Efficient Deduplication in a Distributed Primary Storage Infrastructure

JOÃO PAULO and JOSÉ PEREIRA, High-Assurance Software Lab (HASLab), INESC TEC,
and University of Minho

A large amount of duplicate data typically exists across volumes of virtual machines in cloud computing infrastructures. Deduplication allows reclaiming these duplicates while improving the cost-effectiveness of large-scale multitenant infrastructures. However, traditional archival and backup deduplication systems impose prohibitive storage overhead for virtual machines hosting latency-sensitive applications. Primary deduplication systems reduce such penalty but rely on special cluster filesystems, centralized components, or restrictive workload assumptions. Also, some of these systems reduce storage overhead by confining deduplication to off-peak periods that may be scarce in a cloud environment.

We present DEDIS, a dependable and fully decentralized system that performs cluster-wide off-line deduplication of virtual machines' primary volumes. DEDIS works on top of any unsophisticated storage backend, centralized or distributed, as long as it exports a basic shared block device interface. Also, DEDIS does not rely on data locality assumptions and incorporates novel optimizations for reducing deduplication overhead and increasing its reliability.

The evaluation of an open-source prototype shows that minimal I/O overhead is achievable even when deduplication and intensive storage I/O are executed simultaneously. Also, our design scales out and allows collocating DEDIS components and virtual machines in the same servers, thus, sparing the need of additional hardware.

Categories and Subject Descriptors: D.4.2 [**Software**]: Operating Systems—*Storage management*; D.4.7 [**Software**]: Operating Systems—*Organization and design*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Primary storage, deduplication, distributed systems

1. INTRODUCTION

A study conducted by International Data Corporation (IDC) projects that digital information will reach 40ZB by 2020, which exceeds previous forecasts by 5ZBs, resulting in a 50-fold growth from the beginning of 2010 [EMC 2012]. The study also estimates that by 2020, nearly 40% of the worldwide data will be stored and processed by cloud

computing infrastructures that rely heavily on Virtual Machines (VMs) for hosting cloud services and client applications. With this unprecedented growth of data and the introduction of more expensive storage devices, such as Solid State Drives (SSDs), space-saving techniques such as deduplication are key to reduce the costs of cloud infrastructures [D. Iacono 2013]. As a matter of fact, deduplication is currently accepted as an efficient technique for reducing storage costs at the expense of some additional processing. Moreover, this technique is no longer an exclusive feature of archival and backup storage systems, being now increasingly sought in primary storage systems and cloud computing infrastructures, namely, across VMs' volumes [Srinivasan et al. 2012; El-Shimi et al. 2012; OpenSolaris 2014; Hong and Long 2004; Clements et al. 2009; Ng et al. 2011].

As static VM images are highly redundant, many systems avoid duplicates by storing Copy-on-Write (CoW) golden images and then use snapshot mechanisms for launching identical VM instances [HP 2011; Meyer et al. 2008]. In order to further improve deduplication space savings, other systems also target duplicates found in dynamic general purpose data stored on the VMs' volumes. In many situations, these volumes are used as primary storage for distinct types of services, such as email servers, web servers, application servers, and Network-Attached Storage (NAS) servers [Koller and Rangaswami 2010a]. Several clients run these services in independent VMs with their own volumes, meaning that it is possible to find not only duplicate software, files/emails in a single VM volume but also across the volumes of distinct VMs deployed at the same cluster infrastructure. In fact, by finding duplicates for both static and dynamic data in a cluster-wide fashion, space savings may range from 58% up to 80% [Clements et al. 2009; Meyer and Bolosky 2011, 2012; Srinivasan et al. 2012].

However, in spite of the considerable space savings, primary storage deduplication in a cluster infrastructure raises novel challenges that are not addressed by traditional archival and backup deduplication systems.

1.1. Challenges

For archival and backup data, most applications favor storage throughput over latency as data is usually stored and retrieved sequentially in large batches. However, for primary data, random access patterns are expected while many applications have strict latency requirements for their storage requests [Hong and Long 2004; Clements et al. 2009; Paulo and Pereira 2014b]. These differences explain why most of the traditional archival/backup storage systems use *inline* deduplication, thus removing duplicates before storing data. However, finding and sharing duplicates can be a costly operation that requires additional computational resources and, in most cases, it requires additional storage accesses. Doing these additional operations in the storage write path increases significantly the latency of storage write requests [Quinlan and Dorward 2002; Ng et al. 2011; Srinivasan et al. 2012; Paulo and Pereira 2014b].

As this overhead is unacceptable for many applications writing and updating primary data, another option is to use *off-line* deduplication that minimizes the impact in storage writes by removing the additional computation and storage accesses to find duplicates from the storage write path [Hong and Long 2004; Clements et al. 2009]. However, as data is only aliased after being stored, off-line deduplication temporarily requires additional storage space. Also, deduplication and storage requests are performed asynchronously so appropriate mechanisms for preventing stale data checksums and other concurrency issues are necessary and may degrade performance and scalability.

As another challenge, primary data volumes have data hotspots that are modified frequently. For instance, this property is visible in the real traces used to evaluate our prototype [Koller and Rangaswami 2010a]. More specifically, in the tests with

a mail server trace, described in Section 5, 39% of the write requests are rewrites, corresponding approximately to a throughput of 41 blocks being rewritten per second. This means that an efficient CoW mechanism is needed for preventing in-place updates on aliased data and potential data corruption. For instance, if two VMs are sharing the same data block and one of them needs to update that block, the new content is written into a new and unused block (copied on write) because the shared block is still being used by the other VM. This mechanism introduces even more overhead in the storage write path while increasing the complexity of reference management and garbage collection, thus forcing some systems to perform deduplication only in off-peak periods in order to avoid a considerable performance degradation [Clements et al. 2009]. Since cloud infrastructure hosts VMs from several clients that provide different services for different countries with distinct timezones, off-peak periods to perform deduplication across all these VMs are very scarce or even inexistent. This way, off-line deduplication has a short time-window for processing the VMs storage backlog and eliminating duplicates. Ideally, deduplication should run continuously and duplicates should be kept on disk for short periods of time, thus reducing the extra storage space required.

Finally, distributed cloud infrastructures raise additional challenges as deduplication must be performed globally across volumes belonging to VMs deployed on remote cluster servers [Hong and Long 2004; Clements et al. 2009]. Space savings are maximized if duplicates are found and eliminated globally across the entire cluster. Also, if deduplication is done in a decentralized fashion where each server is responsible for finding and eliminating duplicates for its hosted VMs, it is possible to increase deduplication parallelism and to scale out to larger clusters [Kaiser et al. 2012]. However, this is a complex design that requires a remote indexing mechanism, accessible by all cluster servers, that is used for tracking unique storage content and finding duplicates. Remotely accessing this index in the critical storage path introduces prohibitive overhead for primary storage workloads and invalidates, once again, in-line deduplication. In fact, this negative impact leads to deduplication systems that perform exclusively local server deduplication or that relax deduplication's accuracy and find only some of the duplicates across cluster nodes [You et al. 2005; Bhagwat et al. 2009; Dong et al. 2011; Fu et al. 2012; Frey et al. 2012].

1.2. Assumptions and Contributions

The combined challenges of primary storage and global deduplication are addressed with DEDIS, a dependable and fully decentralized system that performs optimistic cluster-wide off-line deduplication of VMs' primary volumes, while excluding most of the deduplication processing from the storage write path.

Our design assumes that VMs' volumes are stored persistently in a storage backend, also referred to as a storage pool within the article, exporting an unsophisticated shared block device interface that may be distributed or centralized. For instance, this backend may be a Storage Area Network (SAN) system or another similar storage environment. Previously distributed primary off-line deduplication systems, namely DDE and DEDE, respectively, store VMs' volumes on the IBM Storage Tank filesystem with built-in locking operations, and on the VMWares's VMFS filesystem that has built-in locking, aliasing, CoW and garbage collection operations [Hong and Long 2004; Clements et al. 2009]. As one of the main novelties of our system, DEDIS does not rely on storage backends with any of these special mechanisms. Instead, each server hosting VMs is responsible for performing deduplication for the volumes of those VMs. Although this decision significantly impacts the system design and favors distinct optimizations, as discussed in Section 4, it allows decoupling our deduplication system from a specific storage implementation and avoids performance issues that arise from this dependency.

In current cloud computing solutions, such as the one provided by OpenStack, an open-source project for building and managing cloud computing platforms, VMs' volumes can be mapped to persistent high performance block storage devices for storing both the Operating System (OS) and primary data efficiently [OpenStack Foundation 2014, 2016]. In OpenStack, these block devices are managed by the Cinder system that uses as the default storage backend a traditional Logical Volume Management (LVM) system. With the DEDIS approach, traditional LVM storage backends without built-in deduplication may be used, and efficient global deduplication across VMs' volumes is achievable without modifying the implementation of these storage backends. In fact, as shown in the article, our approach only requires modifying the virtual disk I/O interface of VMs, which, in most hypervisors such as Xen and KVM, is possible with user-space toolkits, thus, not requiring an intrusive approach where the hypervisor implementation must be modified [Russell 2008; Citrix Systems, Inc 2014; Jones, M. 2010].

On the other hand, primary storage in-line deduplication systems such as iDedup, LiveDFS, and DBLK rely on storage workloads exhibiting data locality properties to reduce the storage overhead caused by deduplication and present systems that only perform centralized deduplication [Tsuchiya and Watanabe 2011; Ng et al. 2011; Srinivasan et al. 2012]. DEDIS design does not depend on storage workloads exhibiting specific data locality properties to achieve low storage overhead, and deduplication is done in an exact fashion across the whole cluster; more specifically, all duplicate chunks are processed and eventually shared. For clarity purposes, in this article, we refer to *chunks* as the unit of deduplication, which in DEDIS corresponds to fixed-size blocks. Most systems do not compare the full content of chunks, and use instead compact signatures of the chunks' content. These are generally calculated with hashing functions and we refer to them as *chunk signatures* or *digests* [Paulo and Pereira 2014b].

Briefly, DEDIS novel optimistic deduplication approach works as follows: Locally, in each server hosting VMs, storage writes from these VMs to their volumes, at the storage backend, are intercepted with a fixed block size granularity and redirected immediately to the correct storage address by a layer that considers aliased chunks. This decision avoids costly accesses to remote metadata and reference management in the critical storage path. In each server, written blocks are collected asynchronously and off-line deduplication is performed globally and exactly across the entire cluster by using a partitioned and replicated fault tolerant distributed service that maintains both the index of unique chunks' signatures and the metadata necessary for reference management and garbage collection. This service allows DEDIS to be fully decentralized and to scale-out. Finally, volumes belonging to failed cluster nodes can be recovered and restarted in other cluster nodes by using a persistent logging mechanism that stores the necessary metadata in a shared storage pool which, for performance reasons, may or may not be the same where volumes are stored. Once again, this approach excludes a dependency on any specific storage backend with special operations while also avoiding any cross-host communication between servers holding distinct VMs.

As other contributions, we present novel optimizations for improving deduplication performance and further reducing its impact in storage requests. These are novel optimizations that do not rely on storage workloads exhibiting data locality properties in order to be efficient. Namely, DEDIS detects blocks that are potentially write hotspots and avoids sharing such blocks, which reduces significantly the number of CoW operations and their overhead in storage requests. Storage latency overhead is then further reduced by using in-memory caches and batch processing, which are also useful for increasing deduplication throughput. Also, DEDIS can be configured to withstand hash collisions in specific VM volumes by performing byte comparison of chunks before aliasing them.

A final contribution is a detailed experimental evaluation of the DEDIS prototype with both real traces and a realistic benchmark. The results show that in a setup with up to 32 servers, DEDIS introduces low overhead in storage I/O requests, less than 14%, while maintaining acceptable deduplication throughput and resource consumption. Also, our design scales out along with the storage backend for large-scale infrastructures. The evaluation is performed in a fully symmetric setup where servers run both VMs and DEDIS components. This way, our prototype does not require additional servers for running DEDIS services. In fact, in the distributed setup with several servers, even the storage backend, where VMs volumes and DEDIS persistent metadata are stored, is composed by the local disks of the same servers.

This work is focused on achieving efficient deduplication and low storage overhead for random storage workloads. As explained previously and as further detailed in the next section, this is a common type of workload for primary storage systems that current proposals cannot handle efficiently. Nevertheless, in Section 6 we address this assumption in more detail and discuss future research directions for building deduplication systems that handle both random and sequential storage workloads efficiently. Also, the article extends preliminary work [Paulo and Pereira 2014a] by presenting a new evaluation setup with multiple servers' configurations and storage workloads, by presenting a novel cache optimization for improving deduplication throughput while reducing DEDIS impact on storage requests, and by discussing in more detail the concurrent optimistic deduplication approach and fault-tolerant design of our system.

The article is structured as follows: Section 2 summarizes background work on primary storage deduplication and the main differences between DEDIS and state-of-the-art systems. Section 3 describes the baseline distributed architecture assumed by our system. Section 4 presents DEDIS components, fault-tolerance considerations, optimizations, and implementation details. Section 5 presents the evaluation of the open-source prototype. Finally, Section 6 discusses the applicability of this work for sequential storage workloads and future research directions, while Section 7 concludes the article.

