O trabalho apresentado nesta dissertação foi suportado pela Fundação para a Ciência e Tecnologia (FCT) através de uma Bolsa de Doutoramento com referência SFRH/BD/71372/2010

João Tiago Medeiros Paulo **Dependable Decentralized Storage Management for Cloud Computing** UMinho | 2015

Universidade do Minho
Escola de Engenharia

João Tiago Medeiros Paulo

Dependable Decentralized Storage Management for Cloud Computing

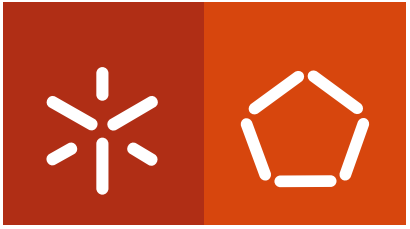
The MAP-i Doctoral Program of the Universities of Minho, Aveiro and Porto

universidade de aveiro

Universidade do Minho

U.PORTO

May 2015



Universidade do Minho
Escola de Engenharia

João Tiago Medeiros Paulo

Dependable Decentralized Storage Management for Cloud Computing

The MAP-i Doctoral Program of the Universities of Minho, Aveiro and Porto

Universidade do Minho

supervisor:
Prof. José Orlando Pereira

May 2015

DECLARAÇÃO

Nome: João Tiago Medeiros Paulo

Endereço electrónico: jtpaulo@gmail.com Telefone: 939414342

Número do Bilhete de Identidade: 13038855

Título Tese: Dependable Decentralized Storage Management for Cloud Computing

Orientador: Prof. José Orlando Pereira

Ano de conclusão: 2015

Designação do Doutoramento: The MAP-I Doctoral Program Of The Universities of Minho, Aveiro and Porto

Universidade do Minho, 15/05/2015

Assinatura: 

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, 18 de Maio de 2015

Full name: João Tiago Medeiros Paulo

Signature: João Tiago Medeiros Paulo

Agradecimentos

Esta longa jornada não teria sido possível sem o apoio de diversas pessoas com as quais tive uma imensa sorte em poder trabalhar e conviver. A todos um grande obrigado porque sem a vossa ajuda o trabalho presente neste documento não seria possível.

Em primeiro lugar quero agradecer ao meu orientador, Prof. José Pereira, por estar sempre disponível para discutir as dúvidas e novas ideias que foram surgindo ao longo destes quatro anos e pela excelente orientação sem a qual não teria sido possível este documento. Foi um enorme prazer poder trabalhar consigo quer na tese quer nos outros projetos em comum.

Em segundo lugar, estou extremamente grato por fazer parte do grupo de Sistemas Distribuídos; teria sido muito difícil encontrar um outro grupo com um ambiente semelhante e com uma vontade tão grande de evoluir e ser melhor a cada dia. Por isso, gostava de deixar o meu agradecimento a todos os docentes e, em especial, ao Prof. Rui Oliveira, que também se mostrou sempre disponível para me ajudar no meu trabalho quando necessário.

Um obrigado muito especial também para todos os meus colegas, e ex-colegas, do laboratório. Para além da ajuda de muitos diretamente ligada ao meu doutoramento, não consigo pensar em melhor ajuda do que poder ir todos os dias trabalhar para um laboratório com um ambiente tão agradável e único. Por isso, um grande obrigado, Ana Nunes, Fábio Coelho, Filipe Campos, Francisco Cruz, Francisco Maia, Miguel Borges, Miguel Matos, Nelson Gonçalves, Nuno Carvalho, Nuno Castro, Paulo Jesus, Pedro Gomes, Pedro Reis, Ricardo Gonçalves, Ricardo Vilaça e Tiago Jorge. Não me podia esquecer também de agradecer ao Jácome Cunha e ao resto do pessoal do grupo OsSemEstatuto pelos momentos de descontração.

Para além destes últimos quatro anos, todos os anos anteriores não teriam

sido possíveis sem toda a ajuda e suporte dos meus pais e irmão. Não me posso esquecer também dos meus avós e da minha tia. Muito obrigado por poder contar sempre convosco.

Finalmente, quero deixar um agradecimento especial à pessoa que me tem aturado nestes últimos tempos e tem sido mais do que importante em vários aspetos da minha vida. Paula, obrigado e espero poder continuar a partilhar contigo este e outros momentos.

Adicionalmente, agradeço também às instituições que apoiaram o trabalho apresentado nesta tese: à Fundação para a Ciência e Tecnologia (FCT), que apoiou este trabalho através da bolsa de doutoramento (SFRH/BD/71372/2010), e ao Departamento de Informática da Universidade do Minho e ao HASLab - High Assurance Software Lab, que ofereceram-me as condições necessárias para realizar a tese.

Braga, Maio de 2015
João Paulo

Dependable Decentralized Storage Management for Cloud Computing

The volume of worldwide digital information is growing and will continue to grow at an impressive rate. Storage deduplication is accepted as valuable technique for handling such data explosion. Namely, by eliminating unnecessary duplicate content from storage systems, both hardware and storage management costs can be improved. Nowadays, this technique is applied to distinct storage types and, it is increasingly desired in cloud computing infrastructures, where a significant portion of worldwide data is stored. However, designing a deduplication system for cloud infrastructures is a complex task, as duplicates must be found and eliminated across a distributed cluster that supports virtual machines and applications with strict storage performance requirements.

The core of this dissertation addresses precisely the challenges of cloud infrastructures deduplication. We start by surveying and comparing the existing deduplication systems and the distinct storage environments targeted by them. This discussion is missing in the literature and it is important for understanding the novel issues that must be addressed by cloud deduplication systems. Then, as our main contribution, we introduce our own deduplication system that eliminates duplicates across virtual machine volumes in a distributed cloud infrastructure. Redundant content is found and removed in a cluster-wide fashion while having a negligible impact in the performance of applications using the deduplicated volumes. Our prototype is evaluated in a real distributed setting with a benchmark suited for deduplication systems, which is also a contribution of this dissertation.

Gestão Confiável e Distribuída do Armazenamento para Computação em Nuvem

O volume de informação digital mundial está a crescer a uma taxa impressionante. A deduplicação de sistemas de armazenamento é aceite como uma técnica valiosa para gerir esta explosão de dados, dado que ao eliminar o conteúdo duplicado é possível reduzir ambos os custos físicos e de gestão destes sistemas. Atualmente, esta técnica é aplicada a diversos tipos de armazenamento e é cada vez mais desejada em infraestruturas de computação em nuvem, onde é guardada uma parte considerável dos dados gerados mundialmente. Porém, conceber um sistema de deduplicação para computação em nuvem não é fácil, visto que os dados duplicados têm de ser eliminados numa infraestrutura distribuída onde estão a correr máquinas virtuais e aplicações com requisitos estritos de desempenho.

Esta dissertação foca estes desafios. Em primeiro lugar, analisamos e comparamos os sistemas de deduplicação existentes e os diferentes ambientes de armazenamento abordados por estes. Esta discussão permite compreender quais os desafios enfrentados pelos sistemas de deduplicação de computação em nuvem. Como contribuição principal, introduzimos o nosso próprio sistema que elimina dados duplicados entre volumes de máquinas virtuais numa infraestrutura de computação em nuvem distribuída. O conteúdo redundante é removido abrangendo toda a infraestrutura e de forma a introduzir um impacto mínimo no desempenho dos volumes deduplicados. O nosso protótipo é avaliado experimentalmente num cenário distribuído real e com uma ferramenta de avaliação apropriada para este tipo de sistemas, a qual é também uma contribuição desta dissertação.

Contents

List of Figures

List of Tables

Abbreviations

BLC Block Locality Cache. 30

CA-SSD Content-Addressable Solid State Drive. 48, 49

CAFTL Content-Aware Flash Translation Layer. 47–49

CAS Content-Addressable Storage. 29, 32, 33, 37, 45, 47

CDC Content-Defined Chunking. 19

CMD Classification-based Memory Deduplication. 44, 49, 50

CoW Copy-on-Write. 4, 5, 8, 9, 23, 28, 37–39, 42–45, 51, 53, 54, 59–61, 68, 69, 79, 80, 82–93, 95–97, 99, 100, 104, 106–110, 115

CPU Central Processing Unit. xvii, 18, 23, 26, 27, 40, 43, 72–75, 95, 98, 100, 104, 105, 107, 111

D. Finder Duplicates Finder. xv, 83–94, 96, 99, 100, 104, 106

DBLK Deduplication Block-Device. 40, 49

DDE Duplicate Data Elimination. 38, 49, 101, 108, 110

DDFS Data Domain File System. 30–33, 35, 36, 39, 41, 49

DDI Distributed Duplicates Index. 83–92, 94–102, 104–106, 111

DHT Distributed Hash Table. 17, 33, 40

DRAM Dynamic Random-Access Memory. 47, 48, 50, 114

FTL FLASH Transaction Layer. 46

GC Garbage Collector. xv, 84–86, 88, 89, 91–94, 96, 99, 101, 106, 107

GOT Global Offset Table. 44

HDAG Hash-based Directed Acyclic Graph. 29

HICAMP Hierarchical Immutable Content Addressable Memory Processor. 45, 49

HydraFS Hydra File System. 37, 49

I/O Input/Output. xii, 4, 5, 13, 14, 16, 17, 21–23, 25, 26, 34, 36–41, 43, 46–52, 54–56, 58, 60, 62, 63, 65, 69, 70, 73, 74, 76–78, 81, 84, 89, 93, 94, 96, 98–101, 103, 104, 107–111, 114, 117

IDC International Data Corporation. 3

IOPS Input/Output Operations Per Second. 99, 106, 107

KSM Kernel Same-page Merging. 43, 44, 49

LessFS Less File System. xvii, 37, 49, 69, 71–75, 78, 106, 115

LRU Least-Recently Used. 20, 21, 25, 29, 30, 41, 48

LVM Logical Volume Management. 15, 82, 94, 95, 97, 104, 108

POD Performance-Oriented I/O Deduplication. 41, 49

RAID Redundant Array of Inexpensive Disks. 4

RAM Random-Access Memory. xvii, 4, 8, 9, 11, 13, 16, 21, 22, 24, 26, 27, 29–34, 36, 39–42, 45, 48, 50–52, 54, 64, 72–75, 83, 94, 95, 97, 98, 100, 104, 105, 107–111, 113, 114, 116, 117

SAN Storage Area Network. 33, 38, 108

SIS Single Instance Storage. 25, 28, 29, 32, 49

SSD Solid State Drive. 4, 5, 7–9, 11, 13, 29, 32, 34, 39, 46–48, 50–52, 54, 110, 113, 114, 116, 117

TPC-C Transaction Processing Performance Council Benchmark C. 8, 56, 76, 78, 114

TTTD Two Thresholds-Two Divisors. 19, 29

VDI Virtual Desktop Infrastructure. 51

VM Virtual Machine. xii, xv, 4–7, 9, 11, 53, 79–82, 84–86, 88–90, 92–104, 108, 111, 113, 115–117

VMFS Virtual Machine File System. 38, 39, 108, 109

VMM Virtual Machine Monitor. 42

XLH Cross Layer I/O-based Hints. 44, 49

ZFS Z File System. 40, 49, 108

Chapter 1

Introduction

A study conducted by International Data Corporation (IDC) projects that digital data will reach 40 ZiB by 2020, corresponding to 50 times more information than the one reported in the beginning of 2010 [EMC 2012]. Cloud computing has a significant role in the management of such data and, from 2012 to 2020, the number of servers worldwide is expected to be 10 times higher, while the amount of digital information managed directly by data centers will increase by a factor of 14. This way, novel approaches that efficiently manage large amounts of digital content and reduce infrastructure costs are increasingly needed.

The automatic removal of duplicate data has proven to be a successful approach to tackle previous challenges, and is now present in several storage appliances [Zhu et al. 2008, Aronovich et al. 2009, You et al. 2005]. Undoubtedly, current usage patterns mean that multiple copies of the same data exist within a storage system, for instance, when multiple users of public cloud infrastructures independently store the same files, such as media, emails, or software packages.

This thesis is focused on storage deduplication, that we define as a technique for automatically eliminating coarse-grained and unrelated duplicate data in a storage system. Briefly, duplicate data belonging to distinct users is removed from the storage system that only persists an unique shared copy. However, the owners of duplicate content are not aware that their data is being shared, thus ensuring deduplication's transparency and privacy. Unlike traditional compression techniques that eliminate intra-file redundancy or redundancy over a small group of files, typically stored together in the same operation, deduplication aims at eliminating both intra-file and inter-file redundancy over large data sets and

possibly even across multiple distributed storage servers [Kulkarni et al. 2004]. Also, duplicates are found for data stored at different times by uncoordinated users and activities.

Deduplication has been in use for a long time in archival and backup systems [Bolosky et al. 2000, Quinlan and Dorward 2002, Cox et al. 2002]. Nowadays, this technique is no longer an exclusive feature of the latter storage types, and it is also being applied to primary storage, Random-Access Memory (RAM) and Solid State Drives (SSDs). The effectiveness of deduplication is usually measured by the *deduplication gain*, defined as the amount of duplicates actually eliminated, that is directly related with the achievable storage space reduction. As detailed in the literature, deduplication can reduce storage size by 83% in backup systems and by 68% in primary storage [Meyer and Bolosky 2011]. RAM used by virtualized hosts can be reduced by 33% [Waldspurger 2002] and the storage space of SSDs can be reduced by 28% [Chen et al. 2011]. The spared space allows reducing infrastructure costs but, it can also be used to improve reliability with, for instance, additional Redundant Array of Inexpensive Disks (RAID) configurations. Moreover, deduplication might have a positive performance impact throughout the storage management stack, namely, in cache and Input/Output (I/O) efficiency [Koller and Rangaswami 2010], and in network bandwidth consumption, when it is performed at the client side and only unique data is sent to the storage server [Muthitacharoen et al. 2001].

However, some of these storage environments have strict latency requirements for the requests being served by them. This way, maximizing deduplication gain is no longer the only goal, since minimizing its overhead in storage requests is also a requirement for enabling efficient deduplication. The core contribution of this document aims precisely at providing efficient deduplication for one of these environments; the cloud computing primary storage, more precisely, across Virtual Machines (VMs) primary volumes managed by cloud infrastructures [Srinivasan et al. 2012, El-Shimi et al. 2012, OpenSolaris 2014, Hong and Long 2004, Clements et al. 2009, Ng et al. 2011].

Cloud computing and, in particular, virtualized commodity server infrastructures bring novel opportunities, needs, and means to apply deduplication to VMs volumes stored in general purpose storage systems. 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 [Hewlett-Packard Development Company , L.P 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 VMs volumes. Space savings up to 80% are achievable when using both approaches and when cluster-wide deduplication is performed [Clements et al. 2009, Meyer and Bolosky 2011, Srinivasan et al. 2012]. With the unprecedented growth of data managed by cloud computing services and the introduction of more expensive storage devices, as SSDs, these additional space savings are key to reduce the costs and increase the capacity of enterprise cloud storage systems [Dan Iacono 2013].

Traditional *in-line* deduplication approaches, commonly used in backup systems, share data before storing it, thus including the computational overhead in storage write requests [Quinlan and Dorward 2002]. Primary storage volumes have strict latency requirements so, the overhead in the critical storage write path is usually not acceptable [Ng et al. 2011, Srinivasan et al. 2012]. As another option, *off-line* deduplication minimizes storage overhead by decoupling writes from aliasing operations, that are performed in the background [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 I/O requests are performed asynchronously so, appropriate mechanisms for preventing stale data checksums and other concurrency issues are necessary and, may degrade performance and scalability.

Unlike in archival and backup environments, primary storage data is modified and deleted very frequently, thus requiring an efficient CoW mechanism 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]. Unfortunately, off-peak periods are scarce or inexistent in cloud infrastructures hosting

VMs from several clients and with distinct workloads. This way, off-line deduplication has a short time-window for processing the storage backlog and eliminating duplicates. Ideally, deduplication should run continuously and duplicates should be kept on disk for short periods of time thus, reducing the extra storage space required.

Distributed cloud infrastructures raise additional challenges as deduplication must be performed 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 all cluster volumes. However, this is a complex operation 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 workloads and invalidates, once again, in-line deduplication. In fact this negative impact lead to 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].

1.1 Problem statement and objectives

In spite of the considerable space savings, primary storage deduplication in a cloud computing distributed infrastructure raises novel challenges that are not fully addressed by current proposals. Firstly, in order to maximize the deduplication gain, duplicates must be found across volumes of VMs that are running in several cluster servers. Moreover, deduplication must have a scalable and reliable design while introducing a negligible performance impact for the VMs dynamic volumes with strict latency requirements. Coping with both challenges is a difficult task, explaining why current systems are only able to maintain a negligible performance impact by trading off deduplication space savings, thus only finding duplicates in off-peak periods or across a subset of the cluster data [Clements et al. 2009, Dong et al. 2011, Srinivasan et al. 2012].

The main objective of this thesis is then to design a deduplication system for cloud computing primary storage infrastructures that is fully-decentralized, scalable, reliable and addresses the previous challenges.

Since there is a vast amount of work on storage deduplication, it is important to know the current features that may be useful for our system. However, there is still a general misconception about the common and distinct characteristics that deduplication systems possess and, there is still few information explaining how the distinct storage environments affect the designs os such systems. For instance, it is not clear why a specific system is efficient for backup storage but not for primary or SSD storage. This way, another objective of this thesis is to identify common design features shared by all deduplication systems, and then to discuss the different optimizations driven by the targeted storage environment while, showing their applicability in cloud computing primary storage infrastructures.

Deduplication designs are commonly validated by implementing prototypes and then evaluating them empirically with static datasets or benchmarking tools. Static datasets are useful to evaluate archival deduplication systems but are not able to simulate the dynamism of primary volumes where data is updated frequently [Tarasov et al. 2012]. On the other hand, there are some open-source micro-benchmarks that can achieve this dynamism but are not able to generate content in a realistic fashion. This means that, in most cases, all written data either has the same content, or it has random content with no duplicates at all, which does not allow a proper evaluation of any deduplication system [Coker 2014, Katcher 1997, Anderson 2002]. This challenge leads to our third objective that is to develop a benchmark that is able to simulate both the dynamism and realistic content found in real storage infrastructures, thus allowing to evaluate properly systems such as the one discussed in this thesis.

1.2 Contributions

As the main contribution of the thesis, the combined challenges of cloud computing primary storage and cluster deduplication are addressed with DEDIS, a dependable and fully-decentralized system that performs cluster-wide off-line deduplication of VMs primary volumes. More specifically, deduplication is performed globally across the entire cluster, in a fully-decentralized and scalable fashion, by using a partitioned and replicated fault tolerant distributed service that indexes storage blocks with unique content and allows finding duplicates. As all storage blocks are indexed by this service, deduplication is performed in

an exact fashion across the whole cluster, ensuring that, all duplicate blocks are processed and eventually shared. Also, an optimistic off-line deduplication approach avoids costly computation and calls to the previous remote service in the storage write path. Along with this optimistic approach, we introduce several optimizations that allow deduplication to run simultaneously with storage requests while having a negligible impact in the performance of both.

Unlike previous related systems, DEDIS works on top of any storage backend exporting an unsophisticated shared block device interface, that may be distributed or centralized. This way, our system does not rely on backends with built-in locking, aliasing, CoW or garbage collection operations. Although this decision significantly impacts the system design and favors distinct optimizations, it allows decoupling the deduplication systems from a specific storage specification and avoids performance issues that arise from this dependency [Hong and Long 2004, Clements et al. 2009]. Also, our design does not rely on storage workloads with specific properties, such as data locality, to achieve low storage overhead and an acceptable deduplication throughput [Srinivasan et al. 2012].

As another contribution, we present an extensive survey of current storage deduplication systems, detailing the main challenges addressed by them and specific design decisions while, clarifying some misunderstandings and ambiguities in this field. Firstly, we extend the existing taxonomy [Mandagere et al. 2008] and identify key design features common to all deduplication systems. For each of these features, we describe the distinct approaches taken to address deduplication main challenges. Then, we group existing deduplication systems into four different storage groups: archival and backup storage, primary storage, RAM and SSDs. We show that each storage group has distinct assumptions that impact deduplication designs.

As a third contribution, we present DEDISbench, a block-based synthetic disk micro-benchmark with novel features for evaluating deduplication systems in a more realistic environment. As the main novelty, data written by the benchmark mimics the content of distributions extracted from real datasets. These distributions can be automatically extracted from any storage system with another tool, named DEDISgen, thus allowing to simulate the content of distinct storage environments. As another feature, DEDISbench supports an hotspot random access distribution, based on Transaction Processing Performance Council Benchmark

C (TPC-C) NURand function, that simulates hotspot disk accesses [Transaction processing performance council 2010]. This feature is key for simulating a dynamic storage environment where a small percentage of data blocks are hotspots, with a high percentage of accesses, while most blocks are only accessed sporadically. Write hotspots increase the number of blocks frequently rewritten and, consequently, the amount of CoW operations which, are known to have a negative impact in primary deduplication [Clements et al. 2009].

1.3 Results

The work discussed in this thesis resulted in a number of publications in distinct international journals and conferences:

- João Paulo and José Pereira. A Survey and Classification of Storage Deduplication Systems. *ACM Computing Surveys*, 47(1):1–30, 2014

  This journal publication surveys existing deduplication systems and classifies them according to the targeted storage environment, *i.e.,* archival and backup, primary, RAM and SSD storage. Also, an existing taxonomy that identifies key design features common to all deduplication systems is extended with novel classification axes.

- João Paulo and José Pereira. Distributed Exact Deduplication for Primary Storage Infrastructures. In *Proceedings of Distributed Applications and Interoperable Systems (DAIS)*, 2014

  This conference publication describes DEDIS, a dependable and fully-decentralized system that performs deduplication across VMs primary volumes in a distributed cloud infrastructure. The main system design is detailed, as well as, some optimizations that reduce the overhead in storage requests while increasing deduplication throughput. The evaluation of our prototype shows that negligible overhead is possible while executing storage requests and running deduplication simultaneously.

- João Paulo, Pedro Reis, José Pereira, and António Sousa. DEDISbench: A Benchmark for Deduplicated Storage Systems. In *Proceedings of International Symposium on Secure Virtual Infrastructures (DOA-SVI)*, 2012

This conference paper presentes DEDISbench, a micro-benchmark for evaluating deduplication systems. Data written by the benchmark follows realistic content distributions that were automatically extracted from real storage systems with another tool called DEDISgen which, is also introduced in the paper. A novel feature for simulating hotspot storage accesses is also discussed while, two open-source deduplication systems, Opendedup and Lessfs, are evaluated by DEDISbench.

- João Paulo, Pedro Reis, José Pereira, and António Sousa. Towards an Accurate Evaluation of Deduplicated Storage Systems. *International Journal of Computer Systems Science and Engineering*, 29(1):73–83, 2013

  This journal publication extends the previous DEDISbench paper by extracting and analyzing the duplicate distributions of three real storage systems. More specifically, the DEDISgen tool is used to extract the content distributions of an archival, a backup and a primary storage belonging to our research group. Finally, the paper shows that each storage type has distinct characteristics, and extends our benchmark with the capability of simulating the novel distributions.

Also, preliminary versions of our work were accepted as fast abstracts or poster abstracts and are listed below:

- João Paulo and José Pereira. DEDIS: Distributed Exact Deduplication for Primary Storage Infrastructures. In *Poster Proceedings of the Symposium on Cloud Computing (SOCC)*, 2013

  This poster abstract presents a preliminary version of the DEDIS system, which is further detailed in the paper from DAIS'14.

- João Paulo and José Pereira. Model Checking a Decentralized Storage Deduplication Protocol. In *Fast Abstract of the Latin-American Symposium on Dependable Computing (LADC)*, 2011. URL `http://haslab.uminho.pt/jtpaulo/files/pp09.pdf`

  This fast abstract explains how model-checking with the TLA+ toolset was used to uncover and correct some subtle concurrency issues in a preliminary version of DEDIS algorithm.

The following work was submitted and it is still in review process:

- João Paulo and José Pereira. Efficient Deduplication in a Distributed Primary Storage Infrastructure. Submitted to *ACM Transactions on Storage Journal*, 2014

  This journal publication extends the DEDIS paper from DAIS'14 by introducing a novel optimization, a detailed description of the fault-tolerant design and a more realistic evaluation setup. More specifically, the paper presents a cache optimization that increases storage performance by avoiding some of the storage reads done with the deduplication engine. Also, an evaluation with up to 32 servers in a fully-symmetric setup where servers run both VMs and DEDIS components is discussed.

DEDIS, DEDISbench and DEDISgen are open-source projects and are publicly available at `http://www.holeycow.org`. Finally, we also published the next work in collaboration with other researchers that, is indirectly related with the thesis:

- Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. MeT: Workload Aware Elasticity for NoSQL. In *Proceedings of the ACM European Conference on Computer Systems (EUROSYS)*. ACM, 2013

1.4 Outline

The rest of the document is structured as follows:

Chapter 2 presents a detailed survey of storage deduplication systems. More specifically, the chapter starts by introducing a classification of deduplication systems according to key design features, discussing the distinct approaches used for each feature, as well as, their relative strengths and drawbacks. Then, it surveys existing systems grouped by type of storage targeted, *i.e.*, archival and backup storage, primary storage, RAM and SSDs, explaining how the distinct features used by these systems suit each storage environment.

Chapter 3 introduces DEDISbench, a micro-benchmark suitable for deduplication systems. Namely, the benchmark design, implementation and features are

described. Then, the DEDISgen tool is presented and used for extracting the duplicate content distributions from three real storage environments; an archival, a backup and a primary storage. DEDISbench is compared with two open-source micro-benchmarks, Bonnie++ and IOzone, and the three benchmarks are used to evaluate two deduplication systems, Opendedup and LessFS.

Chapter 4 presents DEDIS, a dependable and fully-decentralized primary storage deduplication system. We start by describing the baseline distributed storage architecture assumed by our system, and then, we discuss the components, fault-tolerance considerations, optimizations and implementation details. To conclude, DEDIS open-source prototype is evaluated in up to 32 servers and compared with a storage system without deduplication to measure the impact in storage requests performance, as well as, deduplication performance and scalability.

Chapter 5 concludes the thesis and discusses possible future work in the field of storage deduplication.

Chapter 2

Storage deduplication background

Deduplication is now desirable in several storage environments such as; archival and backup storage, primary storage, RAM, and SSDs [Bolosky et al. 2000, Waldspurger 2002, Hong and Long 2004, Chen et al. 2011]. However, there is still a general misconception about the common characteristics shared by all systems, as well as, the specific optimizations and functionalities that make distinct systems appropriate for specific storage environments. This chapter aims precisely at clarifying such information by providing a novel taxonomy and classification of today's storage deduplication approaches.

