynchronous I/O, Tap:aio, that was the base to implement DEDIS *interceptor* and does not perform deduplication [3]. This comparison ensures that the

overhead is a direct consequence of DEDIS deduplication. Unfortunately, a direct comparison with DDE or DeDe systems was not possible as they are not publicly available. The tests' setup was identical to the previous one and the Tap:aio VM data volumes were also stored on the EVA storage. DEDIS and the I/O benchmark ran simultaneously to evaluate the impact of deduplication and garbage collection in peak hours. A 5 minute interval between *D. Finder* and *GC* iterations was chosen to assess the benefits of the hotspot avoidance mechanism, which was configured to share blocks that were not written or re-written in the interval comprehending the current and previous *D. Finder* iterations.

Table 2 shows the results of performing hotspot random writes, during 30 minutes, for Tap:aio and three DEDIS versions. The first, (DEDIS w/o hspot) is the only that does not use hotspot avoidance. The second, (DEDIS hash) uses hash comparison while, the other (DEDIS byte) performs byte comparison. As expected, DEDIS introduces overhead in both storage throughput and latency, however, this value is small when compared to previous systems [4] and, accounts the impact of *D. Finder*, *GC*, *DDI*, *extent* server and corresponding persistent logging mechanisms while running in parallel with the I/O workload.

A significant amount of this overhead is due to CoW operations. In DEDIS each operations requires ≈7ms to be executed. This is already an improvement over previous systems where CoW requires 10ms in servers with more computational resources than ours [4]. This is possible because DEDIS uses a memory cache that provides free blocks for CoW and, performs batch insertions in the *unreferenced* pool, which is a time consuming operation in the critical I/O path. However, as shown in Table 2, the overhead without the hotspot avoidance mechanism is still significant while, with this mechanism, DEDIS performs 70% less CoW operations (≈210,000 less), which enables a clear reduction in I/O latency.

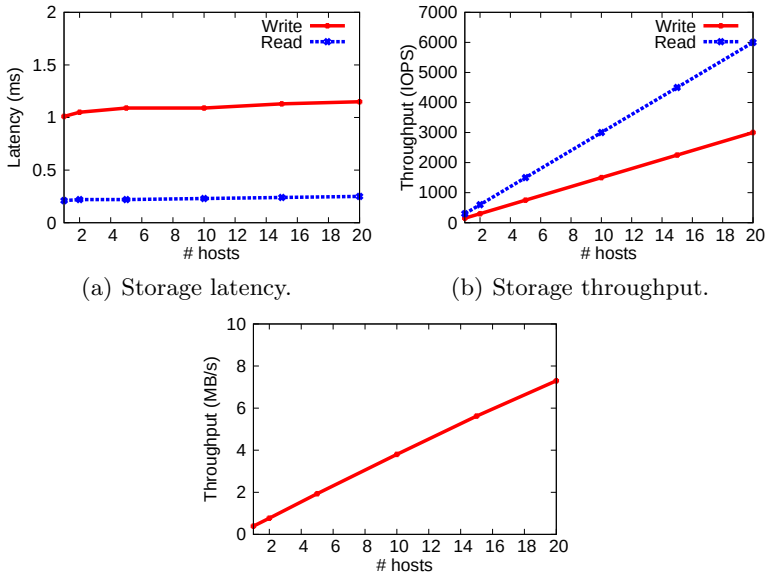
We compared the resource consumptions of DEDIS and Tap:aio for the run described in Section 3.1. DEDIS local modules have a CPU usage of ≈ 14%, only 5% more than Tap:aio, which is a consequence of performing background deduplication. Moreover, DEDIS modules use less than 1% of the node's total RAM and require ≈75 MB to store persistent logs and metadata.

The DDI service uses less than 5% of CPU, 0,35% of the nodes' RAM while, requiring 80MB of disk space and ≈2.2 KB/s of average network usage. These values include the costs of indexing and persisting signatures of 4KB blocks in a replicated fashion. As shown in Section 3.1, the disk usage is nevertheless compensated by the deduplicated space, ≈700 MB.