2. BACKGROUND AND RELATED WORK

Traditional in-line deduplication systems target archival and backup data, favoring storage throughput over latency. This explains why previous proposals to extend systems like Venti and HYDRAstor with file system semantics are able to achieve good performance for stream I/O (sequential reads and writes), while supporting random block storage requests but with unacceptable performance for primary storage environments like the one targeted by DEDIS [Ungureanu et al. 2010; Liguori and Van Hensbergen 2008; Lessfs 2014]. Other backup systems perform off-line deduplication, but these systems are either optimized to eliminate duplicates at the file granularity, reducing the achievable space savings, or rely on centralized indexes [Bolosky et al. 2000; Douceur et al. 2002; Yang et al. 2010]. Unlike these systems, DEDIS is fully decentralized and eliminates duplicates at the fixed-size block (4KB) granularity. Yet another possible combination is discussed in RevDedup, where in-line and off-line deduplication are combined [Li et al. 2014]. Coarse-grained in-line deduplication is used for newer backups in order to maintain fast restore speeds and to spare storage space. Then, fine-grained off-line deduplication is done for older backups to further increase space savings and to optimize the process of deletion of these older backups. Once again, these optimizations focus on storing, restoring, and deleting large portions of data in a small time window, thus favoring storage throughput over latency.

Recently, live volume deduplication in cluster and enterprise scale systems is emerging. Logical LVM systems with snapshot capabilities, such as Parallax, avoid duplicating data, but only among snapshots of VM volumes and golden VM images with common

ancestors [Meyer et al. 2008]. In fact, as explained in the work of Jin and Miller [2009], deduplication is an efficient method for reducing the storage space of VM images, even when these images do not have common ancestors. However, DEDIS aims not only at finding redundancy across VM images' static information, but also across dynamic data stored on VMs' volumes belonging to distinct software/applications running at these VMs.

Other systems like Opendedup and ZFS support multihost in-line deduplication for dynamic data, but are designed for enterprise storage appliances and require large RAM capacities for indexing chunks and enabling efficient deduplication [Opendedup 2014; OpenSolaris 2014].

These limitations shift focus to off-line deduplication where processing overhead is excluded from the storage write path and lower latency is achievable. 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 [Hong and Long 2004]. A centralized metadata server receives signatures of stored chunks and deduplicates them asynchronously by resorting to an index of unique signatures stored at the SAN. A CoW mechanism avoids updates on aliased data, while reference counting information, required for reference management, is stored on an independent metadata structure.

One of the major drawbacks of DDE is the single-point of failure centralized metadata server, so this centralized component is avoided in DeDe [Clements et al. 2009]. DeDe introduces an off-line decentralized deduplication algorithm for VM volumes on top of VMWares's VMFS cluster file system. DeDe uses an index structure, also stored in VMFS, that is accessible to all nodes and protected by a locking mechanism. Efficient deduplication throughput is obtained by doing index lookups and updates in batch, while index partitioning allows a scalable design. VMFS simplifies deduplication as it already has explicit block aliasing, CoW, and reference management. However, these operations are not commonly exposed in most cluster file systems and the performance of the deduplication system is highly dependent on their implementation. For instance, there are alignment issues between the block size used in VMFS and DeDe, implying additional translation metadata and an additional impact in storage requests latency. This penalty, along with the significant overhead of CoW operations, confines DeDe deduplication to run in periods of low I/O load. A proposal for reducing the overhead of CoW operations in storage requests is described in Microsoft Windows Server 2012 centralized off-line deduplication system, where it is suggested that deduplication should be performed selectively on files that meet a specific policy, such as, file age superior to a certain threshold [El-Shimi et al. 2012]. Such a policy avoids sharing fresh files that are more prone to generate CoW operations.

DDE and DeDe are the systems that most resemble DEDIS. However, DEDIS is fully decentralized and does not depend on a specific cluster file system. This distinction allows removing existing single point of failures while also handling unsophisticated storage implementations as backend, centralized, or distributed, as long as a shared block device interface is provided for the storage pool. Decoupling deduplication from the storage backend changes significantly the design of DEDIS and allows exploring novel optimizations while avoiding the alignment issues of DeDe. For example, as detailed in Section 4.5, DeDe's mechanism to tentatively mark addresses as CoW is implemented by recurring to the storage backend locking capabilities. Implementing this mechanism in DEDIS without the lock primitive would require costly cross-host communication, so we introduce a novel mechanism for avoiding I/O hotspots and, consequently, CoW operations. Also, as CoW specialization is not provided by our storage backend, novel cache mechanisms can be used to reduce its impact in storage requests. In fact, these optimizations are key for running deduplication

and I/O intensive workloads simultaneously with low overhead, unlike in previous systems.

Recently, several optimizations were proposed by iDedup, LiveDFS, DBLK, and HANDS to reduce the storage latency overhead of in-line primary storage deduplication. These optimizations focus on speeding up the index lookup operations by avoiding disk accesses that are costly and done in the storage write path [Tsuchiya and Watanabe 2011; Ng et al. 2011; Srinivasan et al. 2012; Wildani et al. 2013]. Briefly, these systems use Bloom filters and explore the spatial and temporal locality of storage workloads with novel disk layouts, pre-fetching algorithms, and cache mechanisms. In fact, many of these optimizations are based on mechanisms previously thought for archival and backup deduplication [Zhu et al. 2008; Rhea et al. 2008; Lillibridge et al. 2009; Guo and Efstathopoulos 2011; Shilane et al. 2012; Wei et al. 2010; Quinlan and Dorward 2002; Zhu et al. 2008; Guo and Efstathopoulos 2011; Debnath et al. 2010; Xia et al. 2011; Fu et al. 2012; Dong et al. 2011]. However, even with these optimizations, these in-line primary deduplication systems are designed for centralized storage appliances as introducing remote index lookups in the critical I/O path results in prohibitive storage overhead.

In order to support cluster-wide in-line deduplication, the Dedupv1 centralized system was extended to perform deduplication over a shared storage device (SAN) where each node has exclusive access to its own data partition and index shard, which is stored in a separate SSD partition [Meister and Brinkmann 2010; Kaiser et al. 2012]. Each node is responsible for performing data partitioning, hash calculation, and for routing the requests of chunks to be deduplicated to the nodes with the corresponding index entries. This approach requires cross-host communication and, since in-line deduplication is done, it adds unwanted overhead in storage write requests that is reduced with a write-back cache and a write-ahead log. In DEDIS, cross-host communication is avoided, and since off-line deduplication is used, there is no need for the write-back cache and the write-ahead log.

In comparison with other archival, backup, and primary in-line deduplication work, DEDIS does not require data locality or keeping metadata structures in SSDs to have acceptable deduplication throughput and reduced storage I/O overhead [Paulo and Pereira 2014b]. Index lookups are optimized by performing them in batch and outside from the critical I/O storage path. Also, the index is not assumed to be fully loaded in RAM and can be partitioned to improve throughput and scalability. DEDIS performs exact deduplication across all cluster nodes, i.e., all stored chunks are compared against each other, thus having optimal deduplication gain. Finally, DEDIS deduplication is decentralized so each cluster node performs deduplication tasks independently and concurrently.

3. BASELINE ARCHITECTURE

Figure 1 outlines the distributed primary storage architecture assumed by DEDIS. A number of physical disks are available over a network to physical hosts running multiple VMs and are used as the storage backend to persist the volumes of these VMs. We assume that these disks are exported to the physical hosts as an unsophisticated shared block device interface; for instance, it can be a SAN system or another similar storage environment. Also, this storage backend may have features like replication and striping that are transparent to the physical hosts.

Together with the hypervisor, storage management services provide logical volumes to VMs by translating *logical addresses* within each volume to the corresponding *physical addresses* at the storage backend (physical 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
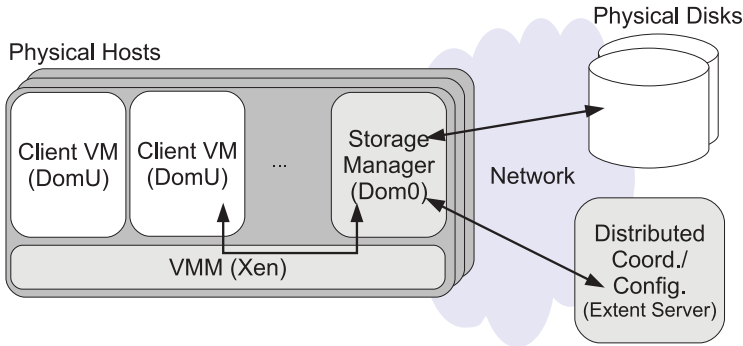
Fig. 1.   Distributed storage architecture assumed by DEDIS.

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 VMs logical to physical storage addresses, storing the physical location of each logical block in a persistent mapping structure. In some LVM systems, this module supports the creation of snapshots by pointing multiple logical volumes to the same physical locations [Meyer et al. 2008]. Logical addresses sharing a physical location must be marked as CoW. Then, updates to these addresses must write the new content to a free block and update the mapping accordingly.

Extent server. A distributed coordination mechanism allocates free blocks from the storage backend (physical disks) when a logical volume is created, lazily when a block is written for the first time, or when an aliased block is updated (i.e., copied on write). Storage extents are allocated with a large granularity and are then, within each physical host, used to satisfy individual block allocation requests, thus reducing the overhead of contacting a remote service [Meyer et al. 2008].

The architecture presented in Figure 1 is a logical architecture, as physical disks and, even the instances of the distributed coordination and configuration service itself, can be contained within the same physical hosts. For simplicity, we assume that the Xen hypervisor is being used and label payload VMs as DomU and the storage management VM as Dom0. Also, in DEDIS evaluation, iSCSI and Fiber Channel are used as the storage networking protocols. However, the architecture is generic and can be implemented within other hypervisors while using other networked storage protocols. Since we focus on the added functionality needed for deduplication, we do not target a specific data structure for mapping from logical to physical addresses. We also do not require built-in volume snapshot or CoW functionalities, as we introduce our own operations. Finally, DEDIS operates with fixed-size blocks because the *interceptor* module also processes requests at the fixed-size block granularity, and generating variable-sized chunks would impose unwanted computation overhead [Hong and Long 2004; Clements et al. 2009].

4. THE DEDIS SYSTEM

4.1. Architecture

The main novelty of DEDIS architecture, depicted in Figure 2, is that it does not require introducing a centralized component. Instead, in addition to the baseline architecture, it uses only a *distributed* module and two *local* modules. These are highlighted in the figure by the dashed rectangle and provide the following functionality:
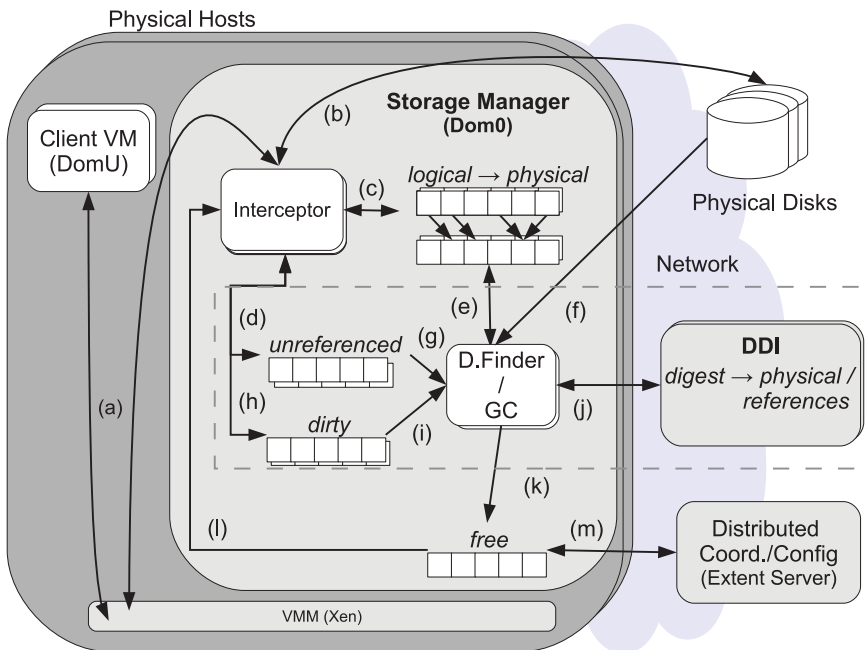
Fig. 2. Overview of the DEDIS storage manager.

Distributed Duplicates Index (DDI). A distributed module that indexes unique content signatures of storage's blocks. Each entry maps a unique signature to the physical storage address of the corresponding block and to the number of logical addresses pointing to (sharing) that block. This information allows aliasing duplicate blocks and performing reference management and garbage collection of unreferenced blocks. Index entries are persistent and are not required to be fully loaded on RAM to have efficient lookup operations. 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 a single node can index many blocks. This way, the index scales out without having any single point of failure.

Duplicate Finder (D. Finder). A local module that asynchronously collects addresses written by local *interceptors*, which are stored in a *dirty* addresses queue, and shares the correspondent blocks with other blocks 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). A local module that processes copied on write blocks or, in other words, aliased blocks that were updated and are no longer being referenced (aliased) by a certain logical address. The physical addresses of copied blocks are kept at the *unreferenced* queue, and the number of references to a certain block can be consulted and decremented at the DDI. Copied blocks can be freed if the number of references reaches zero. Both *D. Finder* and *GC* modules free unused blocks by registering their physical addresses in a local *free* blocks pool that provides unused block addresses for CoW operations and, when necessary, inserts/retrieves unused block addresses from the remote *extent* server.