Storage deduplication can be regarded as bidirectional mapping between two different views of the same data: a logical view, containing identifiable duplicates, and a physical view, as stored in actual devices from which duplicates have been removed. The mapping process is embodied in the I/O path between applications that produce and consume the data and, the storage devices themselves. Figure 2.1 depicts each of these views and identifies key features in each of them that lead to different design decisions and trade-offs.

The logical view of data in a deduplication system is a set of assumptions on the workload that determine which duplicate content is relevant, hence which duplicates exist and which should be removed. First, all deduplication systems partition data into discrete *chunks* that are to be compared, identified as duplicate, and eventually removed. This partitioning can be done with different *granularity*, using various criteria for chunk boundaries as well as for their sizes.

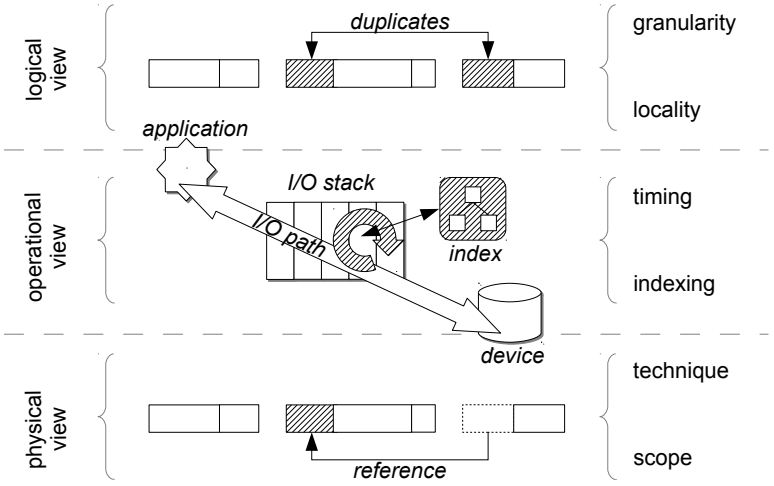


Figure 2.1: Views of deduplication and key design features.

In the remainder of this document, we refer to *chunks* as the unit of deduplication that, in existing deduplication systems can correspond to files, variable-sized blocks, or fixed-size blocks. Although segment is sometimes used as a synonym of chunk, we avoid it as it is also used in some proposals as a higher granularity unit composed by a large number of chunks, thus leading to ambiguity [Lillibridge et al. 2009]. Moreover, assumptions on the likelihood of duplicate chunks being found close together, both in space or time, lead to design decisions exploiting *locality* that influence both the efficiency and effectiveness of the deduplication process.

On the other hand, the physical view of data in a deduplication system is first and foremost concerned with the *technique* used on disk to represent duplicate data that has been removed, such that efficient reconstruction of the logical view is possible. Given the current relevance of distributed storage systems, a key design decision is the *distribution scope* of the deduplication technique. This can be defined as the ability to represent removed duplicates across different nodes, such that the reconstruction of data requires their collaboration.

Finally, deduplication as a process has to be understood as happening in the context of a storage management system. This exposes an API to client applications, such as a file system or a block device, and is composed by multiple stacked software layers and processing, networking, and storage components. The key design issue here is the *timing* of the main deduplication operations, such as searching for duplicates, regarding the critical path of I/O operations. Since

finding duplicates is potentially a resource intensive operation, it is invariably based on an *indexing* data structure that supports efficient matching of duplicate chunks. Thus, the indexing method has a strong impact not only on the efficiency of the deduplication process, but also on its effectiveness by potentially trading off exactness for speed. Also, most systems do not index the full content of chunks, using instead compact signatures of the chunks' content. These are generally calculated with hashing functions and we refer to them as *chunk signatures* or *digests*.

This chapter is focused only on deduplication in storage systems. Namely, we do not address network deduplication [Muthitacharoen et al. 2001], although some systems we refer to do both network and storage deduplication [Cox et al. 2002]. In fact, as explained in this document, most systems that perform deduplication before actually storing the data can off-load some of the processing to the client and avoid sending duplicate chunks over the network.

Also, we do not address distributed Logical Volume Management (LVM) systems with snapshot capabilities that already avoid creating duplicates among snapshots of the same VM or VMs created from the same snapshot [Meyer et al. 2008]. Although these systems share some technical issues with deduplication, such as reference management and garbage collection, they are fundamentally different by not addressing the removal of unrelated duplicate chunks. Finally, we do not address delta-based versioning systems where delta-encoding is only done across versions of the same file [Berliner 1990, Burns and Long 1997]. We focus on deduplication systems that eliminate both intra-file and inter-file redundancy over large data sets without any knowledge regarding data versions.

2.1 Challenges

In order to understand the different deduplication designs, it is important to first know the challenges that current systems must overcome.

2.1.1 Overhead vs. gain

The main challenge in deduplication systems is the trade-off between the achievable deduplication gain and the overhead on a comparable storage system that does not include deduplication. As an example, smaller chunk sizes increase the

space saving benefits of deduplication, but lead to larger index structures that are more costly to maintain. Ideally, the index would be fully loaded into RAM, but for a large storage and a relatively small chunk size the index is too large and must be partially stored on disk. This increases the number of disk I/O operations needed by deduplication which may interfere with the foreground I/O performance [Zhu et al. 2008].

Also, deduplication should be performed as soon as data enters the storage system to maximize its benefits. However, finding duplicates is a resource intensive task that will impact latency if performed in the critical path of storage writes. If deduplication is removed from the critical path and done in the background, additional temporary storage is required and data must be read back from the storage to find duplicates, thus increasing the consumption of storage I/O bandwidth [Srinivasan et al. 2012].

The more data chunks are omitted, the more the physical layout of deduplicated data differs from the original layout. Namely, deduplication introduces fragmentation that deteriorates the performance of read and restore operations [Kaczmarczyk et al. 2012, Lillibridge et al. 2013, Mao et al. 2014a, Fu et al. 2014]. Additional metadata is also required for correctly reconstructing deduplicated data [Meister et al. 2013a]. Thus, there is additional overhead involved in maintaining the integrity of such metadata as one must ensure that a certain shared chunk is no longer serving any I/O request before modifying or deleting it. More specifically, this requires managing references to shared chunks, which is complex and requires a garbage collection mechanism that may also impact performance [Guo and Efstathopoulos 2011, Strzelczak et al. 2013, Botelho et al. 2013].

2.1.2 Scalability vs. gain

The most gain can be obtained when any chunk can, in principle, be compared with any other chunk and be omitted if a match is found. However, such complete matching is harder as the amount of data and components grow, in a large scale storage system. Briefly, a centralized index solution is likely to become itself very large and its manipulation a bottleneck on deduplication throughput [Clements et al. 2009]. Partial indexes that can match only a subset of duplicates improve scalability, but perform only *partial* deduplication. Nonetheless, the amount of chunks that cannot be matched can be reduced by exploring data

locality [Lillibridge et al. 2009] and by grouping together chunks with greater similarity [Manber 1994].

In a distributed storage system, a simple strategy for scalability is to perform deduplication independently in each node thus, having multiple independent indexes. Again, this approach allows only partial deduplication as the same chunk might be duplicated in multiple nodes. Missed deduplication opportunities can be mitigated by grouping, in the same node, chunks that have a greater likelihood of containing matching data, for instance, by routing the most similar files to the same nodes [Bhagwat et al. 2009].

The trade-off between scalability and gain can be improved by using a Distributed Hash Table (DHT) as the index. The DHT is accessible by all nodes, which allows eliminating duplicates globally in an *exact* fashion [Dubnicki et al. 2009, Ungureanu et al. 2010]. However, a remote invocation to the index is required to find duplicate or similar chunks. If the index is accessed in the critical I/O path, which is common in many systems, it may lead to an unacceptable storage latency penalty.

2.1.3 Reliability, security and privacy

Distributed deduplication systems must tolerate node crashes, data loss and even byzantine failures [Douceur et al. 2002]. Eliminating all duplicate data will also eliminate all the redundancy necessary for tolerating data loss and corruption, so a certain replication level must be maintained. Studies show that it is possible to achieve both, but few systems address these issues [Bhagwat et al. 2006, Rozier et al. 2011]. Metadata must also be resilient to failures and, it needs to be stored persistently, which reduces deduplication space savings. Additionally, both data and metadata must be distributed in large scale systems to tolerate single node failures while maintaining high availability.

Some deduplication systems share data from distinct clients, raising privacy and security issues that can be solved by trading off deduplication space savings [Nath et al. 2006]. Security and privacy issues are expected not only in cloud storage infrastructures, but also in remote storage appliances where data from several clients is stored [Harnik et al. 2010].

2.2 Classification criteria

This section introduces a taxonomy for classifying deduplication systems by expanding on previous proposals [Mandagere et al. 2008]. This classification is based on the major design decisions implicit in all deduplication systems as summarized in Figure 2.1: granularity, locality, timing, indexing, technique and distribution scope.

2.2.1 Granularity

Granularity refers to the method used for partitioning data into chunks, the basic unit for eliminating duplicates. Given its importance in the overall design of a deduplication system, it has been simply referred to as the *deduplication algorithm* [Mandagere et al. 2008]. However, there are significant concerns other than granularity that justify avoiding such name.

One of the most straightforward approaches is the *whole file chunking* in which data is partitioned along file boundaries set by a file system [Bolosky et al. 2000]. As many backup systems are file-oriented, whole file chunking avoids the partitioning effort and, by doing deduplication at a higher granularity, there are less chunks to index and to be processed by the deduplication engine [Policroniades and Pratt 2004].

Another common approach used has been to partition data into *fixed-size chunks*, also referred to as *fixed-size blocks* or simply *blocks*. This is particularly fit for a storage system that already uses such partition into fixed-size blocks [Quinlan and Dorward 2002, Hong and Long 2004]. In fact, for the cases where changed data is dynamically intercepted at a small granularity, the fixed-size block approach can offer high processing rates and generate less Central Processing Unit (CPU) overhead than other alternatives with identical sharing rates [Policroniades and Pratt 2004, Constantinescu et al. 2011]. By adjusting the size of chunks, deduplication gain can be increased at the expense of additional overhead in processing, metadata size, and fragmentation [Policroniades and Pratt 2004, Kaczmarczyk et al. 2012, Mao et al. 2014a].

Consider now two versions of the same file where version A only differs from version B by a single byte that was added to the beginning of the latter version. Regardless of files being considered as a whole or partitioned into fixed-size

chunks, in the worst case scenario, no chunks from version A will match chunks from version B. This issue is referred to in the literature as the *boundary-shifting problem* [Eshghi and Tang 2005].

The third option, which solves this problem, is to partition data into *variable-sized chunks* with boundaries set by the content itself, also called Content-Defined Chunking (CDC) [Muthitacharoen et al. 2001]. The first version of the algorithm uses a sliding window that moves over the data until a fixed content pattern defining the chunk boundary is found. This approach generates variable-sized chunks and solves the issue of inserting a single byte in the beginning of version B. More precisely, only the first chunk from version B will differ from the first chunk of version A due to the byte addition, while the remaining chunks will match and will be deduplicated.

In this version of the algorithm, a minimum and maximum size restriction was introduced for preventing too small or large chunks. This modification raises, once again, the boundary-shifting problem for large chunks whose boundaries are defined by the maximum size threshold instead of using the content-based partitioning. The Two Thresholds-Two Divisors (TTTD) algorithm uses two thresholds to impose a maximum and minimum chunk size, as in previous work, but also uses two divisors for defining chunk boundaries [Eshghi and Tang 2005]. The first divisor is similar to the one chosen in the original CDC algorithm, while the second divisor has a larger probability of occurrence. The chunk is calculated with the sliding window, as in the original algorithm, but whenever the second divisor is found the last occurrence is registered as a possible breakpoint. When the maximum size of a chunk is reached, meaning that the first divisor was not found, then the chunk boundary is defined by the last time the second divisor was found in the chunk. Therefore, the probability of occurring the boundary-shifting problem is significantly reduced.

The above algorithms produce variable-sized chunks within a predefined size; however, other algorithms increase the variability of chunk size to reduce metadata space without losing deduplication gain. Fingerdiff is a dynamic partitioning algorithm that creates large chunks for unmodified regions of data, which cannot be shared, and smaller chunks (sub-chunks) for changed data regions to increase space savings [Bobbarjung et al. 2006]. As an example, when a new version of a previous stored file is received, sub-chunks will be small enough for capturing

small changes in the file and sharing them, boosting space savings, while the unmodified data will still be stored as larger chunks, reducing indexing space costs. Two other algorithms aimed at increasing chunk size variability without significantly affecting deduplication gain were presented in Bimodal content-defined chunking [Kruus et al. 2010]. The *breaking apart* algorithm divides backup data streams into large size chunks, and then further divides the chunks into smaller sizes when deduplication gain justifies it. On the other hand, the *building-up* algorithm divides the stream into small chunks that are then composed when the deduplication gain is not affected. Moreover, a variant of the breaking apart algorithm can be combined with a statistical chunk frequency estimation algorithm, further dividing large chunks that contain smaller chunks appearing frequently in the data stream and consequently allowing higher space savings [Lu et al. 2010].

Each method described here can be combined with techniques that eliminate exact duplicates or that can cope with similar but not fully identical chunks, as in delta-encoding [Quinlan and Dorward 2002, Policroniades and Pratt 2004, Nath et al. 2006, Aronovich et al. 2009]. More specifically, both aliasing and delta-encoding, detailed in Section 2.2.5, can be applied to whole files, fixed-size chunks, or variable-sized chunks. However, the optimal chunk size is related to the technique being used, for instance, chunks in delta-encoding deduplication can be larger than in exact deduplication without reducing the deduplication gain.

2.2.2 Locality

Locality assumptions are commonly exploited in storage systems, mainly to support caching strategies and on-disk layouts. Similarly, locality properties found for duplicate data can be exploited by deduplication, making deduplication gain depend on the workload's locality characteristics. However, there are systems that do not assume any specific locality assumptions for their storage workloads. [Dubnicki et al. 2009, Yang et al. 2010a, Clements et al. 2009].

Temporal locality means that duplicate chunks are expected to appear several times in a short time window. More specifically, if chunk A was written, it will probably be written again several times in the near future. Temporal locality is usually exploited by implementing caching mechanisms with Least-Recently Used (LRU) eviction policies [Quinlan and Dorward 2002]. Caching some of the

entries of the index in RAM can reduce disk accesses while reducing memory usage. In workloads that exhibit poor temporal locality, the LRU cache, however, is inefficient and most accesses are directed to the disk index thus, creating a bottleneck.

Spatial locality means that chunks present in a specific data stream are expected to appear in subsequent streams in the same order. For example, if chunk A is followed by chunks B and C in a data stream, the next time that chunk A appears in another stream, it will probably be followed by chunks B and C again. Spatial locality is commonly exploited by storing groups of chunks in a storage layout that preserves their original order in the stream. Then, the signatures of all chunks belonging to the same group are brought to a RAM cache when one of the signatures is looked up in the disk [Zhu et al. 2008, Rhea et al. 2008]. For example, if a stream has chunks with content signatures A, B, and C, then these chunks and their signatures are stored together on disk. When a chunk with signature A is written, the signatures for chunk A, B, and C are brought to memory, because chunks B and C will probably appear next in the stream due to spatial locality and, additional disk accesses to the index can be avoided. Furthermore, temporal and spatial locality can be exploited together [Srinivasan et al. 2012].

2.2.3 Timing

Timing refers to when detection and removal of duplicate data are performed. More specifically, if duplicates are eliminated before or after being stored persistently. *In-line* deduplication, also known as *in-band* deduplication, is done in the critical path of storage write requests. This approach requires intercepting storage write requests, calculating chunk boundaries and signatures, if necessary, finding a match for the chunk at the index and, if found, sharing the chunk or delta encoding it. Otherwise, if the match is not found, the new chunk signature must be inserted at the index. Only then, the I/O request completion is acknowledged.

In-line deduplication is widely used in several storage back ends [Quinlan and Dorward 2002, Rhea et al. 2008] and file systems [Zhu et al. 2008, Ungureanu et al. 2010]. In addition, in-line deduplication is possible only if I/O requests can be intercepted. One of the main drawbacks of in-line deduplication is the

overhead introduced in the latency of write requests as most of the processing is done in the write path. In fact, one of the major bottlenecks is the latency of operations to the on-disk index, which could be solved by loading the full index to RAM, but that does not scale for large data sets. There are some scenarios where this overhead may not be acceptable, for instance, in primary storage systems with strict I/O latency requirements [Srinivasan et al. 2012]. Nevertheless, there are proposals for reducing this impact with optimizations that explore locality, as discussed in Section 2.3.

A variant of in-line deduplication, in client-server storage systems, partitions data and computes content signatures at the client side, sending first only compact chunk signatures to the server [Bolosky et al. 2000, Waldspurger 2002]. Then, the server replies back to the client identifying missing chunks that were not present at the server storage and must be transmitted. This way, only a subset of the chunks is sent and network bandwidth is spared [Cox et al. 2002]. This issue has been referred to as *placement* [Mandagere et al. 2008]; however, it is not considered in this survey as a general design decision shared by all deduplication systems.

As an alternative to in-line deduplication, some systems do *off-line* deduplication where data is immediately written to the storage and then scanned in the background to search and eliminate duplicates. This technique is also referred to as *off-band* or *post-processing* deduplication. Since deduplication is no longer included in the write critical path, the overhead introduced in I/O latency is reduced. This approach requires less modifications to the I/O layer, but needs additional resources to scan the storage, searching for changed chunks that need to be deduplicated. Moreover, as data is first stored and then shared asynchronously, off-line deduplication requires temporarily more storage space than in-line deduplication.

Scanning the storage in off-line deduplication can be avoided by intercepting write requests to determine which chunks have been written and may be deduplicated. Concurrently in the background, the deduplication mechanism collects modified addresses, reads the corresponding data from the storage, and eliminates duplicates. Moreover, the calculation of content signatures may be done in the write path thus, reducing the need of reading the chunk content from disk. These optimizations are able to detect modified content without requiring a stor-

age scan while, still introducing negligible overhead in I/O operations [Hong and Long 2004, Clements et al. 2009]. In both scan and interception strategies, a CoW mechanism is required to ensure that shared data cannot be concurrently changed by a storage write. This is a costly mechanism that adds significant overhead in storage writes latency, but that, is required for avoiding data corruption [Clements et al. 2009]. Finally, in some off-line deduplication systems, I/O and deduplication operations concurrently update common metadata structures, leading to lock mechanisms that result in fairness and performance penalties for both aliasing and I/O operations if implemented naively [Clements et al. 2009].

2.2.4 Indexing

Indexing provides an efficient data structure that supports the discovery of duplicate data. With the exception of some systems that index actual chunk content [Arcangeli et al. 2009], most systems summarize content before building the index [Bolosky et al. 2000, Quinlan and Dorward 2002]. A compact representation of chunks reduces indexing space costs and speeds up chunk comparison.

Summarizing content by hashing leads to *identity signatures* that can be used to search for exact duplicates. As a drawback, hash computation needs additional CPU resources, which may be problematic for some systems, and may generate collisions where, the same signature is used to summarize the content of two distinct chunks [Chen et al. 2011]. The latter issue can be avoided by comparing the content of two chunks with the same identity signatures before aliasing them thus, preventing hash collisions [Rhea et al. 2008]. However, byte comparison of chunks will increase the latency of deduplication and I/O operations if deduplication is done in the storage write path while, the probability of hash collisions is negligible [Quinlan and Dorward 2002].

The similarity of two chunks can be assessed by computing a set of Rabin fingerprints for each chunk, and then comparing the number of common fingerprints [Manber 1994], which we refer to as *similarity signatures* herein. Rabin fingerprints can be calculated in linear time and are distributive over addition, thus allowing a sliding window mechanism to generate variable-sized chunks and compose fingerprints efficiently [Rabin 1981, Broder 1993]. Comparing a large number of fingerprints to find similar chunks may present a scalability problem and require a large index, so a set of heuristics was introduced for coalescing

a group of similarity fingerprints into super-fingerprints. Two matching super-fingerprints indicate high resemblance between the chunks, thus scaling the index to a larger number of chunks [Broder 1997].

Signatures are then used to build the indexing data structure. With a *full index*, all computed signatures are indexed, thus having an entry for each unique chunk at the storage. This finds all potential candidates for deduplication [Bolosky et al. 2000, Quinlan and Dorward 2002], but the size of the index itself becomes an obstacle to performance and scalability. Namely, it becomes too large to be kept in RAM, and storing it on disk has a profound impact on deduplication throughput [Quinlan and Dorward 2002].

This problem has been addressed by using a *sparse index*, in which a group of stored chunks are mapped by a single entry at the index. As an example, a sparse index can be built by grouping several chunks into segments that are then indexed with similarity signatures instead of identity signatures [Lillibridge et al. 2009]. Since segments are coarse-grained, the size of this primary index is reduced and can be kept in RAM. Then, each segment may have an independent secondary index of identity signatures, corresponding to its chunks, that is stored on disk. When a new segment is going to be deduplicated, its similarity signature is calculated and only a group of the most similar segments have their identity secondary indexes brought to RAM. By only loading the secondary indexes of the most similar segments to RAM, deduplication gain is kept acceptable while using less RAM. There are also other proposals of sparse indexes that, for example, exploit file similarity [Bhagwat et al. 2009]. We discuss these specific designs in Section 2.3. Sparse indexes are able to scale to large data sets, but restrict the deduplication gain since some duplicate chunks are not coalesced thus, performing only partial deduplication. However, as the RAM footprint is reduced, each segment can hold smaller chunk sizes that will allow finding more duplicates while still scaling for larger data sets.

A third alternative is a *partial index* where each index entry maps a single unique chunk, but only a subset of unique stored chunks are indexed, unlike in the full index approach. Therefore, the RAM utilization is always under a certain threshold by sacrificing space savings and performing only partial deduplication [Guo and Efstathopoulos 2011, Chen et al. 2011, Gupta et al. 2011, Kim et al. 2012]. The index eviction is made based on a pre-defined policy, for

example, by using an LRU policy or by evicting the less referenced signatures.

2.2.5 Technique

Two distinct representations of stored data that eliminate duplicate content are discussed in the literature. With *aliasing*, also known as chunk-based deduplication, exact duplicates can be omitted by using an indirection layer that makes them refer to a single physical copy. I/O requests for aliased chunks are then redirected accordingly.

Alternatively, *delta-encoding* eliminates duplicate content among two similar but not fully identical chunks. Namely, only one chunk is fully stored, the base chunk, while the distinct content necessary to restore the other chunk is stored separately as a delta or diff. Therefore, the duplicate information is stored only once in the base chunk and the other chunk can be restored by applying the diff to the base version.

Aliasing requires less processing power and has faster restore time than delta deduplication since no fine-grained differences need to be calculated or patched to recover the original chunk [Burns and Long 1997]. On the other hand, delta-encoding saves additional space in chunks that do not have the same exact content thus, allowing the chunk size to be increased without reducing the deduplication gain [You and Karamanolis 2004, Aronovich et al. 2009]. In addition, delta-encoding is performed across a pair of chunks so, it is important to deduplicate chunks with the most similar content to achieve higher deduplication factors. Therefore, the mechanism chosen for detecting similar chunks is key for improving space savings. Finally, the performance of delta deduplication also changes with the delta-encoding algorithms used [Hunt et al. 1998].

Most storage deduplication systems use aliasing, being Microsoft Single Instance Storage (SIS) [Bolosky et al. 2000] and Venti [Quinlan and Dorward 2002] the pioneers. On the other hand, although there are some studies regarding the efficiency of applying delta deduplication on large file collections [Ouyang et al. 2002, Douglis and Iyengar 2003], the first complete deduplication system based exclusively on delta deduplication was proposed by IBM Protect Tier [Aronovich et al. 2009]. However, there are other systems that combine both techniques by first applying aliasing, which eliminates all redundant chunks, and then delta deduplication for chunks that did not exactly match any other chunk, but could

be stored more efficiently if delta-encoded [You et al. 2005, Shilane et al. 2012]. Moreover, other proposals also combine chunk compression with the previous two techniques for reducing even further the storage space [Kulkarni et al. 2004, Gupta et al. 2010, Constantinescu et al. 2011, El-Shimi et al. 2012].

Both aliasing and delta-encoding require metadata structures for abstracting the physical sharing from the logical view. For instance, many storage systems store and retrieve data at the file level abstraction, even if files are then partitioned into smaller chunks for deduplication purposes. In these systems, it is necessary to have, for example, tree structures that map files to their chunk addresses and that must be consulted for file restore operations [Quinlan and Dorward 2002, Meister et al. 2013a]. Other systems intercept I/O calls and deduplicate at the block level abstraction, having already metadata for mapping logical blocks into storage addresses [Hong and Long 2004, Chen et al. 2011]. In these cases, aliasing engines must update these logical blocks to the same physical address while, delta engines must update the logical blocks to point to the base chunks and corresponding deltas. In fact, in all systems where content to be read does not have an associated signature that allows searching directly for chunk addresses in indexing metadata, additional I/O mapping structures are necessary to translate read requests to the corresponding chunks. Finally, as some systems delete or modify chunks, knowing the number of references for a certain aliased or base chunk is important, because when a chunk is no longer being referenced, it can be garbage collected [Guo and Efstathopoulos 2011, Strzelczak et al. 2013, Botelho et al. 2013]. Both I/O translation and reference management mechanisms must be efficient to maintain low storage I/O latency and to reclaim unused storage space.

2.2.6 Scope

Distributed systems perform deduplication over a set of nodes to improve throughput and/or gain while also scaling out for large data sets and a large number of clients. Unlike in centralized deduplication, some distributed deduplication systems need to define routing mechanisms for distributing data over several nodes with independent CPU, RAM and disks. Moreover, by having several nodes, it is possible to increase the parallelism and, consequently, increase deduplication throughput while also tolerating node failures and providing high availability [Cox

et al. 2002, Douceur et al. 2002, Bhagwat et al. 2009]. Other distributed systems assume nodes with individual CPU and RAM that have access to a shared storage device abstraction where nodes perform deduplication in parallel. This allows to share metadata information between nodes by keeping it on the shared storage device, which otherwise would have to be sent over the network [Clements et al. 2009, Kaiser et al. 2012]. Finally, distinct nodes may handle distinct tasks, for instance, while some nodes partition data and compute signatures, other nodes query and update indexes, thus parallelizing even further the deduplication process [Yang et al. 2010b;a].