Finally, the *extent* server uses less than 1Mb of RAM and has a negligible CPU usage since, in these tests, it is only called for the initial allocation of blocks to the local *free* pool. The *extent* queue requires ≈5 MB of storage space for indexing ≈11 GB of unused addresses.

3.3 Scalability Results

Scalability was assessed in a *distributed setting* with up to 20 cluster nodes. After performing some tests, we observed that the EVA storage was having a significant latency degradation when handling I/O requests from all the nodes,



(a) Storage latency.

(b) Storage throughput.

(c) Deduplication throughput.

Fig. 3. DEDIS results for up to 20 cluster nodes with mixed storage writes and reads.

even with the Tap:aio baseline. Therefore, data volumes were exported by a 110GB RAMdisk ISCSI device, in our AMD server, which increased significantly the load supported by the storage backend.

A mixed load of write and read requests was used to test the performance of both I/O operations in parallel with DEDIS performing hash comparison. To ensure that the the network link supporting the iSCSI protocol would not be overloaded by the aggregated throughput of the 20 cluster nodes, the throughput of DEDISbench was limited to 300 reads/s and 150 writes/s, per VM. Tap:aio baseline was not evaluated in this test since the goal was to prove that DEDIS scales for several nodes. DEDISbench ran for 30 minutes in each VM and the subsequent 20 minutes were used to observe DEDIS behavior without I/O load. Tests ran for 1,2,5,10,15 and 20 cluster nodes hosting a single VM. Finally, deduplication throughput, in each node, was limited to 100 ops/s to have an uniform throughput over the entire cluster, *i.e.,* up to 20× more.

As depicted in Figures 3(a) and 3(b), the throughput for both read and write requests scales linearly up to 20 nodes. There is a slight increase in the latency of writes and reads when nodes are added, which is a consequence of having more load in the centralized storage. Figure 3(c) shows the aggregated deduplication throughput increasing up to 20 nodes which, in our setup, is near the maximum capacity of the DDI to process parallel requests with a fixed throughput of 100 requests/s. As in the single server tests, each DDI instance consumes a small

amount of RAM, CPU and network so, infrastructure costs can be reduced by running these instances in servers with other services or where VMs are deployed.

4 Related Work

Recently, live volume deduplication in cluster and enterprise scale systems is becoming popular. Opendedup [13] and ZFS [14] support primary multi-host inline deduplication but are designed for enterprise storage appliances, and require large RAM capacities for indexing chunks and to enable efficient deduplication.

Primary distributed off-line deduplication for a SAN file system was introduced in the Duplicate Data Elimination (DDE) system, implemented over the distributed IBM Storage Tank [7]. DDE has, however, a centralized single-point of failure metadata server for sharing duplicate chunks asynchronously, which was removed in DeDe, an off-line distributed deduplication system for VM volumes on top of VMWares's VMFS cluster file system [4]. The index of chunks is stored on VMFS and is accessible to all nodes while, index lookups are made in batch to increase deduplication throughput. VMFS simplifies deduplication as it already has explicit locking, block aliasing, CoW, and reference management, which are not present in most cluster file systems. These primitives are combined to implement the atomic share function that replaces two duplicate fixed-size blocks with a CoW block. However, this dependency leads to alignment issues between the block size used in VMFS and DeDe, implying additional translation metadata and, consequently, an impact in storage requests latency. Also, the overhead of CoW operations in storage I/O is significant, forcing DeDe deduplication to run only in periods of low I/O load. CoW overhead may, however, be reduced by deduplicating selectively files that meet a specific policy such as, file age superior to a certain threshold, as suggested in Microsoft Windows Server 2012 centralized off-line deduplication system [5].