4.2. I/O Operations

The operations executed by DEDIS modules are depicted in Figure 2. Bidirectional arrows mean that information is both retrieved and updated at the target resource. The *GC* and *D. Finder* modules are included in the same process box because both run in a background multithreaded process within the Xen Dom0, i.e., run in distinct threads of the same process.

An *I/O operation in the Interceptor*. The interceptor (a) gets read and write requests from local VMs, (c) queries the logical-to-physical mapping for the corresponding physical addresses; and (b) redirects them to the storage backend (physical disks) over the network. As potentially aliased blocks must be marked in the mapping as CoW by *D. Finder*, writes to such blocks must first (l) collect a free block address from the *free* pool, (b) redirect the write request to the free block and (c) update the map accordingly. Then, (d) the physical address of the copied block is inserted in the *unreferenced* queue to be processed later by the *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 runs periodically and 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), which marks the address for CoW (e), reads its content at the storage backend (f), computes a signature, and queries the DDI in search of an existing known duplicate (j). This is done using a test-and-increment remote operation, which stores the block's information (hash, physical address, and number of references) 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 and, 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 inserted in the *free* pool.

Freeing an unused block in GC. This background module examines if a copied block at the *unreferenced* queue (g) has become unreferenced with the last CoW operation. The block's content is read from the storage backend (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 returned to the *free* pool (k). This pool keeps only the addresses needed for local CoW operations, while the remainder are returned to the remote *extent* server (m). When the queue is empty, unused addresses are requested from the *extent* server (m).

Each VM volume has its own latency-sensitive *interceptor* module running as an independent process. This module does not invoke any remote services and only blocks in the unlikely case of having an empty local *free* pool which can easily be avoided by tuning the frequency of the *GC* execution. Also, each VM volume has an independent logical-to-physical mapping, *dirty* queue, and *unreferenced* queue. Finally, both in *D. Finder* and *GC* modules, an independent thread processes the operations for each VM volume. This way, the only metadata structure shared across all VMs, in the same server, is the *free* pool that is protected from concurrent accesses by private caches that reduce access frequency.

The *test-and-increment* and *test-and-decrement* operations and the metadata stored in each DDI entry allow performing the lookup of storage block signatures and corresponding physical addresses while incrementing or decrementing the entry's logical references in a single round-trip to the DDI. This feature distinguishes DEDIS from previous systems, like DDE and DeDe that use two distinct metadata structures, and

procedure to write *content* to a VM *logical* address:
1   ***lock*** *logical* address at the *logical-to-physical* mapping
2   **if** *logical* address is marked as CoW:
3   let *CoW block* address be the current value mapped by *logical* address
4   get *unused block* address from the *free* pool
5   write *content* to *unused block* at the storage pool
6   map *logical* address to *unused block* address and remove CoW mark
7   insert *CoW block* operation details in the *unreferenced* queue
8   **else:**
9   let *block* address be the current value mapped by *logical* address
10   write *content* to *block* at the storage pool
11   insert *logical* address in the *dirty* queue
12 ***unlock*** *logical* address at the *logical-to-physical* mapping

Fig. 3.   Pseudo-code for intercepting and processing VM writes at the *interceptor* module.

procedure to share a *logical* address at the *dirty* queue:
13 ***lock*** *logical* address at the *logical-to-physical* mapping
14   let *old block* address be the current value mapped by *logical* address
15   mark *logical* address as CoW
16   register operation in a persistent log
17 ***unlock*** *logical* address at the *logical-to-physical* mapping
18 read *content* from *old block* at the storage pool
19 compute a *signature* from *content*
20 perform a DDI *test-and-increment* operation using *signature*
21 ***lock*** *logical* address at the *logical-to-physical* mapping
22   **if** a *duplicate block* exists at the DDI **and** *logical* address still maps to *old block*:
23   update *logical* address mapping to point to *duplicate block* address
24   ***unlock*** *logical* address at logical-to-physical mapping
25   insert *old block* in the *free* pool
26   **else:**
27   ***unlock*** *logical* address at logical-to-physical mapping
28 insert operation details at the *logical-to-physical* log

Fig. 4.   Pseudo-code for share operations at the *D. Finder* module.

allows combining aliasing and reference management in a single remote invocation, thus avoiding a higher throughput penalty and reducing metadata size [Hong and Long 2004; Clements et al. 2009].

Finally, the *interceptor* processes storage calls from VM applications and from the VM operating system, so deduplication is applied to both types of dynamic data.

4.3. Concurrent Optimistic Deduplication

Figures 3 and 4 show the pseudo-code for intercepting a VM write request and for aliasing a block address at the *dirty* queue, respectively. The *interceptor* and *D. Finder* modules concurrently update and retrieve information from metadata and storage blocks. In order to avoid concurrent accesses and, consequently, data corruption, the *D. Finder* preemptively marks blocks for CoW (line 15) before reading their content from the storage pool, calculating signatures, contacting the DDI, aliasing identical blocks, and freeing the duplicate ones (lines 18 to 25). Blocks marked for CoW are immutable until they are freed by the *D. Finder* or *GC* modules.

However, this mechanism alone is not sufficient. Consider the following scenario: storage block A is being processed by *D. Finder*, it was preemptively marked for CoW, and the request to the DDI was sent to find out if a duplicate block exists at the storage (line 20). Concurrently, the *interceptor* receives a write request for the logical address

procedure to garbage collect a *copied block* address at the *unreferenced* queue:
29   read *content* from *copied block* at the storage pool
30   compute a *signature* from *content*
31   perform a DDI *test-and-decrement* operation using *signature*
32   **if** *copied block* address is distinct from the *DDI block* address:
33   insert *copied block* address in the *free* pool
34   **if** *DDI block* has zero references:
35   insert *DDI block* address in the *free* pool
36   insert operation details at the *logical-to-physical* log
37   remove *copied block* address from the *unreferenced* queue

Fig. 5.   Pseudo-code for garbage collection at the *GC* module.

pointing to storage block A (line 3), writes the content to an unused storage block B, as block A is marked as CoW, and updates the corresponding entry at the *logical-to-physical* mapping to refer to block B, which has now the latest content (lines 4 to 6). Then, after this set of events, the response from the DDI is received and a block C, with the same content as A, is found so the *D. Finder* module updates the logical address to refer to block C (lines 22 and 23). In this trace, the most recent content written in block B is lost and data is corrupted.

A straightforward solution to this issue is to lock the *logical-to-physical* mapping during the whole aliasing operation. However, such a decision includes costly remote calls in the critical section which significantly increases the contention and latency for concurrent storage requests accessing the same lock. Instead, the *D. Finder* performs fine-grained locking that excludes remote invocations to the DDI, storage reads, and other time-consuming operations from the critical section (lines 13 to 17 and 21 to 24/27). Then, the race condition detailed previously must be detected and requires aborting aliasing operations while generating dangling blocks that must be garbage collected. Namely, the second condition in line 22 ensures that the block being processed (old block) is only aliased and freed if the corresponding logical reference has not changed concurrently due to a CoW (lines 22 to 27).

Regarding read operations, the *logical-to-physical* mapping is used in a read-only fashion for redirecting requests to the corresponding storage blocks. Nevertheless, accesses to the mapping use the same lock mechanism to ensure that the latest content is read.

Figure 5 shows the pseudo-code for processing a copied block inserted in the unreferenced queue (line 7). Mutual exclusion is used to manage concurrent accesses to this queue, which is the only metadata structure shared by the *GC* and *interceptor* modules. On the other hand, the *GC* and *D. Finder* modules access, concurrently, the DDI and *free* pool structures, thus requiring mutual exclusion. As an example, consider that *D. Finder* marked a block for CoW, read its content from the storage pool, and is now calculating its signature (lines 15 to 19). Concurrently, the *interceptor* receives a write to the same block and, as it is marked for CoW, redirects the write to an unused block and inserts the copied block in the *unreferenced* queue. Then, the *GC* starts processing the queue, reads the content of the block from the storage pool, calculates a signature and performs a *test-and-decrement* operation (lines 29 to 31). However, at this time, it is possible that the *D. Finder* has not yet performed the *test-and-increment* operation for that same block. This can lead to a scenario where blocks are freed while still being in use and, consequently, to data corruption. In our design, this race condition is solved by running *D. Finder* and *GC* modules sequentially for the same VM.

As discussed previously, the *D. Finder* aborts a small number of operations due to concurrent CoWs done before updating the *logical-to-physical* mapping to reflect aliasing (line 23). These aborts generate dangling blocks that must be collected and freed with the *GC* module (lines 32 and 33). Moreover, blocks that were copied and are no longer being referenced in the DDI are also freed (lines 34 and 35).

To conclude, due to the high level of concurrency present in DEDIS, these and other issues were explored in previous work where our algorithm was validated by a model checker [Paulo and Pereira 2011].

4.4. Fault Tolerance

Writing meta-information persistently is required to ensure that logical volumes survive the crash and restart of physical nodes. Our proposal uses transactional logs for tracking changes to metadata structures and allows logical volumes, held by a crashed physical node, to be recovered by another freshly booted node. The dashed rectangles, in the previous three figures, highlight the key operations for DEDIS fault-tolerance.

Our design assumes that failures occur at the process or server level. In our current implementation, all VMs deployed on the same server have their *D. Finder* and *GC* modules running in distinct threads of a single process, so if one thread fails all the others fail too. On the other hand, if the previous process fails, the *interceptor* continues to process I/O requests independently. However, CoW operations must use free blocks directly from the *extent* service, as the unused blocks in the *free* queue are managed by a thread that was also running in the failed process (line 4).

CoW is the only operation done by the *interceptor* that modifies the *logical-to-physical* mapping. The details of each CoW operation are stored persistently and atomically in the *unreferenced* queue before acknowledging the write request as completed (line 7). When a failure occurs, the information at the queue is used to recover the mapping to a consistent state. The *dirty* queue is solely kept in-memory because it holds noncritical information that, if lost, only has the consequence of missing some share opportunities. Finally, read operations and non-CoW write operations do not require logging, as they do not modify any critical metadata structure.

A persistent log registers *D. Finder* operations immediately after marking logical addresses as CoW and before unlocking the *logical-to-physical* mapping, thus, ensuring that no concurrent mapping accesses are done before the log is written (line 16). Then, if a failure occurs, the log can be used to check what addresses were marked as CoW and need to be reprocessed. However, operations registered at the log may have failed in distinct processing stages; for instance, some operations were contacting the DDI while other operations were already processing the DDI response and aliasing duplicate blocks. To ensure that log entries are fully processed exactly once, all steps are replayed in an idempotent fashion. Namely, each *D. Finder* operation has a unique ordered timestamp that is stored persistently in the log and at the DDI when a *test-and-increment* is done (line 20). The timestamp is then used to check what operations were already processed and cannot be repeated.

The persistent *logical-to-physical* log registers modifications to the *logical-to-physical* mapping due to CoW marking and block aliasing and is appended at the end of each aliasing operation (lines 15, 23 and 28). If a duplicate is found at the DDI, the address of one of the copies (old block) is inserted in the persistent *free* pool and this log is updated while keeping exclusive access to the *free* pool (lines 25 and 28). This way, when an operation is being repeated due to a failure and a duplicate is found at the DDI, if that specific operation is already registered in the *logical-to-physical* log, it is guaranteed that the old block was already freed. On the other hand, if the operation is not registered at the log, the *free* pool must be checked for the old block address. Since the log is written while holding exclusive access to the *free* pool and the thread that

manages the pool was also running in the failed process, if the address was added to the pool then it corresponds to the last address inserted.

Reprocessed operations must also account aborts due to concurrent CoW operations done before updating the *logical-to-physical* mapping to reflect aliasing (line 23). The necessary information to identify these events is stored in the *unreferenced* queue that must be checked when recovering aliasing operations that found duplicate blocks and were not registered as completed in the previous log. At the end of each *D. Finder* iteration, all operations were registered persistently in the *logical-to-physical* log, so remaining logs can be pruned. This also means that aliasing operations only need to be reprocessed if the failure occurred during an iteration of this module.

The *GC* has the same approach to fault tolerance, as each operation has a unique timestamp for ordering CoW operations. The timestamp is calculated when CoW is performed by the *interceptor* and it is stored in the corresponding *unreferenced* queue entry (line 7). Then, the *GC* processes entries at the queue but only removes them after being completely processed and after writing to the *logical-to-physical* log (line 37). This way, if a failure occurs, all entries at the queue can be reprocessed in an idempotent fashion. Namely, *test-and-decrement* calls to the DDI are persistent and identified by the timestamp, thus, allowing to replay requests without repeating operations that were already processed. Then, addresses are inserted in the persistent *free* pool and the exclusive access to the pool is maintained until the *logical-to-physical* log is written (lines 33 to 36). The recovery process is identical to the one used for aliasing operations.