The key distinction is the scope of duplicates that can be matched and represented after being removed. In distributed deduplication systems with a *local scope*, each node only performs deduplication locally, and duplicate chunks are not eliminated across distinct nodes. This includes systems where nodes have their own indexes and perform deduplication independently [You et al. 2005]. Some systems introduce intelligent routing mechanisms that map similar files or groups of chunks to the same node to increase the cluster deduplication gain [Bhagwat et al. 2009, Dong et al. 2011]. In these systems, deduplication is still performed at a smaller granularity than routing and in a local fashion thus, not eliminating all duplicate chunks globally across distinct cluster nodes.

In contrast, in distributed deduplication systems with a *global scope*, duplicate chunks are eliminated globally across the whole cluster. In this case, an index mechanism accessible by all cluster nodes is required so that, each node is able to deduplicate its chunks against other remote chunks. Some systems use centralized indexes that have scalability and fault tolerance issues [Hong and Long 2004] while, other solutions use decentralized indexes that, solve previous issues but increase the overhead of lookup and update operations [Douceur et al. 2002, Dubnicki et al. 2009, Hong and Long 2004, Clements et al. 2009]. When compared to local approaches, global distributed deduplication increases space savings by eliminating duplicates across the whole cluster. However, there is an additional cost for accessing the index, which, for example, in primary storage systems may impose unacceptable storage latency [Ungureanu et al. 2010].

Finally, storage systems that were devised to perform deduplication in a single node are *centralized*, even if they support data from a single or from multiple clients [Quinlan and Dorward 2002, Rhea et al. 2008]. In a cluster infrastructure,

these systems do not take any processing advantage from having several nodes and do not eliminate duplicate chunks across remote nodes.

2.3 Survey by storage type

This section presents an overview of existing deduplication systems, grouped by storage type, and their main contributions for addressing the challenges presented in Section 2.1. Moreover, each system is classified according to the taxonomy described in the previous section. As each storage environment has its own requirements and restrictions, the combination of design features changes significantly with the storage type being targeted.

2.3.1 Backup and archival

As archival and backup storage have overlapping requirements, some solutions address both [Yang et al. 2010b]. In fact, most systems targeting either one of these storage environments have common assumptions regarding data immutability, and favor storage throughput over latency. Nonetheless, restore and delete operations are expected to be more frequent for backups than for archives, where data deletion is not even supported by some systems [Quinlan and Dorward 2002, Strzelczak et al. 2013, Lillibridge et al. 2013, Fu et al. 2014]. Distinct duplication ratios are found in archival and backup production storage. For instance, archival redundancy can reach a value of 79% [Quinlan and Dorward 2002, You and Karamanolis 2004, You et al. 2005], while backup redundancy goes up to 83% [Meister and Brinkmann 2009, Meyer and Bolosky 2011].

Deduplication in backup and archival systems was introduced by SIS [Bolosky et al. 2000] and Venti [Quinlan and Dorward 2002]. More specifically, SIS is an offline deduplication system for backing up Windows images that, can also be used as a remote install service. Stored files are scanned by a background process that shares duplicate files by creating links, which are accessed transparently by clients and point to unique files stored in a common storage. The number of references to each shared file is also kept as metadata on the common storage and, enables the garbage collection of unused files. A variant of CoW, named copy-on-close, is used for protecting updates to shared files. With this technique, the copy of modified file regions is only processed after the file is closed thus, reducing the granularity

and frequency of copy operations and, consequently, their overhead. With a distinct design and assumptions, an in-line deduplication Content-Addressable Storage (CAS) for immutable and non-erasable archival data is introduced by Venti. Unlike in traditional storage systems, data is stored and retrieved by its content instead of physical address, and fixed-size chunking is used instead of a content-aware partitioning, although it is possible to configure the system to read/write blocks with distinct sizes. Unique chunk signatures are kept in an on-disk full index for both systems. Since deduplication in SIS is performed in the background and at the whole-file granularity, the index is smaller and accessed less frequently, while aliasing is also performed outside the critical write path. On the other hand, Venti in-line timing requires querying the on-disk index for each write operation, presenting a considerable performance penalty for deduplication and storage writes throughput. This overhead penalty is alleviated by using a LRU cache, which exploits temporal locality, and disk stripping, that reduces disk seeks by allowing parallel lookups.

The index lookup bottleneck

With no temporal locality, Venti's performance is significantly affected because most index lookups must access the disk. This problem is known as the *index lookup bottleneck* and has been addressed by new indexing designs [Eshghi et al. 2007], by exploiting spatial locality [Zhu et al. 2008, Lillibridge et al. 2009, Guo and Efstathopoulos 2011, Shilane et al. 2012], and by using SSDs to store the index [Meister and Brinkmann 2010, Debnath et al. 2010].

Hash-based Directed Acyclic Graphs (HDAGs) were introduced as a first optimization for representing directory trees and their corresponding files by their content together with a compact index of chunk signatures. The HDAG structures efficiently compare distinct directories to eliminate duplicates among them while, the compact index representation can be kept in RAM, significantly boosting lookups. These optimizations were introduced in Jumbo Store, an in-line deduplication storage system designed for efficient incremental upload and storage of successive snapshots, which is also the first complete storage deduplication system to apply the TTTD algorithm [Eshghi et al. 2007].

Despite the reduction of the index size in Jumbo Store, the amount of RAM needed was still unacceptable for large storage volumes, thus limiting scalabil-

ity [Lillibridge et al. 2009]. This led to designs that maintain the full index on disk, similarly to Venti, while introducing optimizations to improve the throughput of lookup operations, as in the Data Domain File System (DDFS) [Zhu et al. 2008]. Firstly, a RAM-based Bloom filter is used for detecting if a signature is new to the on-disk index, thus avoiding disk lookups for signatures that do not exist. Then, spatial locality is explored instead of temporal locality. Namely, a Stream-Informed layout is used for packing chunks into larger containers that preserve the order of chunks in the backup stream. Then, when a specific chunk signature is looked up, all the other chunk signatures from that container are pre-fetched to a RAM cache. Due to the spatial locality, these signatures are expected to be accessed in the next operations, thus avoiding several disk operations. Although these optimizations also consume RAM, the memory footprint is significantly smaller to the one needed by Jumbo Store. These optimizations were also explored in Foundation, where a byte comparison operation for assessing if two chunks are duplicates was introduced to prevent hash collisions and, consequently, data corruption [Rhea et al. 2008]. Each comparison operation generates additional overhead, because full chunks must be read from disk.

In an incremental backup system, only modified streams are backed up and the content of these streams will change significantly in time. For instance, blocks A, B, C and D are stored in a contiguous fashion but, after some incremental backups, this sequence of blocks will be considerably distinct as some blocks were removed while other blocks were added, generating a new sequence with blocks A X C Y D. With the previous stream-informed layout approach, fewer duplicates will be found when this new sequence appears. A solution to this problem is suggested in the Block Locality Cache (BLC) mechanism that presents a prediction scheme for updating data locality information according to previous backups and, uses this information for pre-fetching into a LRU cache the blocks that will probably be accessed next, thus being less affected by the aging effect of incremental streams [Meister et al. 2013b].

Spatial locality can also be exploited by using a *sparse index* leading to increased performance and scalability at the expense of deduplication gain [Lillibridge et al. 2009], which works as follows. A backup stream is partitioned into segments, holding thousands of variable-sized chunks, which are the basic unit of storage and retrieval. Then a primary index, holding groups of similarity signa-

tures sampled from each stored segment, is used for finding the stored segments that most resemble the incoming one, and that are named champions. Identity signatures of chunks belonging to champion segments, which are stored on disk, are brought to memory and matched to the chunks of the incoming segment. This design maintains a smaller RAM footprint by only loading into memory a sub-set of signatures that will provide the higher deduplication gain. Moreover, as segments have a coarse-granularity, the primary index can also be kept in RAM for large data sets. Finally, by grouping contiguous chunks into segments and finding duplicate chunks among similar segments, spatial locality is also exploited to increase deduplication gain. Although this approach does no detect all deduplication opportunities across chunks from all segments, thus performing only partial deduplication, it allows to reduce chunk sizes and, consequently, increase duplicate detection while maintaining a small RAM footprint.

The previous design is optimized in Sungem by only keeping, for each group of similarity signatures at the sparse index, the signatures that are most effective in finding duplicates. [Simha et al. 2013]. Also, it is discussed a mechanism to garbage collect entries at the index that are no longer useful deduplication targets. These improvements allow further optimizing the size of the index.

Partial or sampled indexes present another solution for the index lookup bottleneck and RAM consumption issues by keeping in cache only a subset of signatures that are evicted when a certain threshold of RAM utilization is attained. As there is no full index on disk, deduplication gain is reduced, because only a few signatures are identified, which is alleviated by exploring spatial locality with the pre-fetching of signatures for contiguous chunks [Guo and Efstathopoulos 2011], as in DDFS. Sampled indexes are also explored in other systems that combine chunk and delta-based deduplication to achieve higher deduplication gain, as follows: a cache that takes advantage of spatial locality, and an on-disk full index are used for aliasing deduplication while, a partial index of similarity signatures is kept in RAM for delta-encoding chunks that were not eliminated by the previous method. Multiple levels of indirection caused by delta deduplication, when introduced on a system as the one presented by DDFS, however, have a substantial impact on restore throughput [Shilane et al. 2012].

Spatial and temporal locality may be lacking in some storage workloads, thus reducing significantly the efficiency of locality-based approaches. The index

lookup bottleneck can then be addressed by storing the index on SSDs, which increases significantly throughput over traditional hard disks and can scale to larger data sets than RAM indexes, as explained in Dedupv1 [Meister and Brinkmann 2010]. However, fine-grained index write and update operations are not a good fit for SSDs FLASH memory. This can be solved by organizing the index as a log, appending new entries sequentially, as in Chunkstash [Debnath et al. 2010]. In fact, the challenges for key-value stores on FLASH memory are widely researched and are present in a wider range of systems unrelated to deduplication [Anand et al. 2010, Debnath et al. 2011, Lim et al. 2011, Lu et al. 2012]. Moreover, both Dedupv1 and Chunkstash explore spatial locality as the original DDFS work does. Although these two systems can exploit spatial locality to reduce FLASH accesses, their designs are not dependent on locality to achieve efficient deduplication throughput and gain.

Distributed deduplication

Peer-to-peer deduplication, where backups are made cooperatively to remote nodes, was introduced in Pastiche [Cox et al. 2002]. More specifically, nodes backup their data into other remote nodes that are chosen by their network proximity and data similarity. As in-line deduplication is performed and each node has its own independent CAS, it is possible to send only the new chunks over the network to the nodes with the most resembling content, reducing both network bandwidth and used storage space. Moreover, convergent encryption derives keys from content instead of using each user's encryption key. This ensures privacy while forcing duplicate chunks to have the same cypher text for deduplication. While this technique leaks the knowledge that a particular cipher text, and thus plain text, already exists, an adversary with no knowledge of the plain text cannot deduce the key from the encrypted chunk [Douceur et al. 2002].

Since Pastiche only performs deduplication across a specific set of cluster nodes, duplicate data still exists across the cluster. This led to systems that focus on deduplication across the cluster as a whole. As a first system proposal, all cluster nodes can have access to a distributed data structure, exported as a centralized index, where unique chunk signatures are kept and mapped to specific nodes. Then, duplicate chunks can be routed to the same nodes and eliminated locally by using Microsoft SIS system, thus achieving exact cluster

deduplication. This design was proposed in Farsite, an in-line global deduplication system [Douceur et al. 2002]. Moreover, Farsite also proposes that some redundant chunks must be kept in order to ensure data reliability, as well as, the use of convergent encryption for protecting files from distinct users.

Moreover, DHTs can be used for indexing chunk signatures and routing requests deterministically to the correct nodes [Dubnicki et al. 2009, Wei et al. 2010]. In Hydrastor, DHTs are used for implementing a large-scale in-line CAS that applies compression to non-duplicated chunks to further increase space savings [Dubnicki et al. 2009]. Moreover, erasure codes and chunk replication, with a factor defined by the user, are used to ensure reliability. Although Hydrastor supports data deletion, this involves a complex algorithm for maintaining the references to shared blocks. A distinct system uses the DHT to distribute files by their content to specific nodes, and works as follows. Locally each file signature is compared and, if a duplicate file exists, it is shared. If a duplicate file does not exist, then file chunks are deterministically distributed over the nodes, being each node responsible for processing all signatures with a specific prefix. This deterministic mechanism ensures that chunks are deduplicated across remote nodes and is proposed in Mad2 [Wei et al. 2010]. Mad2 also proposes a novel metadata structure for preserving spatial locality, named hash bucket matrix, which is combined with a pre-fetching cache mechanism to reduce disk accesses to the index. Finally, a RAM-based Bloom filter, similar to the one proposed in DDFS, is also implemented to avoid looking up entries at the on-disk index that do not exist.

Global distributed deduplication may also use delta deduplication exclusively instead of traditional aliasing deduplication. By using the delta technique, the chunk size can be increased while not significantly affecting gain, thus allowing a scalable RAM index of similarity signatures where the most resembling chunks are found and delta-encoded against the new chunks. This novel approach was introduced by IBM Protect Tier system, presented as a gateway solution that intercepts data from backup stream generators and writes it into an external storage [Aronovich et al. 2009]. Although specific details are not presented, several gateways can be combined to perform deduplication over a common data repository, thus allowing global distributed deduplication. As a distinct approach, the Dedupv1 centralized design can be extended over a shared Storage Area Net-

work (SAN) device where several nodes have exclusive access to their own data partitions [Kaiser et al. 2012]. Nodes are seen as independent Dedupv1 nodes that export their own iSCSI interface, partition data, compute hashes, and map chunk requests to the correct nodes. Each node is responsible for storing, on its own SSD partition, a range of the signatures index, and, in some cases, signature lookups must be done through the network. This leads to additional network bandwidth requirements and to overhead in I/O requests that are minimized with a write-back cache and write-ahead logs. Partition owners are only changed in case of load balancing or failure recovery.

Local deduplication in distributed infrastructures

Global indexes and the consequent bottlenecks can be avoided by parallel local deduplication, increasing deduplication throughput at the expense of gain [You et al. 2005, Bhagwat et al. 2009, Dong et al. 2011, Fu et al. 2012, Xia et al. 2011]. Deep Store introduces a large-scale archival storage where stored files are routed, according to their signatures, to specific cluster nodes with independent processor, memory, and disk [You et al. 2005]. In each storage node, a framework named PRESIDO divides files into variable-sized chunks, and uses both aliasing and delta deduplication for locally eliminating duplicates. Another work was then proposed to route files to specific nodes according to their similarity as follows: When a file is backed up, it is sent to a backup node, for example, the one with the less load, and the file similarity and identity signatures are calculated. Then, based on its similarity signature, the file is routed to the correct backup node. This was introduced by Extreme Binning system that also proposed a two tier index design for local deduplication [Bhagwat et al. 2009]. The first index is stored in RAM and indexes a whole file similarity and identity signature, while the second index is stored on disk and indexes identity signatures. When the file is routed to a node, the primary index is searched for a similar file, and, if found, the file identity signature is compared to see if the whole file can be deduplicated. If not, chunk identity signatures from the most similar file are brought to memory and deduplicated against the chunks of the new file. This is another implementation of a *sparse index* that reduces RAM footprint at the expense of both global and local deduplication gain. Moreover, due to the system's distributed design, several backup files can be deduplicated in parallel.

Extreme Binning relies on file similarity for achieving high deduplication ratios, while other solutions propose to explore spatial locality in each node. Namely, local deduplication exploring spatial locality and Bloom filters, as in DDFS, can be combined with stateless or stateful routing mechanisms at super-chunk granularity [Dong et al. 2011]. In the stateless algorithm, chunks are grouped into super-chunks and are then routed to the right node by their content similarity. On the other hand, the stateful routing algorithm uses information about the location of chunks, and uses a Bloom filter to count the number of times each chunk signature in a super-chunk is stored in a given node. Then, if the node with the most chunks in common is not overloaded, thus preserving load balancing, the super-chunk is routed to that node. Otherwise, the second best node in terms of space savings is chosen. The stateful routing approach chooses the best nodes to store the super-chunks in terms of space savings, while the stateless routing approach does not need knowledge of stored chunks, and uses a best-effort routing scheme with low computational requirements. The stateless routing scheme can achieve 80% of the exact deduplication values; however, this rate may drop significantly for some large data sets where the stateful approach can maintain the 80%. When compared to Extreme Binning, these approaches use a smaller routing granularity, and are not highly dependent on inter-file similarity to achieve good deduplication ratios.

The computational and memory overhead of stateful routing at super-chunk granularity can be minimized by using a probabilistic similarity metric that identifies the nodes holding the most chunks in common with the super-chunk being stored [Frey et al. 2012]. This metric is based on a probabilistic set intersection. It does not require exact knowledge about chunks stored at each node to define the routing strategy that yields the most deduplication gain, but has an implicit estimation error. The estimation error can then be reduced by preferentially storing super-chunks in nodes with high metric similarities that were also used to store previous super-chunks belonging to the same backup file, thus leveraging spatial locality.

As some workloads may exhibit poor similarity, others may have poor locality, leading to proposals that explore both simultaneously and, in this way, compensate the lack of one with the other [Xia et al. 2011, Fu et al. 2012]. More specifically, one of the approaches divides files into content-defined chunks, group-

ing small strongly correlated files into a single segment while dividing large files into several segments. Then a local similarity index is used for finding similar segments to deduplicate against. Similarity index size can be reduced and scalability improved by grouping small files into segments. Then, segments are grouped into locality preserving blocks that are the basic caching unit, allowing a similar approach to DDFS to exploit spatial locality. This algorithm is presented in Silo which does not present a detailed description of the routing algorithm, although aimed at a distributed infrastructure [Xia et al. 2011]. In fact, details are then explained in Σ-Dedup where similarity-based stateful routing at super-chunk granularity is presented, while local deduplication follows an identical approach to Silo [Fu et al. 2012].

Other cluster approaches focus on off-line deduplication for splitting data partitioning and signature calculations from the global index lookup and update operations, parallelizing both and batching accesses to the index, as in DEBAR [Yang et al. 2010b] and Chunkfarm [Yang et al. 2010a]. In both systems, the deduplication algorithm is divided in two phases. In a first phase, data is partitioned into chunks, content signatures are calculated and a RAM cache signature is used to eliminate some of the duplicate chunks locally, without requiring a global index mechanism. In addition this step can be processed in parallel by several nodes, and requires writing all chunks and their signatures to disk, which increases storage consumption. In DEBAR chunk signatures are written to a disk log sorted by buckets to enable batch processing, while in Chunkfarm all signatures that need to be looked up in the index are collected and sent to a metadata server that then uses hash join algorithms for performing both look up and update in batch [Shapiro 1986]. Then, in a second phase, signatures are processed asynchronously, and both index lookup and insertion operations are batched, avoiding fine-grained random I/O operations. In DEBAR the index is stored in a centralized service, while in Chunkfarm it is not specified if the metadata server is distributed or how that can be accomplished. As a matter of fact, a centralized index may pose as a single point of failure and scalability bottleneck in large clusters.

File systems for archival and backup

Finally, archival and backup CAS systems can be extended with file system semantics at the expense of moderate I/O performance. A distributed file system built on top of HYDRAstor, which performs in-line global deduplication at content-defined chunk granularity, is proposed in Hydra File System (HydraFS) [Ungureanu et al. 2010]. HydraFS optimizations allow the system to perform well for stream I/O operations (sequential read and writes), while single random block write and read requests, common in most file systems, are supported, but are inefficient due to the FUSE abstraction layer and the backup-oriented implementation of HYDRAstor. Similar negative impact in I/O performance was also observed when building a file system on top of Venti [Liguori and Van Hensbergen 2008].

Similarly, the open-source centralized Less File System (LessFS) is designed mainly for backup purposes but can also be used for storing VM images and primary data with moderate performance requirements [Lessfs 2014]. In-line deduplication is performed with a fixed-size block granularity while FUSE is used for implementing file system semantics.

Archival and backup deduplication has a large pool of research work and is still being actively researched, mainly on the optimization of both deduplication and I/O throughputs. Nevertheless, most systems in this category assume that data is immutable and I/O throughput is preferred over I/O latency. These assumptions do not hold true in primary storage systems, and explain why building file systems over backup and archival deduplication systems have poor performance for random I/O operations.

2.3.2 Primary storage

With cloud computing there has been a growing interest in live volume deduplication. Primary storage deduplication invalidates some of the assumptions made by archival and backup deduplication systems. Applications using primary storage have strict performance requirements for I/O latency, meaning that a deduplication system must aim at the same I/O performance as a raw storage system without deduplication. Data is no longer write-once and is expected to be modified frequently, thus requiring, for some systems, CoW mechanisms for

preventing updates on aliased data, and efficient reference management mechanisms for tracking chunk references that will change frequently [Hong and Long 2004]. In primary storage, the percentage of duplicate data is usually lower than the one found for backup storage, dropping from 83% to 68% [Meyer and Bolosky 2011]. Other studies show that higher deduplication ratios can be found for general purpose primary VM volumes in large clusters, where 80% of space reduction is possible [Jin and Miller 2009, Clements et al. 2009].

Off-line deduplication

The strict latency requirements of primary storage applications shift the focus to off-line deduplication systems, aiming at lower storage latency by reducing overhead in the write path. Distributed off-line deduplication for a SAN file system was introduced in the Duplicate Data Elimination (DDE) system [Hong and Long 2004], and works as follows. Deduplication is performed in the background outside the critical write path, which, besides reducing latency, allows to temporarily disable deduplication when the system has a higher load. DDE is implemented over the distributed IBM Storage Tank system that avoids cross-host communication in the data path. Clients calculate the signatures for fixed-size chunks written to the storage, and send these signatures to a server that shares duplicate chunks asynchronously. The index of unique signatures is stored on the SAN, and has two versions. One is structured to favor sequential I/O and spatial locality. The other is indexed by partial bits for facilitating random searches. CoW is used to ensure that processed chunks are not changed and the signatures are always valid, as it would otherwise lead to data corruption. Reference counting information is stored in separate metadata structure that is needed for the garbage collection of unused blocks. Deduplication is restricted to within a specific file set that is a sub-set of the global file system. This policy allows distinct file sets for applications with different performance assumptions, as some may not tolerate the performance penalty introduced by deduplication.

A proposal for distributing the centralized metadata server in DDE, which is solely responsible for detecting and coalescing duplicates, and presents a single point of failure, was then introduced in DeDe [Clements et al. 2009]. Namely, an off-line distributed deduplication algorithm for VM volumes on top of VMWares's Virtual Machine File System (VMFS) is described, and uses an index structure,

stored at the cluster file system that is accessible to all nodes and protected by a locking mechanism. This allows efficient batch lookups and updates, while index sharding across multiple nodes enables the design to scale out. Moreover, VMFS simplifies deduplication as it already has explicit block aliasing, CoW and reference management operations, which are not commonly exposed in most cluster file systems, and are used to implement the atomic share function that verifies the content of two fixed-size blocks and replaces them with one CoW block if they match. However, there are alignment issues between the block size used in VMFS and DeDe, implying additional translation metadata and a consequent impact on performance. Additionally, the storage impact of CoW operations is also significant and therefore the DeDe algorithm is intended to run in periods of low I/O load.

A mechanism for reducing the frequency and, consequently, the overhead of CoW operations was proposed in Microsoft Windows Server 2012 centralized off-line deduplication system [El-Shimi et al. 2012]. Namely, only files that meet a certain policy, for instance, file age greater than a certain threshold, are deduplicated. This reduces the probability of rewriting highly volatile files and, consequently, invoking CoW operations. Moreover, it was proposed that files are grouped according to their file extensions, for example, and then each group can be deduplicated individually. This allows loading the index of each group into RAM and performing efficient lookup and update operations. A reconciliation algorithm for eliminating duplicate data between several groups can be used to achieve exact deduplication. Mechanisms for exploring spatial locality similar to DDFS, and to store the index as a log structured file and, possibly, on SSD, as in Chunkstash, were also proposed.

In-line deduplication

Performing deduplication in an in-line fashion requires costly lookups in the write path, which can impose a significant overhead in storage I/O latency. On the other hand, off-line deduplication may introduce additional reads from the storage, requires more storage space, raises concurrency issues and increases the complexity of the deduplication systems. These problems motivated the emergence of in-line deduplication systems for primary storage that introduce optimizations for reducing significantly the I/O latency.

Liquid deduplication file system uses a client-side private cache for holding VMs blocks that are being frequently modified [Xun et al. 2014]. When a flush is issued by the VM, all blocks are evicted from the private cache, have their signatures calculated and are stored in remote data servers organized as a DHT. Deduplication is still done in an in-line fashion since blocks are deduplicated before being stored in the data servers, however, some of the computational overhead of deduplication is avoided until explicit flush operations are called. Although this optimization reduces some of the storage overhead, Liquid deduplication is still only recommended for VM content that is modified sporadically. As a distinct contribution, when a fresh copy of a previously stored VM image is launched, a peer-to-peer approach is used for efficiently retrieving blocks from both the servers holding data and the clients caches, thus reducing the load in data servers and allowing a more scalable design.

In-line global deduplication is also supported by the Opendedup open-source system [Opendedup 2014]. As a novelty, chunk boundaries can be defined with either fixed or variable sized granularities. The index with chunk signatures is sharded across several servers while lookups to the index are sent in batch and multicasted to all servers. Writes to the storage only proceed when all the responses for the same batch are received or when a pre-defined timeout occurs. This approach favors throughput over latency and, this way, has a prohibitive overhead for the latency requirements of most primary storage systems. Also, in order to increase the performance of deduplication, Opendedup requires a significant amount of CPU and RAM resources.

In the Z File System (ZFS), optimizing for reduced latency also means fully loading the index in RAM. It is still possible to cache only a subset of the index in memory, but disk lookups have a significant impact on deduplication and, consequently, storage I/O performance [OpenSolaris 2014]. As an alternative to maintaining the full index in memory, a multi-layer Bloom filter for speeding lookups at the index and reducing the impact in I/O latency was proposed in the Deduplication Block-Device (DBLK) [Tsuchiya and Watanabe 2011]. The first Bloom filter layer detects if a certain signature might be present at the on-disk index without requiring a disk access. Then, if the signature is likely to be already indexed, the second Bloom filter layer narrows the location of the signature, thus speeding its retrieval. Another solution for increasing deduplica-

tion throughput and reducing I/O latency is to use a Bloom filter and explore spatial locality by preserving the disk layout, and then pre-fetching contiguous chunk signatures to cache as in DDFS. These two improvements were presented for an in-line centralized deduplication system along with a novel fault-tolerant journaling mechanism for tracking system transactions, and recovering data and corresponding signatures in failure scenarios [Ng et al. 2011].