DDE and DeDe are the systems that most resemble DEDIS, however, DEDIS is fully-decentralized and does not dependent on a specific cluster file system. This way, there are no single point of failures and an unsophisticated storage implementation, centralized or distributed, can be used as backend as long as a shared block device interface is provided for the storage pool. Decoupling deduplication from the storage backend changes significantly DEDIS's design, allows exploring novel optimizations and avoids DeDe alignment issues. In fact, and as explained in this paper, these design changes and optimizations are key for having a scalable design and for running deduplication and I/O intensive workloads simultaneously with low overhead, which is not possible in previous systems.

5 Conclusions

We presented DEDIS, a dependable and distributed system that performs cluster-wide off-line deduplication across primary storage volumes. The design is fully-decentralized avoiding any single point of failure or contention thus, safely scaling-

out. Also, it is compatible with any storage backend, distributed or centralized, that exports a shared block device interface.

The evaluation of a Xen-based prototype in up to 20 nodes shows that by relying on an optimistic deduplication algorithm and on several optimizations, deduplication and primary I/O workloads can run simultaneously in a scalable system. In fact, DEDIS introduces less than 10% of latency overhead while maintaining a baseline single-server deduplication throughput of 4.78 MB/s with low-end hardware. This is key for performing efficient deduplication and reducing the storage backlog of duplicates in infrastructures with scarce off-peak periods. Also, even with a trivial implementation of a LVM system, deduplication space savings compensate metadata overhead, while maintaining an acceptable consumption of CPU, RAM and network resources.

As future work, we would like to evaluate DEDIS in a scalable distributed storage environment, where both DEDIS metadata and VMs volumes would be stored, and with other primary storage workloads with higher duplication ratios.

6 Acknowledgments

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP 01-0124-FEDER-022701 and FCT by Ph.D scholarship SFRH-BD-71372-2010.

7 Availability

DEDIS system is open-source and is publicly available at `http://www.holeycow.org` for anyone to deploy and benchmark.

References

1. William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of USENIX Windows System Symposium (WSS)*, 2000.
2. Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The Diverse and Exploding Digital Universe: An updated forecast of worldwide information growth through 2011. IDC White Paper - sponsored by EMC. http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf, 2008.
3. Citrix. Blktap page. `http://wiki.xen.org/wiki/Blktap2`. January, 2014.
4. Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2009.
5. Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. Primary Data Deduplication Large Scale Study and System Design. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2012.

6. Hewlett-Packard Development Company , L.P. Complete storage and data protection architecture for vmware vsphere. *White Paper*, 2011.
7. Bo Hong and Darrell D. E. Long. Duplicate Data Elimination in a San File System. In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2004.
8. Lessfs. Lessfs page. `http://www.lessfs.com/wordpress/`. January, 2014.
9. Anthony Liguori and Eric Van Hensbergen. Experiences with Content Addressable Storage and Virtual Disks. In *Proceedings of USENIX Workshop on I/O Virtualization (WIOV)*, 2008.
10. Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual Disks for Virtual Machines. In *Proceedings of European Conference on Computer Systems (EuroSys)*, 2008.
11. Dutch T. Meyer and William J. Bolosky. A Study of Practical Deduplication. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011.
12. Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P. C. Lee, and John C. S. Lui. Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, 2011.
13. Opendedup. Opendedup page. `http://opendedup.org`. January, 2014.
14. OpenSolaris. Zfs documentation. `http://www.freebsd.org/doc/en/books/handbook/filesystems-zfs.html`. January, 2014.
15. J. Paulo and J. Pereira. Model checking a decentralized storage deduplication protocol. *Fast Abstract in Latin-American Symposium on Dependable Computing*, 2011.
16. J. Paulo, P. Reis, J. Pereira, and A. Sousa. Towards an Accurate Evaluation of Deduplicated Storage Systems. *International Journal of Computer Systems Science and Engineering*, 29, 2013.
17. Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2002.
18. Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012.
19. Y. Tsuchiya and T. Watanabe. DBLK: Deduplication for Primary Block Storage. In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2011.
20. Tsuyoshi, Ozawa and Kazutaka, Morita. Accord page. `http://www.osrg.net/accord/`. January, 2014.
21. Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2010.