Both *GC* and *D. Finder* update the *logical-to-physical* log that can be pruned periodically into a persistent version of the *logical-to-physical* mapping to reduce recovery time. Log updates done by the *GC* reflect changes in the mapping due to CoW operations that are performed in parallel with aliasing operations. This way, the timestamps discussed previously order both CoW and aliasing operations to the same logical address, ensuring that the persistent mapping has always the latest modifications.

Regarding storage overhead, only two logging operations are performed in the critical storage path or when holding the lock of the *logical-to-physical* mapping, namely, when the *unreferenced* queue and the log that registers the beginning of aliasing operations are written (lines 7 and 16). The overhead of these operations is reduced as follows: The first log operation only occurs for copied blocks so, as detailed next, a hotspot avoidance mechanism is used for reducing the number of CoW operations. The overhead of the second log operation is reduced by grouping several blocks that will be processed by *D. Finder* and performing a single-batch log write. Although executed outside the critical path, the *logical-to-physical* log is also updated in batch to reduce the overhead of concurrent accesses to the storage pool.

Finally, in order to recover failed nodes into a distinct, freshly booted node, the logs and persistent metadata discussed previously are stored in a shared storage device. If necessary, the impact of logging in storage bandwidth can be reduced by using distinct storage backends for the logs and for the VM volumes. Then, as described in Section 3, a fault-tolerant distributed coordination and configuration service is used to locate and manage the metadata and logical volumes of crashed VMs and for booting them in a distinct cluster node. Moreover, this service is responsible for providing the *extent* server functionality and for tolerating failures of this service. DDI entries can be stored persistently in a shared storage backend or at the local disks of servers since DDI nodes are replicated, with a Replicated State Machine (RSM) approach, and can serve requests for failed replicas. Since the index is sharded across distinct DDI nodes, all the DDI nodes do not need to be fully replicated. In other words, a specific shard of the index can be replicated only across a subset of DDI nodes, while the other shards may be replicated across other groups of DDI nodes. Then, the number of DDI nodes per group may be chosen accordingly to the desired replication factor. This way, when

a new index entry is added, replication is only performed for a specific and relatively small group of nodes. This decision allows the DDI to be fault-tolerant while scaling out for large infrastructures. We show a concrete example of this sharding and replication scheme in Section 5 when we evaluate DEDIS prototype in a distributed cluster infrastructure.

To sum up, when a failure occurs, our current design allows *D. Finder* and *GC* modules to replay unfinished operations without repeating processing steps that were already completed. After completing all these unfinished steps, the *logical-to-physical* log is pruned into the persistent mapping that will correspond to the latest version of the in-memory mapping prior to the failure.

4.5. Optimizations

In the DeDe system, CoW overhead is reduced by only marking a physical address to be copied when a duplicate block is actually found at the index [Clements et al. 2009]. In a distributed infrastructure, this approach requires synchronization between the servers sharing the block and, since DEDIS does not assume a storage backend with locking capabilities, implementing such strategy is complex and requires costly cross-host communication. We avoid this cost by introducing other optimizations.

The *D. Finder* module uses a hotspot detection mechanism for identifying blocks susceptible to be rewritten in the near future or, in other words, write hotspots. By avoiding sharing such hotspots, the amount of CoW operations is reduced. In detail, 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 in the *dirty* queue that were not updated between two consecutive *D. Finder* iterations (approximately 5 minutes) and were inserted in the queue before this period are ready to be shared. This is just an example and the period can be tuned for each VM volume. Our previous work shows that, with this optimization, DEDIS performs 70% less CoW operations, which allows a significant reduction of the overhead in storage requests [Paulo and Pereira 2014a]. CoW overhead is then further reduced with an in-memory cache of unused storage blocks' addresses retrieved from the persistent *free* pool. This allows pre-fetching to memory-free addresses that will be served to CoW operations performed by the *interceptor*. This cache is independent for each *interceptor* and it is resilient to failures by registering the pre-fetched unused addresses in a persistent log. If a failure occurs, this log and the *unreferenced* queue can be compared to find what blocks are still in the cache and what blocks were used for CoW by the interceptor. The log can be pruned when entries at the *unreferenced* queue are processed with the *GC* module.

Another in-memory cache, which can be enabled or disabled in a per VM basis, is used for reducing the content that must be read back from the storage backend with the *D. Finder* module. As explained in Section 4.2, the *D. Finder* reads back the content of dirty blocks from the storage in order to calculate their content signatures. Many of these reads can be avoided if hashes are calculated and inserted into an *in-memory hash* cache when write requests are being processed with the *Interceptor*. Since hash calculation is now executed in the storage write path, it is important to evaluate its impact in storage requests, which is done in Section 5. The only hashes that need to be kept in-memory are the ones from blocks that were written but are still waiting to be processed with the *D. Finder* module, so the cache size depends on the period between share iterations. DEDIS already aims at keeping this interval small in order to maintain a reduced storage backlog. Also, the cache has a pre-defined maximum size, and a disk metadata structure is used for keeping the subset of hashes that do not fit in memory. This way, when a cache miss occurs, instead of reading the full block from disk, the *D. Finder* only reads the corresponding hash for the address being

shared, which reduces the storage I/O bandwidth needed. In our implementation, we use Berkeley DB to store on-disk hashes [Olson et al. 1999]. Finally, both the cache and on-disk structure address the concurrency issues described in Section 4.3 and do not need to be durable. If a failure occurs, the content of the blocks can always be retrieved from the storage pool.

As other optimizations, the throughput of *D. Finder* and *GC* operations is further improved by performing batch accesses to persistent logs, the DDI, the storage pool, the *extent* server, and the *free* pool. Batch requests allow for efficiently using disk and network resources and enable DDI nodes to serve requests efficiently without requiring the full index in RAM.

Finally, our current implementation uses the SHA-1 hashing function which has a negligible probability of collisions [Quinlan and Dorward 2002]. However, full byte comparison of chunks can be enabled for specific VM volumes persisting data from critical applications. Due to the DEDIS optimistic off-line deduplication approach, byte comparison is done outside the critical storage path, reducing the overhead in storage requests. However, this comparison requires reading back, from the storage, the content of the blocks to be shared.

4.6. Implementation

DEDIS prototype is implemented within Xen (version 4.1) and uses the Blktap mechanism (version 2) for building the *interceptor* module. Blktap exports a user-level disk I/O interface that replaces the commonly used loopback drivers, while providing better scalability, fault-tolerance, and performance [Citrix Systems, Inc 2014]. Each VM volume has an independent Blktap process, intercepting disk requests with a fixed block size of 4KB, which is also the block size used in DEDIS. Also, each VM volume may have a distinct Blktap driver, so deduplication can be performed only for specific volumes. For instance, it is possible to define policies where deduplication is only applied to volumes with significant space savings, while other volumes use a default Blktap driver without deduplication.

The *interceptor* module is written in C and implements a novel Blktap driver that is based on the default Xen Blktap driver for asynchronous I/O (Tap:aio). Briefly, the code that specifies how read and write requests for VM volumes are handled was changed according to the *interceptor* module algorithm. As our driver is a modification of the Tap:aio driver, it also performs asynchronous I/O, meaning that, if possible, it tries to submit several I/O operations in batch. This is important, mainly for improving the performance of sequential storage requests. In Section 6, we further discuss the impact of DEDIS in sequential storage workloads.

The goal of this implementation is to highlight the impact of deduplication and not to re-invent an LVM system or the DDI. Simplistic implementations have, thus, been used for metadata and log structures. Namely, the *logical-to-physical* mapping, *dirty* queue, and the *free* blocks queue cache are implemented as arrays fully loaded in memory that are accessible by both *interceptor* and *D. Finder* modules. Since the Blktap driver intercepts requests with a size of 4KB, the *logical-to-physical* mapping also has an entry for each 4KB block. The mapping is kept in memory to ensure that whenever the *interceptor* module accesses it, it introduces a minimum overhead in storage requests. Nevertheless, changes to this mapping, for instance due to CoW operations, are always registered persistently to ensure that if a failure occurs, the mapping is restored to a correct state.

The *in-memory hash* cache is also shared by the *interceptor* and *D. Finder* modules and it is implemented as a direct-mapped cache that allows finding the hash for a specific storage block address. This way, depending on the cache size and number of requests, a percentage of hashes from distinct blocks may end up in the same cache

slot. When this happens, one of the hashes is kept in memory and the others are written to disk. On-disk hashes are stored on Berkeley DB and retrieved when cache misses occur [Olson et al. 1999].

The *unreferenced* and *free* blocks' queues are implemented as persistent queues with atomic operations. The DDI is a modified version of the Accord high-performance coordination service, resembling the Apache Zookeeper system, but based on the Corosync group communication protocol and aimed at write-intensive workloads [Ozawa, T. and Kazutaka, M. 2014]. Accord is a replicated, transactional, and fully distributed key-value store that supports atomic test-and-increment and test-and-decrement operations; therefore, only a few lines of code had to be changed. The *extent* server is implemented as a remote service with a persistent queue of unused storage blocks. This implementation allows measuring the overhead of providing unused blocks to the *free* pools of cluster nodes.

Despite being simplistic, all these structures are usable in a real implementation, this way, the resource utilization (i.e., CPU, RAM, disk, and network) values observed in our evaluation are realistic. In fact, it is still possible to further improve storage and RAM space occupied by metadata and persistent logs if more space-efficient structures are used instead.

5. EVALUATION

DEDIS prototype was evaluated in order to validate the following assumptions. First is the assumption that deduplication does not overly impact storage performance, even when both deduplication and I/O intensive workloads run simultaneously. Then, the assumption that the storage space required for storing VM volumes is significantly reduced. Finally, the assumption that DEDIS design efficiently handles several VMs in the same server and scales out for several cluster servers.

5.1. Traces and Benchmarks

Since DEDIS targets dynamic primary data, using exclusively static traces of VM images is not suitable for its evaluation. On the other hand, traditional synthetic disk benchmarks do not accurately simulate duplicate content, either writing all blocks with the same content or with random content.

For these reasons, DEDIS was evaluated with two real dynamic traces that are publicly available[1] and were used previously to evaluate [Koller and Rangaswami 2010a] work for improving the I/O performance of storage workloads with duplicates. Although this work is distinct from DEDIS because it is not focused on eliminating duplicates from written content, it presents real dynamic traces that are useful for evaluating our system. The traces used in our evaluation were collected for a duration of three weeks and belong to two production systems with different I/O workloads used daily at the Florida International University (FIU) Computer Science department, namely a trace from a Web VM and another from a Mail server.

As a distinct tool, the open-source DEDISbench disk benchmark was also used in our evaluation [Paulo et al. 2012, 2013]. This synthetic benchmark allows simulating realistic content and realistic storage access patterns.

5.1.1. Web VM Trace. The Web VM trace belongs to a VM running two web servers, one hosting the FIU departments' online course management system and the other hosting the departments' web-based email access portal. The operations collected in the trace only correspond to I/O operations for the hosted root partitions containing the

---

[1]http://sylab-srv.cs.fiu.edu/doku.php?id=projects:iodedup:start&s[]=traces#traces.

Table I. Web VM and Mail Server Traces' Statistics for a 1-Week Period

	Storage size (GB)	Storage accessed (%)	Data written (GB)	Data read (GB)
Web VM	70	2.80	11.46	3.40
Mail server	500	6.27	482.10	62.00

OS distribution. The http data for these web servers was stored on a network-attached storage and it is not included in the trace.

Table I shows key statistics for the Web VM I/O workload for a one week period, as described in the work of Koller and Rangaswami [2010a]. This is a write-intensive workload where only a small percentage (2.80%) of the total storage space available for this system (70GB) is written/read. This means that there will be hotspot blocks that are frequently rewritten and, if shared, these blocks must be copied-on-write. Also, a static analysis of the data at the storage device supporting the root partitions of this system shows that, for an aligned fixed-block size of 4KB, ≈20GB are in use, while the remaining blocks are unused, zeroed blocks. Moreover, ≈63% of the blocks in use are duplicates, which is a significant percentage.

5.1.2. Mail Server Trace. The Mail server hosts' user inboxes for the FIU Computer Science department. As shown in Table I, this is an active system with intensive I/O when compared with the Web server. Again, this is a write-intensive workload, where a small percentage of the storage system is accessed during the 1-week period. This way, write hotspots are also expected for this trace. A static analysis of this workload with an aligned fixed-block size of 4KB shows that ≈278GB of the total available space (500GB) are in use. The percentage of duplicate blocks, excluding zeroed blocks, is similar to the Web trace, i.e., ≈63%.

Although the FIU traces are publicly available, the mechanism to replay these traces is not. We have implemented in C a replay tool. Briefly, the input files of both traces specify for each write/read operation, with an aligned block size of 4KB, the timestamp, the offset, and the content of the block as an MD5 hash. Our mechanism uses this information to replay each operation, while having in mind the timing of the operation, the duplicate content to write, and the storage offset where blocks are written/read. The latency and throughput of each operation are measured by our tool, which is also able to replay traces with a parametrized speedup.

Since we do not have access to the static storage content stored by these two systems, the content written by the replay tool only corresponds to the dynamic data that was written for the specific period of time represented in the traces. Since, in both traces, only a small percentage of the stored data is accessed, it is expected that the number of duplicates generated in the traces will also be significantly smaller than the percentage observer for the full static content.