Spatial and temporal locality were also explored in order to minimize both read and write latency. Fragmentation and, consequently, read I/O latency is reduced by deduplicating groups of contiguous blocks and storing them together, so as to preserve their layout. Spatial locality makes it likely that duplicates can still be found in such groups. On the other hand, temporal locality is exploited with a LRU cache that stores a partial list of index entries. The on-disk index is organized in buckets and is stored as a red-black tree that reduces the number of pointers and, consequently, the storage space of traditional hash tables. This space reduction allows to increase the number of buckets to improve hash lookup speed. These optimizations were presented in Netapp's idedup in-line deduplication system where overhead is minimized while maintaining considerable deduplication gain [Srinivasan et al. 2012].

As a distinct approach, the algorithm to pre-fetch chunk signatures into an in-memory cache can follow statistical information derived from storage access patterns, as discussed in HANDS in-line deduplication system [Wildani et al. 2013]. Namely, temporal and spatial locality metrics can be extracted from storage I/O traces in order to improve cache efficiency. Moreover, this mechanism can be extended to consider other variables that are important for grouping chunks with a high probability of being accessed together. The paper shows that it is possible to reduce the space that a full memory index cache would require by 99% while still achieving a cache hit ratio between 30% and 90%.

Another option is discussed in the Performance-Oriented I/O Deduplication (POD) system where RAM space is dynamically adjusted for the read and index caches, according to the current storage access pattern [Mao et al. 2014b]. This idea follows the assumption that primary storage workloads change frequently between read and write bursts. This way, when a write burst occurs, most of the reserved RAM space can be used for holding index signatures in an LRU fashion, thus increasing deduplication throughput. On the other hand, when read

bursts occur, the reserved RAM space holds mainly the content of the most popular blocks. As another contribution, fragmentation is alleviated by performing deduplication only for segments that have a higher number of duplicates than a pre-defined threshold. This ensures that spatial locality is maintained while, fragmentation in stream read requests is reduced.

2.3.3 Random-access memory

Deduplication is also used for applications and VMs that do not benefit from classic RAM sharing provided by process forks or shared libraries. The extra memory, obtained by eliminating redundant memory pages, can be useful for launching additional applications or VMs, or can be provided to the existing ones. More precisely, it is possible to reduce memory consumption by 33% by using RAM deduplication across a group of virtualized hosts [Waldspurger 2002]. Current RAM deduplication systems have different assumptions from backup and primary storage systems. For instance, since duplicate content is only found across applications or VMs running in the same server, deduplication is always performed locally in a centralized fashion. Moreover, memory pages are highly volatile and change frequently, requiring efficient CoW and reference management mechanisms.

The Disco Virtual Machine Monitor (VMM) pioneered in-line transparent page sharing for memory pages, such as code or read-only data, across different VMs [Bugnion et al. 1997]. Memory pages loaded from a special CoW disk are shared, thus eliminating the need of content-aware deduplication. Disco intercepts read requests from CoW disks and finds if the page for that request is already in-memory, thus avoiding the disk access and sharing pages between different VMs. Duplicate pages are mapped into a single memory page at the host's main memory. Since content-aware deduplication is not used, an index of signatures is not necessary.

Non-intrusive scan deduplication

Disco requires several modifications to the guest OS, which led to content-aware approaches, such as VMware ESX, that perform non-intrusive memory scans to find duplicates [Waldspurger 2002]. Namely, memory can be shared in an off-line

fashion by periodically scanning all memory pages, calculating their signatures and looking for duplicate pages in an index of unique page signatures. Then, shared pages must be marked as CoW for preventing updates that may cause data corruption.

As sharing highly volatile pages increases significantly the amount of CoW operations and, consequently, the overhead, a double tree index was proposed in Linux Kernel Same-page Merging (KSM) system to detect such pages and avoid sharing them [Arcangeli et al. 2009]. Memory regions to be deduplicated are given as a parameter and are periodically scanned for finding and merging duplicate pages among applications and KVM VMs. A stable tree metadata structure is used for registering shared pages that are write protected and scanned first for duplicate detection. An unstable tree is used for keeping unique pages that are not write protected and scanned only if no duplicates were found at the stable tree. This mechanism detects what pages are less susceptible to be rewritten and are better sharing candidates. KSM index trees do not keep hash signatures of the pages but the actual page content. This way, duplicate pages are found with *memcmp()* operation instead of comparing content signatures.

KSM design was then updated in the Singleton system to optimize the dual page caching mechanism in virtualized hosts and further increase space savings [Sharma and Kulkarni 2012]. In virtualized hosts, each VM has its own page cache as a first-level cache mechanism. When this mechanism fails to service an I/O request, a second-level hypervisor cache, shared by all VMs, is used to look up the requested page. Having two caches with duplicate pages occupies unnecessary memory space that can be spared with deduplication. The original KSM design was modified by replacing the search trees with hash tables that index the content signatures of pages instead of their actual content, eliminating the CPU usage of *memcmp()* operations. Hypervisor cache page signatures are calculated periodically, checked against the indexes of pages from VMs' local caches, and dropped from the hypervisor cache if duplicates are found, thus improving memory usage.

Memory deduplication space savings were further improved by introducing delta-encoding and compression, besides the usual aliasing technique present in previous systems. First, duplicate pages are shared by looking for duplicate signatures in an identity index, and then a similarity index is used for looking

up for similar pages and delta-encoding them. Pages that were not duplicated, and are not expected to be rewritten in a near future, are compressed to improve further space savings, as introduced in Difference Engine [Gupta et al. 2010].

In order to enhance the efficiency of memory scans and index searches, two distinct optimizations were proposed. In the Cross Layer I/O-based Hints (XLH) system, hints about recently modified memory pages are provided to memory scan approaches, such as the ones used in KSM and ESX [Miller et al. 2013]. These hints provide information about what pages are better share candidates and avoid a linear or random search across all memory pages while, allowing to detect short-lived sharing opportunities. As a distinct optimizations, in the Classification-based Memory Deduplication (CMD) system, pages are grouped into distinct trees based on their access characteristics [Chen et al. 2014]. Pages that belong to the same tree have high probability of having the same content. This classification allows defining smaller trees than in KSM, thus reducing the search space for finding duplicate pages while having a minimal penalty in deduplication gain.

Intrusive deduplication

Other memory deduplication systems intercept disk read operations, as in Disco, but use content-aware page sharing. Intercepting requests has the advantage of detecting short-lived sharing opportunities that in scan approaches, as the one presented by VMware ESX Server, may be discarded [Milos et al. 2009]. One of the first systems to introduce memory deduplication aimed at solving problems of dynamic libraries, such as Dependency Hell and Global Offset Table (GOT) overwrite attack, by replacing dynamic libraries with static libraries, thus creating self-contained applications [Suzaki et al. 2010]. This replacement leads to extra memory usage, because static libraries do not share any data. Slinky introduces in-line deduplication as a solution for mitigating these additional memory costs [Collberg et al. 2005]. Pages are intercepted when they are being loaded into memory, and their signatures are looked up in an index to find duplicates. Content signatures are calculated before loading the pages into memory, and CoW is not necessary because only read-only pages are shared.

Unlike Slinky, some in-band systems must support mutable pages and implement CoW mechanisms. This is addressed in Satori, an in-line deduplication system that modifies virtual disk implementations to intercept read requests made

by VMs [Milos et al. 2009]. Namely, an image of the page cache is built by observing the content of disk reads, and when a block read request is intercepted, its content is hashed and compared with the other indexed page digests to see if the page is duplicate. If the page is duplicated, then it is shared and marked as CoW. For each VM, a Repayment FIFO queue holds a list of volatile pages that the operating system is willing to relinquish at any time, and which are used to efficiently provide free pages for CoW operations.

Besides space savings, memory deduplication can also be used to implement in hardware a memory abstraction of protected shareable segments with snapshot and atomic update operations, as explained in the Hierarchical Immutable Content Addressable Memory Processor (HICAMP) system [Cheriton et al. 2012]. This abstraction allows fault-tolerant and safe concurrent access to data shared by multiple threads while reducing the overhead of common inter process communication approaches. HICAMP system is implemented as a CAS where the memory is divided into fixed-size chunks, referred to as lines, with unique and immutable content ensured by the in-line deduplication algorithm. Memory space is divided into hash buckets with content lookup operations at line granularity. This way, when a new line is written, it is checked to see if a line with that content already exists and the new line can be deduplicated, or if the new line has new content and must be inserted in the memory space. Reference counting is done by the hardware and when the number of references to a line is zero, the line is garbage collected. This design allows to implement memory segments that are logical variable-length contiguous regions of memory and are represented as direct acyclic graphs pointing to specific lines protected with CoW. With this representation, snapshots can be performed efficiently and threads can run with snapshot-isolation guarantees.

Another work that does not present an actual deduplication system but focuses on useful optimizations for RAM deduplication is the new hyper-call for the XEN hypervisor, proposed to minimize the performance impact of hashing pages [Pan et al. 2011]. Moreover, the benefits of memory deduplication in virtualized clusters can be maximized by placing VMs with similar content at the same host, as described in [Wood et al. 2009]. A memory fingerprinting technique that presents a compact representation of the memory content allows to identify VMs with high page sharing potential and migrate them to the same host to

achieve higher space savings.

2.3.4 Solid state drives

Deduplication has also been used within SSDs, known for radically improving the performance of random I/O operations. In SSDs, I/O operations are processed at fixed page sizes, usually 4 KiB, but unlike in traditional hard disks it is not possible to delete data at the same granularity. Pages are grouped into erasure blocks, usually with 64 to 128 pages, and the deletion is done at erasure block granularity. Moreover, updates cannot be done in-place, so modified blocks must be appended to an erasure block with free space. Then, only when all pages in an erasure block are unused, they can be erased and reclaimed by a garbage collection mechanism. The number of erase operations for each block, however, is limited in the range of 10,000 to 1,000,000 operations, thus limiting the drive's lifespan and being one of the major issues of this technology [Chen et al. 2011].

Typical SSD designs include the following components. A FLASH Transaction Layer (FTL) is implemented in the SSD controller and emulates the behavior of a traditional hard drive by exporting an array of logical blocks to the host. In this layer, an *indirect mapping* table is kept for mapping logical to physical addresses. A log-like write mechanism is used for writing pages, and each in-place update to a logical page only invalidates the previously occupied physical page, appending the new physical page to a free erasure block and updating the corresponding entry on the indirect mapping table. A garbage collection mechanism is launched periodically to recycle unused physical pages, consolidate the valid pages into a new erasure block and clean unused erasure blocks. Wear-leveling mechanisms are used to shuffle the hot and cold blocks in order to balance the number of writes and deletions in erasure blocks, thus increasing the lifespan of the SSD. A certain amount of over-provisioned spare space usually exists, which is not usable by the host, and is used by the garbage collection and wear-leveling mechanisms.

Deduplication gain observed in SSDs is also lower than in primary and backup storage. A gain of 56% is possible, however, it is the best case scenario, while for most workloads only up to 30% is achievable [Chen et al. 2011, Gupta et al. 2011, Kim et al. 2012]. On the other hand, finding duplicate data in SSDs has advantages other than the space saving benefits. In fact, if in-line deduplication is performed it is possible to reduce the number of writes to the storage and

increase the device lifespan. Moreover, the space saving benefits not only allow users to store more data, but also provide additional space for wear-leveling and garbage collection mechanisms. As SSDs already have a mapping table for translating logical to physical addresses, the deduplication design can take advantage of it for sharing identical pages with low overhead. Since there is also a periodical garbage collection mechanism, it is possible to extend it to perform reference management and garbage collection of shared pages. Deduplication in the SSD device has also some disadvantages. These devices come already with an internal Dynamic Random-Access Memory (DRAM) memory that can be used to store the deduplication index, however, this DRAM has limited space that does not allow to store a complete index of page signatures. SSDs also come with a limited processing capability, and calculating hash signatures may impose significant overhead. These advantages and drawbacks are further discussed while we describe existing SSD deduplication systems.

A pioneer CAS for SSDs that performs best effort in-line deduplication was presented by the Content-Aware Flash Translation Layer (CAFTL) system [Chen et al. 2011]. The indirect mapping table of the SSD device is used for implementing the I/O translation mechanism for the deduplication engine. In-line deduplication is performed with a best effort approach while the SSD performance is not significantly affected. If the load reaches a certain threshold, in-line deduplication is turned off, and stored duplicate pages are then shared with an off-line deduplication algorithm that runs in the same periods as the garbage collector. The off-line approach spares storage space, but does not avoid write requests to the storage. Fixed-size chunks are used and their hash signatures are stored in a DRAM partial index that only contains the most referenced signatures. An additional metadata table is used for mapping the references to each shared page and performing reference management. All these metadata structures are stored in the SSD, because the DRAM is not persistent over reboots or power failures. However, metadata updates are buffered and flushed only when the buffer is full, leading to metadata loss when a power failure occurs. This can be solved by using a capacitor for the DRAM. Since hash calculations are heavy for the SSD built-in processor, a content-based sampling algorithm that explores data spatial locality is presented, allowing to check the probability of a set of pages being duplicates. Then, hash signatures are only calculated for pages with high probabilities that

will benefit the most from deduplication.

The hash computation overhead is also avoided in the Content-Addressable Solid State Drive (CA-SSD) system with a built-in hardware hashing unit [Gupta et al. 2011]. The mapping structures and the partial index, similar to the ones presented in CAFTL, are stored in a fast persistent storage. This persistent storage must not be the SSD device itself in order to increase SSD performance and space savings. The CA-SSD proposal also differs from CAFTL, because only in-line deduplication is performed. Finally, it was observed that temporal locality was present in the studied workloads, which allowed to implement the partial index of chunk signatures as an LRU cache.

Temporal locality was then further researched in subsequent work by proposing a sampling-based filter for written pages that are still in the DRAM buffer and were not flushed to the SSD yet. This sampling mechanism detects what page contents are being written more than once and will benefit more from deduplication. This way, and since the hash calculation and deduplication can introduce significant overhead in the SSD processor, only the pages that probably will achieve the most benefits from being shared are actually processed, thus reducing deduplication overhead. Moreover, this work implemented the first real prototype and evaluated it without resorting to simulation [Kim et al. 2012].

SSD deduplication is still emerging and raises interesting challenges that are not present in other storage environments. First, the limitations of DRAM space and computational power raises the need for new designs for metadata structures and hashing algorithms. SSD partial indexes decrease RAM requirements, but also detect less duplicates and depend highly on locality assumptions. On the other hand, data fragmentation is not an issue, as in other storage environments, since random read requests are efficient in SSD devices. Regarding the deduplication overhead in I/O requests, the system described previously shows that for significant duplication ratios (more than 5%), the write performance can even be increased, while read performance has no significant overhead [Kim et al. 2012]. These values consider that efficient hashing mechanisms are being used instead of the common one presented in most SSDs.

Table 2.1: Classification of deduplication systems for all storage environments.

	Granularity	Locality	Timing	Indexing	Technique	Scope
SIS	W	N	O	F	A	C
Farsite	W	N	O	F	A	G
LessFS	F	N	I	F	A	C
Venti, [Liguori and Van Hensbergen 2008]	F	T	I	F	A	C
Foundation	F	T	I	F	A	C
[Guo and Efstathopoulos 2011]	F	S	I	P	A	C
JumboStore	V	N	I	F	A	C
Debar, ChunkFarm	V	N	O	F	A	G
Hydrastor, HydraFS, [Kaiser et al. 2012]	V	N	I	F	A	G
Pastiche	V	N	I	F	A	L
DDFS, Dedupv1, Chunkstash, [Meister et al. 2013b]	V	S	I	F	A	C
[Dong et al. 2011], Silo, Σ-Dedup	V	S	I	F	A	L
[Lillibridge et al. 2009], Sungem	V	S	I	S	A	C
Mad2	WV	S	I	F	A	G
ExtremeBinning	WV	N	I	S	A	L
IBM ProtectTier	F	N	I	F	D	G
[Shilane et al. 2012]	V	S	I	FP	AD	C
DeepStore	V	N	I	F	AD	L
ZFS, DBLK	F	N	I	F	A	C
DeDe	F	N	O	F	A	G
DDE	F	S	O	F	A	G
Liquid	F	N	I	F	A	G
[Ng et al. 2011]	F	S	I	F	A	C
iDedup, HANDS, POD	F	TS	I	F	A	C
Microsoft Windows Server	V	S	O	F	A	C
Opendedup	VF	N	I	F	A	G
Disco	F	N	I[2]	n/a[1]	A	C
Slinky, Satori, HICAMP	F	N	I[2]	F	A	C
VMware ESX, KSM, Singleton, XLH	F	N	O	F	A	C
CMD	F	N	O	S	A	C
Difference Engine	F	N	O	F	AD	C
CAFTL	F	S	IO	P	A	C
CA-SSD, [Kim et al. 2012]	F	T	I	P	A	C

Granularity: (W)hole file; (V)ariable; (F)ixed. Locality: (N)one; (S)patial; (T)emporal. Timing: (I)n-line; (O)ff-line. Indexing: (F)ull; (P)artial; (S)parse. Technique: (A)liasing; (D)elta. Scope: (C)entralized; (G)lobal; (L)ocal.

1. Disco does not require an index since it does not perform content-aware deduplication.

2. Disco and Satori perform in-line deduplication, because I/O read requests to the persistent storage are intercepted and redirected, if possible, to duplicate shared memory pages before actually loading the pages from the storage. Slinky performs in-line deduplication, because pages loaded from static libraries are shared before actually being loaded to memory.

2.4 Discussion

Table 2.1 classifies deduplication storage systems according to the taxonomy described in Section 2.2 and groups these systems by storage environment. This table highlights the relevance of each design option, characterizes the design space for each storage type, and points out unexplored designs that should be researched.

Aliasing deduplication is used in all storage types, while delta deduplication by itself is used only in backup systems. In both backup storage and RAM, aliasing and delta deduplication are combined for increasing space savings. Backup storage systems use several chunk granularities, being variable-sized chunks the most common one, while fixed-sized chunks are preferred in all other storage environments. Several indexing designs are used in backup deduplication, including even combinations of full and partial indexes [Shilane et al. 2012]. In primary and RAM deduplication systems, except for Disco and CMD systems [Bugnion et al. 1997, Chen et al. 2014], full indexes are always used. In contrast, SSD deduplication uses only partial indexes due to DRAM space restrictions. Locality is only explored in backup, primary and SSD deduplication systems. In RAM and SSD storage, deduplication is performed only in a centralized fashion, while in backup deduplication, local and global distributed approaches are also available. On the other hand, in primary deduplication all distributed systems perform global deduplication. Finally, all storage types have systems using off-line and in-line deduplication, while in SSD deduplication it is possible to combine both.

Most archival and backup systems assume that stored data has a write-once policy and, in some archival systems, this data cannot be deleted at all. Some of these systems, however, can be used as back end for implementing file system syntax and, consequently, support data updating with limited performance for random I/O operations [Nath et al. 2006, Ungureanu et al. 2010]. Since deletion and update operations are expected to be less frequent than in other storage environments, reference management and garbage collection mechanisms are also active for shorter periods and their overall overhead is reduced. Moreover, in these deduplication systems throughput is preferred over latency and, as most deduplication systems perform in-line deduplication, I/O latency is significantly increased.

These assumptions are not valid in primary storage, RAM, and SSD where I/O

latency overhead must be negligible even if deduplication throughput is reduced. Thus, data is updated in place, requiring a CoW mechanism to protect updates on shared data and potential data corruption. Although data is not updated in place, in SSDs it is necessary to ensure that shared blocks are not erased and collected by the garbage collector while still being referenced. In RAM deduplication, most pages are highly volatile, thus changing more often than in other storage environments, and increasing CoW and reference management overhead. Finally, existing SSD deduplication systems present deduplication embedded in the SSD device, which significantly restricts the computational power and RAM space available.

Most strikingly, in-line global deduplication can still be further explored in distributed primary storage systems. Liquid and Opendedup are the only two systems that perform in-line global deduplication [Xun et al. 2014]. However, both approaches have a significative impact in storage requests when performing global deduplication, which is not acceptable for primary volumes with strict latency requirements. In fact, even with the proposed optimizations, current 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. Moreover, there are few proposals on distributed deduplication, and several contributions and combinations of techniques are possible. For instance, all primary deduplication systems use full index approaches while partial and sparse indexing mechanisms can still be explored.

It is also important to notice that primary deduplication must be evaluated differently from backup deduplication. In archival and backup deduplication systems, static VM images, containing the operating system and application binaries, are widely used to assess the system's performance as many of these systems are designed to store such content [Guo and Efstathopoulos 2011]. In primary storage deduplication, dynamic traces should be used to accurately simulate real workloads. In fact, using evaluation workloads suited for specific environments is extremely important to validate distinct assumptions. For example, in a scenario where VM images with a common ancestor are backed up integrally, spatial locality can be explored efficiently. In a primary storage scenario where specific blocks of VM images are being changed independently with no correlation, for instance, in a Virtual Desktop Infrastructure (VDI) where each user has its own

workstation mapped to an independent VM, spatial locality will be significantly reduced.

Mechanisms that replay dynamic traces or resort to synthetic I/O benchmarks can be used to simulate primary storage environments if real workloads are not available. Traditional I/O benchmarks do not simulate realistic content distributions for their tests, which limits their realism. As only a few proposals address this challenge [Tarasov et al. 2012], novel contributions in this topic are possible. Also, workloads for RAM and SSD storage must be considered.

To conclude, current primary storage systems with strict latency requirements either have centralized components that limit scalability, or find duplicates in a centralized fashion or in off-peak periods to ensure a minimal penalty in the performance of storage requests [Hong and Long 2004, Clements et al. 2009, Srinivasan et al. 2012, Xun et al. 2014]. In a cloud computing primary storage, the solutions must be fully-decentralized in order to scale and cannot rely on off-peak periods that may be scarce or even non-existent. As another issue, there are still few benchmark tools suited for accurately simulating the duplicate content and dynamism expected in a primary storage environment [Tarasov et al. 2012].

Chapter 3

Benchmarking storage deduplication systems

Deduplication is now being applied to distinct storage environments with specific workloads and characteristics, which justifies the need of having proper benchmarking tools for evaluating and comparing existing deduplication systems. Previous work analyzed 120 datasets used in deduplication studies and concluded that most of them are either private or non-reproducible [Tarasov et al. 2012]. Therefore, other researchers and enterprises cannot use them to evaluate their own deduplication systems. On the other hand, the few datasets that are publicly available contain mainly static operating system distributions and VM images, while most have less than 1 GiB of data.

Using static workloads for evaluating primary deduplication systems, as the one we present in Chapter 4, is not realistic. Primary storage data is dynamic and some parts of it are frequently rewritten, requiring a CoW mechanism that adds significant overhead in storage requests and must be accounted in the evaluation [Clements et al. 2009]. This dynamism can be simulated with traditional micro-benchmarks. However, most of them do not use a realistic distribution for generating duplicates and, in most cases, the data written in each benchmark operation either has the same content or, it has random content with no duplicates at all [Coker 2014, 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].

Moreover, distinct storage types *i.e.*, archival, backup, primary, RAM and SSD storage, have specific distributions of duplicates and, simulating these differences is key for having a more realistic evaluation. For instance, distinct duplicate distributions will have a specific impact in the reference management and CoW operations that, as explained in the previous chapter, do not affect significantly the performance of archival deduplication but affect the performance of primary storage deduplication.

Some benchmarks generate duplicates by defining a percentage of duplicate content over the written records, or the entropy of generated content [Norcott 2014, Al-Rfou et al. 2010]. However, these methods are only able to generate simplistic distributions that do not specify, for example, distinct number of duplicates per unique data as found in a real storage system. To our knowledge, there is only one file system benchmark addressing the previous limitations [Tarasov et al. 2012]. Nevertheless, as we explain in Section 3.4, the characteristics of this benchmark are more focused towards archival and backup deduplication. For example, data modification is not simulated in a stress fashion as it would be expected in a primary storage.

This way, there is still the need for benchmarks that are able to simulate the dynamism and the duplicate content found in distinct storage environments. Also, with the increased number of deduplication proposals, it is important that these benchmarks are public so that distinct systems can be evaluated in reproducible and comparable testing environments.

3.1 DEDISbench

We target the previous challenges with DEDISbench, an open-source synthetic disk I/O micro-benchmark suited for block-based deduplication systems.

3.1.1 Design, features and implementation

The basic design and features of DEDISbench resemble the ones found in Bonnie++ and IOzone, which are two open-source synthetic micro-benchmarks widely used to evaluate disk I/O performance [Coker 2014, Norcott 2014].

DEDISbench is implemented in C and allows performing either read or write storage requests at the fixed-size block granularity, 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. Also, several processes can access concurrently the storage device or independent files, being the number of processes and the size of process files defined by the users. The evaluation can be configured to stop when a certain amount of data was processed or when a pre-defined period of time has elapsed, which is not common in most I/O benchmarks.

As another novel feature, storage operations can be executed with different load intensities. In addition to a stress/peak load where storage operations are issued as fast as possible to stress the system, DEDISbench can also issue operations at a nominal load, specified by the user, and thus evaluate the system with a stable load. Few disk I/O benchmarks support both features, as stated in Section 3.4.

Note that DEDISbench simulates low-level block I/O operations so, it is not focused on generating realistic directory trees and files like other benchmarks [Katcher 1997, Anderson 2002, Filebench 2014, Al-Rfou et al. 2010, Tarasov et al. 2012]. Nevertheless, such benchmarks are also referred along this chapter and compared with DEDISbench in terms of content generation and storage access patterns.

Figure 3.1 depicts an overview of DEDISbench architecture. For each process, an independent *I/O request launcher* module launches either read or write block operations, at nominal or peak rates, until the termination condition is reached. For each I/O operation, this module must contact the *access pattern* generator for obtaining the storage offset for the I/O operation (1) that will depend on the type of access pattern chosen by the user and that can be; sequential, random uniform or random with hotspots. Next, the *I/O request launcher* module contacts the *content generator* module for obtaining an identifier for the content to generate (2). Since DEDISbench is aimed at block-based deduplication, this identifier will then be appended as an unique pattern to the block's content, ensuring that blocks with different identifiers will not have the same content. The generated identifiers will follow the information from an input file, provided to the benchmark, that details how to simulate a specific duplicate content distribution.