5.1.3. DEDISbench. Synthetic benchmarks are also widely used for evaluating storage systems. However, benchmarks such as, Bonnie++, PostMark, and Fstress do not use a realistic distribution for generating duplicates and the data written in each benchmark operation either has the same content or it has random content with no duplicates at all [Coker 2015; Katcher 1997; Anderson 2002]. In both cases, the deduplication engine will process an abnormal number of duplicates, which will affect not only the storage and deduplication performance metrics, but also the values reported for the space savings and resource usage of the deduplication system [Tarasov et al. 2012]. On the other hand, benchmarks like IOzone and Filebench define a percentage of duplicate content over the written records or the entropy of generated content [Norcott 2015; Al-Rfou et al. 2010]. However, these methods are only able to generate simplistic distributions that do not specify, for example, distinct numbers of duplicates per unique data as

found in a real storage system. To our knowledge, the work of Tarasov et al. [2012] presents the only benchmark that is able to simulate complex duplicate distributions, but it is designed for evaluating incremental backup storage systems where backups are done periodically in large batches, thus, not simulating the mutation rate found, for instance, in the two real traces previously described.

DEDISbench is an open-source synthetic benchmark implemented in C that mimics the functionality of two widely used storage benchmarks, namely, IOzone and Bonnie++. Briefly, the benchmark allows performing read or write storage requests at fixed-size block granularities, with a size chosen by the user. Storage operations can be issued directly over a block storage device or over files created by the benchmark in a file system.

Additionally, DEDISbench implements novel features for evaluating primary deduplication systems under a more realistic scenario. First, the content of written blocks can follow a content distribution extracted from real datasets, thus, mimicking the percentage and distribution of duplicates found in a real storage system. In more detail, DEDISbench is able to process an input file specifying a distribution of duplicate content and to use this information for generating a synthetic workload that follows such distribution. The input file states the number of unique content blocks for a certain amount of duplicates. For instance, there are 5,000 blocks with 0 duplicates, 500 blocks with 1 duplicate, 20 blocks with 5 duplicates and 2 blocks with 30 duplicates. With the previous information, DEDISbench generates a cumulative distribution that defines the probabilities of selecting specific block identifiers. Blocks are duplicates when they share the same identifier. Identifiers with high probability of being chosen correspond to blocks with many duplicates, while identifiers with lower probabilities correspond to blocks with few duplicates. Unique blocks without any duplicate are also contemplated in the distribution and have unique identifiers. For each write operation issued, a random generator and a cumulative distribution function are used to select the correct identifier and, consequently, the content to write.

The input file describing the duplicate distribution can be generated by the users or automatically with DEDISgen, an analysis tool used for processing a real dataset and extracting from it the correspondent duplicate content distribution. In this evaluation, we use the input file generated by DEDISgen after analyzing the primary storage system for our group research projects, which store dynamic data from real test applications [Paulo et al. 2013]. This workload has ≈1.5TB and 25% of the stored blocks (with a size of 4KB) are duplicates.

As another important feature that allows simulating frequent accesses to a small portion of the dataset, DEDISbench supports an access pattern, based on the TPC-C NURand function, that simulates hotspot random disk accesses. This access pattern mimics a primary storage environment where a small percentage of blocks are hotspots with a high percentage of accesses, while most blocks are only accessed sporadically. As seen in the previous two real traces, hotspots are common and increase the number of block rewrites and, consequently, the amount of CoW operations. Since CoW is a costly operation for storage latency, it is important to test the effects of hotspots in deduplication systems [Clements et al. 2009]. In fact, in our previous work, DEDISbench was used to evaluate two open-source deduplication systems, Opendedup and Lessfs, and these novel features were key to uncover performance issues that were not detectable with the Bonnie++ and IOzone benchmarks [Paulo et al. 2012].

5.2. Traces Evaluation

To evaluate DEDIS prototype with the Web VM and Mail server traces, the experimental setup used was the following: tests ran in a server equipped with an AMD Opteron(tm) Processor 6172, 24 cores and 128GB of RAM. A single VM ran in this

Table II. DEDIS and Tap:aio Write Throughput and Latency Results for the Web VM Trace

Replay Speed		1	10	20	40	80	160	320	640
Storage	Trace	4	78	111	247	380	796	1,582	3,995
throughput	Tap:aio	4	78	111	247	380	815	1,581	4,173
(IOPS)	DEDIS	4	78	111	247	380	815	1,580	4,197
Storage	Tap:aio	0.0096	0.0095	0.0095	0.0119	0.0125	0.0083	0.0093	0.0079
latency (ms)	DEDIS	0.0099	0.0101	0.0108	0.0368	0.0439	0.0127	0.0105	0.0095

server and was configured with 4GB of RAM and two volumes, one holding the VM OS and another holding the traces data. The VM OS volume, with 20GB, was stored in a local 7200 RPMs SATA disk, while the trace volume was stored in a raw partition (500GB) created on an HP StorageWorks 4400 Enterprise Virtual Array (EVA4400), that was connected to the AMD server with Fiber Channel. This partition was used as the storage backend and stored the trace volume, configured with 20GB for the Web VM trace tests and with 290GB for the Mail server trace tests, which corresponds to the storage space in use by each of these services when the traces were collected.

As the traces already contemplate data from the OS where the services were running, the VM OS volume was not considered in the evaluation, so it was configured with a default XEN driver without deduplication. The trace replay mechanism only wrote/read data to the trace volume, and DEDIS was also only active for this volume, ensuring that deduplication was only done for the I/O operations generated by the replay mechanism.

Local DEDIS modules, a single DDI instance, and the *extent* service ran in the AMD server, while persistent metadata and logs belonging to all these components were also stored in the storage backend provided by the EVA storage system, ensuring that another server could access these logs and recover failed components, if necessary. In this single server setup, we used a single DDI instance, so we did not test the impact of replication that is discussed in the distributed tests.

The replay mechanism ran at the VM, so I/O operations were measured at the VM (DomU). 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, as well as the overhead of persistent logging.

In order to assess DEDIS overhead, we compared it with Tap:aio, the default Blktap driver for asynchronous I/O. As explained previously, this was the base driver used to implement the DEDIS *interceptor* module and it does not perform deduplication. This comparison ensures that the storage overhead observed was directly related with DEDIS. Unfortunately, a comparison with DDE or DeDe was not possible, as these systems are not publicly available.

5.2.1. Web VM Trace Results. For both Tap:aio and DEDIS, we ran 60-minute tests, the first 40 minutes with parallel I/O (with a block size of 4KB) and deduplication, and then, for the subsequent 20 minutes, deduplication was performed, isolated from the I/O workload. We chose this setup to understand the differences when performing deduplication in parallel with I/O and in isolated periods. Five minutes were used as the interval between D. Finder and GC iterations to obtain several iterations of the modules during the test and to minimize the storage backlog.

Tables II and III show the throughput and latency for write and read requests for the first 40 minutes with concurrent deduplication and I/O. As shown in the tables, when replaying the trace at a normal speed (1x), the overhead of our system, when compared with Tap:aio, is almost negligible. As the throughput for the trace is relatively low and, in the first 40 minutes, read requests are nonexistent, we repeated the tests with increased speedups. Also, we compared the throughput of DEDIS and Tap:aio with

Table III. DEDIS and Tap:aio Read Throughput and Latency Results for the Web VM Trace

Replay Speed		1	10	20	40	80	160	320	640
Storage throughput (IOPS)	Trace	-	30	64	100	178	299	439	943
	Tap:aio	-	30	46	100	178	325	439	1,032
	DEDIS	-	30	45	100	178	326	439	1,033
Storage latency (ms)	Tap:aio	-	0.76	1.11	0.93	0.41	0.18	0.08	0.03
	DEDIS	-	0.78	1.14	1.00	0.45	0.18	0.09	0.03

Table IV. DEDIS Deduplication Throughput and Space Savings Results for the Web VM Trace

Replay Speed	1	10	20	40	80	160	320	640
Deduplication throughput (MB/s)	1.73	18.39	27.31	28.04	31.10	28.62	22.71	23.23
Shared space (MB)	5	43	113	175	203	111	132	96

the optimal trace throughput in order to understand if the EVA storage backend was becoming saturated and could not handle the load at an increased throughput. We have increased the replay speed up to 640x and stopped in this value because increasing the replay speed to 1,280x would replay the full three weeks trace in ≈27 minutes and we wanted to maintain a 40-minute run. Comparing the throughput of DEDIS, Tap:aio, and the default trace throughput, it is visible that values are similar. In the 160x and 640x tests, the throughput of reads for DEDIS and Tap:aio are a bit lower than expected, which is then compensated by the write throughput, which is slightly higher.

When the replay speed increases, DEDIS overhead for both read and write latency also increases slightly. However, the latency overhead in write requests is never superior to 0.04ms, and in read requests, is never superior to 0.07ms, which are small values. Moreover, when compared with the results discussed in the DeDe system's paper, where each VMFS CoW operation requires ≈10ms to complete, and where every block must be aligned before being read/written, this is a clear improvement. For instance, in the 80x test, more than 30% of the write requests are CoW operations. Although our hotspot avoidance mechanism avoids ≈50% of these operations, if the performed CoW operations took 10ms, then the overhead would be significantly higher.

Table IV shows deduplication statistics for the 60 minutes run. Firstly, the deduplication throughput is similar for the first 40 minutes when deduplication runs in parallel with I/O and for the subsequent 20 minutes when running isolated. Note that deduplication throughput includes all operations processed by the D. Finder module and not only the operations that actually shared blocks. Also, all the operations included in this metric require marking the blocks as CoW, contacting the DDI, and processing its response. The deduplication throughput tends to increase when more write operations are done and, consequently, the *D. Finder* module has more blocks to share in a single cycle, thus leveraging the batch optimizations described previously in the article. For the test with a speedup of 80x, the deduplication throughput is ≈30MB/s, which is a clear improvement over the DeDe system where blocks are shared at ≈2.6MB/s.

Similarly, the shared space tends to increase when more blocks are written and shared. Note that the content written by the replay tool and shared by DEDIS corresponds to the dynamic data specified in the trace. Since traces only access a small percentage of the original dataset, the number of duplicates generated will be significantly smaller than the number observed in the work of Koller and Rangaswami [2010a] for the full dataset.

The table also shows a decrease in the space savings and deduplication throughput when the speed of the trace is higher than 80x. When the speed of the trace is increased, hotspot blocks are more likely to emerge. In fact, for the 80x and 160x tests, we observed

Table V. DEDIS and Tap:aio Resource Consumption Results for the Web VM Trace

Replay Speed		1	10	20	40	80	160	320	640
CPU (%)	Tap:aio	0.00	0.14	0.44	0.91	1.55	2.25	3.57	5.18
	DEDIS	0.00	0.33	0.81	1.79	2.95	3.76	5.48	8.08
RAM	Tap:aio	6.14	6.15	6.14	6.15	6.15	6.15	6.15	6.15
	DEDIS	89.40	92.88	95.24	96.03	96.55	96.95	96.47	96.59
Persistent metadata (MB)	DEDIS	116	130	139	145	150	144	143	152

Table VI. DEDIS and Tap:aio Write Throughput and Latency Results for the Mail Server Trace

Replay Speed		1	10	20	40
Storage throughput (IOPS)	Trace	126	1,661	4,097	8,862
	Tap:aio	126	1,661	4,077	8,813
	DEDIS	126	1,661	4,076	8,645
Storage latency (ms)	Tap:aio	0.0085	0.0065	0.0067	0.0070
	DEDIS	0.0086	0.0080	0.0073	0.0101

Table VII. DEDIS and Tap:aio Read Throughput and Latency Results for the Mail Server Trace

Replay Speed		1	10	20	40
Storage throughput (IOPS)	Trace	96	622	560	618
	Tap:aio	96	622	578	629
	DEDIS	96	622	590	572
Storage latency per node (ms)	Tap:aio	0.18	0.18	0.21	0.32
	DEDIS	0.22	0.18	0.25	0.86

that the number of CoW operations avoided by the DEDIS hotspot detection mechanism increased from ≈160,000 to ≈470,000. This means that many write requests are not shared, as they are hotposts that would generate CoW operations. Since the throughput of writes only doubles between these two tests, it is a noticeable decrease in the amount of shared space. The same happens for the 320x and 640x tests.

Table V shows the CPU, RAM, and persistent metadata used by DEDIS and Tap:aio. The CPU and RAM values include the resources consumed by all DEDIS components and by the DDI node that ran collocated in the AMD server. As expected, the CPU increases slightly when tests run at a higher speed. For the original trace speed (1x), the amount of CPU used by DEDIS and Tap:aio is very small, and our analysis scripts that use the Linux *top* utility reported the consumption as 0.00 due to the precision of the utility. The amount of RAM required by DEDIS is acceptable even for a server with substantially less RAM than the AMD server used. As the amount of space shared with this trace is small, the persistent metadata space is not clearly compensated by the deduplication space savings. However, in the next tests we show that for a storage workload with slightly more duplicates, the persistent metadata space is compensated.

5.2.2. Mail Server Trace Results. The Mail server trace experiments ran with an identical configuration to the previous tests, while the trace volume was configured with a size of 290GB. Tables VI and VII show for Tap:aio and DEDIS the throughput and latency for read and write requests for the first 40 minutes of the tests when I/O and deduplication ran simultaneously. When replayed at the original speed (1x) this trace does more operations per second than the Web VM trace. When replayed with a speedup of 80x, it is noticeable in both DEDIS and Tap:aio results that the storage backend is no longer keeping up with the desired trace throughput and, this way, we stopped the evaluation at the 40x speedup threshold. In fact, for the 40x test, the storage backend saturation is already slightly visible.

Table VIII. DEDIS Deduplication Throughput and Space Savings Results
for the Mail Server Trace

Replay Speed	1	10	20	40
Deduplication throughput (MB/s)	19.62	24.96	25.13	18.16
Shared space (MB)	455	1,146	1,350	1,869

Table IX. DEDIS and Tap:aio Resource Consumption Results for the Mail Server Trace

Replay Speed		1	10	20	40
CPU (%)	Tap:aio	0.41	5.27	9.57	18.18
	DEDIS	0.90	8.14	14.93	28.22
RAM	Tap:aio	6.15	6.15	6.16	6.18
	DEDIS	652.45	653.20	653.32	653.54
Persistent metadata (MB)	DEDIS	670	698	709	727

For write requests, the overhead of DEDIS is never superior to 0.003ms, which once again is a negligible overhead. For most read tests, the overhead is inferior to 0.03ms. However, in the 40x test, the overhead is ≈0.5ms. As the trace has both sequential and random I/O, this increase in overhead is probably caused by a set of sequential reads that, due to deduplication, may be reading fragmented blocks. The work described in this article is focused mainly on random workloads. Nevertheless, in Section 6, we discuss fragmentation and sequential workloads, and we point out some existing solutions that could be incorporated in the DEDIS design to handle these challenges.

Table VIII shows deduplication statistics for both the first 40 minutes with parallel I/O and the subsequent 20 minutes with isolated deduplication. Again, the deduplication throughput is similar for the period when deduplication is running both isolated and simultaneously with the I/O workload. The space savings and throughput of deduplication increase with the speed of the trace, as expected. However, for the 40x test, there is a small decrease in deduplication throughput that can be explained by the storage backend saturation that is already slightly noticed in the storage write and read requests results.

Although, the deduplication throughput is slightly lower than the one observer for the Web VM trace, the shared space is significantly higher. In fact, the shared space for a speedup of 10x and for higher speedups compensates the persistent metadata space required, shown in Table IX. Since these tests use a trace volume with 290GB, the *logical-to-physical* mapping requires more space. As this structure is stored both on disk and RAM, the extra space is visible both in the RAM and persistent metadata consumption. As explained previously, DEDIS prototype uses simple structures to implement this mapping, so it should be possible, as future work, to reduce its size with more efficient approaches. Finally, the CPU consumption increases slightly with the speed of the trace, while the overhead introduced by DEDIS is small.

5.3. Benchmark Evaluation

DEDIS was also evaluated with DEDISbench and with a very similar setup to the previous ones. DEDISbench ran in the VM and wrote data into the trace volume that was configured with a size of 20GB.

Table X shows the storage I/O and deduplication metrics for a 40 minute run of DEDISbench performing hotspot random writes (with a block size of 4KB) and for the subsequent 20 minutes, when deduplication ran isolated from the I/O workload. As DEDISbench does stress testing with a random workload, the throughput and latency are higher than the ones observed for the two traces with the original speed (1x). In this test, DEDIS overhead in storage writes throughput is ≈10%, while the latency overhead is ≈8% (0.9ms). In DeDe, only the alignment of the VM blocks with VMFs

Table X. DEDIS and Tap:aio Results for DEDISbench

	Storage throughput (IOPS)	Storage latency (ms)	Deduplication throughput (MB/s)	Shared space (MB)	CPU (%)	RAM (MB)	Persistent metadata (MB)
Tap:aio	1,307	0.75	–	–	6.81	6.28	–
DEDIS	1,174	0.84	18.77	421	6.29	95.98	150

introduces this percentage of overhead, meaning that the values from this test also show improvement over previous work.

The deduplication throughput value is similar to the one achieved for the Mail Server trace, while the shared storage space is slightly smaller but also compensates the space required by the persistent metadata. The RAM overhead is similar to the one found for the Web VM trace since the trace volume has the same size (20GB). The CPU overhead introduced by DEDIS is negligible. In fact, since the Tap:aio driver processes some more I/O operations, the CPU consumption is slightly higher than the one observed for DEDIS.

All the previous results show that DEDIS overhead is small in a setup with a single server and a single VM. Such is possible even when deduplication is being performed simultaneously with an acceptable throughput. Next, we test DEDIS in a setup with multiple VMs per server and in a setup with multiple servers.

Running the same real trace in several VMs is not realistic because it would generate exactly the same content and access pattern several times. Although it is possible to split the traces into subworkloads and run each subworkload in a distinct VM, this would reduce the trace realism and it would be a nontrivial task.

On the other hand, DEDISbench instances may run in different VMs and, as long as each instance is simulating the same content distribution, then the storage backend where VMs volumes are stored will end up with that realistic distribution. More specifically, each DEDISbench instance generates unique blocks in an unique way across all VMs. Then, since the same real distribution is being simulated in each benchmark instance, blocks with similar identifiers/content will be generated according to the distribution specification, even if these instances are running in different VMs. Additionally, the hotspot random access pattern used in DEDISbench is also different for each VM. This way, and since the previous conclusions extracted from the traces and DEDISbench results are similar, we choose to use the latter exclusively for the next tests.

Finally, we did not show results for a DEDISbench storage read test because a similar test is already presented at the end of this evaluation section.

5.4. Multiple VMs per Machine Evaluation

DEDIS was first evaluated in a setup with several VMs running on the same server. The experimental setup was very similar to the previous one. An independent DEDISbench instance ran in each VM and wrote data into an individual trace volume, with 20GB, independent from the OS volume and stored at the storage backend. A single DEDIS process ran in the AMD server, although each VM trace volume had a distinct thread for running the *D. Finder* and *GC* algorithms. As explained previously, the only shared metadata structure that required synchronization across these threads was the local *free* blocks pool. A single DDI instance and the *extent* service ran in the same AMD server, as in previous tests.

DEDIS and Tap:aio were evaluated with up to eight VMs and with a random write workload, i.e., in each VM, DEDISbench performed random hotspot writes for 40 minutes with a subsequent pause of 20 minutes when deduplication ran isolated. As depicted in Figures 6(a) and 6(b), the overhead in storage throughput and latency introduced by DEDIS is always below 15%, which is still near the overhead for a

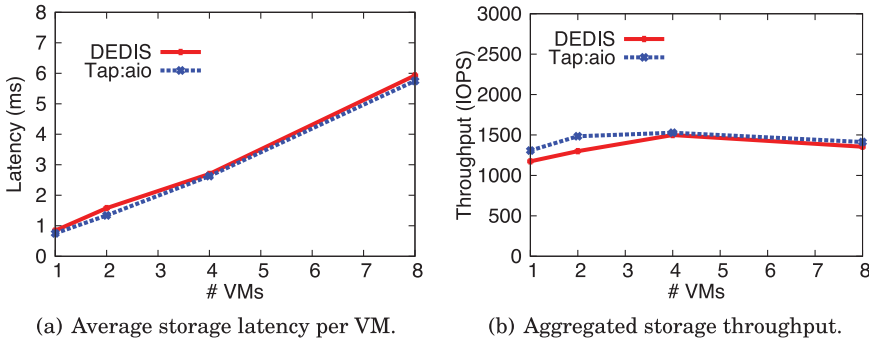(a) Average storage latency per VM.   (b) Aggregated storage throughput.

Fig. 6.   DEDIS and Tap:aio results for up to eight VMs with a random hotspot write workload.

Table XI. DEDIS Aggregated Deduplication Results for up to Eight VMs
with a Random Hotspot Write Workload

# VMs	1	2	4	8
Deduplication throughput (MB/s)	18.77	25.02	41.72	15.69
Deduplication space saved (MB)	421	633	686	320

Table XII. DEDIS and Tap:aio Resource Consumption for up to Eight VMs
with a Random Hotspot Write Workload

Number VMs		1	2	4	8
CPU (%)	Tap:aio	6.82	6.90	7.26	11.16
	DEDIS	6.29	6.75	8.87	12.44
RAM	Tap:aio	6.28	12.58	24.67	49.23
	DEDIS	95.98	181.22	348.52	669.16
Persistent metadata (MB)	DEDIS	150	202	341	413

single VM. Also, the overhead does not increase with the number of VMs per server; in fact, the maximum overhead was observed for the test with two VMs per server. As expected, when the number of VMs increases, these share the storage bandwidth and, as a consequence, the aggregated throughput remains similar while the latency of storage requests per VM is higher. However, for the test with eight VMs and for both Tap:aio and DEDIS, a small decrease of the throughput is noticeable, showing that the storage backend is becoming saturated with the concurrent load of the eight VMs.

In Table XI, this saturation is also noticeable. The deduplication throughput slightly increases while more VMs are being added and the throughput of the storage is also increasing. Then, with eight VMs per server, the deduplication throughput drops along with the storage throughput to a value closer to the deduplication throughput observed for a single VM per server. The decrease is also visible in the shared space because the VMs are writing less data and the deduplication engine is sharing it slower. As a matter of fact, in the next tests, we show that when the storage backend scales with the number of VMs, the deduplication throughput and shared space also scale.

Finally, Table XII shows that DEDIS RAM and persistent metadata increase linearly when more VMs are being served by the same server, which is explained by most metadata structures being independent for each VM. The CPU consumption remains small, even when more VMs are added.

5.5. Multiple Servers Evaluation

The setup used to evaluate DEDIS scalability for several cluster servers differs from the previous ones. Tests ran in cluster nodes equipped with a 3.1GHz Dual-Core Intel

i3 Processor, 8GB of RAM, a 7200 RPMs SATA disk, and connected by a gigabit switch. VMs were configured with 4GB of RAM and a single virtual disk volume with 20GB. A *symmetric setup* where each server ran a single VM, local DEDIS modules, and a DDI instance was used for all tests. The only component that ran in an isolated server was the *extent* service. The main advantage of using this setup was that no additional servers were required for running exclusively DEDIS or DDI components, thus resembling the setup of a traditional LVM system. Each server ran a single VM, since these servers' resources are more limited when compared to the AMD server.

DDI entries were partitioned and replicated with a replication factor of two. In each replication group, holding a distinct shard of the DDI, one of the replicas was used to process requests, while the other was kept only for fault-tolerant purposes. This way, for the setup with two servers, a single shard was used, for the setup with four servers, two shards were used, and so on. For instance, for a setup with four servers, server one and three processed requests for two distinct shards. Then, in order to cope with the failure of these two servers, server two replicated server one, and server four replicated server three. With this new setup, the overhead of replicating the DDI is contemplated in the results.

The distributed storage backend was also provided by the local disks of the cluster servers. More specifically, each server exported to the other servers an iSCSI device with 45GB. VM volumes were then stored in these devices with block-level striping. This way, the number of iSCSI devices grew with the number of servers, i.e., for a setup with two servers there were two iSCSI disks, for a setup with four servers there were four iSCSI devices, and so on. This design allowed scaling the storage pool with the number of VMs while spreading the volumes across distinct iSCSI devices. Persistent metadata and logs belonging to DEDIS and the *extent* server were also stored in the distributed storage pool, ensuring that all servers could access these logs and recover failed components, if necessary. DDI persistent data was kept in the local disk of each server, since replication was used to ensure fault-tolerance. Due to the scope of the evaluation, our storage pool implementation only performed striping without maintaining any redundancy for tolerating disk failures. In a production environment the storage backend should have replication enabled in order to tolerate the failure of storage servers for both the Tap:aio and DEDIS environments. However, this is just a test setup and our main motivation is to show that DEDIS introduces low overhead in a distributed storage system that scales with the number of servers.

In order to simulate a realistic environment where both static and dynamic VM data are deduplicated, we chose to store both OS and DEDISbench data in the same VM volume. Also, in a realistic environment, as the one discussed in LiveDFS, when a VM volume is created with a specific OS and other static information, it is more common to perform in-line deduplication while loading the volume into the storage backend than only performing deduplication later after storing the volume with duplicate content [Ng et al. 2011]. This way, we implemented a similar mechanism for loading new VM volumes into the storage. Briefly, upon loading, VM volumes are divided into 4KB blocks that are examined and actually stored only if they have useful content, thus excluding zeroed blocks that are then lazily allocated when needed. Moreover, deduplication is performed for each block being loaded to the storage. Duplicates are found inside the same VM volume and across other VM volumes already at the storage, with an algorithm that is very similar to the one used by DEDIS. The metadata and DDI structures used, while loading the VMs, are the same that are used by DEDIS when the VMs are deployed and running. This way, if a VM volume is being loaded while other VMs are already running at the cluster, the loading mechanism will also contemplate duplicate blocks from running VMs. The only difference from DEDIS algorithm is that

Table XIII. DEDIS Optimizations Results for Two Cluster Nodes with a Random Hotspot Write Workload

	DEDIS wo/cache	DEDIS w/cache	DEDIS byte
Aggregated storage throughput (IOPS)	1,710	1,854	1,807
Average storage latency per node (ms)	1.08	1.00	1.05
Aggregated deduplication throughput (MB/s)	2.52	31.46	3.92

in-line deduplication is used instead, meaning that duplicates are eliminated before being stored.