Step 2 is only required for write requests because read requests do not generate any content to be written. Finally, the operation will be sent to the storage (3) and the metrics regarding operations throughput and latency will be monitored by the *I/O request launcher* module. The approaches used for generating the access and duplicate distributions are further detailed next.

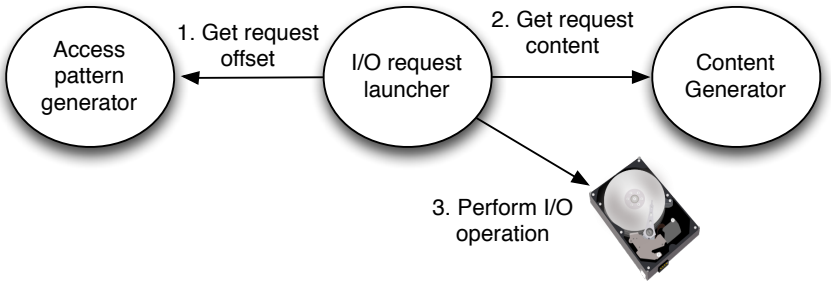


Figure 3.1: Overview of storage requests generation.

3.1.2 Storage access distribution

DEDISbench supports sequential and random uniform patterns for storage read and write accesses, as in IOzone and Bonnie++. These patterns are important for measuring the performance of low-level hardware characteristics, for instance, the hard disks arm movement while, also being useful for testing other storage characteristics such as raid and replication.

DEDISbench introduces a novel third access pattern that simulates access hotspots, where few blocks are accessed frequently while the majority of blocks are accessed sporadically. This hotspot distribution is generated with TPC-C NURand function [Transaction processing performance council 2010]. TPC-C is an industry standard on-line transaction processing SQL benchmark that mimics a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. In DEDISbench, the NURand function is used for generating the storage addresses to be written in each operation. As we show in Section 3.3, this is a more realistic pattern for most applications, where random accesses are tested while leveraging the advantages of caching mechanisms, thus allowing to uncover distinct performance issues.

3.1.3 Duplicate content distribution

DEDISbench main contribution is the ability to process an input file specifying a distribution of duplicate content, and using this information for generating a synthetic workload that follows such distribution. As depicted in Figure 3.2 the input file states the number of unique content blocks for a certain amount of duplicates. In this example there are 5000 blocks with 0 duplicates, 500 blocks with 1 duplicate, 20 blocks with 5 duplicates and 2 blocks with 30 duplicates. This file can be populated by the users or can be generated automatically with DEDISgen, an analysis tool used for processing a real dataset and extracting from it the duplicate content distribution. This tool is further detailed in the next section.

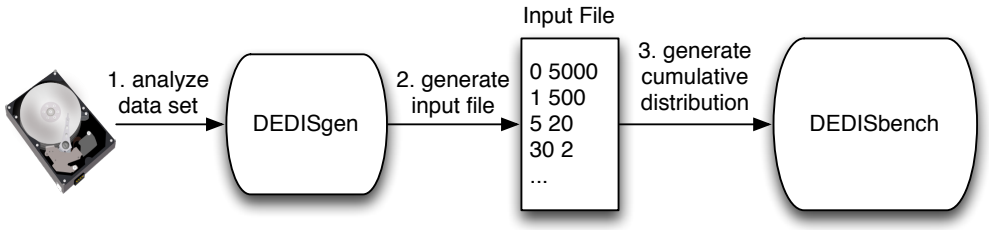


Figure 3.2: Process for extracting and generating a duplicate content distribution in DEDISbench.

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.

3.2 Automatic dataset analysis and extraction

There is extensive work focusing on the duplicates found at real storage systems [Meyer and Bolosky 2011, Clements et al. 2009, Hewlett-Packard Development Company , L.P 2011]. However, the information provided in these studies

does not present the necessary details in order to generate the input content distributions used by DEDISbench. This section explains how our benchmark can be extended to simulate additional duplicate content distributions, and presents the details of the distributions found for three distinct real storage systems. These distributions are publicly available to be used with our benchmark.

DEDISgen is an open-source tool for analyzing and extracting duplicate content distributions from real storage systems. This tool is implemented in C and processes data either from a storage block device, or from files inside a specific directory tree in the following way: The stored data is read and divided into fixed-size blocks, with a size chosen by the user. A SHA-1 hash sum is calculated for each block and inserted in a Berkeley DB database[1] in order to find duplicate hashes [Olson et al. 1999]. After processing all data, the database information is transformed into an input file suitable for DEDISbench. All this process is transparent and automatic for anyone who uses this tool.

Next, we analyze the distributions, for three distinct storage environments, extracted with DEDISgen for a block size of 4 KiB. These three distributions have specific access patterns and performance requirements that, as explained previously, limit the deduplication approach to be used.

3.2.1 Archival storage

The first dataset analyzed has mainly archival and some backup data from the members of our research group. Most data is accessed sporadically and the number of updates on stored data is extremely low, thus not significant. However, data can be deleted which is not assumed in some archival systems. This way, this dataset's requirements are similar to the ones found in traditional archival/backup deduplication systems where write-once data is assumed and I/O throughput is leveraged over I/O latency [Quinlan and Dorward 2002].

As depicted in Table 3.1, the dataset has approximately 486 GiB, 90% of the blocks do not have any duplicate, 3% of the blocks have duplicates with distinct content and 7% of the blocks are copies that can be eliminated with deduplication. This storage has a small percentage of duplicate content, when compared to some of the literature [Meyer and Bolosky 2011, Hewlett-Packard Development Company , L.P 2011]. This happens because most files are stored

[1] Berkeley DB is used as an hash table.

in compressed formats to reduce storage space usage. Since these files are only restored sporadically, the cost of decompressing them is acceptable for the users.

Table 3.1: Content statistics for the archival, personal files and high performance storage systems.

		Archival	Personal Files	High Performance
Total space (GiB)		486	113	1528
% Blocks w/o duplicates		90	76	69
% Duplicate blocks	distinct	3	6	6
	copies	7	18	25

3.2.2 Personal files storage

The second dataset has personal files from our research group and has distinct requirements from the archival dataset. Most data is accessed frequently and some is updated and deleted sporadically, so it cannot be considered as write-once. Also, the latency for reading and writing files at this storage is expected to be lower than the one tolerated for the previous archival storage. Since the requirements of this dataset change, the deduplication systems targeting this storage type are also different from the ones considered in the last section. Namely, deduplication systems are expected to efficiently handle reference management and protect updates on shared data with CoW mechanisms that can have a significant impact in storage requests latency.

As shown in Table 3.1, the dataset has approximately 113 GiB and has a higher percentage of copies than the one found at the Archival storage, namely, 18%. Moreover, 76% of the blocks do not have any duplicate while 6% are duplicated blocks with distinct content. This storage has significantly more duplicates than the archival one because several personal files are duplicated and compression of files is less usual. For example, several source-code repositories are held by this storage and, in most cases, each researcher has its own copy, thus generating several duplicates.

3.2.3 High performance storage

The last dataset analyzed is used as the primary storage for our group research projects and stores dynamic data from simulations and real applications. Data is updated frequently, which requires a higher number of CoW operations from deduplication systems and raises the complexity of reference management. Also, storage I/O latency is expected to be as minimal as possible so, the deduplication systems suitable for the two storage types described previously are usually not suited for primary storage environments.

As shown in Table 3.1, the dataset occupies approximately 1.5 TB, having the larger size of the three storage systems analyzed, and also the larger percentage of copies, namely, 25%. Additionally, 69% of the scanned blocks do not have any duplicate and 6% have duplicates with distinct content. This storage has the higher percentage of duplicates, which can be explained by the amount of duplicate runs from simulations and real systems benchmarks that are persisted in this storage system.

3.2.4 Datasets analysis

It is important to look at the distinct percentages of duplicates found for each storage environment, as it affects the global number of duplicates to generate. However, it is also important to observe the ratio of duplicates per block with unique content. In a storage scenario where blocks are updated frequently, having many blocks with one duplicate or having many blocks with several duplicates will change the complexity and performance of CoW, reference management and garbage collection mechanisms. This way, it is important that the workload simulates not only the number of duplicates but also the duplication ratio found in real storage systems.

The differences in the duplication ratios for the three storages are visible at Table 3.1. For instance, in both personal files and high performance storage systems 6% of the scanned blocks have unique content and are duplicated. However, the percentage of duplicates (copies) is higher for the high performance storage that, consequently, has a higher ratio of duplicates per block. These differences are even more noticeable in Figure 3.3, pointing the percentage of unique content blocks that have a specific range of duplicates (*i.e.*, equal to 0, between 1 and

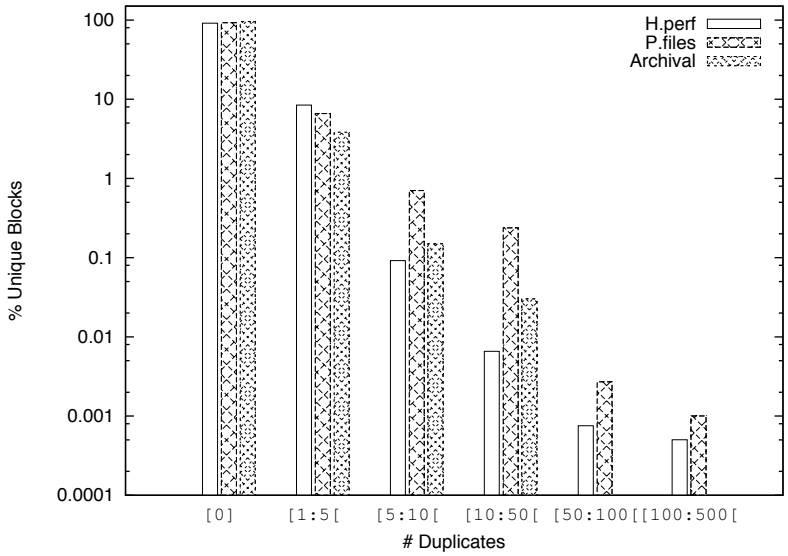


Figure 3.3: Distribution of duplicate ranges per unique blocks for archival, personal files and high performance storage systems.

5, 5 and 10, 10 and 50 and so on). The figure omits blocks with more that 500 duplicates for legibility reasons. In fact, all storage systems have some unique blocks with more than 500 duplicates. In detail, the personal files storage has 244 blocks with more than 500 duplicates, the high performance storage has 161 blocks with more than 500 duplicates, and the archival storage has 118 blocks with 50 to 100 duplicates, 120 blocks with 100 to 500 duplicates and 58 blocks with more than 500 duplicates.

Looking at the figure, the archival storage has few unique blocks with more than 100 duplicates, while the personal files storage has the higher percentage of unique blocks with many duplicates. On the other hand, the high performance storage has the higher percentage of unique blocks with few duplicates, which will probably increase the complexity of reference management and garbage collection as many shared blocks will be copied-on-write and then garbage collected in a frequent basis.

To sum up, this analysis is important for understanding the percentage of duplicates and the ratio of duplicates per unique block for three storage environments with distinct access patterns and requirements. For instance, the different duplication ratios identified can change the utilization patterns of reference management, garbage collection and CoW mechanisms that may impact

significantly the storage performance. Moreover, as we show next, distinct duplicate content distributions indeed affect the evaluation of deduplication systems so, it is extremely important to choose the correct workload. Finally, these are just three examples of what duplicate distributions can be expected from distinct real storage systems. Both DEDISgen and DEDISbench are publicly available so, we encourage other researchers and enterprises to use them for analyzing and releasing the distributions of their own storage systems.

3.3 Evaluation

In order to understand the impact of DEDISbench novel features, we compared our benchmark with IOzone and Bonnie++, the two micro-benchmarks with the most resembling design and features [Coker 2014, Norcott 2014].

Bonnie++ is a standard micro-benchmark that performs several tests to evaluate disk I/O performance *in the following order*: Write tests assess the performance of single byte writes, block writes and rewrites, while read tests assess byte and block reads, all with a sequential access distribution. Seek tests perform random uniform block reads and, in 10% of the operations, random uniform block writes. The size of blocks, the number of concurrent Bonnie++ processes and the size of the file each process accesses are defined by the user. All these tests are performed with a stress load and run until an amount of data is written/read for each test. It is not possible to specify the content of written blocks. Finally, Bonnie++ also tests file deletion and creation, which is not contemplated in this evaluation because it is not supported by IOzone or DEDISbench.

IOzone is the I/O micro-benchmark that most resembles DEDISbench and allows performing sequential and random uniform write and read tests. The block size, number of concurrent processes, and the size of the files of each process, are also defined by the users. Tests are performed at a stress load and, for each test, the user defines the amount of data to be read/written by each process. Unlike in Bonnie++, it is possible to define full tests that perform either read or write random disk I/O operations. Additionally, the percentage of inter-file and intra-file duplicate content can be specified for written blocks. Nevertheless, as discussed in the next sections, this content generation mechanism does not allow specifying a content distribution with a realistic level of detail as in DEDISbench.

DEDISbench, IOzone and Bonnie++ have several features in common but also differ in specific details, as shown in Table 3.2. In DEDISbench, a cumulative distribution function extracted from a real dataset allows simulating not only the percentage of duplicate and non-duplicate blocks but also the distribution of duplicates per unique block. Such is not possible with the IOzone's approach that only allows defining the intra and inter-file duplication percentage, or in Bonnie++ where the content distribution cannot be specified by the user. The content distributions generated by the three benchmarks are evaluated and further discussed in Section 3.3.2.

Table 3.2: Comparison of DEDISbench, IOzone and Bonnie++ features.

	DEDISbench	IOzone	Bonnie++
Content generation	Cumulative distribution function of a real distribution	Intra and Inter-file duplication percentage.	Cannot be specified
Access distributions	Sequential, uniform random and hotspot random	Sequential and uniform random	Sequential and uniform random
Termination	Data written and time	Data written	Data written
Intensity	Stress and nominal	Stress	Stress
Granularity	Block	Block	Byte and block

The three benchmarks support sequential and random uniform access tests. DEDISbench introduces a novel hotspot access distribution where few blocks are accessed frequently, while the majority of blocks are accessed sporadically. I/O tests in DEDISbench can be pre-defined to terminate when a specific amount of data was written, as in IOzone and Bonnie++, or when a specific amount of time has elapsed. Moreover, our benchmark can issue I/O operations with stress or nominal load intensities, allowing to evaluate storage systems with peak or stable loads. In Bonnie++, it is possible to perform block and byte I/O tests while, in the other two benchmarks, tests are only performed at the block granularity. Most of these details are evaluated and discussed in Section 3.3.4 as they influence the results of the evaluation of deduplication systems.

3.3.1 Scope and setup

The experiments discussed in this section aim at validating two distinct points: First, we want to show that DEDISbench simulates more accurately a real content distribution than IOzone and Bonnie++. As a second goal, we want to show that this enhanced content generation mechanism along with the other novel features allow uncovering new issues in deduplication systems, thus proving that such features are important for an accurate evaluation.

All tests ran in a 3.1 GHz Dual-Core Intel Core Processor with hyper-threading, 4 GiB of RAM and a local SATA disk with 7200 RPMs. Unless stated otherwise, for each benchmark test, the amount of data read/written was 8 GiB distributed over 4 concurrent processes, each reading/writing 2 GiB from an independent file.

The three benchmarks were configured in order to simulate as accurately as possible the content distribution of the personal files dataset described in Section 3.2. Also, since this dataset was analyzed with a block size of 4 KiB, the size chosen for DEDISbench and Bonnie++ block tests was also 4 KiB. DEDISbench used the input distribution file generated by DEDISgen to simulate the realistic distribution while, in Bonnie++, it is not possible to specify the content to be written.

On the other hand, the block size chosen for IOzone was 16 KiB, thus defining that each block would have 25% of its data (4 KiB) duplicated across distinct process files. With this configuration and using 4 independent files, each block of 16 KiB has a distinct 4 KiB region with three duplicates, one for each file, which resembles the average number of 3 duplicates per block found at Personal Files workload. Globally, IOzone generates 18.75% of copies while the remaining 75% of the blocks do not have any duplicate, which also resembles the values shown at Table 3.1 for the Personal Files data. By choosing 16 KiB for the block size, the duplicate blocks are generated with a size of 4 KiB as in the real dataset, which would not be possible if the IOzone block size was defined as 4 KiB.

Since IOzone allows defining intra and inter-file duplicate percentages, we could have increased further the block size parameter to define the number of duplicate sub-blocks at the same file. However, this decision would have increased significantly the block size and the complexity of the configuration for achieving a slightly closer distribution to the real one. Moreover, even with this extra

parameter, IOzone would only be able to simulate two or three types of blocks with distinct proportions of duplicates, while in DEDISbench it is possible to simulate as many types as specified in the input distribution file, thus increasing hugely the content distribution detail.

Note that we chose the personal files dataset for evaluating the benchmarks accuracy but we could have chosen one of the other two datasets described previously. As explained in Section 3.3.4, the evaluated deduplication systems, namely Opendedup and Lessfs, were developed for storage workloads with requirements resembling our personal files one. This way, we use this real distribution as a baseline for the complete evaluation procedure in order to increase the chapter's uniformity and clarity. However, the results and consequent conclusions of Section 3.3.2 would have been similar if one of the other distributions was chosen and the benchmarks were configured to simulate them. Finally, thorough the next sections, we also refer to the personal files dataset as the real dataset.

3.3.2 Duplicate content distributions

We used DEDISgen for analyzing the data generated by DEDISbench, IOzone and Bonnie++, for a sequential disk I/O write test, in order to compare the distinct content generation mechanisms. We chose the sequential I/O test over a random test because there are no block rewrites, enabling the extraction of precise information about all the written blocks and their contents. As explained previously, all benchmarks were configured to simulate as accurately as possible the content of the personal files workload.

Figure 3.4 presents the percentage of unique content blocks with a specific range of duplicates for Bonnie++, IOzone and DEDISbench generated content, and for the personal files dataset. Unique content blocks generated by Bonnie++ have between 1 and 5 duplicates, in fact, each unique block has precisely 3 duplicates because every file is written with the exact same content, meaning that, all blocks in the same file are distinct but are duplicated across the other files. Consequently and as shown in Table 3.3, with Bonnie++, 75% of the written space can be deduplicated. Note that, Figure 3.4 shows the number of duplicates generated for each *unique block* written by the benchmarks, while Table 3.3 shows the percentages of blocks without any duplicate, blocks with distinct duplicates, and duplicate blocks for *all the blocks* written by the benchmarks, thus explaining

why the percentages differ.

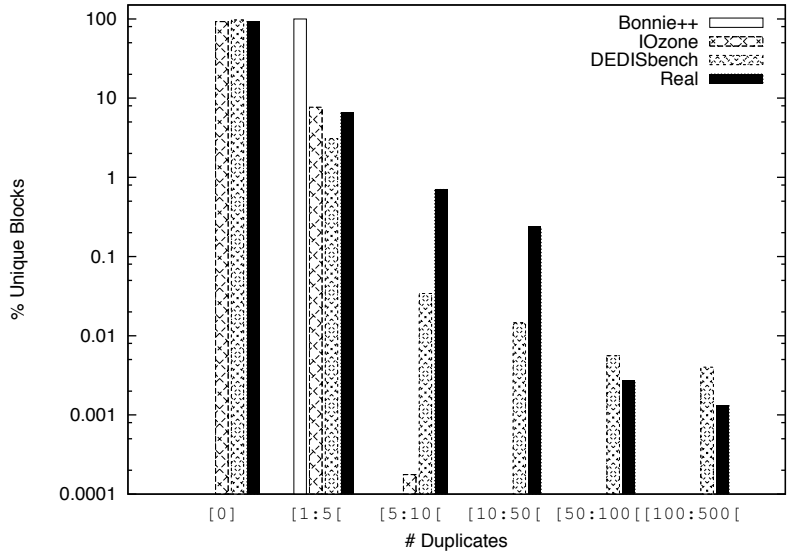


Figure 3.4: Distribution of duplicate ranges per unique blocks for Bonnie++, IOzone, DEDISbench and the real dataset.

The results for IOzone in Figure 3.4 show that most unique blocks do not have any duplicate, while the remaining blocks have mainly between 1 and 5 duplicates, and a very small percentage has between 5 and 10. According to IOzone configuration, duplicated distinct blocks should have 3 duplicates each, which happens for almost all the blocks with the exception of 216 blocks that have only 1 duplicate and 3 blocks that have 7 duplicates. In Table 3.3, IOzone percentages for duplicates and unique blocks are the closest ones to the real distribution percentages.

Table 3.3: Duplicates found for Bonnie++, IOzone, DEDISbench and the real dataset.

		Bonnie++	IOzone	DEDISbench	Real
% Blocks w/o duplicates		0	75	90	76
% Duplicate blocks	distinct	25	6	3	6
	copies	75	19	7	18

The results of DEDISbench, in Figure 3.4, show that the number of unique blocks is distributed over several regions of duplicates, resembling most the real

distribution. Most blocks have few duplicates and a small percentage of blocks has many duplicates. In fact, we omitted one value from the figure in the far end of the distribution tail, for legibility reasons, where a single block has 15665 duplicates. Simulating accurately the head and tail of the distribution is important for having a realistic evaluation. For example, having many blocks with few duplicates will increase the number of shared blocks that, after being rewritten, must be collected by the garbage collection algorithm. On the other hand, mixing blocks with different number of duplicates will also affect the size of metadata structures and the work performed by the deduplication engine.

However, when looking at Table 3.3, DEDISbench results are slightly more distant from the real values than IOzone results. Since the real data set has approximately 100 GiB and the benchmark is only writing 8 GiB, even if the cumulative distribution followed by DEDISbench has a high probability of writing the content of some blocks several times, which would generate a large amount of duplicates, these blocks are being written fewer times than expected. Figure 3.5 and Table 3.4 compare the results of running DEDISbench sequential write tests for 16 and 32 GiB (divided by 4 files) and show that when the amount of written data is closest to the amount of data in the real dataset, the generated distribution also becomes closer to the real one.

Table 3.4: Duplicates found for DEDISbench tests with 8, 16 and 32 GiB and the real dataset.

		DEDISbench 8	DEDISbench 16	DEDISbench 32	Real
% Blocks w/o duplicates		90	87	83	76
% Duplicate blocks	distinct	3	4	5	6
	copies	7	8	12	18

To conclude, these results show that both Bonnie++ and IOzone do not simulate accurately the distribution of duplicates per unique blocks. On the other hand, DEDISbench design allows to overcome this limitation and simulates more accurately a real storage workload, thus proving our first evaluation goal. The lack of detail in Bonnie++ and IOzone can influence the load in the deduplication and garbage collection mechanisms of the deduplication system. For instance, a block shared by two entities or by one hundred determines the timing when

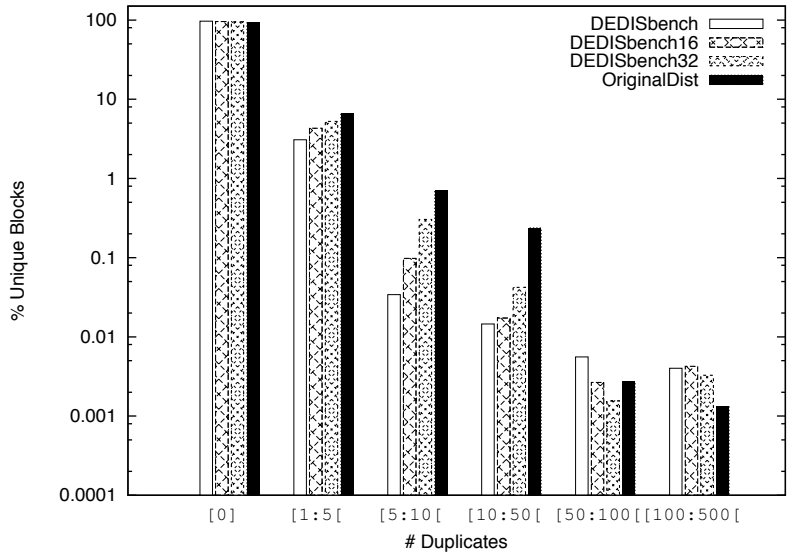


Figure 3.5: Distribution of duplicate ranges per unique blocks for DEDISbench tests with 8, 16 and 32 GiB and for the real dataset.

garbage collection is needed, how often the CoW mechanism must be used and the amount of information in metadata structures for sharing identical content.

3.3.3 Storage access distributions

Another contribution of DEDISbench is the introduction of the NURand hotspot access distribution, besides the traditional sequential and random uniform disk access patterns, used in Bonnie++ and IOzone. We ran DEDISbench with the three access distributions: sequential, random uniform and NURand, and extracted the access patterns of each distribution. IOzone and Bonnie++ were not used in these tests because, in order to extract this information, it would require modifying their source code. Moreover, DEDISbench sequential and random uniform distributions mimic the ones found for these two benchmarks.

Figure 3.6 presents the percentage of blocks for a certain range of accesses. In the sequential distribution 100% of the blocks are accessed precisely once (range between 1 and 5 in the figure), while in the random uniform distribution most of the blocks are accessed between 1 and 5 times, in fact most blocks are accessed only once and the percentage of blocks decreases for an higher number of accesses. On the other hand, the NURand distribution results show that a high percentage

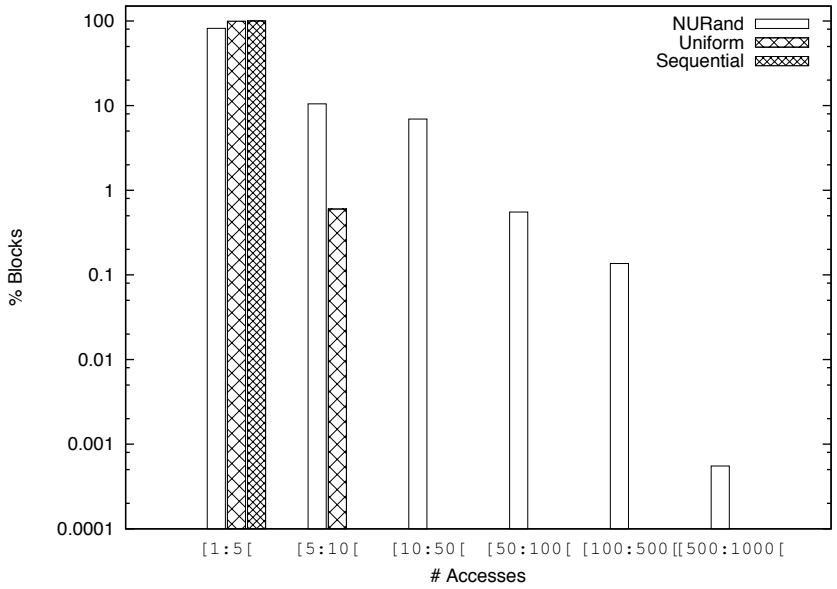


Figure 3.6: Distribution of accesses per block for sequential, random uniform and NURand approaches.

of blocks is accessed few times while a small percentage is accessed many times, generating blocks that are hotspots (*i.e.*, a few blocks are accessed more than 500 times).

To sum up, the NURand distribution allows creating hotspots for I/O requests, thus simulating a storage environment where some data is accessed frequently and most data is only accessed sporadically. For deduplication systems this means that some blocks are constantly being shared, copied-on-write and garbage collected, which has a distinct impact that cannot be simulated by the other two access distributions.

3.3.4 Storage performance evaluation

In order to assess how the distinct features of each benchmark affect the evaluation of real deduplication systems, we chose to evaluate LessFS and Opendedup in-line deduplication systems, discussed in Chapter 2. We chose these systems because they are mature open-source projects that, although not designed for primary storage scenarios with strict latency requirements, export file system semantics supporting data modification. This feature is important for testing the impact of CoW and garbage collection mechanisms.

The main goal of this section is to compare how Bonnie++, IOzone and DEDISbench evaluate both deduplication systems. However, a direct comparison of the results of each benchmark have no meaning, as we are dealing with distinct benchmarking tools. For instance, saying that Opendedup achieves higher disk throughput in Bonnie++ sequential block write test than in IOzone and DEDISbench sequential block tests may not be significant. Firstly, each benchmark has distinct implementations for similar tests, and measures differently the performance metrics. Moreover, some benchmarks support tests that are not present in the others, for example, only Bonnie++ has single byte tests, and only DEDISbench uses an hotspot access distribution.

For these reasons, we chose to evaluate each deduplication system as it would have been evaluated in a traditional scenario. Namely, the two deduplication systems were compared, in terms of overhead, with Ext4, a file system without deduplication. This way and as an example, if the single byte write tests of Bonnie++ introduce higher disk latency overhead than the sequential block write tests of the three benchmarks, we can conclude that writing single bytes is inefficient and that this specific test uncovers a problem that is not evaluated by the other tests. Similarly, if sequential block writes in Bonnie++ have less I/O latency overhead than in IOzone or in DEDISbench sequential write tests, we can compare these values and explain that this difference exists because Bonnie++ writes a higher percentage of duplicates than the other two benchmarks. To sum up, by comparing the overheads of the deduplication systems over Ext4, it is possible to extract meaningful information even if the implementations of the three benchmarks are different. Note that this evaluation does not aim at assessing which benchmark consumes less resources or has higher throughput. Instead, we want to understand how each deduplication system is evaluated by these benchmarks and which issues are uncovered by using distinct features.

Regarding the tests setup, the three file systems were mounted in the same partition, with a size of 20 GiB, that was formatted before running each benchmark. Also, the deduplication file systems were configured to have a fixed-block size of 4 KiB. Bonnie++ tests were issued in the following order: single-byte write, block write and block rewrite in sequential mode, single-byte read and block read in sequential mode and, finally, the random seek test. For IOzone and DEDISbench we tried to choose a test order as similar as possible to Bonnie++.

IOzone order was: Block write, block rewrite, block read and block reread in sequential mode, and then block read and block write in random uniform mode. For DEDISbench the order was exactly the same as in IOzone, but with two additional tests, a block read and block write with the NURand access distribution. Also, DEDISbench used the personal files workload, discussed in Section 3.2, because this is the storage environment that best fits the assumptions of the evaluated deduplication systems.

Table 3.5 shows the results of running Bonnie++ on Ext4, LessFS and Opendedup. By comparing the deduplication systems with Ext4 it is possible to conclude that writing sequentially one byte at a time is inefficient because, for each written byte, a block of 4 KiB will be modified and will be shared by the deduplication system, thus forcing the deduplication system to process a single block 4096 times. This is also true for sequential byte reads where, in each operation, it must be made an access to the metadata that tracks the stored blocks for retrieving a single byte. In this last test, the overhead introduced by Opendedup, when compared to LessFS overhead, is considerably higher so, it is possible that LessFS is taking advantage of some sort of mechanism that avoids retrieving the whole block to memory in each byte read operation.

Table 3.5: Evaluation of Ext4, LessFS and Opendedup with Bonnie++.

	Ext4	LessFS	Opendedup
Sequential byte write (KiB/s)	1100	76	56
Sequential block write (KiB/s)	72035	13860	155496
Sequential block rewrite (KiB/s)	17319	1016	62744
Sequential byte read (KiB/s)	3029	1262	72
Sequential block read (KiB/s)	73952	60064	144614
Urandom seek (seeks/s)	171	127	116

In sequential block write and rewrite tests Opendedup outperforms Ext4 by taking advantage of Bonnie++ writing the same content frequently and repeating it in all tests. Data written in sequential byte tests is shared so, in subsequent tests, Opendedup algorithm only requires consulting the in-memory metadata for finding duplicate content and sharing it, thus avoiding the need of actually writing the new blocks to disk. On the other hand, LessFS implementation does not seem to take advantage of such scenario, probably because some persistent

metadata structure is still being written for each write operation. Opendedup also outperforms Ext4 in sequential block reads probably with a pre-fetching cache mechanism, although only efficient at the block granularity. Finally, in random seek tests both deduplication systems present worse results than Ext4, with LessFS slightly outperforming Opendedup. RAM and CPU measurements while Bonnie++ was running are depicted in Table 3.6. Both Opendedup and LessFS consume a significant amount of RAM, meaning that most metadata is loaded in memory and explaining, for example, the performance boosts of Opendedup in sequential block read and write tests. The significant CPU consumption measured for Opendedup can be a consequence of Bonnie++ writing a high percentage of duplicate content, thus generating a large amount of duplicates to be processed.

Table 3.6: CPU and RAM consumption of LessFS and Opendedup for Bonnie++, IOzone and DEDISbench.

		LessFS	Opendedup
Bonnie++	CPU	22 %	163 %
	RAM	2.2 GiB	1.8 GiB
IOzone	CPU	9 %	25 %
	RAM	1.3 GiB	2.1 GiB
DEDISbench	CPU	15.7	19.5 %
	RAM	2.2 GiB	1.9 GiB

Table 3.7: Evaluation of Ext4, LessFS and Opendedup with IOzone.

	Ext4	LessFS	Opendedup
Sequential block write (KiB/s)	74463	5525	19761
Sequential block rewrite (KiB/s)	74357	373	29925
Sequential block read (KiB/s)	67159	7777	10464
Sequential block reread (KiB/s)	67522	11495	10404
Urandom block read (KiB/s)	2086	1304	1766
Urandom block write (KiB/s)	2565	162	1608

Table 3.7 shows the results of running IOzone for Ext4, LessFS and Opendedup. Unlike Bonnie++, this benchmark does not write always the same content in distinct tests, explaining why Opendedup does not outperforms Ext4 in block

rewrite operations. This way, although some of the data was shared already, the content written is not always the same and most requests are still written to disk. With IOzone, Opendedup outperforms LessFS in almost all tests with the exception of block reread test where LessFS is slightly better. LessFS performance degradation is visible in sequential and random write tests, and mainly in rewrite tests. In Table 3.6, the RAM and CPU usages drop significantly which can be a consequence of less duplicate content being written and processed. The RAM usage in Opendedup is an exception and the value is higher than in Bonnie++ tests. The size of the metadata index increases with the number of unique blocks to index, which can explain this last value.

Table 3.8 shows the results of running DEDISbench on Ext4, LessFS and Opendedup. As explained previously, IOzone generates almost all duplicated blocks with exactly 3 duplicates, while DEDISbench uses a realistic distribution where most blocks have few duplicates but some blocks have a large number of copies. This will help explaining the next results. In sequential tests both Opendedup and LessFS are outperformed by Ext4, as in IOzone evaluation. However, the results of Opendedup for the sequential write test show considerably less overhead when compared to the same IOzone test. This can be a consequence of DEDISbench generating some blocks with a large amount of duplicates that will require writing only one copy to the storage, thus enhancing the performance of Opendedup. On the other hand, in the sequential rewrite tests, Opendedup performance decreases since DEDISbench generates many blocks with few duplicates that will then be rewritten, copied-on-write and will require garbage collection, thus increasing the overhead.

The most interesting results appear in the random I/O tests. Firstly, LessFS outperforms Ext4 in uniform random block read test, which is an harsh test for the disk arm movement, pointing one of the advantages of using deduplication. If two blocks stored in distant disk positions are shared, the shared block will then point to the same disk offset and a disk arm movement will be spared. Also, in-memory cache mechanisms can store more blocks since duplicate blocks do not need to be included in the cache. In IOzone there are few duplicates per block and this operations does not occur so often but, in DEDISbench some blocks have a large number of duplicates which can reduce significantly the disk arm movement, increase cache efficiency and, consequently, improve performance. As

Table 3.8: Evaluation of Ext4, LessFS and Opendedup with DEDISbench.

	Ext4	LessFS	Opendedup
Sequential block write (KiB/s)	86917	5025	77508
Sequential block rewrite (KiB/s)	76905	658	18853
Sequential block read (KiB/s)	78649	7527	18592
Sequential block reread (KiB/s)	78620	11789	20405
Urandom block read (KiB/s)	791	2055	511
Urandom block write (KiB/s)	1416	123	n.a.
NURandom block read (KiB/s)	2287	1830	1350
NURandom block write (KiB/s)	1246	152	n.a.

a matter of fact, even in Opendedup where this improvement is less visible, the overhead for random uniform read tests is lower than the one for sequential read tests.

With the NURand hotspot distribution the performance of read operations in Ext4 is leveraged because caching mechanisms can be used more efficiently. Therefore, the performance gain for LessFS and Opendedup are less noticeable but, nevertheless, achieve less overhead than in sequential tests. The CPU and RAM consumptions, shown in Table 3.6, for LessFS and Opendedup are similar to the ones obtained with IOzone, with a slight reduction in Opendedup values and increase in LessFS ones. These variations can be explained by the design and implementation of each deduplication system and how they process the distinct generated datasets.

The other interesting results are visible in the uniform and NURand random write tests. The performance of LessFS when compared to Ext4 decreases significantly, while Opendedup system blocks with a CPU usage of almost 400%, not being able to complete these tests. This means that the realistic content distribution in DEDISbench uncovered a problem in Opendedup that could not be detected with the simplistic content distributions of IOzone and Bonnie++.

To further prove this point, Table 3.9 tests Opendedup with the default DEDISbench and a modified version that writes always the same content for each I/O operation. The results show that Opendedup completes successfully all the tests and greatly increases the performance, even when compared to the Ext4 results with the default DEDISbench version. However, the drawback of

processing a fully duplicate dataset is visible in the CPU and RAM usage of Opendedup that increases to 272% and 2.6 GiB respectively. These results show that using a realistic content distribution is necessary for a proper evaluation of deduplication systems, and that Opendedup is not thought for datasets with a considerable percentage of non-duplicated data.

Table 3.9: Evaluation of Opendedup with DEDISbench and a modified version of DEDISbench that generates the same content for each written block.

	DEDISbench Original	DEDISbench Modified
Sequential block write (KiB/s)	77508	247428
Sequential block rewrite (KiB/s)	18853	253818
Sequential block read (KiB/s)	18592	412694
Sequential block reread (KiB/s)	20405	418169
Urandom block read (KiB/s)	511	106696
Urandom block write (KiB/s)	n.a.	3638
NURandom block read (KiB/s)	1350	73386
NURandom block write (KiB/s)	n.a.	3289

To sum up, this section states that using realistic content and access distributions influences significantly the evaluation of deduplication systems. Moreover, generating a realistic content distribution is necessary for finding performance issues and system design fails, like the ones found for Opendedup, but also for finding deduplication advantages, such as the performance boost in uniform random read tests found for LessFS. Moreover, it is useful having a benchmark that can simulate several content distributions ranging from fully duplicate to fully unique content and, most importantly, that is able to generate a content distribution where the number of duplicates per block is variable, and follows a realistic workload.

3.4 Related work

Despite the extensive research on storage benchmarking, benchmarks that simulate duplicate content distributions are vaguely addressed in the literature, being either limited to generating simplistic distributions, or focused towards generating specific storage datasets [Norcott 2014, Filebench 2014, Tarasov et al. 2012].

IOzone and Bonnie++ are the two open-source synthetic micro-benchmarks that most resemble DEDISbench in terms of features and evaluation parameters [Norcott 2014, Coker 2014]. However, Bonnie++ does not allow specifying the content generated for write operations, in fact, it writes the same content in each I/O test and for each file. On the other hand, IOzone allows specifying the percentages of a record (block) that are duplicated in a intra-file and inter-file fashion. Although these parameters allow having some control over the number of duplicates per record, the level of detail is not as realistic as the one achievable with our benchmark.

Both IOzone and Bonnie++ support either sequential or random uniform access distributions, and are only able to perform stress testing. DEDISbench introduces an hotspot access distribution based on TPC-C NURand function, and allows issuing tests at a nominal throughput specified by the users.

Other work, with distinct goals, leverages the characterization of actual file systems by simulating directory threes and depth, the amount of files in each directory, distinct file sizes, and multiple operations on files and directories. Namely, Postmark design aims at evaluating the performance of creating, appending, reading and deleting small files, thus simulating the characteristic workloads found in mail, news and web-based commerce servers [Katcher 1997]. Distinct operations and sizes are assigned to files by using a random uniform distribution. Then, in Fstress work, novel workloads for simulating distinct storage environments are discussed (*e.g.*, peer-to-peer, mail and news servers) [Anderson 2002]. Also, an hotspot probabilistic distribution is used for assigning operations to distinct files and, these operations are issued with a pre-defined nominal load. As a third option, with higher accuracy, a set of probabilistic models can be used for creating new directories and files, choosing the depth and number of files in each directory, and defining the size and access patterns to distinct files [Agrawal et al. 2009].

Although previous benchmarks are able to simulate realistic file systems, none of them focus on realistic content generation. In Filebench file system benchmark, an entropy based approach is used for generating data with distinct content, and allows controlling the compression and duplication ratio of written blocks [Filebench 2014, Al-Rfou et al. 2010]. Similarly to IOzone, this feature cannot reproduce the level of detail that is achievable with DEDISbench.

Finally, Tarasov *et al.* work presents a framework for generating synthetic realistic datasets for deduplication systems [Tarasov et al. 2012]. When compared to previous benchmarks, this is the only work that aims at generating duplicate content distributions with a high-level of detail. In order to accurately simulate datasets and the mutations on their content over time, several snapshots of a real storage system must be analyzed. This analysis is done at the file system level, while the extracted information is used to specify a probabilistic model that tracks both the content at the real storage system and the data mutations across snapshots. Deduplication storage systems are then evaluated in the following way. A first version of the dataset is represented in an in-memory metadata structure describing all the folders, files and their content signatures, as found at the analyzed dataset. This first version is applied to the storage system by creating all the content at the storage, thus simulating a realistic dataset. Then, for each iteration of the benchmark, the probabilistic model is used for mutating the in-memory metadata into the new dataset version. After completing the mutation process, the new simulated dataset is specified in the metadata structure, and can applied to the current storage system by creating, modifying and deleting the necessary files and folders.

This model is accurate for simulating mutable backup systems, in fact, for this specific purpose it is a more realistic model than the one followed by DEDISbench. However, for other storage environments like primary storage systems, the previous mutation process is too slow so, the high dynamism of DEDISbench that frequently writes and modifies data blocks is necessary to simulate an accurate storage workload.

To sum up, most I/O benchmarks do not support the generation of duplicate content writing either random or constant data patterns. To our knowledge, IOzone, Filebench and Tarasov et all framework are the only I/O benchmarks supporting such feature. When compared with DEDISbench, these benchmarks use different algorithms for generating duplicate content that limit the realism of generated distributions, or simulate specific storage environments that are distinct from the ones simulated by DEDISbench.

3.5 Discussion

In this chapter, we discuss the characterization of duplicate content in storage systems and its impact in the evaluation of deduplication systems. Also, we propose DEDISbench, a synthetic disk I/O micro-benchmark that processes metadata extracted from real datasets for generating realistic content for I/O write operations. Previous micro-benchmarks either do not focus on distinct content generation, or generate limited distributions that, in most cases, do not accurately simulate real datasets. Moreover, DEDISbench allows performing I/O tests with stress and nominal intensities and introduces a novel distribution, based on TPC-C NU-Rand function, that allows testing the impact of hotspot random disk accesses.

We also show that distinct storage environments have specific content distributions that must be simulated accurately in order to have a more realistic evaluation. The DEDISgen tool can be used precisely to extract these different distributions from real storage datasets. Also, this tool allows extending DEDISbench supported workloads in a simple and fully-automatic fashion.

The comparison of DEDISbench with IOzone and Bonnie++ shows that our benchmark simulates more accurately a real content distribution, allowing to specify in detail the proportion of duplicates per unique block. Along with the novel hotspot distribution, this increased accuracy is key for finding relevant issues in two deduplication file systems, LessFS and Opendedup. This allow us to conclude that DEDISbench novel features are important for properly evaluating primary storage deduplication systems.

Chapter 4

DEDIS: Primary storage deduplication

Deduplication space savings are increasingly desired in cloud computing virtualized infrastructures. In fact, previous studies show that up to 80% of duplicate content exists across VMs primary volumes stored at these infrastructures [Clements et al. 2009]. CoW golden images are commonly used to launch identical VM images as they allow eliminating a high percentage of duplicate static data [Hewlett-Packard Development Company , L.P 2011, Meyer et al. 2008]. Deduplication gain can then be increased if dynamic data from users and applications is also targeted by the deduplication system, as well as, if duplicates are found and eliminated across the whole cluster. However, cluster-wide deduplication of dynamic data stored in VMs primary volumes raises new challenges that must be addressed.

Challenges

Applications accessing VMs primary volumes have strict performance requirements, therefore, the impact of deduplication systems in the latency and throughput of storage accesses must be unnoticeable. Traditional in-line systems include the computational overhead of deduplication in the storage critical path, which leads to an unacceptable penalty in the storage system performance [Ng et al. 2011, Srinivasan et al. 2012].

Alternatively, off-line systems alleviate the previous penalty by running deduplication as a background process, that asynchronously shares written data [Hong

and Long 2004, Clements et al. 2009]. Since deduplication and storage requests are performed concurrently, it is necessary to have proper mechanisms for coping with concurrency issues. CoW is required to prevent in-place updates of aliased data and possible data corruption. However, CoW operations are done in the critical storage write-path, and increase the complexity of reference management and garbage collection. In fact, in some off-line systems the storage overhead introduced by this and other operations is still significant, thus forcing deduplication to run only in off-peak periods [Clements et al. 2009]. Off-peak periods are usually scarce in cloud infrastructures so, deduplication has a short time-window for processing duplicates. As off-line systems require extra temporary storage space, the deduplication algorithm should run continuously to ensure that duplicates are only kept on disk for short periods of time.

Deduplication gain is maximized if duplicates are found and eliminated globally across all VMs volumes at the cluster. Doing so in a distributed cloud infrastructure is not trivial and raises even more challenges [Hong and Long 2004, Clements et al. 2009]. In order to find duplicates across the whole cluster, each node performing deduplication must have access to a remote indexing mechanism, that tracks unique storage content and allows 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 approaches.

The previous challenges are addressed with DEDIS, a dependable and fully-decentralized system that performs cluster-wide off-line deduplication of VMs primary volumes. Our system supports any storage backend, distributed or centralized, as long as it exports a simple shared block device interface. This way, our approach does not rely on specific storage backends with built-in functionalities, thus decoupling the deduplication system from a certain storage specification and avoiding performance issues that arise from this dependency [Hong and Long 2004, Clements et al. 2009]. Also, our design does not depend on storage workloads exhibiting specific data locality properties to achieve low storage overhead and an acceptable deduplication throughput [Srinivasan et al. 2012].

Briefly, DEDIS uses an optimistic off-line deduplication approach that excludes most of the computational effort from the storage write path. Storage writes are intercepted with a fixed-size block granularity, and redirected immedi-

ately 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, thus reducing the impact of deduplication in storage requests. Moreover, deduplication is performed globally and exactly across the entire cluster, more specifically, all duplicate chunks are processed and eventually shared. This is achieved 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 our design to be fully decentralized and to scale-out.

4.1 Baseline architecture

Figure 4.1 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. Together with the hypervisor, storage management services provide logical volumes to VMs by translating *logical addresses* within each volume to *physical addresses* in arbitrary disks upon each block I/O operation. Since networked disks provide only simple block I/O primitives, a distributed coordination and configuration service is assumed to locate meta-information for logical volumes, free block extents and to ensure that a logical volume is mounted at any time by at most one VM. The main functionality is as follows:

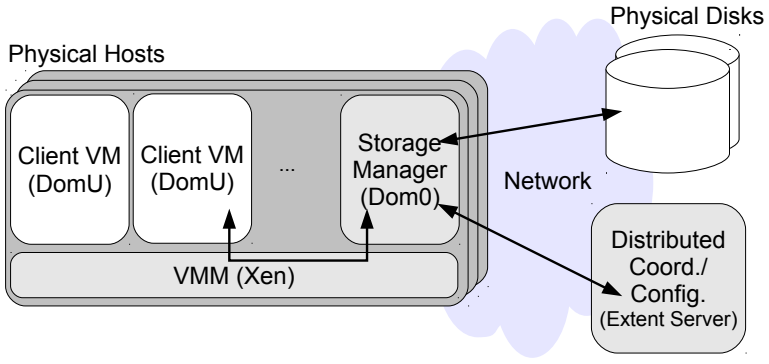


Figure 4.1: Distributed storage architecture assumed by DEDIS.

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 a common pool 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 4.1 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 is used as the storage networking protocol. 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 metadadata structure for mapping 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].

4.2 The DEDIS system

4.2.1 Architecture

DEDIS architecture, depicted in Figure 4.2, requires, in addition to the baseline architecture, a *distributed* module and two *local* modules. These are highlighted in the figure by the dashed rectangle and provide the following functionality:

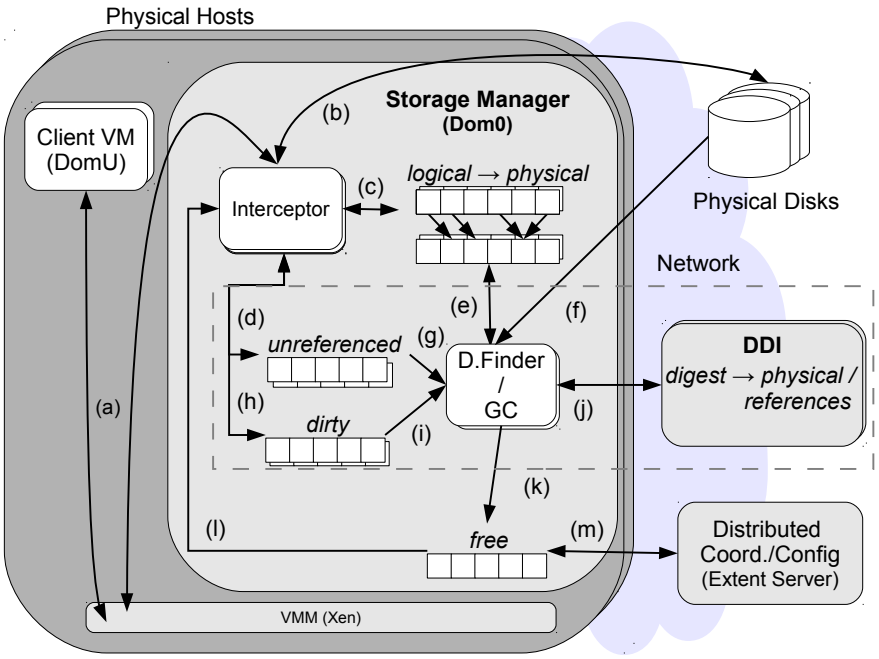


Figure 4.2: Overview of the DEDIS storage manager.

Distributed Duplicates Index (DDI) A distributed module that indexes unique content signatures of storage's blocks. Each entry maps an 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 enable 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 several blocks. This way, the index scales-out without having any single point of failure.

Duplicates 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.

4.2.2 I/O operations

The operations executed by DEDIS modules are depicted in Figure 4.2. 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 multi-threaded 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 a physical disk 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), that marks the address for CoW (e), reads its content at the storage (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, that 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 (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 is 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 and *extent* server requests for unused blocks. 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 that use two distinct metadata structures, and allows combining aliasing and reference management in a single remote in-

vocation, thus avoiding a higher throughput penalty and reducing the required metadata size.

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.

4.2.3 Concurrent optimistic deduplication

Figures 4.3 and 4.4 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.

```
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
```

Figure 4.3: Pseudo-code for intercepting and processing VM writes at the *interceptor* module.

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 pointing to storage block A (line 3), writes

```
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
```

Figure 4.4: Pseudo-code for share operations at the *D. Finder* module.

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 can be corrupted.

A straightforward solution to this issue is to lock the *logical-to-physical* mapping during the whole aliasing operation. However, such 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 (line 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 4.5 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 that, 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.

```
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
```

Figure 4.5: Pseudo-code for garbage collection at the *GC* module.

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 have corresponding entries at the DDI, which are no longer being referenced, are also freed (lines 34 and 35). Our algorithm ensures that a *test-and-decrement* operation is always preceded by a *test-and-increment* for the same DDI entry in order to properly handle reference management.

Some of the previous concurrency issues were uncovered while validating a version of our algorithm with a model checker. More specifically, the algorithm was encoded with the CAL language and two safety properties were then verified with the TLA+ toolset [Lamport 2002, Lamport et al. 2009]: (i) That values read from a storage block correspond to values actually written there, thus excluding corruption; and (ii) that both the DDI and logical-to-physical mappings are not indexing or referring to free blocks, thus avoiding the incorrect scenario where blocks are simultaneously being used and marked as free. The configuration used had 2 virtual storage blocks, 3 physical disk blocks, 3 processes, and 2 block values. For further details, the CAL specification can be consulted in Appendix A.