When the VM instance starts running, the associated volume will have the necessary boot information already at the storage backend. Since zeroed blocks are not allocated immediately, these are lazily allocated when needed by the *interceptor* module. Lazy-allocation is resilient to failures in a similar fashion to CoW. When the *interceptor* receives a write request for a block that must be lazily allocated, it gets an unused address from the *free* pool, updates the *logical-to-physical* mapping, and registers the operation at the persistent *unreferenced* queue. Then, the *GC* is responsible for processing the queue and persisting the mapping modifications in the *logical-to-physical* log. Deduplication uses the same logs as *D. Finder* to ensure that modifications to the *logical-to-physical* mapping and to other persistent structures are resilient to failures.

To conclude, this mechanism allows evaluating DEDIS in a more realistic scenario where VMs' volumes are launched by a mechanism that shares static information. Then, the dynamic information written by DEDISbench is shared with DEDIS. Also, the OS information rewritten while DEDISbench is running, is copied-on-write, if necessary, and shared by DEDIS. The operations done by our launching mechanism use the same metadata structures as DEDIS, require no additional storage space, and have no additional impact while the experiment is running.

5.5.1. DEDIS Optimizations. We started by evaluating some of DEDIS optimizations in a setup with two cluster nodes. In order to evaluate the *in-memory hash* cache described in Section 4.5, we compared two versions of DEDIS, one using this mechanism (DEDIS w/cache) and another without it (DEDIS w/o cache). Also, to understand the overhead of doing byte comparison of blocks before sharing them, we have evaluated another version of DEDIS, similar to DEDIS w/cache, but using byte comparison optimization (DEDIS byte).

Tests ran in a setup with two servers, each with a single VM. Two servers were used to ensure that there were at least two DDI nodes and that replication costs would be properly assessed in the results. Table XIII shows the storage I/O and deduplication metrics for a 40 minute run of DEDISbench performing hotspot random writes, and for the subsequent 20 minutes when deduplication ran isolated from the I/O workload.

The results show that the storage I/O latency is reduced and the aggregated throughput increased when the *in-memory hash* cache is used. Improvements are even more noticeable for the aggregated deduplication throughput, increasing from 2.52MB/s to 31.46MB/s.[2] This gain is achievable because block digests are precalculated and looked up in the in-memory cache. Even when cache misses occur and the hashes must be fetched from the on-disk Berkeley DB, the storage bandwidth used is significantly smaller than the one that would be used for reading back the full content of 4KB blocks from the storage. In terms of metadata space, the on-disk Berkeley DB required 15MB of disk space per server. The in-memory cache was configured to use 16MB of RAM per server that, for these tests, allowed obtaining a cache hit ratio of 69%. It is also important to refer that, for both DEDIS versions, the deduplication throughput value

---

[2]In the DEDIS w/cache test, the *D. Finder* module processed more than one million operations.

(a) Average storage latency per node.
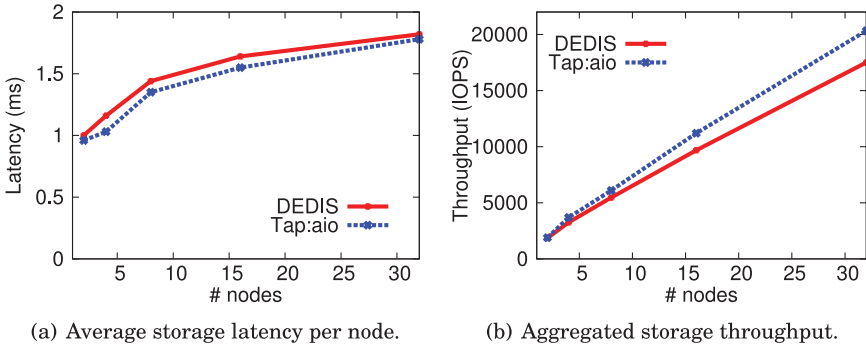
(b) Aggregated storage throughput.

Fig. 7. DEDIS and Tap:aio results for up to 32 cluster nodes with a random hotspot write workload.

was similar when deduplication was running in parallel with the I/O benchmark and when it was running isolated.

Table XIII also shows the overhead of performing byte comparison over hash comparison. The overhead is small for storage latency and throughput but it is more noticeable in deduplication throughput. Byte comparison requires reading back the content of the two blocks being shared from the storage which reduces the benefits of the hash cache. This cost presents a tradeoff between performance and resilience to collisions of the SHA-1 algorithm that must be considered for the VMs where deduplication is being applied. In most cases, the negligible probability of collision of the SHA-1 algorithm is acceptable and the hash comparison is preferred [Quinlan and Dorward 2002; Paulo and Pereira 2014b].

The impact of other optimizations, such as the hotspot avoidance mechanism, were already discussed in previous work, so we do not address them here in detail [Paulo and Pereira 2014a]. Nevertheless, these optimizations were used in these tests and had a positive impact in the results. For instance, in the DEDIS w/cache test, the hotspot mechanism avoided approximately 80% of CoW operations ($\approx$300,000 operations per server). Finally, in terms of CPU, RAM, and network bandwidth, the three DEDIS versions had similar consumptions. We further discuss resource consumption and deduplication space savings in the next section where we evaluate DEDIS prototype in a larger cluster.

5.5.2. DEDIS Scalability and Performance. The prototype, with hash comparison and all the optimizations, was then evaluated in a setup with up to 32 servers, where each server ran a single VM, a DDI instance, and local DEDIS components. The *extent* service was the only component that ran in an independent server. DEDIS and Tap:aio were evaluated with a random write workload, i.e., DEDISbench performed random hotspot writes for 40 minutes with a subsequent pause of 20 minutes when deduplication ran isolated.

Figures 7(a) and 7(b) show the average storage latency and aggregated throughput for both Tap:aio and DEDIS, running with 2, 4, 8, 16, and 32 cluster nodes. Storage latency slightly increases when more VMs are serving I/O requests, with both Tap:aio and DEDIS. It occurs because in a symmetrical system, where all volumes are evenly stripped across all servers, an increasing share of requests are routed to remote nodes, thus incurring network overhead. Note that with 32 servers, only a small share of load is handled locally.

When compared with Tap:aio results, the latency overhead introduced by DEDIS is at most 11%, regardless of the number of servers. Similarly, the throughput overhead is at most 14%, the maximum value observed in experiments, with 4 and 32 servers. These results show that DEDIS introduces low overhead in a worst case scenario when

(a) Aggregated deduplication throughput.
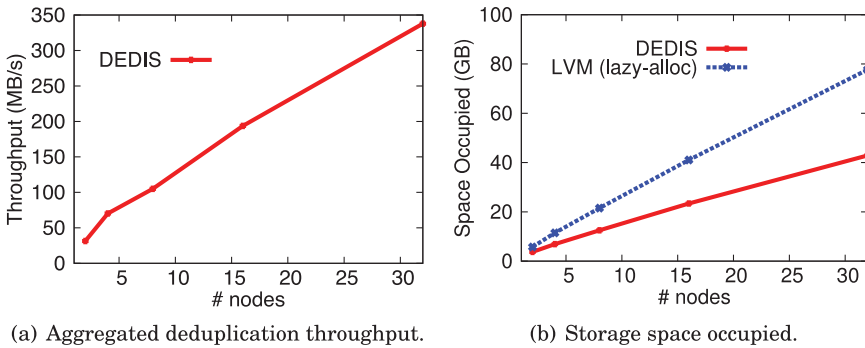

(b) Storage space occupied.

Fig. 8. Deduplication results for up to 32 cluster nodes with a random hotspot write workload.

Table XIV. Percentage of Deduplication Operations that Eliminated
Duplicates for up to 32 Servers

# Servers	2	4	8	16	32
% Benchmark operations shared	16.9	17.0	17.2	17.3	17.5
% All operations shared	33.4	38.6	45.7	51.6	57.0

deduplication and intensive I/O are running concurrently. Also, DEDIS overhead, as a percentage, is not directly affected by the number of cluster nodes, meaning that our design scales along with the storage backend.

Figure 8(a) shows the aggregated deduplication throughput for the same tests which scale close to linearly with a growing number of servers. Again, in settings with a very low number of servers, the share of accesses to the local disk that avoid network overhead is still significant and provides a slight advantage. Also, the throughput is evenly distributed across the distinct cluster nodes, showing that storage bandwidth is fairly distributed and that no node is starved. The key component in our design, which allows deduplication to scale out, is the decentralized DDI service that can be partitioned across distinct nodes. As our results show, these shards can be collocated with VMs and other DEDIS components without having a significant impact in the overall performance. For the 32 servers run, the average deduplication throughput per server was approximately 10MB/s, which is still an improvement over the DeDe system where blocks are shared at ≈2.6MB/s [Clements et al. 2009].

When more VMs are writing data into the storage, there is a higher probability of finding more duplicates across their volumes. This is shown in Table XIV where the percentage of deduplication operations that found and eliminated duplicates is detailed. The table comprises the percentage of duplicates found solely for processed DEDISbench operations and for all operations, including both the benchmark and the VM-loading mechanism. Both values show that when more servers are added, the percentage of duplicates increases, which is only possible because DEDIS does exact cluster-wide deduplication. The percentages of duplicates found for all operations are higher because, in our tests, we have used the same VM image for all servers, so preallocated blocks of VM images were fully deduplicated by our loader. Also, there was some redundancy inside the same VM image that was eliminated before being loaded to the storage pool. Note that unused zeroed blocks from VM images were not deduplicated and were later lazily-allocated, so we have not included these blocks in these results. Using the same image for all VMs allows evaluating all cluster nodes in the same condition and ensures that extracted I/O metrics are not affected by using distinct configurations. Although this approach does not simulate a cluster with distinct

Table XV. Average Resource Consumption, Per Node, for the Hotspot Random
Write Test with 32 Cluster Nodes

	CPU (%)	RAM (MB)	Network (KB/s)	Persistent metadata (MB)
Tap:aio	3.90	6.25	–	–
DEDIS	6.44	197.25	247.89	96.10

VM images, it is common for many cloud providers to have a set of standard images that are widely used for launching new VMs. Therefore, the amount of fully duplicate images is significant in real-world deployments.

For the experiment with 32 VMs, the loading mechanism deduplicated approximately 31GB while DEDISbench was running, and in the subsequent 20 minutes, approximately 5.9GB of dynamic content was shared, which corresponds to 17.5% of the total number of blocks processed by the *D. Finder* module that processed approximately 8.7 million requests. Note that the DEDISbench hotspot workload simulates a high percentage of rewrite operations, which is important for generating more CoW operations and assessing their overhead, but also reduces the duplicates processed by DEDIS due to the following reasons: First, the hotspot avoidance mechanism avoids many share operations for blocks frequently rewritten, and that would probably be copied-on-write after being shared. Also, even without the avoidance mechanism, a block may be rewritten several times between two share iterations, but it will only be shared once when the *D. Finder* asynchronously collects it. Nevertheless, the percentage of processed blocks that were actually shared is near the duplicate content simulated with DEDISbench, which is 25%. Regarding the other tests, the results and conclusions are similar to the ones described for the 32 servers experiment.

Figure 8(b) shows the storage space required after loading the VMs images into the storage and running the I/O benchmark in each VM. This figure compares DEDIS with an LVM system without deduplication, but that supports lazy-allocation of unused blocks. DEDIS used approximately 50% of the space that the LVM system would require. Moreover, a storage system without lazy allocation would require 640GB for storing the 32 VM volumes instead of the 43GB used by DEDIS. These values do not include, however, the storage space needed for persistent metadata and logs, which we detail next and show that it is clearly compensated by the space savings.

Table XV shows the average resource consumption per server for the 32 servers experiment. We chose this specific run but, once again, the other tests have similar results and conclusions. The CPU, RAM, and network values include the resources consumed by all DEDIS components and by the DDI nodes that ran collocated in the cluster nodes. The persistent metadata values also include the VM loading mechanism that shared persistent structures with DEDIS and used the DDI to deduplicate VM images.

In terms of CPU usage, DEDIS introduced a small percentage of overhead when compared to Tap:aio. As expected, DEDIS required more RAM for its in-memory caches and for performing deduplication. Nevertheless, DEDIS used less than 3% of the total RAM of each cluster node. The network usage for the 32 servers, more specifically, the network bandwidth used for contacting the DDI nodes and for supporting their replication was less than 250KB/s. Regarding metadata consumption, the DDI, DEDIS, and the loading mechanism required 96.10MB of storage space per server. Globally, for the 32 servers, it used ≈3GB of storage space that were clearly compensated by the 37GB of space deduplicated by DEDIS and the load mechanism.

Finally, it is important to refer that, in our tests, resource consumptions were evenly distributed across the servers. Also, in the 32 servers experiment, the *extent* service used less than 1% of CPU, 2% of the server RAM, and 2GB of persistent metadata for indexing ≈1TB of storage blocks.