4.2.4 Fault tolerance

Writing meta-information persistently is required to ensure that logical volumes survive 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* pool 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 non-critical 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 an unique ordered timestamp that is stored persistently in the log, and at the DDI when a *test-and-increment* is done (lines 16 and 20). Since requests to the DDI follow the order specified in the timestamps, each DDI node persistently stores the last timestamp processed for each VM volume. This allows quickly checking for unprocessed operations while, requiring a small amount of extra storage space, even for a large number of VM volumes.

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 *logical-to-physical* 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 an 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, an 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 4.1, 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 each DDI node can be fully replicated, with a virtually-synchronous group communication protocol, and can serve requests for failed replicas.

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.

4.2.5 Optimizations

In previous related work, 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 an 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, that 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. 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 with the *D. Finder* module. As explained in Section 4.2.2, 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 4.3. 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 4.2.3 and do not need to be durable. If a failure occurs, the content of the blocks can always be retrieved directly 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 *extent* server, and the *free* pool. Batch requests allow 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 our 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.

4.2.6 Implementation

DEDIS prototype is implemented within Xen and uses the Blktap mechanism for building the *interceptor* module. Blktap exports an user-level disk I/O interface that replaces the commonly used loopback drivers while providing better scalability, fault-tolerance and performance [Citrix Systems, Inc 2014]. Each VM volume has an independent process intercepting disk requests with a fixed block size of 4 KiB, which is also the block size used by our system. 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 goal of this implementation is to highlight the impact of deduplication and not to re-invent a LVM system or the DDI. Simplistic implementations have thus been used for metadata and log structures. Namely, 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. The *in-memory hash* cache is also shared by both 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 entries of 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 [Tsuyoshi, Ozawa and Kazutaka, Morita 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, this implementation presents a worst-case scenario for the storage and RAM space occupied by metadata and persistent logs as more space-efficient structures could have been used instead.

4.2.7 Launching new VMs

To evaluate the prototype in a more realistic environment, we ensure that the system does not start with an unrealistic amount of duplicates in VM images, that would be avoided by cloning in most LVM systems. Those duplicates would result in abnormally high number of duplicates being found and processed during the initial stage of experiments, while at the same time, not having any aliased blocks would unfairly reduce the overhead from CoW operations.

Briefly, upon loading, VM images are divided into 4 KiB blocks that are examined and actually stored only if they have useful content. This way, unused blocks found in sparse images are lazily allocated only when needed. Moreover, deduplication is performed for each block being loaded to the storage. Duplicates are found inside the same image and across distinct images, with an algorithm

that is very similar to the one used by our prototype. Moreover, 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 an image 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 in-line deduplication is used instead, meaning that duplicates are eliminated before being stored. In-line deduplication is commonly used in VM loading mechanisms so, we chose to use a similar approach in our implementation [Ng et al. 2011].

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 is important for evaluating our prototype in a more realistic scenario as it provides the results that would have been obtained by having DEDIS running for a much longer time and discarding a longer prefix of the execution. These operations use the same metadata structures as DEDIS, require no additional storage space and, have no additional impact while the experiment is running.

4.3 Evaluation

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

4.3.1 Benchmark

Since DEDIS targets dynamic primary data, using exclusively static traces of VM images is not suitable for its evaluation. On the other hand, traditional disk benchmarks do not simulate accurately duplicate content, either writing all blocks with the same content or with random content. For these reasons, we have used DEDISbench to evaluate our deduplication system.

As explained in Chapter 3, DEDISbench novel features are important for evaluating primary deduplication systems under a more realistic scenario. Firstly, the content written by the benchmark mimics distributions found in real storage systems. In fact, one of the workloads presented at the previous chapter and supported by DEDISbench, simulates the content of a real primary storage, with $\approx$ 1.5 TB and 25% of duplicates, which fits our evaluation requirements.

Moreover, DEDISbench supports an hotspot random pattern for storage accesses. This pattern allows simulating 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. Write hotspots increase the number of block rewrites and, consequently, the amount of CoW operations. Since CoW is a costly operation for storage writes latency, it is important to test the effects of hotspots in primary deduplication systems [Clements et al. 2009].

4.3.2 Experimental setup

Tests ran in cluster nodes equipped with a 3.1 GHz Dual-Core Intel i3 Processor, 8 GiB of RAM and a 7200 RPMs SATA disk. VMs were configured with 4 GiB of RAM and a single virtual disk volume with 20 GiB. 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.

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 1 and 3 processed requests for two distinct shards. Then, in order to cope with the failure of these two servers, server 2 replicated server 1 and server 4 replicated server 3.

The distributed storage pool was also provided by the local disks of the cluster servers. More specifically, each server exported to the other servers an iSCSI device with 45 GiB. 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, etc. 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.

Each VM ran an independent DEDISbench instance 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, both local and remote, as well as, the overhead of persistent logging.

Finally, the evaluation results presented in this section required $\approx$ 18 hours of computation, only for DEDIS tests. In this period, more than 1.7 TB ($\approx$ 455 million blocks) were written by DEDISbench into the storage and more than 405 GiB ($\approx$ 106 million blocks) were deduplicated by both DEDIS and the loading mechanism. Tap:aio tests required $\approx$ 7 hours and wrote more than 395 GiB ($\approx$ 103 million blocks) into the storage. These tests generated 80 MiB of logs that were then analyzed to extract the results presented in this chapter.

4.3.3 Optimizations

In order to evaluate the *in-memory hash* cache, described in Section 4.2.5, 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, identical to DEDIS w/ cache, but using the 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 4.1 shows the storage I/O and deduplication metrics for a 40 minutes run of DEDISbench performing hotspot random writes (with a block size of 4 KiB) and for the subsequent 20 minutes, when deduplication ran isolated from the I/O workload. 5 minutes were chosen 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. In the first 40 minutes, DEDIS ran in parallel with the I/O benchmark to assess its overhead in a system with intensive storage load. 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.

Table 4.1: DEDIS optimizations results for 2 cluster nodes with a random hotspot write workload.

	DEDIS w/o cache	DEDIS w/ cache	DEDIS byte
Aggregated storage throughput (IOPS)	1710	1854	1807
Average storage latency per node (ms)	1.08	1.00	1.05
Aggregated deduplication throughput (MiB/s)	2.14	28.27	4.40

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 2.14 MiB/s to 28.27 MiB/s.[1] This gain is achievable because block digests are pre-calculated 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 4 KiB blocks from the storage. In terms of metadata space, the on-disk Berkeley DB required 15 MiB of disk space per server. The in-memory cache was configured to use 16 MiB 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 was similar when deduplication was running in parallel with the I/O benchmark and when it was running isolated.

In the DEDIS w/ cache test, the hotspot mechanism avoided approximately 80% of CoW operations (≈ 300,000 operations per server). As explained previously, CoW operations are costly for storage writes latency so, this reduction is important for achieving the results discussed in the next section.

Table 4.1 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].

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 our prototype in a larger cluster.

4.3.4 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

[1] In the DEDIS w/ cache test, the *D. Finder* module processed more than 1 million operations.

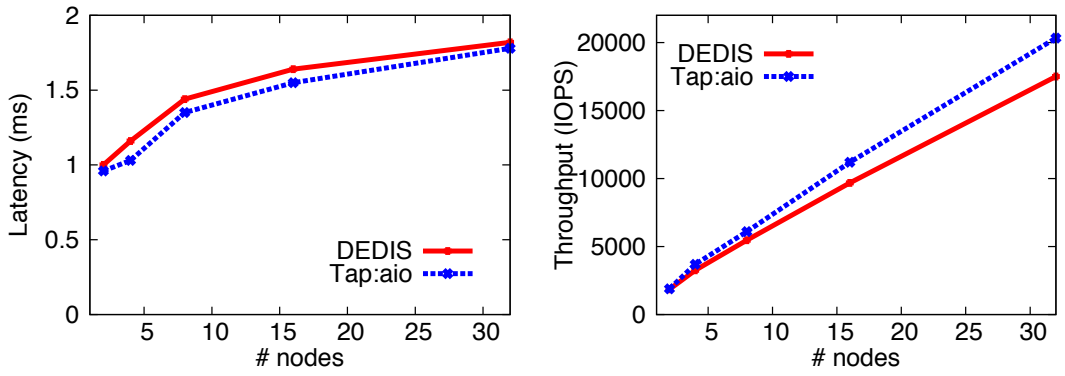
instance, and local DEDIS components. The *extent* service was the only component that ran in an independent server. In order to assess overhead, we compared it with the default Blktap driver for asynchronous I/O, named Tap:aio, that was the base to implement the *interceptor* module and does not perform deduplication [Citrix Systems, Inc 2014]. 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. Again, we started our evaluation with 2 servers to include DDI replication costs in the results.

Our prototype 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. In DEDIS tests, deduplication was executed in parallel with the I/O benchmark and the periods of *GC* and *Share* iterations were the same as the ones used in the previous tests.

Storage writes

Figures 4.6(a) and 4.6(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, for both DEDIS and Tap:aio. It occurs because in a symmetrical infrastructure, 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 our system introduces low overhead in a worst case scenario when deduplication and intensive I/O are running concurrently. Also, the overhead as a percentage is not directly affected by the number of cluster nodes meaning that our design scales along with the storage backend.



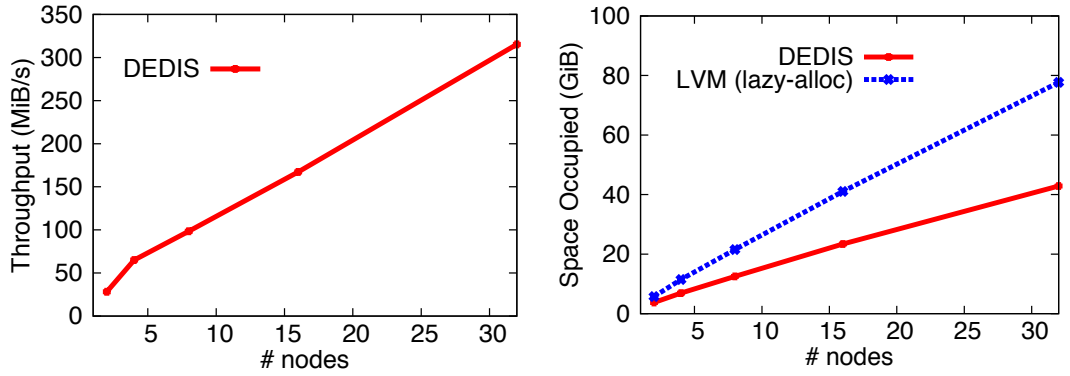
(a) Average storage latency per node. (b) Aggregated storage throughput.

Figure 4.6: DEDIS and Tap:aio results for up to 32 cluster nodes with a random hotspot write workload.

Deduplication

Figure 4.7(a) shows the aggregated deduplication throughput for the same tests which scales 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 10 MiB/s, which is a clear improvement over previous work where blocks are shared at $\approx$ 2.6 MiB/s [Clements et al. 2009].

When more VMs are writing data into the storage, there is an higher probability of finding more duplicates across their volumes. This is shown in table 4.2 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



(a) Aggregated deduplication throughput.

(b) Storage space occupied.

Figure 4.7: Deduplication results for up to 32 cluster nodes with a random hotspot write workload.

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, pre-allocated blocks of these 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 blocks from VM images were not deduplicated and were latter lazily-allocated so we have not included them 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 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.

Table 4.2: 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

For the experiment with 32 VMs, the loading mechanism deduplicated approximately 31 GiB. While DEDISbench was running and, in the subsequent 20 min-

utes, approximately 5.9 GiB 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 DEDISbench hotspot workload simulates a high percentage of re-write 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 4.7(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 a LVM system without deduplication but that supports lazy-allocation of unused blocks. Our approach used approximately 50% of the space that the LVM system would require. Moreover, a storage system without lazy allocation would require 640 GiB for storing the 32 VM volumes instead of the 43 GiB 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 is clearly compensated by the space savings.

Resources

Table 4.3 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 our system required more RAM for its

Table 4.3: Average resource consumption, per node, for the hotspot random write test with 32 cluster nodes.

	CPU (%)	RAM (MiB)	Network (KiB/s)	Persistent metadata
Tap:aio	3.90	6.25	-	-
DEDIS	6.44	197.25	247.89	96.10

in-memory caches and for performing deduplication. Nevertheless, less than 3% of the total RAM of each cluster node was used. 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 250 KiB/s. Regarding metadata consumption, the DDI, DEDIS and the loading mechanism required 96.10 MiB of storage space per server. Globally, for the 32 servers, it were used $\approx$ 3 GiB of storage space that, were clearly compensated by the 37 GiB of deduplicated space.

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 2 GiB of persistent metadata for indexing $\approx$ 1 TB of storage blocks.

4.3.5 Read performance

A setting with two servers was also used for assessing the 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 4.4 shows that DEDIS outperforms Tap:aio in both storage latency and throughput. This 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 2010]. As a matter of fact,

this effect was also verified in Chapter 3 for the LessFS deduplication system. To conclude, these results prove that our design has negligible impact in storage read requests and that, in some cases, deduplication can even increase random read performance.

Table 4.4: DEDIS and Tap:aio results for 2 cluster nodes with a random hotspot read workload.

	Aggregated storage throughput (IOPS)	Average latency per node (ms)
Tap:aio	8238	0.24
DEDIS	8698	0.23

4.3.6 Throttling deduplication and garbage collection

Finally, we measured the performance impact in both storage throughput and latency for distinct deduplication and garbage collection throughputs. More specifically, we have limited the throughput of *D. Finder* and *GC* modules per cluster node, and compared the storage overhead of these limited versions with the overhead of a run without any limit. DEDIS was evaluated in 32 cluster nodes with a random write workload. The remaining experimental settings were identical to the ones used in Section 4.3.4.

Table 4.5 shows that the non-limited version of DEDIS achieves 2677 aliasing operations per second for each cluster node (≈10 MiB/s). Surprisingly, when deduplication is limited to 1000 ops/s per node, the impact in storage requests latency and throughput is higher. As explained previously, DEDIS deduplication relies on batching for reducing the impact of logging and remote calls to DDI. When the throughput is limited, the size of each batch is smaller while more batches are required for processing the same amount of operations, which helps explaining the increased storage overhead. On the other hand, when the throughput limit approaches the 2677 ops/s, the storage performance becomes closer to the one observed for the baseline non-limited test. In fact, for the 3000 and 4000 ops/s limits, the average deduplication throughput is no longer reaching the defined upper bound.

Similarly, in Table 4.6 the values for storage latency and throughput are similar when the number of CoW operations processed per second with the *GC*

Table 4.5: DEDIS results with deduplication throttling for 32 cluster nodes.

Deduplication throughput upper limit per node (ops/s)	-	1000	2000	3000	4000
Aggregated storage throughput (IOPS)	17733	13347	17029	17587	17580
Average storage latency per node (ms)	1.82	2.41	1.87	1.82	1.81
Average deduplication throughput per node (ops/s)	2677	934	1765	2191	2384

module are limited. Without any restriction, the module processes 63 ops/s for each cluster node, which is an acceptable value as only some write operations generate CoWs and, up to 80% of these are prevented with our hotspot avoidance mechanism. As expected, when the upper bound is increased, the average *GC* throughput is closer to the non-limited one. It is also important to refer that, for all the previous tests, the usage of CPU, RAM, network and metadata is similar across them.

Table 4.6: DEDIS results with garbage collection throttling for 32 cluster nodes.

Garbage collection throughput upper limit per node (ops/s)	-	20	40	80	120
Aggregated storage throughput (IOPS)	17733	17712	17600	17610	17557
Average storage latency per node (ms)	1.82	1.80	1.82	1.81	1.82
Average garbage collection throughput per node (ops/s)	63	18	28	38	45

These results prove, once again, that deduplication and garbage collection can run simultaneously with storage I/O load. Moreover, since costly operations are excluded from the critical storage path, these operations can run at the maximum throughput without a significant impact. This is an important distinction from previous systems that are only capable of running deduplication in off-peak periods, thus requiring extra temporary storage space and higher deduplication throughput for processing the additional storage backlog.

4.4 Related work

As explained in Chapter 2, traditional deduplication systems target archival and backup data. Although these systems share common assumptions with primary

deduplication ones, there are important distinctions. Firstly, primary storage deduplication must maintain low storage latency or, in other words, a primary deduplication system must aim at the same storage latency as a raw storage system without deduplication. As another difference, although data updates and deletes are supported in some backup systems, primary data is expected to be re-written more frequently, thus increasing the number of costly CoW operations and the complexity of reference management and garbage collection. In previous work, some archival and backup deduplication systems were extended to export file system semantics with moderate I/O performance [Ungureanu et al. 2010, Liguori and Van Hensbergen 2008, Lessfs 2014]. These systems achieve good performance for stream I/O (sequential reads and writes), while supporting random block storage requests but with unacceptable performance for the primary storage environment targeted by DEDIS.

Recently, live volume deduplication in cluster and enterprise scale systems is emerging. LVM systems with snapshot capabilities avoid duplicating data but only among snapshots of VM volumes and golden VM images with common ancestors [Meyer et al. 2008]. Other systems like ZFS are designed for enterprise storage appliances and require large RAM capacities for indexing chunks and enabling efficient deduplication [OpenSolaris 2014].

These issues 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 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. In fact, this issue alone adds 10% of overhead in storage requests, which is practically the same value for the overhead of the whole DEDIS system. This penalty along with the extra latency added by CoW operations, that require ≈10 ms to complete, 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 policy avoids sharing fresh files that are more prone to generate CoW operations.

Recently, several optimizations were proposed 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. Briefly, these systems use multi-layer Bloom filters, combine Bloom filters and disk layouts exploiting spatial locality, use client-side caches for holding chunks being frequently modified and avoid processing them until strictly necessary, and use cache mechanisms that explore both temporal and spatial data locality at the cost of loosing some deduplication accuracy [Tsuchiya and Watanabe 2011, Ng et al. 2011, Srinivasan et al. 2012, Xun et al. 2014, Opendedup 2014]. Then, deduplication accuracy can be increased by extracting statistical information from storage workloads and using it for optimizing the pre-fetch of disk signatures into cache [Wildani et al. 2013]. Also, the RAM space used for caching can be dynamically optimized for the read and index caches, according to the current storage access pattern [Mao et al. 2014b]. 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]. However, even with these optimizations, most of the current 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 fact, Liquid and Opendedup are the only systems supporting global in-line deduplication [Xun et al. 2014, Opendedup 2014]. However, these approaches introduce a significative impact in storage requests latency, thus being limited to workloads with moderate latency requirements.

DDE and DeDe are the systems that most resemble DEDIS. However, our system is fully-decentralized and does not dependent 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 4.2.5, DeDe's mechanism to tentatively mark addresses as CoW is implemented with the aid of 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.

To sum up, in comparison with other archival, backup, and primary 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. 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. Exact deduplication is performed across all cluster nodes, *i.e.*, all stored chunks are compared against each other, thus having optimal deduplication gain. Finally, deduplication is decentralized so each cluster node performs deduplication tasks independently and concurrently. This is a major distinction from using a single centralized storage appliance with built-in deduplication [Bolosky et al.

2000].

4.5 Discussion

In this chapter we show the design, implementation, and evaluation of DEDIS, a dependable and distributed system that performs off-line deduplication across VMs primary storage volumes, in a cluster-wide fashion. Our design is compatible with any storage backend, distributed or centralized, as long as it exports a shared block device interface. Also, our system is fully-decentralized avoiding any single point of failure or contention thus, safely scaling-out. The DDI component is decisive to achieve the previous properties and also to enable global deduplication. Then, the optimistic off-line deduplication approach and novel optimizations allow improving deduplication performance and reliability while reducing the impact in storage requests. Namely, the hotspot avoidance and cache mechanisms are key for the results obtained in our experiments.

The evaluation of DEDIS Xen-based prototype in up to 32 cluster nodes focus on random reads and writes and shows that deduplication and primary I/O 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 10 MiB/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 cloud computing 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.

Chapter 5

Conclusions

Deduplication is a key technique to deal with the current explosion of digital information in cloud computing infrastructures. However, finding and eliminating duplicates across primary volumes of VMs in a distributed infrastructure is not a trivial task and raises several challenges. In this thesis, we address precisely these challenges and present a deduplication system for cloud computing that is fully-decentralized, scalable, and reliable.

Although the first published storage deduplication systems have now more than twelve years, the research work in deduplication is still growing at an accelerated pace. In fact, deduplication is no longer an exclusive feature of archival and backup storage, being now also present in primary storage, RAM and SSDs. Due to the considerable amount of work on this topic, it is important to clearly know how distinct systems relate to each other and what specific problems they address. Also, this information is crucial to fully-understand why the existing solutions are not suited for the cloud computing environment.

This challenge motivated the work described in Chapter 2, where we survey the existing research on storage deduplication systems. The chapter starts by defining the concept of deduplication and presenting a novel taxonomy with the common design features, that are shared across all deduplication systems. Namely, we discuss the granularity of chunks, reliance on data locality assumptions, the timing when deduplication is performed, how chunk content is indexed to find duplicates, how duplicate chunks are shared, and the distributed scope of deduplication systems. As another contribution, the same taxonomy is then used to classify 52 deduplication systems and to explain how each system copes with

the challenges of specific storage environments. Archival and backup deduplication systems are widely explored and already exploited in several commercial storage appliances. Most of these systems assume immutable data, and trade latency for deduplication and storage I/O throughput by mainly using in-line deduplication approaches. On the other hand, in primary storage, data is mutable and storage I/O latency is critical, so the number of in-line deduplication systems is reduced and the percentage of off-line approaches raised. In RAM deduplication, most systems scan memory for duplicates to avoid intrusive mechanisms for intercepting I/O calls. Also, RAM pages are highly volatile, thus being updated more often than in other storage systems. Finally, in SSD deduplication the processor and DRAM included in the devices are used for deduplication, but both have limited capabilities that significantly restrict deduplication designs.

Since the core contribution of this dissertation is a distributed primary storage deduplication system, we require proper benchmarking tools for evaluating it. Although there are several benchmarks for traditional storage systems, only a really small number is capable of simulating realistic duplicate content, which is crucial for evaluating accurately any deduplication approach. In Chapter 3, we addressed precisely this issue by introducing DEDISbench, a synthetic micro-benchmark suitable for storage deduplication systems. As the main contribution, data written by the benchmark follows content distributions that mimic the duplicates found in real storage systems. Unlike in previous benchmarks, the written content simulates the percentage of duplicate and non-duplicated blocks found in a real storage, while also, detailing the proportion of duplicates per block. As other important novel features, DEDISbench tests can be issued at stress and nominal intensities, and a novel storage access distribution that simulates hotspot random accesses is introduced. This hotspot distribution is based on the TPC-C NURand function and it is important to test random storage I/O while maintaining cache efficiency. Also, it allows simulating a storage environment with frequent block updates, as the one assumed in DEDIS and similar systems [Clements et al. 2009].

The process of analysis and extraction of novel content distributions is fully automatic by using DEDISgen. Any real storage system can be processed with this tool that automatically generates distributions usable by DEDISbench. We exemplify this feature by analyzing three distinct real storage systems from our

research group; An archival, a backup and a primary storage. We conclude that each distribution has specific characteristics, and thus, it is important to evaluate deduplication systems with suitable storage workloads. Our comparison of DEDISbench with IOzone and Bonnie++ shows that our benchmark simulates more accurately duplicate content. Also, we show that the features of our benchmark are crucial to find new issues in two deduplication file systems, LessFS and Opendedup.

Primary storage and cloud computing distributed infrastructures raise several challenges for storage deduplication that are only addressed by few systems. To ensure that deduplication does not overly affect the performance of VMs primary storage volumes, many of these systems avoid cluster-wide deduplication. Such restriction limits the achievable deduplication gain because duplicates are not found exactly across the whole cluster. Other systems reduce the storage penalty by performing off-line deduplication in off-peak periods. However, as such periods are scarce or inexistent in cloud infrastructures, the temporary storage space required for keeping unprocessed data must be large, thus also affecting deduplication space savings.

These limitations were addressed in Chapter 4 with DEDIS, a dependable system that performs distributed deduplication across VMs primary storage volumes. Our novel optimistic off-line deduplication avoids issuing costly operations in the critical storage path and, consequently, reduces storage overhead. Duplicate blocks are found and eliminated across the whole cluster by using a distributed index of duplicates. This index enables our design to be fully-decentralized while avoiding any single point of failure or contention thus, safely scaling-out. Our system is compatible with any storage backend, distributed or centralized, as long as it exports a shared block device interface. This way, our solution is agnostic to the storage implementation and does not depend on any special storage operations, which allows exploring novel optimizations. Namely, we present a novel mechanism that avoids sharing write hotspot blocks and reduces significantly the number of costly CoW operations issued. Also, we introduce several cache mechanisms that spare costly storage accesses, thus speeding up deduplication throughput while reducing the impact in VMs storage requests. DEDIS design and optimizations are key for running deduplication in peak hours while still introducing low storage overhead. As another contribution, our design

is fault-tolerant and can withstand hash collisions in specific VM volumes by performing byte comparison of chunks before aliasing them.

The evaluation of DEDIS Xen-based prototype in up to 32 cluster nodes proves the efficiency and scalability of our design and optimizations. In fact, the results show that the overhead in random storage requests is less than 14%, even when cluster nodes are performing deduplication in parallel with a baseline single-node deduplication throughput of approximately 10 MiB/s. These values are not achievable in previous systems and allow us to conclude that exact cluster-wide deduplication and low storage overhead are attainable in cloud primary storage infrastructures, even in peak periods. Also, 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 our services, in fact, even the storage backend, where VMs volumes and DEDIS persistent metadata are stored, is composed by the local disks of the same servers.

As a final contribution, DEDIS, DEDISbench, DEDISgen and the archival, backup and primary storage distributions are open-source and publicly available for anyone to use.

5.1 Future work

Due to the broadness of the storage deduplication research field, there are still several challenges to solve. Briefly, primary, RAM, and SSD deduplication systems have received less attention and further contributions for improving deduplication throughput, reducing storage overhead, and increasing deduplication space savings can still be expected. Moreover, even in backup deduplication where the amount of work is substantially larger, these issues and others, such as reference management and scalability can be further improved.

Fault tolerance and security are other two topics that deserve further attention. Although the impact of deduplication in reliability was already studied and it was shown that a level of replication must be enforced to ensure data reliability, the design of many systems must still be improved to tolerate crash and byzantine faults [Rozier et al. 2011, Bhagwat et al. 2006]. Security must also be enforced when deduplication can be performed across data from distinct users. Convergent encryption is commonly used for ensuring this property [Cox et al.

2002, Douceur et al. 2002], but it is also possible to use other security mechanisms that on the one hand reduce deduplication gain, and on the other hand increase data privacy and security [Nath et al. 2006]. For instance, it is possible to achieve security models that hide the users identities from the storage providers while still enabling deduplication [Storer et al. 2008]. Moreover, security has also been identified as an important challenge for RAM deduplication [Suzaki et al. 2011].

Regarding specific improvements and novel features for the DEDIS system, it is necessary to complete the implementation and evaluation of the fault recovery mechanism. Also, our design can be improved to tolerate byzantine failures. Since many infrastructures are now using SSDs, it would be interesting to understand the impact of deduplication in these new drives.

The privacy of deduplicated data is also not addressed by our current design. This is an important aspect as data from distinct VMs and clients is shared by our system. It would be interesting to check if current approaches are compatible with DEDIS design and if there is any space for improving deduplication space savings while maintaining privacy [Rashid et al. 2012].

Achieving optimized deduplication for sequential I/O is a challenge that is out of the scope of this dissertation and that has been the focus of other systems [Ungureanu et al. 2010]. Similarly, we do not deal with the fragmentation introduced by deduplication, which impacts directly sequential storage reads and is also researched in the literature [Kaczmarczyk et al. 2012, Mao et al. 2014a]. 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. However, these optimizations perform poorly for random block operations. 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. Although these approaches trade off some of the deduplication space savings, it would be interesting to check if they can be applied to DEDIS. As another contribution, building an hybrid system that bundles these distinct researches and achieves low storage overhead, for both types of I/O access patterns, is also an interesting challenge for future work.

Although our system is independent from the storage backend and does not

have control over features like replication, it is also important to understand if DEDIS deduplication may have any impact in the efficiency of this feature. On the other hand, 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 the sequential layout of some groups of blocks. Such would be useful for reducing the storage fragmentation effects.

Regarding our storage deduplication benchmark, several improvements are still possible. Firstly, DEDISbench does not simulate storage properties like locality, which is an important characteristic assumed by many deduplication systems [Zhu et al. 2008]. Also, DEDISbench is a block-based benchmark so, extending it to simulate file system semantics would be a valuable contribution. For this task, several ideas from previous work could be used [Al-Rfou et al. 2010, Tarasov et al. 2012]. Analyzing more real storage systems with DEDISgen and building a public database with several workloads is also an important future contribution. This will allow evaluating more accurately distinct deduplication systems and comparing the ones with similar storage targets.

To conclude, we believe that deduplication systems and their benefits are now widely accepted by the scientific and enterprise communities, but there are still several interesting open research challenges. We envision that deduplication research will continue to grow at an accelerated pace in the forthcoming years.

Bibliography

Nitin Agrawal, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *Conference on File and Storage Technologies*, 2009. - **Cited** on page 78.

Rami Al-Rfou, Nikhil Patwardhan, and Phanindra Bhagavatula. Deduplication and Compression Benchmarking in Filebench. Technical report, 2010. - **Cited** on pages 56, 57, 78 and 120.

Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and Large CAMs for High Performance Data-intensive Networked Systems. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, 2010. - **Cited** on page 34.

Darrell Anderson. Fstress: A Flexible Network File Service Benchmark. Technical report, Duke University, 2002. - **Cited** on pages 9, 55, 57 and 78.

Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing Memory Density by using KSM. In *Proceedings of the Linux Symposium*, 2009. - **Cited** on pages 25 and 45.

Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel Klein. The Design of a Similarity Based Deduplication System. In *Proceedings of International Systems and Storage Conference (SYSTOR)*, 2009. - **Cited** on pages 5, 22, 27 and 35.

Brian Berliner. CVS II: Parallelizing Software Development. In *Proceedings of USENIX Winter Technical Conference*, 1990. - **Cited** on page 17.

Deepavali Bhagwat, Kristal Pollack, Darrell Long, Thomas Schwarz, Ethan Miller, and Jehan françois Pâris. Providing High Reliability in a Minimum Re-

dundancy Archival Storage System. In *Proceedings of International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006. - **Cited** on pages 19 and 118.

Deepavali Bhagwat, Kave Eshghi, Darrell Long, and Mark Lillibridge. 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)*, 2009. - **Cited** on pages 8, 19, 26, 29 and 36.

Deepak Bobbarjung, Suresh Jagannathan, and Cezary Dubnicki. Improving Duplicate Elimination in Storage Systems. *ACM Transactions on Storage*, 2(4), 2006. - **Cited** on page 21.

William Bolosky, Scott Corbin, David Goebel, and John Douceur. Single Instance Storage in Windows 2000. In *Proceedings of USENIX Windows System Symposium (WSS)*, 2000. - **Cited** on pages 6, 15, 20, 24, 25, 26, 27, 30 and 112.

Fabiano Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. Memory Efficient Sanitization of a Deduplicated Storage System. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013. - **Cited** on pages 18 and 28.

Andrei Broder. Some Applications of Rabin's Fingerprinting Method. In *Sequences II: Methods in Communications, Security, and Computer Science*, 1993. - **Cited** on page 25.

Andrei Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences*, 1997. - **Cited** on page 26.

Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4), 1997. - **Cited** on pages 44 and 52.

Randal Burns and Darrell Long. Efficient Distributed Backup with Delta Compression. In *Proceedings of Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, 1997. - **Cited** on pages 17 and 27.

Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011. - **Cited** on pages 6, 15, 25, 26, 28, 48 and 49.

Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. CMD: Classification-based Memory Deduplication Through Page Access Characteristics. In *Proceedings of International Conference on Virtual Execution Environments (VEE)*, 2014. - **Cited** on pages 46 and 52.

David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John Stevenson, and Omid Azizi. HICAMP: Architectural Support for Efficient Concurrency-Safe Shared Structured Data Dccess. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012. - **Cited** on page 47.

Citrix Systems, Inc. Blktap documentation. `http://wiki.xen.org/wiki/Blktap2`, 2014. - **Cited** on pages 96 and 103.

Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2009. - **Cited** on pages 6, 7, 8, 10, 11, 18, 22, 25, 29, 40, 54, 55, 59, 81, 82, 84, 94, 99, 104, 110, 113 and 116.

Russell Coker. Bonnie++ web page. `http://www.coker.com.au/bonnie++/`, 2014. - **Cited** on pages 9, 55, 56, 64 and 78.

Christian Collberg, John Hartman, Sridivya Babu, and Sharath Udupa. Slinky: Static Linking Reloaded. In *Proceedings USENIX Annual Technical Conference (ATC)*, 2005. - **Cited** on page 46.

Cornel Constantinescu, Joseph Glider, and David Chambliss. Mixing Deduplication and Compression on Active Data Sets. In *Proceedings of Data Compression Conference (DCC)*, 2011. - **Cited** on pages 20 and 28.

Landon Cox, Christopher Murray, and Brian Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002. - **Cited** on pages 6, 17, 24, 28, 34 and 118.

Dan Iacono. Enterprise Storage: Efficient,Virtualized and Flash Optimized. *IDC White Paper*, 2013. - **Cited** on page 7.

Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding Up Inline Storage Deduplication using Flash Memory. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2010. - **Cited** on pages 31 and 34.

Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based storage. In *Proceedings of ACM's Special Interest Group on Management Of Data (SIGMOD)*, 2011. - **Cited** on page 34.

Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in Scalable Data Routing for Deduplication Clusters. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011. - **Cited** on pages 8, 29, 36, 37 and 51.

John Douceur, Atul Adya, William Bolosky, Dan Simon, and Marvin Theimer. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. Technical Report MSR-TR-2002-30, Microsoft Research, 2002. - **Cited** on pages 19, 29, 34, 35 and 119.

Fred Douglis and Arun Iyengar. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2003. - **Cited** on page 27.

Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a Scalable Secondary Storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2009. - **Cited** on pages 19, 22, 29 and 35.

Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta. Primary Data Deduplication Large Scale Study and System Design. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2012. - **Cited** on pages 6, 28, 41 and 111.

EMC. New Digital Universe Study Reveals Big Data Gap. `http://www.emc.com/about/news/press/2012/20121211-01.htm`, 2012. - **Cited** on page 5.

Kave Eshghi and Hsiu Tang. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Technical Report HPL-2005-30, Intelligent Enterprise Technologies Laboratory, 2005. URL `http://www.hpl.hp.com/techreports/2005/HPL-2005-30R1.pdf`. - **Cited** on page 21.

Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. Jumbo Store: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2007. - **Cited** on page 31.

Filebench. Filebench web page. `http://filebench.sourceforge.net`, 2014. - **Cited** on pages 57, 77 and 78.

Davide Frey, Anne-Marie Kermarrec, and Konstantinos Kloudas. Probabilistic Deduplication for Cluster-based Storage Systems. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2012. - **Cited** on pages 8 and 37.

Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2014. - **Cited** on pages 18 and 30.

Yinjin Fu, Hong Jiang, and Nong Xiao. A Scalable Inline Cluster Deduplication Framework for Big Data Protection. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, 2012. - **Cited** on pages 8, 36, 37 and 38.

Fanglu Guo and Petros Efstathopoulos. Building a High-Performance Deduplication System. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2011. - **Cited** on pages 18, 26, 28, 31, 33, 51, 53 and 111.

Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011. - **Cited** on pages 26, 48 and 50.

Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex Snoeren, George Varghese, Geoffrey Voelker, and Amin Vahdat. Difference Engine:

Harnessing Memory Redundancy in Virtual Machines. *Communications ACM*, 53, 2010. - **Cited** on pages 28 and 46.

Danny. Harnik, Benny. Pinkas, and Alexandra. Shulman-Peleg. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security Privacy*, 8(6), 2010. - **Cited** on page 19.

Hewlett-Packard Development Company , L.P. Complete storage and data protection architecture for VMware vSphere. *White Paper*, 2011. - **Cited** on pages 7, 59, 60 and 81.

Bo Hong and Darrell Long. Duplicate Data Elimination in a San File System. In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2004. - **Cited** on pages 6, 7, 8, 10, 15, 20, 25, 28, 29, 40, 54, 81, 82, 84, 110 and 113.

James Hunt, Kiem-Phong Vo, and Walter Tichy. Delta Algorithms: An Empirical Analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2), 1998. - **Cited** on page 27.

Keren Jin and Ethan Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of International Systems and Storage Conference (SYSTOR)*, 2009. - **Cited** on page 40.

Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing Impact of Data Fragmentation Caused by In-line Deduplication. In *Proceedings of International Systems and Storage Conference (SYSTOR)*, 2012. - **Cited** on pages 18, 20 and 119.

Jürgen Kaiser, Dirk Meister, André Brinkmann, and Sascha Effert. Design of an Exact Data Deduplication Cluster. In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2012. - **Cited** on pages 29, 36 and 51.

Jeffrey Katcher. PostMark: a new file system benchmark. Technical report, NetApp, 1997. - **Cited** on pages 9, 55, 57 and 78.

Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in SSDs: Model and quantitative analysis. In *Proceedings of Con-*

ference on Mass Storage Systems (MSST), 2012. - **Cited** on pages 26, 48, 50 and 51.

Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2010. - **Cited** on pages 6 and 107.

Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2010. - **Cited** on page 22.

Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2004. - **Cited** on pages 6 and 28.

Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X. - **Cited** on page 91.

Leslie Lamport et al. Tla+ toolset, 2009. URL `http://www.tlaplus.net/tools/tla-toolbox/`. - **Cited** on page 91.

Lessfs. Lessfs page. `http://www.lessfs.com/wordpress/`, 2014. - **Cited** on pages 39 and 110.

Anthony Liguori and Eric Van Hensbergen. Experiences with Content Addressable Storage and Virtual Disks. In *Proceedings of USENIX Workshop on I/O Virtualization (WIOV)*, 2008. - **Cited** on pages 39, 51 and 110.

Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2009. - **Cited** on pages 16, 19, 26, 31, 32, 51 and 111.

Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013. - **Cited** on pages 18 and 30.

Hyeontaek Lim, Bin Fan, David Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-Value Store. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, 2011. - **Cited** on page 34.

Guanlin Lu, Yu Jin, and David Du. Frequency Based Chunking for Data De-Duplication. In *Proceedings of International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010. - **Cited** on page 22.

Guanlin Lu, Youngjin Nam, and David Du. BloomStore: Bloom-Filter Based Memory-efficient Key-Value Store for Indexing of Data Deduplication on Flash. In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2012. - **Cited** on page 34.

Udi Manber. Finding Similar Files in a Large File System. In *Proceedings of USENIX Winter Technical Conference*, 1994. - **Cited** on pages 19 and 25.

Nagapramod Mandagere, Pin Zhou, Mark A. Smith, and Sandeep Uttamchandani. Demystifying Data Deduplication. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, 2008. - **Cited** on pages 10, 20 and 24.

Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. Read-Performance Optimization for Deduplication-Based Storage Systems in the Cloud. *Transactions on Storage*, 10(2), 2014a. - **Cited** on pages 18, 20 and 119.

Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. POD: Performance Oriented I/O Deduplication for Primary Storage Systems in the Cloud. In *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS)*, 2014b. - **Cited** on pages 43 and 111.

Dirk Meister and André Brinkmann. Multi-Level Comparison of Data Deduplication in a Backup Scenario. In *Proceedings of International Systems and Storage Conference (SYSTOR)*, 2009. - **Cited** on page 30.

Dirk Meister and André Brinkmann. dedupv1: Improving Deduplication Throughput Using Solid State Drives (SSD). In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2010. - **Cited** on pages 31 and 34.

Dirk Meister, André Brinkmann, and Tim Süß. File Recipe Compression in Data Deduplication Systems. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013a. - **Cited** on pages 18 and 28.

Dirk Meister, Jürgen Kaiser, and André Brinkmann. Block Locality Caching for Data Deduplication. In *Proceedings of International Systems and Storage Conference (SYSTOR)*, 2013b. - **Cited** on pages 32 and 51.

Dutch Meyer and William Bolosky. A Study of Practical Deduplication. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2011. - **Cited** on pages 6, 7, 30, 40, 59 and 60.

Dutch Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael Feeley, Norman Hutchinson, and Andrew Warfield. Parallax: Virtual Disks for Virtual Machines. In *Proceedings of European Conference on Computer Systems (EuroSys)*, 2008. - **Cited** on pages 7, 17, 81, 84 and 110.

Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More Effective Memory Deduplication Scanners Through Cross-layer Hints. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2013. - **Cited** on page 46.

Grzegorz Milos, Derek Murray, Steven Hand, and Michael Fetterman. Satori: Enlightened Page Sharing. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2009. - **Cited** on pages 46 and 47.

Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-bandwidth Network File System. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, 2001. - **Cited** on pages 6, 17 and 21.

Partho Nath, Michael Kozuch, David O'Hallaron, Jan Harkes, Mahadev Satyanarayanan, Niraj Tolia, and Matt Toups. Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2006. - **Cited** on pages 19, 22, 52 and 119.

Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick Lee, and John Lui. Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud.

In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, 2011. - **Cited** on pages 6, 7, 43, 51, 81, 98 and 111.

William Norcott. Iozone web page. `http://www.iozone.org/`, 2014. - **Cited** on pages 56, 64, 77 and 78.

Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 1999. - **Cited** on pages 60, 96 and 97.

Opendedup. Opendedup page. `http://opendedup.org`, 2014. - **Cited** on pages 42, 111 and 112.

OpenSolaris. ZFS documentation. `http://www.freebsd.org/doc/en/books/handbook/zfs.html`, 2014. - **Cited** on pages 6, 42 and 110.

Zan Ouyang, Nasir Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-Based Delta Compression of a Collection of Files. In *Proceedings of International Conference on Web Information Systems Engineering (WISE)*, 2002. - **Cited** on page 27.

Ying-Shiuan Pan, Jui-Hao Chiang, Han-Lin Li, Po-Jui Tsao, Ming-Fen Lin, and Tzi-cker Chiueh. Hypervisor Support for Efficient Memory De-duplication. In *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, 2011. - **Cited** on page 47.

Calicrates Policroniades and Ian Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2004. - **Cited** on pages 20 and 22.

Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2002. - **Cited** on pages 6, 7, 20, 22, 23, 25, 26, 27, 28, 29, 30, 60, 96 and 102.

Michael Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981. - **Cited** on page 25.

Fatema Rashid, Ali Miri, and Isaac Woungang. A Secure Data Deduplication Framework for Cloud Environments. In *Proceedings of International Conference on Privacy, Security and Trust (PST)*, 2012. - **Cited** on page 119.

Sean Rhea, Russ Cox, and Alex Pesterev. Fast, Inexpensive Content-addressed Storage in Foundation. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2008. - **Cited** on pages 23, 25, 29, 32 and 111.

Eric Rozier, William Sanders, Pin Zhou, Nagapramod Mandagere, Sandeep Uttamchandani, and Mark Yakushev. Modeling the Fault Tolerance Consequences of Deduplication. In *Proceedings of International Symposium on Reliable Distributed Systems (SRDS)*, 2011. - **Cited** on pages 19 and 118.

Leonard Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3), 1986. - **Cited** on page 38.

Prateek Sharma and Purushottam Kulkarni. Singleton: System-Wide Page Deduplication in Virtual Environments. In *Proceedings of Symposium on High Performance Distributed Computing (HPDC)*, 2012. - **Cited** on page 45.

Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta Compressed and Deduplicated Storage using Stream-Informed Locality. In *Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2012. - **Cited** on pages 28, 31, 33, 51, 52 and 111.

Dilip Simha, Maohua Lu, and Tzi-cker Chiueh. A Scalable Deduplication and Garbage Collection Engine for Incremental Backup. In *Proceedings of International Systems and Storage Conference (SYSTOR)*, 2013. - **Cited** on page 33.

Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012. - **Cited** on pages 6, 7, 8, 10, 18, 23, 24, 43, 54, 81, 82 and 111.

Mark Storer, Kevin Greenan, Darrell Long, and Ethan Miller. Secure Data Deduplication. In *Proceedings of Workshop on Storage Security and Survivability (StorageSS)*, 2008. - **Cited** on page 119.

Przemyslaw Strzelczak, Elzbieta Adamczyk, Urszula Herman-Izycka, Jakub Sakowicz, Lukasz Slusarczyk, Jaroslaw Wrona, and Cezary Dubnicki. Concurrent Deletion in a Distributed Content-Addressable Storage System with Global Deduplication. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013. - **Cited** on pages 18, 28 and 30.

Kuniyasu Suzaki, Toshiki Yagi, Kengo Iijima, Nguyen Quynh, Cyrille Artho, and Yoshihito Watanebe. Moving from Logical Sharing of Guest OS to Physical Sharing of Deduplication on Virtual Machine. In *Proceedings of Workshop on Hot Topics in Security (HotSec)*, 2010. - **Cited** on page 46.

Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication as a Threat to the Guest OS. In *Proceedings of European Workshop on Systems Security (EuroSec)*, 2011. - **Cited** on page 119.

Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating Realistic Datasets for Deduplication Analysis. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2012. - **Cited** on pages 9, 54, 55, 56, 57, 77, 79 and 120.

Transaction processing performance council. TPC-C standard specification, Revision 5.5. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, 2010. - **Cited** on pages 11 and 58.

Yoshihiro Tsuchiya and Takashi Watanabe. DBLK: Deduplication for Primary Block Storage. In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2011. - **Cited** on pages 42 and 111.

Tsuyoshi, Ozawa and Kazutaka, Morita. Accord page. `http://www.osrg.net/accord/`, 2014. - **Cited** on page 97.

Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2010. - **Cited** on pages 19, 23, 29, 39, 52, 110 and 119.

Carl Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36(SI), 2002. - **Cited** on pages 6, 15, 24 and 44.

Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. MAD2: A Scalable High-throughput Exact Deduplication Approach for Network Backup Services. In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2010. - **Cited** on pages 35 and 111.

Avani Wildani, Ethan Miller, and Ohad Rodeh. HANDS: A Heuristically Arranged Non-Backup In-line Deduplication System. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2013. - **Cited** on pages 43 and 111.

Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark Corner. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In *Proceedings of Conference on Virtual Execution Environments (VEE)*, 2009. - **Cited** on page 47.

Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2011. - **Cited** on pages 36, 37 and 38.

Zhao Xun, Zhang Yang, Wu Yongwei, Chen Kang, Jiang Jinlei, and Li Keqin. Liquid: A Scalable Deduplication File System for Virtual Machine Images. *IEEE Transactions on Parallel and Distributed Systems*, 1(1), 2014. - **Cited** on pages 42, 53, 54, 111 and 112.

Tianming Yang, Dan Feng, Zhongying Niu, and Ya-ping Wan. Scalable High Performance De-duplication Backup via Hash Join. *Journal of Zhejiang University - Science C*, 11(5), 2010a. - **Cited** on pages 22, 29 and 38.

Tianming Yang, Hong Jiang, Dan Feng, Zhongying Niu, Ke Zhou, and Yaping Wan. DEBAR: A Scalable High-performance De-duplication Storage System for Backup and Archiving. In *Proceedings of International Parallel & Distributed Processing Symposium (IPDPS)*, 2010b. - **Cited** on pages 29, 30 and 38.

Lawrence You and Christos Karamanolis. Evaluation of Efficient Archival Storage Techniques. In *Proceedings of Conference on Mass Storage Systems (MSST)*, 2004. - **Cited** on pages 27 and 30.

Lawrence You, Kristal Pollack, and Darrell Long. Deep Store: An Archival Storage System Architecture. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2005. - **Cited** on pages 5, 8, 28, 29, 30 and 36.

Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2008. - **Cited** on pages 5, 18, 23, 31, 32, 111 and 120.

Appendix A

CAL specification

MODULE *dedup*

EXTENDS *FiniteSets*, *Sequences*, *Naturals*, *TLC*

CONSTANT *vids*, *pids*, *cids*, *service*, *itt*, *istore*

```
–algorithm Deduplication{
variable
  \* Logical to physical map initial values
  tt = itt, \* initial physical mapping
  cow = [i ∈ vids ↦ FALSE], \* CoW marker is unset for all addresses
  lock = [i ∈ vids ↦ FALSE], \* no addresses are locked
  \* DDI is empty
  dht ∈ [{} → {}], \* DDI hash to physical address map
  rc ∈ [{} → {}], \* DDI reference counting map
  \* Free blocks queue and unreferenced queue
  free = pids \ {i ∈ pids : ∃ vid ∈ vids : tt[vid] = i}, \* free blocks addresses
  queue = ⟨⟩, \* copied blocks addresses to be garbage collected
  \* Storage
  store = istore; \* current content of physical blocks
process(v ∈ vids) \* we use a logical process for each logical page
variable
  p, \* currently known physical address
  c, \* currently known content hash
  r = store[tt[self]], \* last storage content read
  w = store[tt[self]]; \* last storage content written
{
```

```
request : while(TRUE){ \ * wait here for OS request
  either{ \ * Read request
    lock[self] := TRUE; \ * lock logical address
    p := tt[self]; \ * get physical address from map
    \ * Wait for read storage callback
    readcb : r := store[p]; \ * r has the content read from the storage
    lock[self] := FALSE; \ * unlock logical address
  }or{ \ * Write request
    with(i ∈ cids){
      c := i; \ * choose one arbitrary value to write
    };
    lock[self] := TRUE; \ * lock logical address
    if(cow[self]){ \ * if address is marked for CoW
      alloc : await(free ≠ {}); \ * wait until free blocks queue has elements
      p := CHOOSE i ∈ free : TRUE; \ * get a free block address
      free := free \ {p}; \ * remove it from free blocks queue
      \ * wait for write callback
      cwritecb : store[p] := c; \ * content is written in the free block at the storage
      queue := queue ∘ ⟨tt[self]⟩; \ * add CoW address to unreferenced queue
      tt[self] := p; \ * update map with the free block address
      cow[self] := FALSE; \ * remove CoW marker
      lock[self] := FALSE; \ * unlock logical address
      w := c;
      r := c;
    }else{
      \ * wait for regular write callback
      nwritecb : store[tt[self]] := c; \ * content is written at the storage address
      lock[self] := FALSE; \ * unlock logical address
      w := c;
      r := c;
    };
```

```
    }
  }
};
process(srv = service) \ * we use a logical daemon process for share and GC
variable
  v, \ * currently selected logical address
  p; \ * currently selected physical address
{
  step : while(TRUE){ \ * Periodically
    either{ \ * Garbage collection
      await(queue ≠ ⟨⟩); \ * wait for elements at unreferenced queue
      with(q = Head(queue)){ \ * get first element from queue
        queue := Tail(queue);
        p := q;
      };
      gcalc : with(hash = store[p]){ \ * read block and compute hash
        if(rc[hash] = 1){ \ * if block has only this reference
          if(dht[hash] ≠ p){ \ * compare queue address with the DDI one
            free := free ∪ {dht[hash], p}; \ * also free queue address if a match is not found
          }else{
            free := free ∪ {dht[hash]}; \ * free DDI address if a match is found
          };
          dht := [i ∈ DOMAIN dht \ {hash} ↦ dht[i]]; \ * remove DDI entry address
          rc := [i ∈ DOMAIN rc \ {hash} ↦ rc[i]]; \ * remove DDI entry reference counter
        }else{
          if(dht[hash] ≠ p){ \ * compare queue address with the DDI one
            free := free ∪ {p}; \ * free queue address if a match is not found
          };
          rc[hash] := rc[hash] − 1; \ * decrement block reference at DDI
        };
      };
```

```
    }or{ \ * Sharing
      with(vid ∈ {i ∈ vids : ¬cow[i] ∧ ¬lock[i]}){ \ * address is not locked or marked for CoW
        v := vid; \ * logical address to share
        p := tt[vid]; \ * corresponding physical address to share
        cow[vid] := TRUE; \ * mark for CoW
      }; \ * lock is released
      scalc : with(hash = store[p]){ \ * read block and compute hash
        if(hash ∈ DOMAIN dht){ \ * if hash is at the DDI
          rc[hash] := rc[hash] + 1; \ * increment reference counter at the DDI
          if(¬lock[v] ∧ tt[v] = p){ \ * lock address and check for concurrent modification
            tt[v] := dht[hash]; \ * update map with DDI address
            free := free ∪ {p}; \ * free duplicate block
          };
        }else{
          dht := dht @@ (hash :> p); \ * insert new DDI entry physical address
          rc := rc @@ (hash :> 1); \ * insert new DDI entry reference counter
        };
      };
    };
  };
}
}
```

BEGIN TRANSLATION

END TRANSLATION

$$Correct \triangleq \Box\forall\, vid \in vids : r[vid] = w[vid]$$

$$NoAlias \triangleq \Box\forall\, pid \in free,\ vid \in vids,\ hash \in \text{DOMAIN } dht : tt[vid] \neq pid \wedge dht[hash] \neq pid$$