Table XVI. DEDIS and Tap:aio Results for up to 32 Cluster Nodes with a Random
Hotspot Read Workload

	Aggregated storage throughput (IOPS)	Average latency per node (ms)
Tap:aio	8,238	0.24
DEDIS	8,698	0.23

5.6. DEDIS Read Performance

A setting with two servers was also used for assessing DEDIS overhead in random hotspot read workloads. In these tests, DEDISbench wrote data in the first 30 minutes, then stopped for 30 minutes, and finally ran again for another 40 minutes performing random hotspot reads. The first 60 minutes were used to populate the storage and have DEDIS sharing duplicate blocks. The last 40 minutes were used to run the benchmark and check storage read performance in a deduplicated storage. In the test with the Tap:aio driver, the storage was also populated, but without any sort of deduplication.

Surprisingly, Table XVI shows that DEDIS outperforms Tap:aio in both storage latency and throughput. This probably happens because, in a deduplicated storage, some of the reads for distinct addresses will end up reading the same shared block from the storage. This allows using OS read caches more efficiently while reducing the disk arm movement [Koller and Rangaswami 2010b]. It is important to refer that the *interceptor* module and the Tap:aio driver use the O_DIRECT flag for reading/writing content to the storage backend, thus avoiding the OS caches of the Dom0 server. On the other hand, the VM cache is enabled for both DEDIS and Tap:aio in order to have a fair comparison. Finally, these results refer to a random workload that does not suffer from storage fragmentation, which we discuss further in the next section.

To conclude, these results prove that our design has negligible impact in random storage read requests and that, in some cases, deduplication can even increase their performance.

The results presented in this section allow us to conclude that DEDIS introduces low storage overhead, even when both deduplication and intensive storage random I/O are performed concurrently. This is true for several storage workloads, for setups with multiple VMs per server, and for setups with multiple servers. Also, the evaluation results presented in this section required ≈26 hours of computation, only for DEDIS tests. In this period, more than 734GB (≈192 million blocks) were written into the storage and more than 86GB (≈20 million blocks) were deduplicated. Tap:aio tests required ≈23 hours and wrote more than 780GB (≈204 million blocks) into the storage. These tests generated ≈50MB of logs that were then analyzed to extract the results presented in this article.

6. DISCUSSION

Deduplication research is traditionally focused on sequential storage workloads [Quinlan and Dorward 2002; Ungureanu et al. 2010]. On the other hand, random I/O workloads, common in primary storage systems, are less researched and still have outstanding challenges that prevent efficient deduplication with acceptable storage overhead [Hong and Long 2004; Clements et al. 2009]. This is the main reason why the work discussed in this article is focused on random storage workloads.

Obtaining low overhead for both types of workloads with the same system is yet another problem. As an example, HydraFS achieves good results for stream I/O while supporting random I/O but with moderate performance [Ungureanu et al. 2010]. DEDIS was built with some characteristics in mind that should allow it to perform reasonably for sequential I/O. Our *interceptor* module is implemented using asynchronous I/O and many of DEDIS logging operations are done in batch to reduce the overhead

of processing one request at a time. This way, if sequential writes are issued and these do not rewrite any previously shared data, it is expectable that DEDIS performance will be comparable with Tap:aio. However, if sequential requests generate several CoW operations, DEDIS overhead will be more noticeable.

A possible future contribution and interesting research direction would be to build a hybrid system that bundles these distinct researches and achieves low storage overhead for both types of I/O access patterns. Since DEDIS solution can be used in a per volume basis, it would be easy to use this driver for volumes with random workloads and use a specific driver optimized for sequential throughput for other volumes. However, even more interesting would be to have a hybrid deduplication system that can change its algorithm for subworkloads issued in the same volume.

Similarly, DEDIS design does not address the fragmentation introduced by deduplication, which has negligible impact in random storage reads but significantly impacts sequential storage reads. This topic also has several research work that could be applied to DEDIS in the future. Deduplication systems optimized for sequential I/O usually use chunks with larger sizes or group chunks into segments to improve the throughput of stream I/O operations and to reduce fragmentation [Zhu et al. 2008; Lillibridge et al. 2009]. Other systems reduce fragmentation by doing selective deduplication only over some chunks or by rewriting some chunks in order to maintain the sequential storage layout for the groups of blocks that will suffer most from fragmentation [Kaczmarczyk et al. 2012; Srinivasan et al. 2012]. The latter ideas could be incorporated in the DEDIS off-line deduplication algorithm as future work. As another idea, it would be interesting to understand if storage replication could be used to provide both fault-tolerance and, when possible, to asynchronously replicate some blocks and place them in specific storage locations that would ensure their sequential layout and reduce the fragmentation effects. Once again, the hybrid deduplication system could include these optimizations for improving the efficiency of sequential read requests.

7. CONCLUSIONS

We present the design, implementation, and evaluation of DEDIS, a dependable and distributed system that performs offline deduplication across VMs' primary storage volumes in a cluster-wide fashion. DEDIS design is fully decentralized, avoiding any single point of failure or contention, thus, safely scaling-out. Our design is compatible with any storage backend, distributed or centralized, as long as it exports a shared block device interface. Moreover, we present novel optimizations for improving deduplication performance and reliability while reducing the impact in storage requests.

The evaluation of DEDIS Xen-based prototype with both real traces and benchmarking tools shows that DEDIS has a small impact in storage requests while providing an acceptable deduplication throughput, even when several VMs are hosted in the same cluster server. Moreover, our evaluation in up to 32 cluster nodes shows that deduplication and primary I/O random workloads can run simultaneously in a fully decentralized and scalable system while keeping low latency and throughput overhead, less than 14%, and a baseline single-server deduplication throughput of approximately 10MB/s with low-end hardware. Such is not possible in previous primary deduplication proposals and is fundamental for performing efficient deduplication and reducing the duplicate storage backlog in infrastructures with scarce off-peak periods [Clements et al. 2009; Hong and Long 2004]. Also, the resulting net space savings are clearly worthwhile in face of an acceptable consumption of CPU, RAM, and network resources. These results allow us to conclude that efficient distributed deduplication is achievable in primary storage cloud infrastructures.

Finally, DEDIS source-code is open-source and is publicly available for anyone to deploy and benchmark.

8. AVAILABILITY

DEDIS source code and additional information can be consulted at http://www.holeycow.org/Home/deduplication.

ACKNOWLEDGMENT

REFERENCES

Rami Al-Rfou, Nikhil Patwardhan, and Phanindra Bhagavatula. 2010. *Deduplication and Compression Benchmarking in Filebench*. Technical Report.

Darrell Anderson. 2002. *Fstress: A Flexible Network File Service Benchmark*. Technical Report. Duke University.

Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. 2009. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. 2000. Single instance storage in Windows 2000. In *Proceedings of USENIX Windows System Symposium (WSS)*.

Citrix Systems, Inc. 2014. Blktap documentation. Retrieved from http://wiki.xen.org/wiki/Blktap2.

Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. 2009. Decentralized deduplication in SAN cluster file systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*.

Russell Coker. 2015. Bonnie++ web page. Retrieved from http://www.coker.com.au/bonnie++/.

D. Iacono. 2013. Enterprise storage: Efficient,virtualized and flash optimized. *IDC White Paper*.

Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. Chunk stash: Speeding up inline storage deduplication using flash memory. In *Proceedings of USENIX Annual Technical Conference (ATC)*.

Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. 2011. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.

John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. 2002. *Reclaiming Space from Duplicate Files in a Serverless Distributed File System*. Technical Report MSR-TR-2002-30. Microsoft Research.

Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. 2012. Primary data deduplication large scale study and system design. In *Proceedings of USENIX Annual Technical Conference (ATC)*.

EMC. 2012. New Digital Universe Study Reveals Big Data Gap. http://www.emc.com/about/news/press/2012/20121211-01.htm. (2012).

Davide Frey, Anne-Marie Kermarrec, and Konstantinos Kloudas. 2012. Probabilistic deduplication for cluster-based storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*.

Yinjin Fu, Hong Jiang, and Nong Xiao. 2012. A scalable inline cluster deduplication framework for big data protection. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*.

Fanglu Guo and Petros Efstathopoulos. 2011. Building a high-performance deduplication system. In *Proceedings of USENIX Annual Technical Conference (ATC)*.

HP. 2011. Complete storage and data protection architecture for VMware vSphere. *White Paper* (2011).

Bo Hong and Darrell D. E. Long. 2004. Duplicate data elimination in a san file system. In *Proceedings of Conference on Mass Storage Systems (MSST)*.

Keren Jin and Ethan L. Miller. 2009. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of International Systems and Storage Conference (SYSTOR)*.

Jones, M. 2010. Virtio: An I/O virtualization framework for linux. *IBM White Paper* (2010).

Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. 2012. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of International Systems and Storage Conference (SYSTOR)*.

Jürgen Kaiser, Dirk Meister, André Brinkmann, and Sascha Effert. 2012. Design of an exact data deduplication cluster. In *Proceedings of Conference on Mass Storage Systems (MSST)*.

Jeffrey Katcher. 1997. *PostMark: A New File System Benchmark*. Technical Report. NetApp.

Ricardo Koller and Raju Rangaswami. 2010a. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage* 6, 3 (Sept. 2010), 13:1–13:26.

Ricardo Koller and Raju Rangaswami. 2010b. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.

Lessfs. 2014. Lessfs page. Retrieved from http://www.lessfs.com/wordpress/.

Yan-Kit Li, Min Xu, Chun-Ho Ng, and Patrick P. C. Lee. 2014. Efficient hybrid inline and out-of-line deduplication for backup storage. *Trans. Storage* 11, 1 (2014), 2:1–2:21.

Anthony Liguori and Eric Van Hensbergen. 2008. Experiences with content addressable storage and virtual disks. In *Proceedings of USENIX Workshop on I/O Virtualization (WIOV)*.

Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. 2009. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.

D. Meister and A. Brinkmann. 2010. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Proceedings of Conference on Mass Storage Systems (MSST)*.

Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. 2008. Parallax: Virtual disks for virtual machines. In *Proceedings of European Conference on Computer Systems (EuroSys)*.

Dutch T. Meyer and William J. Bolosky. 2011. A study of practical deduplication. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.

Dutch T. Meyer and William J. Bolosky. 2012. A study of practical deduplication. *ACM Transactions on Storage* 7, 4 (2012), 14:1–14:20.

Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P. C. Lee, and John C. S. Lui. 2011. Live deduplication storage of virtual machine images in an open-source cloud. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*.

William Norcott. 2015. IOzone web page. Retrieved from http://www.iozone.org/.

Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of USENIX Annual Technical Conference (ATC)*.

Opendedup. 2014. Opendedup web page. Retrieved from http://opendedup.org.

OpenSolaris. 2014. ZFS documentation. Retrieved from http://www.freebsd.org/doc/en/books/handbook/filesystems-zfs.html.

OpenStack Foundation. 2014. OpenStack web page. Retrieved from https://www.openstack.org.

OpenStack Foundation. 2016. Cinder documentation. Retrieved from http://docs.openstack.org/developer/cinder/.

T. Ozawa and M. Kazutaka. 2014. ACCORD web page. Retrieved from http://www.osrg.net/accord/.

Joao Paulo and Jose Pereira. 2011. Model checking a decentralized storage deduplication protocol. In *Fast Abstract in Latin-American Symposium on Dependable Computing*.

J. Paulo and J. Pereira. 2014a. Distributed exact deduplication for primary storage infrastructures. In *Distributed Applications and Interoperable Systems*.

João Paulo and José Pereira. 2014b. A survey and classification of storage deduplication systems. *Comput. Surveys* 47, 1 (2014), 11:1–11:30.

J. Paulo, P. Reis, J. Pereira, and A. Sousa. 2012. DEDISbench: A benchmark for deduplicated storage systems. In *Proceedings of International Symposium on Secure Virtual Infrastructures (DOA-SVI)*.

J. Paulo, P. Reis, J. Pereira, and A. Sousa. 2013. Towards an accurate evaluation of deduplicated storage systems. *International Journal of Computer Systems Science and Engineering* 29, 1, 1:73–1:83.

Sean Quinlan and Sean Dorward. 2002. Venti: A new approach to archival storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.

Sean Rhea, Russ Cox, and Alex Pesterev. 2008. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of USENIX Annual Technical Conference (ATC)*.

Rusty Russell. 2008. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. 2012. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. 2012. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.

Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. 2012. Generating realistic datasets for deduplication analysis. In *Poster Session of USENIX Annual Technical Conference (ATC)*.

Y. Tsuchiya and T. Watanabe. 2011. DBLK: Deduplication for primary block storage. In *Proceedings of Conference on Mass Storage Systems (MSST)*.

Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. 2010. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.

Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. 2010. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proceedings of Conference on Mass Storage Systems (MSST)*.

Avani Wildani, Ethan L. Miller, and Ohad Rodeh. 2013. HANDS: A heuristically arranged non-backup in-line deduplication system. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2011. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of USENIX Annual Technical Conference (ATC)*.

Tianming Yang, Hong Jiang, Dan Feng, Zhongying Niu, Ke Zhou, and Yaping Wan. 2010. DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. In *Proceedings of International Parallel & Distributed Processing Symposium (IPDPS)*.

Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. 2005. Deep store: An archival storage system architecture. In *Proceedings of International Conference on Data Engineering (ICDE)*.

Benjamin Zhu, Kai Li, and Hugo Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.